PostgreSQL 13.1文書

PostgreSQLグローバル開発グループ

PostgreSQL 13.1文書

PostgreSQLグローバル開発グループ

法的告知

目次

図の一覧

表の一覧

例の一覧

はじめに

本書はPostgreSQLのオフィシャルドキュメントです。PostgreSQLソフトウェアの開発と並行して、PostgreSQL開発者とそれ以外のボランティアにより書かれてきました。現在のバージョンのPostgreSQLが公式にサポートする全ての機能を網羅しています。

PostgreSQLの膨大な情報を取り扱いやすくするため、本書はいくつかの部分で構成されています。それぞれの部分は異なる層のユーザ、あるいはPostgreSQLの経験の程度が異なるユーザを対象にしています。

- パート Iは新規ユーザ向けの形式ばらない入門編です。
- パート IIはデータ型や関数などのSQL問い合わせ言語環境に加え、ユーザレベルのパフォーマンスチューニングについても文書化したものです。PostgreSQLユーザは皆、一読すべきです。
- パート IIIはサーバのインストールと管理について記載しています。PostgreSQLサーバを実行しているユーザは全て、個人用やその他用途などの目的にかかわらず、この部分を読むべきです。
- パート IVはPostgreSQLのクライアントプログラム用プログラミングインタフェースについて記載しています。
- パート Vには、上級ユーザ向けのサーバの拡張機能についての情報があります。ここで取り扱う話題には、ユーザ定義型やユーザ定義関数などがあります。
- パート VIにはSQLコマンド、クライアントおよびサーバプログラムに関するリファレンス情報があります。この部分は、コマンドあるいはプログラムの順に並んだ構造的な情報によって他の部分を補助します。
- パート VIIにはPostgreSQL開発者の役に立つかもしれない様々な情報があります。

1. PostgreSQLとは?

PostgreSQLは、カリフォルニア大学バークレイ校のコンピュータサイエンス学科で開発されたPOSTGRES, Version 4.2[1]をベースにしたオブジェクトリレーショナルデータベース管理システム（ORDBMS）です。POSTGRESは後からいくつかの商用データベースで利用できるようになった、多くの概念についての先駆となりました。

PostgreSQLはオリジナルのバークレイ校のソースコードを引き継ぐオープンソースのデータベースで、標準SQLの大部分やその他の最新の機能をサポートしています。

- 複雑な問い合わせ
- 外部キー
- トリガ
- 更新可能ビュー
- トランザクションの一貫性
- 多版同時実行制御

[1] https://dsf.berkeley.edu/postgres.html

またPostgreSQLは、例えば新規に以下のものを付け加えることで、いろいろな方法でユーザが拡張することができます。

- データ型
- 関数
- 演算子
- 集約関数
- インデックスメソッド
- 手続き言語

さらに自由主義的ライセンス条件により、PostgreSQLは誰にでも、その使用、変更 、配布を個人使用、商用、学術など目的に限らず無償で可能です。

2. PostgreSQL小史

現在PostgreSQLとして知られるオブジェクト指向リレーショナルデータベース管理システムは、カリフォルニア大学バークレイ校で作成されたPOSTGRESパッケージから派生しています。20年以上にわたる開発の背景を持ち、PostgreSQLは現在最も先端的な、どこからでも入手可能なオープンソースデータベースです。

2.1. バークレイ校POSTGRESプロジェクト

Michael Stonebraker教授率いるPOSTGRESプロジェクトにはその後援者としてDefense Advanced Research Projects Agency（DARPA）、National Science Foundation（NSF）、そしてESL, Inc.が名を連ねていました。POSTGRESの実装は1986年から始まりました。当初のシステムに対する概念は[ston86]で発表され、最初のデータモデルの定義は[rowe87]で紹介されました。当時のルールシステムの設計は[ston87a]で説明されました。ストレージ管理の理論や構造は[ston87b]で詳しく示されました。

POSTGRESはそれ以来いくつかの主要なリリースを重ねてきました。最初の「デモウェア」システムが1987年に使用可能になり、1988年のACM-SIGMODコンファレンスで紹介されました。[ston90a]で説明されているバージョン1は、1989年6月に一部の外部ユーザにリリースされ、最初のルールシステムに対する批評の結果（[ston89]）を基にルールシステムは再設計（[ston90b]）され、バージョン2が1990年6月に新しいルールシステムを実装してリリースされました。バージョン3は1991年に登場し、複数ストレージ管理機構、改善された問い合わせエクゼキュータ、書き直しされたルールシステムのサポートが追加されました。Postgres95まで引き続いた各リリース（下記を参照）のほとんどの部分では、移植性と信頼性に焦点を合わせていました。

POSTGRESは様々な研究用、そして実際の業務アプリケーションを実装するために使われてきています。その中には、金融データ分析システム、ジェットエンジン性能分析パッケージ、小惑星追跡データベース、医療情報データベース、いくつかの地図情報処理システム（GIS）などがあります。POSTGRESはさらに、いくつかの大学で教材としても使われています。最後に、Illustra Information Technologies社（後に、Informix[2]社に吸収合併され、現在はIBM[3]社所有）がコードを整理し商用化しました。POSTGRESは1992年後半から始まったSequoia 2000 scientific computing project[4]の主要なデータ管理システムになりました。

[2] https://www.ibm.com/analytics/informix

[3] https://www.ibm.com/

[4] http://meteora.ucsd.edu/s2k/s2k_home.html

1993年には外部ユーザコミュニティの大きさは約2倍に膨れました。データベースの研究に費されるべき時間がプロトタイプコードの保守とサポートに取られていることが次第に明らかになってきました。このサポートの重荷を減らすために、バークレイPOSTGRESプロジェクトはバージョン4.2をもって公式に終了しました。

2.2. Postgres95

1994年、Andrew YuとJolly ChenがPOSTGRESにSQL言語インタプリタを追加しました。引き続いてPostgres95がWeb上でリリースされ、オリジナルのPOSTGRESバークレイコードのオープンソースによる後続として世界への独自の道を歩み始めました。

Postgres95のコードは全てANSI C準拠となるように書き直され、これまでに膨れ上がったコードの25%を整理することで身軽になりました。多くの内部改造によって性能と保守性が改善されました。Postgres95リリース1.0.xは、POSTGRESバージョン4.2に比べWisconsin Benchmarkで約30〜50%速く動作しました。バグ修正以外では、下記の大きな改善がありました。

- （サーバに実装された）SQLが問い合わせ言語PostQUELに取って代わりました。（インタフェースライブラリlibpqはPostQUELにちなんで命名されました。）PostgreSQLになる以前は副問い合わせはサポートされていませんでしたが（下記を参照）、Postgres95ではユーザ定義SQL関数で模倣できました。集約は再実装されました。`GROUP BY`問い合わせ句のサポートも追加されました。

- GNUのReadlineを使った新しいプログラム（psql）が、対話式SQL問い合わせのために用意されました。これは古いmonitorプログラムにほぼ取って代わるものになりました。

- 新しいフロントエンドライブラリ、`libpgtcl`がTclベースのクライアントをサポートしました。サンプルシェル`pgtclsh`はTclとPostgres95サーバとインタフェースをとる新規Tclコマンドを提供しました。

- ラージオブジェクトインタフェースがオーバーホールされました。転置ラージオブジェクトが唯一のラージオブジェクト格納機構でした（転置ファイルシステムは削除されました）。

- インスタンスレベルのルールシステムが削除されました。ルールは書き換えルールとしてまだ利用できました。

- 標準SQLの機能やPostgres95の機能を紹介したチュートリアルがソースコードとともに配布されました。

- GNU makeが（BSD makeの代わりに）構築に使われました。また、Postgres95はパッチの当たっていないGCCでコンパイルできました（doubleにおけるデータ整列が修正されたおかげです）。

2.3. PostgreSQL

1996年になると「Postgres95」という名前が時代の試練に耐えられなくなったことが明らかになりました。そこで、オリジナルのPOSTGRESとSQLの能力を持つ、より最近のバージョンとの関係を反映する、PostgreSQLという新しい名前を選びました。同時に、元々バークレイPOSTGRESプロジェクトで始まった連番に戻す番号の6.0で始まるバージョン番号を設定しました。

歴史的な理由か発音しやすいためか、多くの人々はPostgreSQLを「Postgres」と参照しつづけています。（いまではすべて大文字と記載することは稀です。）こうした使い方は愛称や別名として広く受け入れられています。

Postgres95開発で重視されたのは、サーバのコードに内在する問題点を特定し、原因を理解することでした。PostgreSQLにおいては、全ての分野に目を留めているとしても、保守作業を続けつつ特徴や能力を強化することに重点が移りました。

その後PostgreSQLがたどった足跡の詳細は付録Eを参照してください。

3. 規約

以下の規約はコマンドの概要に対して使用されます。大括弧（[および]）はオプションである部分を示します。（Tcl コマンドの概要では疑問符（?）をTclのしきたりとして代わりに使います。） 中括弧（{および}）そして縦線（|）はいずれか1つを選択しなければならないことを意味します。点々（...）は前にある要素が繰り返されるという意味です。

明確性を強調する場面では、SQLコマンドの前には=>を付け、シェルコマンドの前には$を付けます。とは言っても通常プロンプトは示しません。

一般的に管理者とは、サーバのインストールと運営の責にあたる人のことです。ユーザとは、PostgreSQLシステムの一部でも活用したり、もしくはこれから使用する誰もがその対象です。これらの用語は厳密に解釈されるものではありません。本書はシステム管理の手順について特定の固定観念を想定していません。

4. より進んだ情報

ドキュメント、つまり本書の他に、PostgreSQLについてのその他のリソースがあります。

Wiki

PostgreSQL wiki[5]はプロジェクトのFAQ[6]（よくある質問）リスト、TODO[7]リスト、およびさらに多くの話題の情報を収容しています。

Webサイト

PostgreSQLのWebサイト[8]には、最新のリリースに関する詳細についてや、PostgreSQLの利用・操作をする上で生産性をより高める情報があります。

メーリングリスト

メーリングリストは、ユーザが質問の答えを得ること、他のユーザと経験を共有すること、開発者に連絡することに適した所です。詳細はPostgreSQLのWebサイトで調べてください。

あなた自身！

PostgreSQLはオープンソースプロジェクトです。つまり、ユーザコミュニティの継続的なサポートに依存しているのです。PostgreSQLを使い始めてからは、ドキュメントやメーリングリストを通じて他の人から

[5] https://wiki.postgresql.org

[6] https://wiki.postgresql.org/wiki/Frequently_Asked_Questions

[7] https://wiki.postgresql.org/wiki/Todo

[8] https://www.postgresql.org

の助けに頼ることになるでしょう。知識を得たらそれを今度は貢献することを考えてください。メーリングリストを講読し質問に回答してください。もし、ドキュメントに載っていないことを学んだら、それを書いて寄稿してください。もし、コードに機能を追加したら、それを寄稿してください。

5. バグレポートガイドライン

PostgreSQLに関してバグを発見した場合、ぜひ我々に連絡してください。最大限の注意を払っても、全てのプラットフォーム、全ての環境でPostgreSQLの機能全てが正常に動くことは保証できませんので、バグレポートはPostgreSQLをより信頼性の高いものにするために、大変重要になります。

下記の助言は、有効に活用され得るバグレポートを作成する際に、作成者を支援することを狙ったものです。これに従う義務はありませんが、沿った方がより有益なものとなるでしょう。

私たちは、すべてのバグを直ちに修正することを約束することはできません。そのバグが明確で、重大で、あるいは他の多くのユーザにも影響を与えるものであれば、誰かがすぐに調査する可能性が高くなるでしょう。また、より新しいバージョンに変えて、そこでも同じようなことが起こるかを確認してもらうように伝える場合もあります。あるいは、現在計画中の大きな変更が終了するまで、バグを修正できないと判断する場合もあります。また、単に修正が非常に困難であり、より重要な他の事項があることもあります。早急に処置が必要な場合は、商用サポートの購入を検討してください。

5.1. バグの特定

バグ報告を行う前に、ドキュメントを読み、もう一度読み返し、実行しようとしている処理が実行可能かどうか確認してください。実行可能かどうかが不明な場合は、その旨を報告してください。それはドキュメントのバグです。また、ドキュメントに書かれていることと実際の結果が異なる場合にはバグとなります。以下のような状況が考えられます。但し、これらに限定しているわけではありません。

- プログラムが致命的なシグナル、またはオペレーティングシステムのエラーメッセージで終了し、それがプログラム内部の問題を指摘している場合。（逆に、「disk full」のようなメッセージはプログラムの問題ではありませんから、この場合は自分で修正してください。）
- あるプログラムで、入力された値に対して間違った結果を返す場合。
- （ドキュメントで定義されている）有効な値を入力してもプログラムで受け付けない場合。
- プログラムが、無効な入力値を通知やエラーメッセージなどを表示せずに受け入れる場合。但し、無効な入力と思われるものでも、拡張、あるいは過去の経緯による互換性と考えられている可能性があることに注意してください。
- サポートされているプラットフォームで、PostgreSQLが手順通りにコンパイル、ビルド、インストールできない場合。

ここでは、「プログラム」とはバックエンドプロセスだけではなく、すべての実行可能ファイルを意味します。

プログラムの実行が遅かったり、リソースを大量に使用するのは必ずしもバグではありません。アプリケーションを改善するためには、ドキュメントを読んだり、どこかのメーリングリストで尋ねてみたりしてください。標準SQLの要求に応じない場合も、その機能の互換性を明確にうたっていない限り、バグとは言えません。

以降に進む前に、TODOリストやFAQを参照して、そのバグが既知のものかどうか確認してください。もしTODOリストの情報を読み取ることができなければ、問題を報告してください。少なくともTODOリストを分かりやすくすることができます。

5.2. 報告すべきこと

バグ報告で最も重要なことは、全ての事実を、そして事実のみを明確に記述することです。何が起こったのか、または、プログラムのどこが問題か、「何々が起こっているようだ」などの憶測や推測を記述しないでください。実装にさほど詳しくない方の推測は正しくない場合があり、有効なバグ報告になりません。実装に精通している方の場合であっても、根拠のある説明は参考情報となりますが、やはり正しい事実が一番役に立ちます。バグを修正するためには、まず開発者自身がそのバグを再現する必要があります。ありのままの事実を報告することは、単刀直入（多くの場合は画面からメッセージをコピー&ペーストを行うのみ）ですが、えてして、重要でないだろうと想像したり、省いても理解してもらえるだろうという思い込みによって、重要な情報が洩れてしまう場合がかなり多くあります。

全てのバグ報告では、下記の内容が記述されていなければいけません。

- 問題を再現できるように、プログラムの起動から行った作業を順序通りに記述してください。例えば、出力がテーブルのデータに依存するならば、単にSELECT文を記述していても、それ以前に行われた、CREATE TABLEやINSERT文が記述されていなければ十分とはいえません。我々は、ユーザのデータベーススキーマをリバースエンジニアリングするための時間を取ることができませんし、推測してデータを構築したとしても、おそらく間違えることになるでしょう。

  SQL関連の問題のテストケースの最適な書式は、psqlフロントエンドに直接読み込ませて問題を再現することができるファイルを用意することです（~/.psqlrcの起動ファイルに何も書かれていないことを確認してください）。このファイルを簡単に作成するには、pg_dumpを使ってテーブル定義とその状況を再現させるために必要なデータを取り出し、問題の起こった問い合わせを追加します。サンプルデータの量を減らすことは、推奨されますが必ずしも必要ではありません。どのような方法であれ、バグが再現できればよいのです。

  アプリケーションがPHPなど何か別のクライアントインタフェースを使用している場合、問題となる問い合わせを切り出してください。問題を再現させるために我々がWebサーバをセットアップすることは、おそらくないでしょう。どのような場合においても、正確な入力ファイルを提供することを忘れないでください。「大規模ファイル」や「中規模データベース」で発生する問題である、といった推測は行わないでください。こうした情報は不正確過ぎて役に立ちません。

- 得られた出力そのもの。「うまくいかなかった」、あるいは「クラッシュした」といった報告はしないでください。エラーメッセージがあるならば、たとえ意味が理解できなくてもそれを報告してください。オペレーティングシステムのエラーでプログラムが強制終了してしまったら、どのエラーでそうなったのかを報告してください。何も起こらない場合も、その旨を報告してください。たとえテストケースの結果がプログラムのクラッシュなど明確な場合でも、我々のプラットフォームで再現できない場合があります。最も容易な方法は、出力をターミナルからコピーすることです。

  > **注記**
  >
  > エラーメッセージを報告する場合、そのメッセージを最大限詳細に取得してください。psqlでは、前もって\set VERBOSITY verboseを指定してください。サーバログからメッセージを取り出す場合

は、全ての詳細をログに取得できるようにlog_error_verbosity実行時パラメータをverboseに設定してください。

注記

致命的なエラーが起こった場合、クライアント側で報告されるエラーメッセージには得られる情報が全て書かれているとは限りません。データベースサーバのログも見てみてください。もしログを取っていないならば、取る習慣を付けるいいタイミングです。

- どのような出力を望んでいたのかを記述することも非常に重要です。ただ単に「このコマンドはこのような出力を返した」や、「期待していた結果ではない」だけでは、再現して結果を検証した際、開発者は、これは期待した通りの正しい結果である、と考えるかもしれません。送られてきたコマンドの背後にある文脈を全て把握することはできません。また、特に「SQLではこう書かれていない/Oracleではこのようにならない」というコメントはご遠慮願います。SQLの正確な動作を探し出すのは楽しい作業ではありませんし、また、世にある他のリレーショナルデータベースの動作全てをPostgreSQLの開発者が把握しているわけでもありません（問題がプログラムのクラッシュである場合、この内容は言うまでもなく省略できます）。

- すべてのコマンドラインオプションと起動時のオプション、デフォルトから変更した関連する環境変数や設定ファイル。ここでも、正確な情報を提供してください。OSの起動時にデータベースサーバを起動するようにパッケージされたディストリビューションを使用している場合は、それらがどのように実行されているかを確認する必要があります。

- インストールの手順書から変更して実行したすべての内容。

- PostgreSQLのバージョン。`SELECT version();`で、接続しているサーバのバージョンがわかります。多くの実行可能なプログラムでは`--version`オプションも使用できます。少なくとも`postgres --version`と`psql --version`は実行できるはずです。これらの関数やオプションが使用できない場合、アップグレードが保証されているものよりも、さらに古いバージョンです。RPMなどパッケージ化されたものを使用している場合は、その旨を連絡し、パッケージに付加されたバージョン番号があれば、それも記載してください。Git版に対するバグ報告の場合は、その旨も記載し、コミットハッシュの情報も含めてください。

  13.1よりもバージョンが古い場合、アップグレードすることをお勧めします。新しいリリースでは多くのバグ修正や改良がなされているからです。ですので、古めのPostgreSQLのリリースを使用していて遭遇した不具合が修正されている可能性がかなりあります。古いPostgreSQLのリリースを使用しているサイトに対して、我々は限定されたサポートしか提供することができません。それ以上のサポートが必要であれば、商用サポート契約を結ぶことを検討してください。

- プラットフォーム情報。カーネル名とバージョン、Cライブラリ、プロセッサ、メモリ情報なども含めて報告してください。多くの場合、ベンダ名とそのバージョンを明記するだけで十分ですが、「Debian」の正確な構成要素を全ての人間が把握している、であるとか、全ての人間がx86_64を使用しているなどの思い込みは止めてください。インストールに関する問題の場合は、マシンのツール群（コンパイラやmakeなど）の情報も必要となります。

バグ報告が長文になってもそれは仕方がないことなので、気にしないでください。最初に全ての情報を入手できる方が、開発者が事実を聞き出さなければいけない状況よりも良いです。その一方、ファイルが大きい

ならば、その情報に誰か興味があるかを最初に尋ねるのが得策かもしれません。記事[9]には、バグ報告に関するその他のコツの概要があります。

問題を解決する入力を見つけ出すための試行錯誤に時間をかけないでください。これはおそらく問題解決の助けになりません。バグが即座に修正されない場合、その時間を利用すれば、あなた自身のワークアラウンドを探して共有することができます。繰り返しになりますが、バグがなぜあるのかを解明するのに余計な時間をかける必要はありません。開発者の方が十分速くそれを見つけ出します。

バグ報告をする際、理解しやすい用語を使用してください。このソフトウェアパッケージ全体は「PostgreSQL」と呼ばれていますが、略して「Postgres」とも呼ばれます。特にバックエンドプロセスに関して述べる時は、そのように明記し、「PostgreSQLがクラッシュする」とは記述しないでください。1つのバックエンドプロセスのクラッシュと、その親プロセス「postgres」のクラッシュとはかなり異なります。1つのバックエンドがダウンしてしまったことを、「サーバがクラッシュした」とは記述しないでください。その逆の場合にも当てはまります。また、「psql」対話式フロントエンドなどのクライアントプログラムはバックエンドとは完全に分離されています。問題がクライアント側かサーバ側かの切り分けを試みてください。

5.3. バグ報告先

一般論として、バグ報告は<pgsql-bugs@lists.postgresql.org>というバグ報告用メーリングリストに送ってください。バグ報告の題名には、エラーメッセージの一部分などわかりやすいものを使ってください。

その他の方法として、プロジェクトのWebサイト[10]にあるバグ報告フォームの項目を埋める方法があります。この方法で入力したバグ報告は、<pgsql-bugs@lists.postgresql.org>メーリングリストに送信されます。

バグ報告にセキュリティが関連する場合や公開アーカイブからすぐに閲覧できることを好まない場合、pgsql-bugsには送信しないでください。セキュリティの問題については、非公開で<security@postgresql.org>に報告することができます。

<pgsql-sql@lists.postgresql.org>や<pgsql-general@lists.postgresql.org>などのユーザ向けのメーリングリストには決してバグ報告を送らないでください。これらのメーリングリストはユーザからの質問に答えるためのもので、ほとんどの購読者はバグ報告を受け取りたくないと思われます。さらに重要なのは、これらのリストの購読者によってバグが修正されることはほとんどないということです。

また、開発者向けの<pgsql-hackers@lists.postgresql.org>にもバグ報告を送らないでください。ここはPostgreSQLの開発に関して議論する場で、バグ報告とは切り離している方が良いとされています。もしその問題により多くのレビューが必要な場合は、そのバグ報告をpgsql-hackersで議論することになります。

ドキュメントに関して問題がある場合は、ドキュメント用のメーリングリスト、<pgsql-docs@lists.postgresql.org>が最適な報告先です。その際、問題になった箇所がどの部分かを明記してください。

また、サポートされていないプラットフォームへの移植に関係するバグ報告である場合は<pgsql-hackers@lists.postgresql.org>に報告してください。そのプラットフォームへPostgreSQLを移植するように（報告者と一緒に）最善の努力をします。

[9] https://www.chiark.greenend.org.uk/~sgtatham/bugs.html

[10] https://www.postgresql.org/

注記

嘆かわしい量のスパムメールが出回っているため、これらのメーリングリストは購読しない限りモデレータ付きとなっています。これはEメールが配送されるのにいくらか遅延があることを意味します。メーリングリストを購読したい場合には、https://lists.postgresql.org/を訪ねて、その指示に従ってください。

パート I. チュートリアル

PostgreSQLのチュートリアルにようこそ。本書は数章から構成され、ここで、PostgreSQLやリレーショナルデータベースの概念、SQL言語を初めて使用する方向けに、簡単に紹介します。ここでは、コンピュータの一般的な使い方についての知識だけを前提としています。Unixやプログラミングに関する経験は必要ありません。ここでは、主にPostgreSQLシステムの重要なポイントについて実践的な経験を得ることを目的としています。扱っているトピックについての完全で詳細な処理を記述しているものではありません。

このチュートリアルの内容を読んだ後、SQL言語のより体系的な知識を学習したいのであれば、パート IIを、PostgreSQL用のアプリケーションの開発に関する情報を学習したいのであれば、パート IVを続いて読んでください。また、サーバをセットアップおよび管理される方は、パート IIIも参照してください。

目次

第1章 さあ始めましょう

1.1. インストール

PostgreSQLを使用できるようにするためには、当然ながらインストールしなければなりません。使用しているオペレーティングシステム配布物内に含まれていたり、システム管理者がインストールしていたりしますので、PostgreSQLが既にサイトにインストールされている可能性があります。そのような場合、オペレーティングシステムの文書やシステム管理者からPostgreSQLへのアクセス方法に関する情報を得なければなりません。

PostgreSQLが既に使用可能かどうかや実験的に使用できるかどうかがわからなければ、独自にインストールすることができます。インストールは難しくありません。良い経験になるでしょう。PostgreSQLは、非特権ユーザによって、つまり、スーパーユーザ（root）権限を必要とすることなく、インストールできます。

自身でPostgreSQLをインストールする場合は、インストール手順を第16章で確認してください。そしてインストールが完了してから、本書に戻ってきてください。適切な環境変数の設定に関する節に正確に従っていることを確認してください。

システム管理者がデフォルトの方法で設定していなかった場合、他にも多少の作業をすることになります。例えば、データベースサーバマシンがリモートマシンの場合、PGHOST環境変数をデータベースサーバマシンの名前に設定する必要があります。また、PGPORT環境変数の設定も行わなければならないかもしれません。要するに、アプリケーションプログラムを起動しようとして、データベースに接続できないというエラーが発生する場合には、サイト管理者と相談し、自分が管理者であれば、文書を読んで、環境が適切に設定されていることを確認してください。これまでの内容を理解できない場合は、次の節を読んでください。

1.2. 構造的な基本事項

先に進む前に、PostgreSQLシステム構成の基礎を理解すべきです。PostgreSQLの各部分がどのように相互作用しているかを理解することにより、本章の内容がわかりやすくなります。

データベースの用語で言うと、PostgreSQLはクライアント/サーバモデルを使用しています。PostgreSQLのセッションは以下の協調動作するプロセス（プログラム）から構成されます。

- サーバプロセス。これは、データベースファイルを管理し、クライアントアプリケーションからのデータベースの接続を受け付け、クライアントに代わってデータベースに対する処理を行います。データベースサーバプログラムはpostgresと呼ばれています。
- ユーザの、データベース操作を行うクライアント（フロントエンド）アプリケーション。クライアントアプリケーションはその性質上非常に多様性があります。テキスト指向のツール、グラフィカルなアプリケーション、データベースにアクセスしWebページを表示するWebサーバ、あるいはデータベースに特化した保守ツールなどがあります。いくつかのクライアントアプリケーションがPostgreSQLの配布物に同梱されていますが、ほとんどのクライアントアプリケーションはユーザによって開発されます。

クライアント/サーバアプリケーションでは典型的なことですが、クライアントとサーバはホストが異なっても構いません。その場合、クライアントとサーバはTCP/IPネットワーク接続経由で通信を行います。このことに

は注意してください。なぜなら、クライアントマシンからアクセスできるファイルは、データベースサーバマシンではアクセスできない（または、異なるファイル名でアクセスできるだけである）可能性があるからです。

PostgreSQLサーバはクライアントから複数の同時接続を取り扱うことができます。これを達成するため、サーバは接続ごとに新しいプロセスを開始（「fork」）します。その時点から、クライアントと新しいサーバプロセスは元のpostgresプロセスによる干渉がない状態で通信を行います。こうして、マスターサーバプロセスは常に稼働してクライアントからの接続を待ち続け、一方で、クライアントからの接続と関連するサーバプロセスの起動が行われます。（もちろんこれは全てユーザからはわかりません。完全性を目的として説明しているだけのことです。）

1.3. データベースの作成

データベースサーバにアクセスできるかどうかがわかる最初の試験は、データベースの作成を試みることです。稼働中のPostgreSQLサーバは多くのデータベースを管理することができます。典型的には、プロジェクトやユーザごとに別々のデータベースが使用されます。

サイト管理者により使用できるデータベースが既に作成されている場合もあります。この場合、この段階を飛ばして、次の節まで進んでください。

新しいデータベースを作成するには、以下のコマンドを使用してください。この例ではmydbという名前です。

```
$ createdb mydb
```

この手順で何も応答がなければ、この段階は成功し、本節の残りは飛ばすことができます。

以下のようなメッセージが現れた場合、PostgreSQLが正しくインストールされていません。

```
createdb: command not found
```

PostgreSQLがインストールされていないか、シェルの検索パスに含まれていないのいずれかです。代わりに絶対パスでそのコマンドを実行してみてください。

```
$ /usr/local/pgsql/bin/createdb mydb
```

このパスはサイトによって異なるかもしれません。この問題を解決するには、サイト管理者に連絡するか、インストール取扱説明書を調べてください。

他の応答として以下もあります。

```
createdb: could not connect to database postgres: could not connect to server: No such file or
 directory
        Is the server running locally and accepting
        connections on Unix domain socket "/tmp/.s.PGSQL.5432"?
```

これは、サーバが起動していないか、createdbが想定している状態でサーバが起動していないかを示しています。こちらも、インストール手順を点検するか管理者に相談してください。

他の応答として以下もあります。

```
createdb: could not connect to database postgres: FATAL:  role "joe" does not exist
```

ここでjoeのところには、ログインした時のユーザ名が記載されています。これは、管理者がそのユーザ用のPostgreSQLユーザアカウントを作成していない時に起こります（PostgreSQLユーザアカウントは、オペレーティングシステムのユーザアカウントとは別です）。自身が管理者なら、アカウントの作成方法に関して第21章を参照してください。最初のユーザアカウントを作成するためには、PostgreSQLをインストールした時のオペレーティングシステムのユーザ（通常postgres）になる必要があります。PostgreSQLユーザアカウントがオペレーティングシステムのユーザ名と異なる名前で用意されているかもしれません。その場合は、PostgreSQLのユーザ名を指定するために、-Uスイッチを使用するか、PGUSER環境変数を設定する必要があります。

ユーザアカウントを持っているが、データベースを作成するために必要な権限を持っていない場合は、以下のメッセージが現れます。

```
createdb: database creation failed: ERROR:  permission denied to create database
```

全てのユーザがデータベースを新規に作成できる権限を持っているわけではありません。PostgreSQLがデータベースの作成を拒否した場合、サイト管理者からデータベースを作成する権限を付与してもらう必要があります。これが発生した場合はサイト管理者に相談してください。自身でPostgreSQLをインストールしたのであれば、このチュートリアルを実行する時は、サーバを起動したユーザアカウントでログインしてください。[1]

他の名前のデータベースを作成することもできます。PostgreSQLは、与えられたサイトでいくつでもデータベースを作成することを許可しています。データベース名は、先頭がアルファベット文字から始まる、63バイトまでの長さでなければなりません。簡単な選択は、現在のユーザ名と同じ名前のデータベースを作成することです。多くのツールでは、データベース名のデフォルトとしてそれを仮定していますので、これにより入力数を減らすことができます。このデータベースを作成するには、単純に以下を実行します。

```
$ createdb
```

データベースを使用しなくなったら、削除することができます。例えば、mydbデータベースの所有者（作成者）であれば、以下のコマンドでそれを廃棄することができます。

```
$ dropdb mydb
```

（このコマンドでは、データベース名のデフォルトはユーザアカウント名ではありません。常に指定しなければなりません。）この動作は、そのデータベースに関する全てのファイルを物理的に削除しますので、取り消すことはできません。事前に熟考した場合にのみこれを実施してください。

createdbやdropdbの詳細は、それぞれcreatedbとdropdbにあります。

[1] 何故これで上手くいくのかについての説明：PostgreSQLのユーザ名はオペレーティングシステムのユーザアカウントとは分離されています。データベースに接続するとき、接続に利用するPostgreSQLのユーザ名を選択します。選択しなければ、デフォルトで現在のオペレーティングシステムのアカウントと同じ名前となります。これにより、サーバを開始したオペレーティングシステムのユーザと同じ名前のPostgreSQLユーザアカウントが、常に存在するようになっています。そしてまた、このユーザは常にデータベースを作成する権限を持ちます。そのユーザとしてログインする代わりに、-Uオプションを毎回使用して、接続するPostgreSQLのユーザ名を選択することができます。

1.4. データベースへのアクセス

データベースを作成した後、以下によってアクセスできます。

- *psql*というPostgreSQL対話式端末プログラムを実行。これにより、対話式にSQLコマンドの入力、編集、実行を行うことができます。
- pgAdminのような既存のグラフィカルなフロントエンドツールや、ODBCあるいはJDBCを備えたオフィススイートなどを使用して、データベースの作成や操作を行う。これらについてはこのチュートリアルでは取り上げません。
- 複数の使用可能言語の1つを使用した、独自のアプリケーションの作成。これについては、パート IVで詳しく説明します。

このチュートリアルの例を試すには、psqlから始めることを勧めます。以下のコマンドを入力することで、mydbデータベースに対して実行することができます。

```
$ psql mydb
```

データベース名を与えなかった場合、データベース名はデフォルトでユーザアカウント名となります。この仕組みについては前節でcreatedbを使って既に説明しています。

psqlでは、始めに以下のメッセージが表示されます。

```
psql (13.1)
Type "help" for help.

mydb=>
```

最後の行は以下のようになっているかもしれません。

```
mydb=#
```

これは、データベーススーパーユーザであることを示します。自身でPostgreSQLのインスタンスをインストールした場合にはこのようになっている可能性が高いです。スーパーユーザであることは、アクセス制御の支配を受けないことを意味します。このチュートリアルの実施においては、これは重要ではありません。

psqlの起動に問題が発生した場合は、前節に戻ってください。createdbの診断とpsqlの診断方法は似ており、前者が動作すれば後者も同様に動作するはずです。

psqlが最後に出力する行はプロンプトで、psqlが入力を監視していること、psqlが管理する作業領域にSQL問い合わせを入力できることを示しています。以下のコマンドを試してください。

```
mydb=> SELECT version();
                                          version
-------------------------------------------------------------------------------------------
 PostgreSQL 13.1 on x86_64-pc-linux-gnu, compiled by gcc (Debian 4.9.2-10) 4.9.2, 64-bit
(1 row)
```

```
mydb=> SELECT current_date;
    date
------------
 2016-01-07
(1 row)

mydb=> SELECT 2 + 2;
 ?column?
----------
        4
(1 row)
```

psqlプログラムは、SQLコマンドではない、多くの内部コマンドを持っています。それらはバックスラッシュ文字「\」から始まります。例えば、各種PostgreSQL SQLコマンドの構文に関するヘルプを以下のようにして得ることができます。

```
mydb=> \h
```

psqlを終了するには、以下を入力します。

```
mydb=> \q
```

psqlは終了し、コマンドシェルに戻ります（他の内部コマンドについてはpsqlのプロンプトで\?を入力してください）。psqlの完全な能力についてはpsqlで説明されています。このチュートリアルではこれらの機能は明示的に使用しませんが、便利な場合これらを使用しても構いません。

第2章 SQL言語

2.1. 序文

本章では、SQLを使用した簡単な操作方法について、その概要を説明します。このチュートリアルは単なる入門用であり、SQLについての完全な教科書ではありません。[melt93]や[date97]など、SQLを説明した書籍は多くあります。PostgreSQLの言語機能の中には標準を拡張したものがあることには注意してください。

以下で示す例では、前章で説明したmydbという名前のデータベースを作成し、psqlを起動できるようになっていることを前提としています。

このマニュアルで示す例は、PostgreSQLソース配布物に含まれており、src/tutorial/以下に展開されます。(PostgreSQLのバイナリ配布物ではこのファイルは含まれていないかも知れません。) このファイルを使用するためには、以下に示すように、まずこのディレクトリに移動し、makeを実行してください。

```
$ cd ..../src/tutorial
$ make
```

これによりスクリプトが作成され、そして、ユーザ定義の関数と型を含むCのファイルがコンパイルされます その後、以下を行うことで、チュートリアルを始めることができます。

```
$ cd ..../tutorial
$ psql -s mydb

...


mydb=> \i basics.sql
```

\iは、指定したファイルからコマンドを読み込みます。psqlの-sオプションによって、シングルステップモードとなり、それぞれの文をサーバに送る前に一時停止します。本節で使用するコマンドはbasics.sourceファイル内にあります。

2.2. 概念

PostgreSQLは*リレーショナルデータベース管理システム*(RDBMS)です。これは*リレーション*の中に格納されたデータを管理するシステムであることを意味しています。リレーションは基本的には*テーブル*を表す数学用語です。テーブルにデータを格納することは今日では平凡なことですので、わかりきったものだと思われるかもしれませんが、データベースを構成する方法には他にも多くの方式があります。Unix互換のオペレーティングシステムのファイルとディレクトリは、階層型データベースの一種と言えます。より近代的な成果はオブジェクト指向データベースです。

各テーブルは、*行*の集合に名前を付けたものです。あるテーブルの各行は、名前を付けた*列*の集合ということができます。各列は特定のデータ型を持ちます。列は行において固定の順番を持ちますが、SQLはテー

ブルにある行の順番をまったく保証しないことを覚えておくことは重要です（しかし、表示用に明示的にソートさせることは可能です）。

テーブルはデータベースとしてまとめられ、1つのPostgreSQLサーバインスタンスで管理されるデータベースの集合はデータベース*クラスタ*を構成します。

2.3. 新しいテーブルの作成

テーブル名と、テーブルの全ての列の名前と型を指定することで、新しいテーブルを作成することができます。

```
CREATE TABLE weather (
    city            varchar(80),

    temp_lo         int,           -- 最低気温
    temp_hi         int,           -- 最高気温
    prcp            real,          -- 降水量
    date            date
);
```

上のコマンドを複数の行に分けてpsqlに入力することができます。psqlは、セミコロンで終わるまでそのコマンドは継続するものと認識します。

SQLコマンドでは空白文字（つまり空白、タブ、改行）を自由に使用できます。つまり、上で示したコマンドとは異なる整列で入力しても良いことを意味します。全てを1行で入力することさえできます。連続した2つのハイフン（「--」）はコメントの始まりです。その後に入力したものは、行末まで無視されます。SQLはキーワードと識別子に対して大文字小文字を区別しません。ただし、（上では行っていませんが）識別子が二重引用符で括られていた場合は大文字小文字を区別します。

varchar(80)は、80文字までの任意の文字列を格納できるデータ型を指定しています。intは一般的な整数用の型です。realは単精度浮動小数点数値を格納する型です。dateはその名前からわかるとおり日付です。（わかると思いますが、date型の列の名前もdateになっています。これはわかりやすいかもしれませんし、逆に混乱を招くかもしれません。これは好みによります）。

PostgreSQLは標準SQLのデータ型、int、smallint、real、double precision、char(N)、varchar(N)、date、time、timestampやintervalをサポートします。また、一般的なユーティリティ用の型や高度な幾何データ型もサポートします。任意の数のユーザ定義のデータ型を使用して、PostgreSQLをカスタマイズすることができます。したがって、標準SQLにおける特殊な場合をサポートするために必要な場所を除き、型名は構文内でキーワードではありません。

以下に示す2番目の例では、都市とその地理的な位置情報を格納します。

```
CREATE TABLE cities (
    name            varchar(80),
    location        point
);
```

point型は、PostgreSQL独自のデータ型の一例です。

最後に、テーブルが不要になった場合や別のものに作り直したい場合、以下のコマンドを使用して削除できることを示します。

```
DROP TABLE tablename;
```

2.4. テーブルに行を挿入

以下のように、INSERT文を使用して、テーブルに行を挿入します。

```
INSERT INTO weather VALUES ('San Francisco', 46, 50, 0.25, '1994-11-27');
```

全てのデータ型でどちらかといえばわかりやすい入力書式を使用していることに注意してください。通常、単純な数値以外の定数は、この例のように、単一引用符(')で括らなければなりません。date型で受け付けられるものは実際はかなり柔軟です。しかし、このチュートリアルの段階では、曖昧さがない書式にこだわることにします。

point型では、入力として次のような座標の組み合わせが必要です。

```
INSERT INTO cities VALUES ('San Francisco', '(-194.0, 53.0)');
```

ここまでの構文では、列の順番を覚えておく必要がありました。以下に示す他の構文では、列のリストを明示的に与えることができます。

```
INSERT INTO weather (city, temp_lo, temp_hi, prcp, date)
    VALUES ('San Francisco', 43, 57, 0.0, '1994-11-29');
```

リスト内の列は好きな順番で指定できます。また、一部の列を省略することもできます。例えば、降水量がわからない場合は以下のようにすることができます。

```
INSERT INTO weather (date, city, temp_hi, temp_lo)
    VALUES ('1994-11-29', 'Hayward', 54, 37);
```

多くの開発者は、暗黙的な順番に依存するよりも、列のリストを明示的に指定する方が良いやり方だと考えています。

次節でもデータを使用しますので、上のコマンドを全て入力してください。

また、COPYを使用して大量のデータを平文テキストファイルからロードすることもできます。COPYコマンドはINSERTよりも柔軟性はありませんが、この目的に特化していますので、通常、より高速になります。以下に例を示します。

```
COPY weather FROM '/home/user/weather.txt';
```

ここで元となるファイルを表すファイル名は、クライアントではなく、バックエンドプロセスを動かしているマシンで利用できるものでなければなりません。バックエンドプロセスがこのファイルを直接読み込むからです。COPYにはCOPYコマンドについてのより詳しい説明があります。

2.5. テーブルへの問い合わせ

テーブルからデータを取り出すために、テーブルへ*問い合わせ*をします。このためにSQLのSELECT文が使用されます。この文は選択リスト（返される列のリスト部分）とテーブルリスト（データを取り出すテーブルのリスト部分）、および、省略可能な条件（制限を指定する部分）に分けることができます。例えば、weatherの全ての行を取り出すには、以下を入力します。

```
SELECT * FROM weather;
```

ここで*は「全ての列」の省略形です。[1] したがって、以下のようにしても同じ結果になります。

```
SELECT city, temp_lo, temp_hi, prcp, date FROM weather;
```

出力は、以下のようになります。

```
city      | temp_lo | temp_hi | prcp |    date
--------------+---------+---------+------+------------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
 San Francisco |      43 |      57 |    0 | 1994-11-29
 Hayward       |      37 |      54 |      | 1994-11-29
(3 rows)
```

選択リストには、単なる列参照だけではなく式を指定することもできます。例えば、以下を行うことができます。

```
SELECT city, (temp_hi+temp_lo)/2 AS temp_avg, date FROM weather;
```

この結果は次のようになります。

```
city      | temp_avg |    date
--------------+----------+------------
 San Francisco |       48 | 1994-11-27
 San Francisco |       50 | 1994-11-29
 Hayward       |       45 | 1994-11-29
(3 rows)
```

AS句を使用した出力列の再ラベル付けの部分に注意してください（AS句は省略することができます）。

必要な行が何かを指定するWHERE句を追加して問い合わせに「条件付け」することができます。WHERE句は論理（真値）式を持ち、この論理式が真となる行のみを返します。よく使われる論理演算子（AND、OR、NOT）を条件付けに使用することができます。例えば以下は、San Franciscoの雨天時の気象データを取り出します。

```
SELECT * FROM weather
```

[1] SELECT *は即興的な問い合わせで有用ですが、テーブルに列を追加することにより結果が変わってしまいますので、実用システムのコードでは悪いやり方であると一般的には考えられています。

```
    WHERE city = 'San Francisco' AND prcp > 0.0;
```

結果は次のようになります。

```
city      | temp_lo | temp_hi | prcp |    date
--------------+---------+---------+------+------------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
(1 row)
```

問い合わせの結果をソートして返すように指定することができます。

```
SELECT * FROM weather
    ORDER BY city;
```

```
city      | temp_lo | temp_hi | prcp |    date
--------------+---------+---------+------+------------
 Hayward       |      37 |      54 |      | 1994-11-29
 San Francisco |      43 |      57 |    0 | 1994-11-29
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
```

この例では、ソート順は十分に指定されていません。ですので、San Franciscoの行は順序が異なるかも知れません。しかし、次のようにすれば常に上記の結果になります。

```
SELECT * FROM weather
    ORDER BY city, temp_lo;
```

問い合わせの結果から重複行を除くように指定することができます。

```
SELECT DISTINCT city
    FROM weather;
```

```
city
--------------
 Hayward
 San Francisco
(2 rows)
```

ここでも、結果行の順序は変動するかもしれません。DISTINCTとORDER BYを一緒に使用することで確実に一貫した結果を得ることができます。[2]

```
SELECT DISTINCT city
```

[2] PostgreSQLの古めのバージョンを含む一部のデータベースシステムでは、DISTINCTの実装に行の自動順序付けが含まれており、ORDER BYは不要です。しかし、これは標準SQLにおける要求ではなく、現在のPostgreSQLではDISTINCT句が行の順序付けを行うことを保証していません。

```
    FROM weather
    ORDER BY city;
```

2.6. テーブル間を結合

ここまでの問い合わせは、一度に1つのテーブルにのみアクセスするものでした。問い合わせは、一度に複数のテーブルにアクセスすることも、テーブル内の複数の行の処理を同時に行うようなやり方で、1つのテーブルにアクセスすることも可能です。一度に同一のテーブルまたは異なるテーブルの複数の行にアクセスする問い合わせは、結合問い合わせと呼ばれます。例として、すべての気象データを関連する都市の位置情報と一緒に表示したい場合が挙げられます。それを行うためには、weatherテーブルの各行のcity列を、citiesテーブルの全ての行のname列と比較し、両者の値が一致する行の組み合わせを選択しなければなりません。

> **注記**
>
> これは概念的なモデルでしかありません。実際の結合は通常、1つひとつの行の組み合わせを比べるよりも、もっと効率的な方法で行われます。しかし、これはユーザからはわかりません。

これは、以下の問い合わせによって行うことができます。

```
SELECT *
    FROM weather, cities
    WHERE city = name;
```

```
city      | temp_lo | temp_hi | prcp |    date    |     name      | location
--------------+---------+---------+------+------------+---------------+-----------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27 | San Francisco | (-194,53)
 San Francisco |      43 |      57 |    0 | 1994-11-29 | San Francisco | (-194,53)
(2 rows)
```

この結果について2つのことに注目してください。

- Hayward市についての結果行がありません。これはcitiesテーブルにはHaywardに一致する項目がないからで、結合の際にweatherテーブル内の一致されなかった行は無視されるのです。これをどうしたら解決できるかは、しばらく後で説明します。
- 都市名を持つ2つの列があります。weatherテーブルとcitiesテーブルからの列のリストが連結されるため、これは正しい動作です。しかし実際には、これは望ましい結果ではないため、*を使わずに、明示的に出力列のリストを指定することになるでしょう。

```
SELECT city, temp_lo, temp_hi, prcp, date, location
    FROM weather, cities
    WHERE city = name;
```

練習: WHERE句を省略した場合のこの問い合わせの意味を決定してください。

列がすべて異なる名前だったので、パーサは自動的にどのテーブルの列かを見つけることができました。2つのテーブルで列名が重複している場合は、以下のようにどちらの列を表示させたいかを示すために列名を修飾しなければなりません。

```
SELECT weather.city, weather.temp_lo, weather.temp_hi,
       weather.prcp, weather.date, cities.location
    FROM weather, cities
    WHERE cities.name = weather.city;
```

結合問い合わせではすべての列名を修飾するのが良いやり方であると一般に考えられています。そうすれば、テーブルのいずれかに後で重複する名前を持つ列が追加されても、問い合わせが失敗しません。

ここまでに示したような結合問い合わせは、以下のように別の形で表すことができます。

```
SELECT *
    FROM weather INNER JOIN cities ON (weather.city = cities.name);
```

この構文は先の例よりも一般的に使用されるものではありませんが、以降の話題の理解を助けるためにここで示しています。

ここで、どのようにすればHaywardのレコードを得ることができるようになるのかを明らかにします。実行したい問い合わせは、weatherをスキャンし、各行に対して、cities行に一致する行を探すというものです。一致する行がなかった場合、citiesテーブルの列の部分を何らかの「空の値」に置き換えたいのです。この種の問い合わせは外部結合と呼ばれます（これまで示してきた結合は内部結合です）。以下のようなコマンドになります。

```
SELECT *
    FROM weather LEFT OUTER JOIN cities ON (weather.city = cities.name);

     city      | temp_lo | temp_hi | prcp |    date    |     name      | location
---------------+---------+---------+------+------------+---------------+-----------
 Hayward       |      37 |      54 |      | 1994-11-29 |               |
 San Francisco |      46 |      50 | 0.25 | 1994-11-27 | San Francisco | (-194,53)
 San Francisco |      43 |      57 |    0 | 1994-11-29 | San Francisco | (-194,53)
(3 rows)
```

この問い合わせは左外部結合と呼ばれます。結合演算子の左側に指定したテーブルの各行が最低でも一度出力され、一方で、右側のテーブルでは左側のテーブルの行に一致するもののみが出力されるからです。右側のテーブルに一致するものがない、左側のテーブルの行を出力する時、右側のテーブルの列は空の値（NULL）で置換されます。

練習: 右外部結合や完全外部結合も存在します。これらが何を行うかを考えてください。

テーブルを自分自身に対して結合させることができます。これは自己結合と呼ばれます。例として、他の気象データの気温範囲内にある気象データを全て取り出すことを考えます。weather各行のtemp_loとtemp_hiを、他のweather行のtemp_loとtemp_hi列とを比較しなければなりません。以下の問い合わせを使用して行うことができます。

```
SELECT W1.city, W1.temp_lo AS low, W1.temp_hi AS high,
    W2.city, W2.temp_lo AS low, W2.temp_hi AS high
    FROM weather W1, weather W2
    WHERE W1.temp_lo < W2.temp_lo
    AND W1.temp_hi > W2.temp_hi;

     city      | low | high |     city      | low | high
---------------+-----+------+---------------+-----+------
 San Francisco |  43 |   57 | San Francisco |  46 |   50
 Hayward       |  37 |   54 | San Francisco |  46 |   50
(2 rows)
```

ここで、結合の左側と右側を区別することができるように、weatherテーブルにW1とW2というラベルを付けています。また、入力量を省くために、他の問い合わせでもこの種の別名を使用することができます。以下に例を示します。

```
SELECT *
    FROM weather w, cities c
    WHERE w.city = c.name;
```

こういった形の省略はかなりよく行われます。

2.7. 集約関数

他のほとんどのリレーショナルデータベース製品同様、PostgreSQLは集約関数をサポートします。集約関数は複数の入力行から1つの結果を計算します。例えば、行の集合に対して、count(総数)、sum(総和)、avg(平均)、max(最大)、min(最小)といった演算を行う集約があります。

例として、次のように全ての都市の最低気温から最も高い気温を求めることができます。

```
SELECT max(temp_lo) FROM weather;
```

```
max
-----
  46
(1 row)
```

どの都市のデータなのかを知りたいとしたら、下記のような問い合わせを試行するかもしれません。

```
SELECT city FROM weather WHERE temp_lo = max(temp_lo);     間違い
```

しかし、max集約をWHERE句で使用することができませんので、このコマンドは動作しません（WHERE句はどの行を集約処理に渡すのかを決定するものであり、したがって、集約関数の演算を行う前に評価されなければ

ならないことは明らかです。このためにこの制限があります)。しかし、よくあることですが、問い合わせを書き直すことで、意図した結果が得られます。これには以下のような副問い合わせを使用します。

```
SELECT city FROM weather
    WHERE temp_lo = (SELECT max(temp_lo) FROM weather);
```

```
city
---------------
 San Francisco
(1 row)
```

副問い合わせは、外側の問い合わせで起こることとは別々に集約を計算する独立した演算ですので、この問い合わせは問題ありません。

また、GROUP BY句と組み合わせた集約も非常に役に立ちます。例えば、以下のコマンドで都市ごとに最低気温の最大値を求めることができます。

```
SELECT city, max(temp_lo)
    FROM weather
    GROUP BY city;
```

```
city      | max
---------------+-----
 Hayward       |  37
 San Francisco |  46
(2 rows)
```

ここには都市ごとに1行の出力があります。それぞれの集約結果はその都市に一致するテーブル行全体に対する演算結果です。以下のように、HAVINGを使用すると、グループ化された行にフィルタをかけることができます。

```
SELECT city, max(temp_lo)
    FROM weather
    GROUP BY city
    HAVING max(temp_lo) < 40;
```

```
city   | max
---------+-----
 Hayward |  37
(1 row)
```

このコマンドは上と同じ計算を行うものですが、全てのtemp_loの値が40未満の都市のみを出力します。最後になりますが、「S」から始まる名前の都市のみを対象にしたい場合は、以下を行います。

```
SELECT city, max(temp_lo)
```

```
    FROM weather
    WHERE city LIKE 'S%'            -- 1
    GROUP BY city
    HAVING max(temp_lo) < 40;
```

1 LIKE演算子はパターンマッチを行います。これについては9.7で説明します。

集約とSQLのWHEREとHAVING句の間の相互作用を理解することが重要です。WHEREとHAVINGの基本的な違いは、WHEREが、グループや集約を演算する前に入力行を選択する(したがって、これはどの行を使用して集約演算を行うかを制御します)のに対し、HAVINGは、グループと集約を演算した後に、グループ化された行を選択する、ということです。したがって、WHERE句は集約関数を持つことはできません。集約を使用して、どの行をその集約の入力にするのかを決定することは意味をなしません。一方で、HAVING句は常に集約関数を持ちます (厳密に言うと、集約を使用しないHAVING句を書くことはできますが、これは有用となることはほぼありません。同じ条件はWHEREの段階でもっと効率良く使用できます)。

前の例ではWHERE内に都市名の制限を適用することができます。集約を行う必要がないからです。これはHAVINGに制限を追加するよりも効率的です。なぜならWHEREの検査で失敗する全ての行についてグループ化や集約演算が行われないからです。

2.8. 更新

UPDATEコマンドを使用して、既存の行を更新することができます。11月28日以降の全ての気温の読み取りが2度高くなっていることがわかったとします。その場合、以下のコマンドによって、データを修正することができます。

```
UPDATE weather
    SET temp_hi = temp_hi - 2,  temp_lo = temp_lo - 2
    WHERE date > '1994-11-28';
```

データの更新後の状態を確認します。

```
SELECT * FROM weather;

     city      | temp_lo | temp_hi | prcp |    date
---------------+---------+---------+------+------------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
 San Francisco |      41 |      55 |    0 | 1994-11-29
 Hayward       |      35 |      52 |      | 1994-11-29
(3 rows)
```

2.9. 削除

DELETEコマンドを使用してテーブルから行を削除することができます。Haywardの気象を対象としなくなったとします。その場合、以下のコマンドを使用して、テーブルから行を削除することができます。

```
DELETE FROM weather WHERE city = 'Hayward';
```

Haywardに関する気象データは全て削除されました。

```
SELECT * FROM weather;
```

```
city      | temp_lo | temp_hi | prcp |    date
--------------+---------+---------+------+------------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
 San Francisco |      41 |      55 |    0 | 1994-11-29
(2 rows)
```

以下の形式の文には注意しなければなりません。

```
DELETE FROM tablename;
```

条件がない場合、DELETEは指定したテーブルの全ての行を削除し、テーブルを空にします。システムは削除前に確認を求めるようなことは行いません！

第3章 高度な諸機能

3.1. はじめに

前章では、PostgreSQLでSQLを使用してデータを保存したりアクセスしたりする基本について説明しました。ここでは、管理を単純化しデータの喪失や破壊を防止するSQLのいくつかのより高度な機能について論議します。最後にPostgreSQLのいくつかの拡張に触れます。

本章では折々第2章にある例に変更や改善を加えますので、その章を読んでおくことは役立ちます。本章にあるいくつかの例は、tutorialディレクトリのadvanced.sqlに入っています。ここでは繰り返しませんが、このファイルにはロードして使ってみることができるサンプルデータもあります。（ファイルの使い方は2.1を参照してください。）

3.2. ビュー

2.6の問い合わせをもう一度参照してください。天候の記録と都市の所在場所を結合したリストを得ることが今作っているアプリケーションにとって特に重要なのですが、この結合リストを必要とする度に問い合わせを打ち込みたくはないとしましょう。この問い合わせに対してビューを作成し、通常のテーブルのように参照できる問い合わせに名前を付けることができます。

```
CREATE VIEW myview AS
    SELECT city, temp_lo, temp_hi, prcp, date, location
        FROM weather, cities
        WHERE city = name;

SELECT * FROM myview;
```

ビューを自由に利用することは、SQLデータベースの良い設計における重要な項目です。ビューはテーブル構造の詳細をカプセル化しますので、アプリケーションが発展するに従いテーブル構造が変わったとしても、一貫したインタフェースを保てます。

ビューは実テーブルが使用できるほとんどの場所で使えます。他のビューに対するビューの作成も珍しくはありません。

3.3. 外部キー

第2章のweatherテーブルとcitiesテーブルを思い出してください。次のような問題点を考えてみましょう。citiesテーブルに一致する項目がない行は絶対にweatherテーブルに挿入できなくしたいとします。これをデータの参照整合性の保全と呼びます。最も単純なデータベースシステムでこれを実装するとしたら、まずcitiesテーブルに一致する行が存在するかどうかを確認し、それからweatherテーブルに新規レコードを追加する、あるいは拒絶する、といったことになるでしょう。この手法には多くの問題があること、そしてとても不便であることから、PostgreSQLに代わって作業させることができます。

これらのテーブルの新しい宣言は以下のようになります。

```
CREATE TABLE cities (
        city     varchar(80) primary key,
        location point
);

CREATE TABLE weather (
        city      varchar(80) references cities(city),
        temp_lo   int,
        temp_hi   int,
        prcp      real,
        date      date
);
```

では無効なレコードを挿入してみましょう。

```
INSERT INTO weather VALUES ('Berkeley', 45, 53, 0.0, '1994-11-28');
```

```
ERROR:  insert or update on table "weather" violates foreign key constraint "weather_city_fkey"
DETAIL:  Key (city)=(Berkeley) is not present in table "cities".
```

外部キーの動作はアプリケーションごとに細かく調整できます。このチュートリアルではこの簡単な例より先には進みませんが、さらに情報がほしい方は第5章をご覧ください。外部キーを正しく使用するようにすると、間違いなくデータベースアプリケーションの質を向上させますので身に付くように励んでください。

3.4. トランザクション

トランザクションは全てのデータベースシステムで基礎となる概念です。トランザクションの基本的要点は複数の手順を単一の「全てかなしか」の操作にまとめ上げることです。手順の進行途中の状態は他の同時実行中のトランザクションからは見えません。トランザクションの完結の障害となる何らかのエラーが起こると、 それらの手順はどれもデータベースにまったく影響を与えません。

例を挙げましょう。ある銀行のデータベースでそこに多数の顧客の口座の残高と支店の総預金残高が記録されているとします。アリスの口座からボブの口座に$100.00の送金があったことを記録したいとします。ちょっと乱暴に単純化すると、このSQLは次のようになります。

```
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';
UPDATE branches SET balance = balance - 100.00
    WHERE name = (SELECT branch_name FROM accounts WHERE name = 'Alice');
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Bob';
UPDATE branches SET balance = balance + 100.00
    WHERE name = (SELECT branch_name FROM accounts WHERE name = 'Bob');
```

書かれているSQLコマンドの詳しいことについて、ここでは重要でありません。重要な点は、この単純な操作の目的を果たすため、複数の独立した更新手続きが関わっていることです。銀行職員としてはこれら全ての更新が行われるかもしくはまったく行われないのかいずれかの確証が必要です。$100.00がアリスの口座から引き落とされずにボブの口座に振り込まれるようなシステムの不備があってはなりません。一方、$100.00がボブに振り込まれないでアリスの口座から引き落とされたとしたら、アリスはこの銀行のお得意様ではなくなるでしょうね。操作の途中で一部不都合が発生した場合、結果に影響を与えるいかなる手続きも実行されないという確証が必要です。更新手続きを*トランザクション*にグループ化すると、その確証が得られます。あるトランザクションは他のトランザクションから見て完結するかまったく起こらなかったかという見方から*原子的*と呼ばれます。

もう一方で、いったんトランザクションが完結しデータベースシステムに承認された場合は、確実に恒久的に保存され、たとえ直後にクラッシュが起こったとしても記録は失われないという確証も必要です。例えばボブが自分の口座から現金を引き落として店舗から立ち去った直後にボブの口座からの引き落とし記録がシステムのクラッシュで消えてしまうことは受け入れられません。トランザクションが実装されているデータベースでは、あるトランザクションによる全ての更新がそのトランザクションを完結したと通知を行う前に永続的記録装置（すなわちディスク上）にログを書き込むことで保証しています。

トランザクション実装のデータベースの別の重要な特性は、原子的更新という概念に深く関係しています。複数のトランザクションが同時に動作している時、それぞれのトランザクションは別のトランザクションが行っている未完了の変更を見ることができてはなりません。例えば、あるトランザクションがすべての支店の残高を集計する作業を行なっているとき、アリスの口座がある支店からの引き落としを勘定に入れるけれども、ボブの口座がある支店への振り込みを勘定に入れないというのは受け入れられませんし、その逆も駄目です。つまり、データベース上での恒久的効果という意味のみならず、一連の操作の過程で可視性ということにおいてもトランザクションは「すべて」か「なし」かでなければなりません。作業中のトランザクションによる更新は、他のトランザクションからはトランザクションが完結するまで不可視です。そのトランザクションが完結したその時点で、トランザクションが行った更新の全てが見えるようになります。

PostgreSQLではトランザクションを構成するSQLコマンドをBEGINとCOMMITで囲んで設定します。従って、この銀行取り引きのトランザクションの実際は次のようになります。

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';

-- 等々
COMMIT;
```

トランザクション処理の途中でコミットを行わない（アリスの口座残高が足りなかったような場合）と判断した場合は、COMMITではなくROLLBACKを使用して行った全ての更新を破棄します。

PostgreSQLは実際全てのSQL文をトランザクション内で実行するようになっています。BEGINを発行しない場合、それぞれの文は暗黙的にBEGINが付いているとみなし、（成功すれば）COMMITで囲まれているものとします。BEGINとCOMMITで囲まれた文のグループは*トランザクションブロック*と呼ばれることもあります。

> **注記**
>
> いくつかのクライアントライブラリは自動的にBEGINとCOMMITコマンドを発行し、ユーザに尋ねることなくトランザクションブロックが有効になります。使用しているインタフェースのドキュメントで確認してください。

セーブポイントを使用することで、トランザクション内で文を、より粒度を細かく制御することが可能になります。セーブポイントは、トランザクションを構成するある部分を選択的に破棄する一方、破棄されない残りの部分をコミットします。SAVEPOINTコマンドでセーブポイントを定義した後、必要であればROLLBACK TOコマンドによりセーブポイントまでロールバックできます。定義されたセーブポイントとロールバックするポイントとの間の全てのトランザクションのデータベースの変更は破棄されますが、セーブポイント以前の変更は保持されます。

セーブポイントまでロールバックした後でもセーブポイントは定義されたままです。このため何度でもそこにロールバックできます。逆に再度特定のセーブポイントにロールバックする必要がないのであれば、それを解除しシステムリソースを多少とも解放することができます。あるセーブポイントを解除したりセーブポイントにロールバックすることにより、自動的にその後に定義されたすべてのセーブポイントが解除されることに注意してください。

これら全てはトランザクションブロック内で起こるので、他のデータベースセッションからは何も見えません。トランザクションブロックをコミットした場合、他のセッションからはコミットされた行為が1つの単位として見えるようになりますが、ロールバックの行為は決して可視になりません。

銀行のデータベースを思い出してください。アリスの口座から$100.00を引き出してボブの口座に振り込むとします。後になってボブではなくウィリーの口座に振り込むべきだったと気が付きました。この場合セーブポイントを次のように使います。

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';
SAVEPOINT my_savepoint;
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Bob';

-- おっと、忘れるところだった。ウィリーの口座を使わなければ。
ROLLBACK TO my_savepoint;
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Wally';
COMMIT;
```

この例はもちろん極端に単純化していますが、セーブポイントの使用を通じてトランザクションブロック内で多くの制御を行えることがわかります。さらには何らかのエラーでシステムがトランザクションブロックを中断状態にした場合、完全にロールバックして再び開始するのを別とすれば、ROLLBACK TOコマンドがトランザクションブロックの制御を取り戻す唯一の手段です。

3.5. ウィンドウ関数

ウィンドウ関数は現在の行に何らかとも関係するテーブル行の集合に渡って計算を行います。これは集約関数により行われる計算の形式と似たようなものです。とは言っても、非ウィンドウ集約呼び出しのように、ウィンドウ関数により行が単一出力行にグループ化されることはありません。その代わり、行はそれぞれ個別の身元を維持します。裏側では、ウィンドウ関数は問い合わせ結果による現在行だけでなく、それ以上の行にアクセスすることができます。

これはその部署の平均給与とそれぞれの従業員の給与をどのように比較するかを示した例です。

```
SELECT depname, empno, salary, avg(salary) OVER (PARTITION BY depname) FROM empsalary;
```

```
depname  | empno | salary |          avg
-----------+-------+--------+-----------------------
 develop   |    11 |   5200 | 5020.0000000000000000
 develop   |     7 |   4200 | 5020.0000000000000000
 develop   |     9 |   4500 | 5020.0000000000000000
 develop   |     8 |   6000 | 5020.0000000000000000
 develop   |    10 |   5200 | 5020.0000000000000000
 personnel |     5 |   3500 | 3700.0000000000000000
 personnel |     2 |   3900 | 3700.0000000000000000
 sales     |     3 |   4800 | 4866.6666666666666667
 sales     |     1 |   5000 | 4866.6666666666666667
 sales     |     4 |   4800 | 4866.6666666666666667
(10 rows)
```

最初の3つの出力列は、テーブルempsalaryから直接もたらされ、テーブル内のそれぞれの行に対し1つの出力行が存在します。4番目の列は、現行の行と同じdepnameの値を持つ全てのテーブル行に渡って取得した平均値を表わしています。（これは実際、非ウィンドウavg集約関数と同じですが、OVER句によりウィンドウ関数として扱われ、ウィンドウフレームに渡り計算されます。）

ウィンドウ関数呼び出しは常に、ウィンドウ関数名と引数の直後に続くOVER句を含みます。これが通常の関数、または非ウィンドウ集約関数と構文的に区別されるところです。OVER句は、ウィンドウ関数により処理のため問い合わせの行がどのように分解されるかを厳密に決定します。OVER内のPARTITION BY句は、行をPARTITION BY式の同じ値を共有するグループ、すなわちパーティションに分割します。それぞれの行に対し、ウィンドウ関数は現在行と同じパーティションに分類される行に渡って計算されます。

OVER内でORDER BYを使用することによりウィンドウ関数で処理される行の順序を制御することもできます。（ウィンドウのORDER BYは行が出力される順序に一致する必要すらありません。）ここに例をあげます。

```
SELECT depname, empno, salary,
       rank() OVER (PARTITION BY depname ORDER BY salary DESC)
FROM empsalary;
```

```
depname  | empno | salary | rank
-----------+-------+--------+------
 develop   |     8 |   6000 |    1
 develop   |    10 |   5200 |    2
 develop   |    11 |   5200 |    2
```

```
 develop   |     9 |   4500 |    4
 develop   |     7 |   4200 |    5
 personnel |     2 |   3900 |    1
 personnel |     5 |   3500 |    2
 sales     |     1 |   5000 |    1
 sales     |     4 |   4800 |    2
 sales     |     3 |   4800 |    2
(10 rows)
```

ここで示されたように、rank関数は、それぞれの別々のORDER BYの値に対する現在行のパーティション内における順位を、ORDER BY句で定義された順序を使って生成します。rankは明示的なパラメータを必要としません。この動作はOVER句により完全に決定されるためです。

ウィンドウ関数で考慮される行は、そのWHERE、GROUP BY、およびHAVING句でフィルターをかけられた問い合わせのFROM句によって生成された「仮想テーブル」の行です。例えば、WHERE条件に一致しないため削除された行はウィンドウ関数からは見えません。異なったOVER句を用いて、異なった方法によりデータを分割する複数のウィンドウ関数を問い合わせが含んでも構いません。しかし、この仮想テーブルで定義された行の同一の集まり上で全てが作動します。

ORDER BYは、行の順序付けが重要でない場合、省略可能であることを見てきました。PARTITION BYも同様に割愛することができます。この場合、全ての行を含む単一のパーティションが存在します。

ウィンドウ関数に関連した別の重要な概念があります。それぞれの行に対して、そのウィンドウフレームと呼ばれる、そのパーティション内の行の集合が存在します。ウィンドウ関数の中には、パーティション全体ではなく、ウィンドウフレームの行のみに対して作用するものもあります。デフォルトでは、ORDER BYが指定されると、フレームは、パーティションの始めから現在の行までのすべての行、およびそれより後にあるがORDER BY句に従うと現在の行とおなじ順序になるすべての行から構成されます。ORDER BYが省略された場合、デフォルトのフレームはそのパーティション内のすべての行を含みます。[1] sumを使用した例を示します。

```
SELECT salary, sum(salary) OVER () FROM empsalary;
```

```
salary |  sum
--------+-------
   5200 | 47100
   5000 | 47100
   3500 | 47100
   4800 | 47100
   3900 | 47100
   4200 | 47100
   4500 | 47100
   4800 | 47100
   6000 | 47100
   5200 | 47100
(10 rows)
```

[1] ほかの方法でウィンドウフレームを定義するいくつかのオプションがありますが、このチュートリアルでは扱いません。詳細は、4.2.8を参照してください。

上では、OVER句内にORDER BYが存在しませんので、ウィンドウフレームはパーティションと同一です。またパーティションはPARTITION BYもありませんのでテーブル全体となります。言い換えると、総和はそれぞれ、テーブル全体に対して行われ、その結果、各出力行で同じ結果を得ることになります。しかし以下のように、ORDER BY句を加えると、非常に異なる結果を得ます。

```
SELECT salary, sum(salary) OVER (ORDER BY salary) FROM empsalary;
```

```
salary |  sum
--------+-------
   3500 |  3500
   3900 |  7400
   4200 | 11600
   4500 | 16100
   4800 | 25700
   4800 | 25700
   5000 | 30700
   5200 | 41100
   5200 | 41100
   6000 | 47100
(10 rows)
```

ここで、sumは最初の（最も低い）salaryから現在の行まで、現在のものと重複する全てを含んで、計算されます（重複するsalaryに対する結果に注意してください）。

ウィンドウ関数は問い合わせのSELECTリストとORDER BY句に限って許可されます。GROUP BY、HAVING、およびWHERE句などその他の場所では禁止されています。その理由は、ウィンドウ関数は論理的に、ここに挙げたような句が処理された後に実行されるからです。またウィンドウ関数は非ウィンドウ集約関数の後に実行されます。これが意味する所は、ウィンドウ関数の引数に集約関数呼び出しを含めても有効ですが、その逆は成り立たないと言うことです。

ウィンドウ演算が行われた後、行にフィルタ処理を行ったりグループ化を行う必要が生じた場合、副問い合わせを使用します。例をあげます。

```
SELECT depname, empno, salary, enroll_date
FROM
  (SELECT depname, empno, salary, enroll_date,
          rank() OVER (PARTITION BY depname ORDER BY salary DESC, empno) AS pos
     FROM empsalary
  ) AS ss
WHERE pos < 3;
```

上記問い合わせは3より小さいrankを持った内部問い合わせからの行のみを表示します。

問い合わせが複数のウィンドウ関数を含む場合、各ウィンドウ関数に異なるOVER句を記述することができます。しかし複数の関数で同じウィンドウ処理動作が必要な場合は重複となり、またエラーを招きがちです。代わりにWINDOW句でウィンドウ処理動作に名前を付け、これをOVER内で参照することができます。以下に例を示します。

```
SELECT sum(salary) OVER w, avg(salary) OVER w
  FROM empsalary
  WINDOW w AS (PARTITION BY depname ORDER BY salary DESC);
```

ウィンドウ関数についてより詳細は、4.2.8、9.22、7.2.5、および SELECT マニュアルページにあります。

3.6. 継承

継承とはオブジェクト指向データベースの概念です。データベース設計においてこれまでになかった興味深い可能性を広げてくれます。

2つのテーブルcities（都市）テーブルとcapitals（州都）テーブルを作ってみましょう。州都は本来同時に都市でもありますので、全ての都市をリストする時は何もしなくても州都も表示する何らかの方法が必要です。賢い人なら次のような案を工夫するかもしれません。

```
CREATE TABLE capitals (
  name       text,
  population real,

  elevation  int,    --  （フィート単位）
  state      char(2)
);

CREATE TABLE non_capitals (
  name       text,
  population real,

  elevation  int     --  （フィート単位）
);

CREATE VIEW cities AS
  SELECT name, population, elevation FROM capitals
    UNION
  SELECT name, population, elevation FROM non_capitals;
```

問い合わせを続ける分には問題はありませんが、例えば、複数の行を更新する時に醜くなります。

より良い解決策は次のような構文です。

```
CREATE TABLE cities (
  name       text,
  population real,

  elevation  int     --  （フィート単位）
);
```

```
CREATE TABLE capitals (
  state      char(2) UNIQUE NOT NULL
) INHERITS (cities);
```

この例では、capitalsテーブルの行は親のcitiesテーブルから全ての列、すなわちname(都市名)、population(人口)そしてelevation(標高)を継承します。name列のデータ型は、可変長文字列のためにPostgreSQLが初めから備えているtext型です。capitalsテーブルは、これに加えて州の略称を示すstate列を持ちます。PostgreSQLでは、テーブルは関連付けられたテーブルがあればそれぞれから属性を継承することができます。

以下の問い合わせの例は、州都も含めて、標高500フィート以上に位置する全ての都市を求めるものです。

```
SELECT name, elevation
  FROM cities
  WHERE elevation > 500;
```

これは以下を返します。

```
name    | elevation
-----------+-----------
 Las Vegas |      2174
 Mariposa  |      1953
 Madison   |       845
(3 rows)
```

その一方、州都ではない標高500フィート以上に位置する都市を見つけ出したい時は次のような問い合わせになります。

```
SELECT name, elevation
    FROM ONLY cities
    WHERE elevation > 500;
```

```
name    | elevation
-----------+-----------
 Las Vegas |      2174
 Mariposa  |      1953
(2 rows)
```

ここでcitiesの前に置かれたONLYは、継承階層においてcitiesテーブルの下層にあるテーブルではなく、citiesテーブルのみを参照することを意味します。既に説明したSELECT、UPDATEおよびDELETEなど数多くのコマンドは、このONLY表記をサポートしています。

> **注記**
>
> 継承は多くの場合で便利ですが、一意性制約や外部キーと統合されていないので万能ではありません。詳細は5.10を参照してください。

3.7. まとめ

SQL初心者向けのこのチュートリアル入門では触れていない多くの機能が、PostgreSQLにはあります。これらの機能は、本書の残りで詳しく説明します。

もっと多くの入門資料がお望みであれば、PostgreSQLのWebサイト[2]により多くのリソースがリンクされています。

[2] https://www.postgresql.org

パート II. SQL言語

ここでは、PostgreSQLでSQL言語を使用する方法を説明します。まず最初にSQL構文全般について述べ、 データを保持する構造の作成方法、データベースに登録する方法、そして、データベースへの問い合わせを行う方法について説明していきます。本パートの中盤では、SQLコマンドで使用可能なデータ型と関数を紹介します。そして残りの部分では、最適な性能を実現するためにデータベースを調整する際に重要となるいくつかの点について説明します。

ここでの内容は、初心者のユーザでも先のページを何度も参照することなく、最初から最後まで全てのトピックを理解できるような構成になっています。各章ごとに内容が独立していますので、上級ユーザは必要な章だけを選んで読むことができます。ここではトピックに関する説明が中心となっていますので、特定のコマンドの完全な記述が必要なユーザはパート VIを参照してください。

対象読者は、PostgreSQLデータベースへの接続およびSQLコマンド発行の方法を知っているユーザです。まだこれらについて慣れていないユーザは、本書の前にパート Iをお読みになることをお勧めします。SQLコマンドは通常PostgreSQLの対話式端末psqlを使用して入力しますが、同様の機能を備えた他のプログラムも使用することができます。

目次

第4章 SQLの構文

本章ではSQLの構文について説明します。本章の内容は、データの定義や変更のためにSQLコマンドを適用する方法について詳しく説明する以後の章を理解する上での基礎となります。

この章はSQLデータベース間で異なって実装されたり、またはPostgreSQLに固有な幾つかの規則と概念を含んでいるので、SQLについて熟知しているユーザも本章を注意深く読むことをお勧めします。

4.1. 字句の構造

SQLの入力は、ひと続きの*コマンド*からなります。コマンドは*トークン*が繋がったもので構成され、最後はセミコロン(「;」)で終わります。入力ストリームの終了もやはりコマンドを終わらせます。どのトークンが有効かは特定のコマンドの構文によります。

トークンは*キーワード*、識別子、引用符で囲まれた識別子、リテラル(もしくは定数)、特別な文字シンボルです。トークンは通常空白(スペース、タブ、改行)で区切られますが、曖昧さがなければ(一般的には特別な文字が他のトークン型と隣接している場合のみ)必要ありません。

例えば、以下のものは(構文的に)正しいSQLの入力です。

```
SELECT * FROM MY_TABLE;
UPDATE MY_TABLE SET A = 5;
INSERT INTO MY_TABLE VALUES (3, 'hi there');
```

この例は1行に1つのコマンドを記述した、3つのコマンドが連続しています(必ずしも1つのコマンドを1行で書く必要はありません。1行に複数のコマンドを入力することも可能ですし、1つのコマンドを複数行に分けて記述することも可能です)。

さらに、入力されたSQLに*コメント*が付いていても構いません。コメントはトークンではなく、その効果は空白と同じです。

SQL構文は、どのトークンがコマンドを識別し、どれがオペランドでどれがパラメータかに関してはさほど首尾一貫していません。最初のいくつかのトークンは一般にコマンド名です。したがって、上記の例において「SELECT」、「UPDATE」、「INSERT」コマンドについて通常説明することになります。しかし、例えばUPDATEコマンドでは、SETトークンが特定の位置に常に記述されなければなりませんし、この例で使われているINSERTコマンドを完結するためにはVALUESトークンが必要です。それぞれのコマンドの正確な構文規則はパート VIで説明されています。

4.1.1. 識別子とキーワード

上記の例に出てくるSELECT、UPDATE、もしくはVALUESのようなトークンは、*キーワード*の一例です。キーワードとは、SQL言語で決まった意味を持っている単語です。MY_TABLEトークンやAトークンは*識別子*の一例です。これらは、使われるコマンドによって、テーブル、列、他のデータベースオブジェクトの名前を識別します。したがって、単に「名前」と呼ばれることもあります。キーワードと識別子は同じ字句の構造を持つため、言語を知らなくてはトークンが識別子なのかキーワードなのかわからないということになります。全てのキーワードのリストは付録Cにあります。

SQL識別子とキーワードは、文字（a～zおよび発音区別符号付き文字と非Latin文字)、アンダースコア（_）で始まらなければいけません。識別子またはキーワードの中で続く文字は、文字、アンダースコア、数字（0～9）あるいはドル記号($)を使用することができます。標準SQLの記述に従うと、ドル記号は識別子内では使用できないことに注意してください。ですから、これを使用するとアプリケーションの移植性は低くなる可能性があります。標準SQLでは、数字を含む、あるいはアンダースコアで始まったり終わったりするキーワードは定義されていません。したがって、この形式の識別子は標準の今後の拡張と競合する可能性がないという意味で安全と言えます。

システムはNAMEDATALEN-1バイトより長い識別子を使いません。より長い名前をコマンドで書くことはできますが、短く切られてしまいます。デフォルトではNAMEDATALENは64なので、識別子は最長で63バイトです。この制限が問題になる場合は、src/include/pg_config_manual.h内のNAMEDATALEN定数の値を変更して増やすことができます。

キーワードと引用符付きでない識別子は大文字と小文字を区別しません。したがって、

```
UPDATE MY_TABLE SET A = 5;
```

は、以下の文と同じ意味になります。

```
uPDaTE my_TabLE SeT a = 5;
```

慣習的によく使われる方法では、キーワードを大文字で、名前を小文字で書きます。例えば下記のようになります。

```
UPDATE my_table SET a = 5;
```

識別子には副次的な種類もあります。区切り識別子あるいは引用符付き識別子です。任意の文字の連なりを二重引用符（"）で囲んだものです。区切り識別子は常に識別子であって、キーワードではありません。ですから、"select"は「select」という名前の列あるいはテーブルを問い合わせるために使えますが、引用符の付かないselectはキーワードとして理解されるので、テーブルもしくは列名が期待される部分では解析エラーを起こします。引用符付き識別子は下記の例のように書くことができます。

```
UPDATE "my_table" SET "a" = 5;
```

引用符付き識別子は、コード0の文字以外であればどのような文字でも使えます（二重引用符を含めたい場合は、二重引用符を2つ入力します）。これにより、空白やアンパサンド（&）を含むテーブル名や列名など、この方法がなければ作れないような名前のものを作ることが可能になります。この場合においても長さの制限は適用されます。

引用符が付かない名前は常に小文字に解釈されますが、識別子を引用符で囲むことによって大文字と小文字が区別されるようになります。例えば、識別子FOO、foo、"foo"はPostgreSQLによれば同じものとして解釈されますが、"Foo"と"FOO"は、これら3つとも、またお互いに違ったものとして解釈されます（PostgreSQLが引用符の付かない名前を小文字として解釈することは標準SQLと互換性がありません。標準SQLでは引用符の付かない名前は大文字に解釈されるべきだとされています。したがって標準SQLによれば、fooは"FOO"と同じであるべきで、"foo"とは異なるはずなのです。もし移植可能なアプリケーションを書きたいならば、特定の名前は常に引用符で囲むか、あるいはまったく囲まないかのいずれかに統一することをお勧めします。）

引用符付き識別子には異形があり、コード番号で識別されるエスケープされたUnicode文字を含むことができます。この異形は、U&（大文字または小文字のUの後にアンパサンド）で始まり、その直後に空白を間に入れずに二重引用符を続けます。例えば、U&"foo"となります。（これにより演算子&との不明確性が生じることに注意してください。この問題を回避するには空白を演算子の前後に入れます。）引用符の中で、Unicode文字はバックスラッシュとそれに続く4桁16進数の文字コード番号で、またはもう1つの方法として、バックスラッシュに続いてプラス符号、そして続いた6桁16進数の文字コード番号によりエスケープ形式で指定されます。例えば、識別子"data"は次のように書くことができます。

```
U&"d\0061t\+000061"
```

次の少し意味のある例はロシア語の「slon」（象）をキリル文字で書いたものです。

```
U&"\0441\043B\043E\043D"
```

バックスラッシュ以外のエスケープ文字を使用したい場合、文字列の後にUESCAPE句を使用して指定することが可能です。例をあげます。

```
U&"d!0061t!+000061" UESCAPE '!'
```

エスケープ文字には、16進表記用の文字、プラス記号、単一引用符、二重引用符、空白文字以外の任意の単一文字を使用することができます。エスケープ文字はUESCAPEの後に二重引用符ではなく単一引用符で記述していることに注意してください。

識別子内にエスケープ文字をそのまま含めるためには、それを2つ記述してください。

U+FFFFより大きなコードポイントを持つ文字を構成するUTF-16サロゲートペアを指定するために、4桁と6桁の形式のどちらかを使用できますが、技術的には6桁形式の機能によりこれは不要になります。（サロゲートペアは直接格納されるわけではなく、一つのコードポイントに結合されます。）

サーバ符号化方式がUTF-8でない場合、このエスケープシーケンスの1つで指定されたUnicodeコードポイントは実際のサーバ符号化方式へと変換されます。それが可能でない場合にはエラーが報告されます。

4.1.2. 定数

PostgreSQLには、3つの暗黙に型付けされる定数があります。文字列、ビット文字列、そして数字です。定数は明示的な型で指定することもでき、その場合はシステムによる、より正確な表現と効率の良い操作が可能になります。こうした他の方法については後ほど説明します。

4.1.2.1. 文字列定数

SQLにおける文字列定数は、単一引用符（'）で括られた任意の文字の並びです。例えば、'This is a string'です。文字列定数内に単一引用符を含めるには、2つ続けて単一引用符を記述します。例えば、'Dianne''s horse'です。二重引用符(")とは同一ではない点に注意してください。

2つの文字列定数が、少なくとも1つの改行を含んだ空白のみで区切られている場合は、2つの定数は連結され、実質的に1つの定数として書かれたように処理されます。例を示します。

```
SELECT 'foo'
'bar';
```

は、

```
SELECT 'foobar';
```

と同じです。しかし、

```
SELECT 'foo'      'bar';
```

は有効な構文ではありません（このちょっとした奇妙な振舞いはSQLで決められているもので、PostgreSQLではこの標準に従っています）。

4.1.2.2. C形式エスケープでの文字列定数

PostgreSQLでは、また、「エスケープ」文字列定数を受け付けます。これは標準SQLの拡張です。エスケープ文字列定数は、E(大文字でも小文字でもかまいません)を開始単一引用符の直前に記述することで指定されます。例えばE'foo'です。（複数行に渡るエスケープ文字列定数では、最初の開始引用符の前にのみEを記述してください。）エスケープ文字列の中では、バックスラッシュ文字（\）によりC言語のようなバックスラッシュシーケンスが開始し、その中でバックスラッシュとそれに続く文字の組み合わせが（表 4.1で示したように）特別なバイト値を表現します。

表4.1 バックスラッシュエスケープシーケンス

バックスラッシュエスケープシーケンス	解釈
\b	後退
\f	改ページ
\n	改行
\r	復帰
\t	タブ
\o, \oo, \ooo (o = 0–7)	8進数バイト値
\xh, \xhh (h = 0–9, A–F)	16進数バイト値
\uxxxx, \Uxxxxxxxx (x = 0–9, A–F)	16もしくは32ビットの16進数 Unicode 文字値

バックスラッシュの後のそのほかの全ての文字はそのまま扱われます。従って、バックスラッシュ文字を含ませるときは2つのバックスラッシュ（\\）を記載します。同時に、エスケープ文字列の中では、単一引用符を、通常の方法の''に加え、\'としても含めることができます。

特に8進数や16進数エスケープを用いて作成されるバイトシーケンスが、サーバ文字セット符号化方式において有効な文字で構成されていることはコードを書く人の責任です。便利な代替手段は、Unicodeエスケープか、4.1.2.3で説明するもう一つのUnicodeエスケープ構文を代わりとして使用することです。そうすればサーバが文字変換が可能か検査するでしょう。

> **注意**
>
> 設定パラメータstandard_conforming_stringsが offの場合、PostgreSQLはバックスラッシュエスケープを通常の文字列定数とエスケープ文字列定数の両方で認識します。しかし、PostgreSQL9.1からデフォルトはonになりました。これはバックスラッシュエスケープはエスケープ文字列でのみ認識されるということになります。この振る舞いはSQL標準仕様に即していますが、バックスラッシュエスケープを常に認識するという歴史的な動作に依存しているアプリケーションは動作しなくなるでしょう。回避策として、このパラメータをoffにすることはできますが、バックスラッシュエスケープの使用を避けるよう移植するのが良いでしょう。特殊文字を表現するためにバックスラッシュを使用する必要がある場合、Eをつけて文字列定数を記述してください。
>
> standard_conforming_stringsの他に、設定パラメータescape_string_warningおよびbackslash_quoteが文字定数内のバックスラッシュの動作を決定します。

コードゼロの文字は文字列定数の中に入れられません。

4.1.2.3. Unicodeエスケープがある文字列定数

PostgreSQLは同時に、文字コード番号で任意のUnicode文字を指定可能な文字列に対するもう一つのエスケープ構文を提供します。Unicodeエスケープ文字列定数は、U&(大文字・小文字のUの後にアンパサンド)で始まり、その直後に、空白を間にはさまず、開始引用符が続きます。例えば、U&'foo'となります。(これにより演算子&との曖昧性が生じることに注意してください。この問題を回避するには空白を演算子の前後に入れます。) 引用符の中で、Unicode文字はバックスラッシュとそれに続く4桁16進数の文字コード番号で、またはもう1つの方法として、バックスラッシュに続いてプラス符号、そして続いた6桁16進数の文字コード番号によりエスケープ形式で指定されます。例えば、文字列'data'は次のように書かれます。

```
U&'d\0061t\+000061'
```

次の少し意味のある例はロシア語の「slon」(象)をキリル文字で書いたものです。

```
U&'\0441\043B\043E\043D'
```

バックスラッシュ以外のエスケープ文字を使用したい場合、文字列の後にUESCAPE句を使用して指定することが可能です。例をあげます。

```
U&'d!0061t!+000061' UESCAPE '!'
```

エスケープ文字には、16進表記用の文字、プラス記号、単一引用符、二重引用符、空白文字以外の任意の単一文字を使用することができます。

識別子内にエスケープ文字をそのまま含めるためには、それを2つ記述してください。

U+FFFFより大きなコードポイントを持つ文字を構成するUTF-16サロゲートペアを指定するために、4桁と6桁の形式のどちらかを使用できますが、技術的には6桁形式の機能によりこれは不要になります。(サロゲートペアは直接格納されるわけではなく、一つのコードポイントに結合されます。)

サーバ符号化方式がUTF-8でない場合、このエスケープシーケンスの1つで指定されたUnicodeコードポイントは実際のサーバ符号化方式へと変換されます。それが可能でない場合にはエラーが報告されます。

また、文字列定数に対するユニコードエスケープ構文は設定パラメータstandard_conforming_stringsが有効なときのみ動作します。そうでないとこの構文は、SQL文を構文解釈するクライアントを混乱させ、SQLインジェクションや、それに類似したセキュリティ問題に繋がることさえあるからです。パラメータがoffに設定されていれば、この構文はエラーメッセージを出して拒絶されます。

4.1.2.4. ドル記号で引用符付けされた文字列定数

文字列定数の標準の構文はたいていの場合便利ですが、対象とする文字列内に多くの単一引用符やバックスラッシュがあると、それらを全て二重にしなければなりませんので理解しづらくなります。こうした状況においても問い合わせの可読性をより高めるためにPostgreSQLは、「ドル引用符付け」という他の文字列定数の指定方法を提供します。ドル引用符付けされた文字列定数は、ドル記号($)、省略可能な0個以上の文字からなる「タグ」、ドル記号、文字列定数を構成する任意の文字の並び、ドル記号、この引用符付けの始めに指定したものと同じタグ、ドル記号から構成されます。例えば、「Dianne's horse」という文字列をドル引用符付けを使用して指定する方法を、以下に2つ示します。

```
$$Dianne's horse$$
$SomeTag$Dianne's horse$SomeTag$
```

ドル引用符付けされた文字列の内側では、単一引用符をエスケープすることなく使用できることを理解して下さい。実際には、ドル引用符付けされた文字列の内側の文字はまったくエスケープが必要なく、文字列定数はすべてそのまま記述することができます。その並びが開始タグに一致しない限り、バックスラッシュもドル記号も特別なものではありません。

各入れ子レベルに異なるタグを付けることで、ドル引用符付けされた文字列を入れ子にすることができます。これは、関数定義を作成する時に非常によく使用されます。以下に例を示します。

```
$function$
BEGIN
    RETURN ($1 ~ $q$[\t\r\n\v\\]$q$);
END;
$function$
```

ここで、q[\t\r\n\v\\]qは、ドル引用符付けされた[\t\r\n\v\\]リテラル文字列を表し、PostgreSQLがこの関数本体を実行する時に認識されます。しかし、この並びは、外側のドル引用符用の区切り文字$function$に一致しませんので、外側の文字列を対象としている場合は単なる文字の並びとなります。

もしあれば、ドル引用符付けされた文字列のタグは、引用符付けされていない識別子と同じ規則に従います。ただし、タグにはドル記号を含めることはできません。タグは大文字小文字を区別します。したがって、tagString contenttagは正しいのですが、TAGString contenttagは間違いです。

キーワードや識別子の後にドル引用符付けされた文字列を続ける場合は、空白でそれを区切らなければなりません。さもないと、ドル引用符の区切り文字は、直前の識別子の一部として解釈されます。

ドル引用符付けは、標準SQLで定義されていません。しかし、複雑な文字列リテラルを記述する場合は標準準拠の単一引用符構文よりも便利なことがよくあります。特に、他の定数の内部に文字列定数を記述するよ

うな場合は役に立ちます。こうした状況は手続き関数の定義でよく必要とされます。単一引用符構文では、上の例のバックスラッシュはそれぞれ、4個のバックスラッシュで記述しなければなりません。この4つのバックスラッシュは、元の文字列定数を解析する際に2つに減少され、そして、関数を実行する際に内部の文字列定数が再解析され1つに減少します。

4.1.2.5. ビット文字列定数

ビット文字列定数はB（大文字もしくは小文字）が始まりの引用符の前に付いている（間に空白はありません）通常の文字列定数のように見えます。例えばB'1001'のようになります。ビット文字列定数の中で許可される文字は0と1のみです。

その他にも、ビット文字列定数はX'1FF'といった具合に、先頭にX（大文字または小文字）を使用して16進表記で指定することもできます。この表記は、各16進数値をそれぞれ4つの2進数値に置き換えたビット文字列定数と同等です。

どちらの形式のビット文字列定数でも、通常の文字列定数と同じように複数行にわたって続けて書くことができます。ドル引用符付けはビット文字列定数では使用できません。

4.1.2.6. 数値定数

数値定数は下記の一般的な形で受け付けられます。

```
digits
digits.[digits][e[+-]digits]
[digits].digits[e[+-]digits]
digitse[+-]digits
```

ここでdigitsは1つ以上の10進数字（0～9）です。小数点を使用する場合は、少なくとも1つの数字が小数点の前か後になくてはなりません。指数記号eの付く形式を使う場合にはeの後に少なくとも1つの数字がなければいけません。空白や他の文字は、定数の中に埋め込むことはできません。プラスまたはマイナスの符号を先頭につけても、定数の一部とはみなされないことに注意してください。これらの符号は定数に適用される演算子とみなされます。

下記は有効な数値定数のいくつかの例です。

```
42
3.5
4.
.001
5e2
1.925e-3
```

小数点も指数も含まない数値定数の場合、まずその値がinteger型（32ビット）に収まればinteger型であるとみなされます。そうでない場合、bigint型（64ビット）で収まればbigint型とみなされます。どちらでもない場合は、numeric型とみなされます。定数が小数点または指数あるいはその両方を含む場合は、常に最初にnumeric型であるとみなされます。

数値定数に最初に割り振られるデータ型は、型解決アルゴリズムの開始点に過ぎません。ほとんどの場合、定数は文脈に基づいて自動的に最も適切な型に変換されます。必要であれば、特定のデータ型にキャストして、数値がそのデータ型として解釈されるように強制することができます。例えば、以下のようにして数値をreal型(float4)として処理することができます。

```
REAL '1.23'  -- 文字列書式
1.23::REAL   --  (歴史的な) PostgreSQL書式
```

実のところ、これらは以下で説明する一般的なキャスト記法の特別な場合です。

4.1.2.7. 他の型の定数

任意の型の定数は下記の表記のいずれかを使って入力することができます。

```
type 'string'
'string'::type
CAST ( 'string' AS type )
```

文字列定数のテキストはtypeと呼ばれる型の入力変換ルーチンへと渡されます。結果は指示された型の定数です。明示的な型キャストは、定数がどの型でなければならないかについて曖昧な点がなければ(例えば定数が直接テーブル列に代入されている場合)省略しても構いません。その場合自動的に型強制されます。

文字列定数は通常のSQL記法でもドル引用符付けでも記述することができます。

関数のような構文を使って型強制を指定することも可能です。

```
typename ( 'string' )
```

しかし、全ての型の名前でこの方法は使用できるというわけではありません。詳細は4.2.9を参照してください。

::、CAST()や関数呼び出し構文は、4.2.9で説明する通り、任意の式の実行時の型変換を指定するために使うこともできます。構文的なあいまいさをなくすために、type 'string'という形式は単なるリテラル定数を指定する場合にのみ使うことができます。この他type 'string'構文には、配列型では動作しないという制限があります。配列型の定数の型を指定する場合は::かCAST()を使用してください。

CAST()構文はSQLに従っています。type 'string'構文は、標準を一般化したものです。SQLでは、この構文を数個のデータ型でのみ規定しています。しかし、PostgreSQLではすべての型で使用することができます。::付きの構文は、歴史的にPostgreSQLで使用されてきました。関数呼び出し構文も同じく歴史的に使用されているものです。

4.1.3. 演算子

演算子はNAMEDATALEN-1(デフォルトは63)までの長さの、以下に示すリストに含まれる文字の並びです。

```
+ - * / < > = ~ ! @ # % ^ & | ` ?
```

しかし、演算子の名前にはいくつかの制約があります。

- --と/*は演算子名の中に使うことができません。なぜならこれらはコメントの始まりと解釈されるからです。
- 複数文字の演算子名は、その名前が少なくとも下記の文字の1つ以上を含まない限り、+や-で終わることができません。

  ```
  ~ ! @ # % ^ & | ` ?
  ```

  例えば、@-は演算子名として認められていますが、*-は認められていません。この制限によりPostgreSQLは、SQLに準拠する問い合わせをトークン同士の間に空白を要求せず、解析することができます。

非SQL標準の演算子名を使う場合、通常は曖昧さを回避するために、隣り合った演算子を空白で区切る必要があります。例えば@という左単項演算子を定義した場合、X*@Yとは書けません。PostgreSQLがこれを確実に1つではなく2つの演算子名として解釈できるように、X* @Yと書く必要があります。

4.1.4. 特殊文字

英数字ではないいくつかの文字は、演算子であることとは異なる特殊な意味を持っています。使用方法の詳細はそれぞれの構文要素についてのところで説明します。本節では、単にその存在を知らせ、これらの文字の目的をまとめるに留めます。

- 直後に数字が続くドル記号($)は、関数定義の本体またはプリペアド文中の位置パラメータを表すために使われます。他の文脈ではドル記号は識別子名の一部であるかもしれませんし、ドル引用符付けされた文字列定数の一部であるかもしれません。
- 括弧(())は、通常通り式をまとめ優先するという意味を持ちます。場合によっては括弧は、特定のSQLコマンドの固定構文の一部として要求されることがあります。
- 大括弧([])は、配列要素を選択するために使われます。配列に関する詳しい情報は8.15を参照してください。
- カンマ(,)は、リストの要素を区切るために構文的構成体で使われることがあります。
- セミコロン(;)は、SQLコマンドの終わりを意味します。文字列定数または引用符付き識別子以外では、コマンドの途中では使うことができません。
- コロン(:)は、配列から「一部分」を取り出すために使われます (8.15を参照してください)。いくつかのSQL方言(埋め込みSQLなど)では、コロンは変数名の接頭辞として使われます。
- アスタリスク(*)は、いくつかの文脈において、テーブル行や複合型の全てのフィールドを表現するために使用されます。また、集約関数の引数として使われる場合も特殊な、つまり、その集約が明示的なパラメータをまったく必要としないという意味を持ちます。
- ピリオド(.)は数値定数の中で使われます。また、スキーマ名、テーブル名、列名を区切るためにも使われます。

4.1.5. コメント

コメントは二重ハイフンで始まる文字の並びで、行の終わりまで続きます。例えば以下のようになります。

```
-- これは標準SQLのコメントです
```

他にも、C言語様式のブロックコメントも使用できます。

```
/* ネストされた複数行にわたる
 * コメント /* ネストされたブロックコメント */
 */
```

コメントは/*で始まり、対応する*/で終わります。これらのブロックコメントはC言語とは異なり、標準SQLで規定されているように入れ子にすることができます。したがって、既存のブロックコメントを含む可能性のある大きなコードのブロックをコメントアウトすることができます。

コメントは、その後の構文解析が行われる前に入力ストリームから取り去られ、事実上、空白で置き換えられます。

4.1.6. 演算子の優先順位

表 4.2は、PostgreSQLの演算子の優先順位と結合性を示しています。ほとんどの演算子は同じ優先順位を持ち、左結合します。演算子の優先順位と結合性はパーサに組み込まれています。複数の演算子のある式を優先順位の規則が意味するのとは異なる順序で解析したい場合には、括弧で囲ってください。

表4.2 演算子の優先順位（高いものから低いものへ）

演算子/要素	結合性	説明
.	左	テーブル/列名の区切り文字
::	左	PostgreSQL方式の型キャスト
[ ]	左	配列要素選択
+ -	右	単項加算、単項減算
^	左	累乗
* / %	左	掛け算、割り算、剰余
+ -	左	加算、減算
(その他の演算子)	左	その他全ての組み込み、あるいはユーザ定義の演算子
BETWEEN IN LIKE ILIKE SIMILAR		範囲内に包含、集合の要素、文字列の一致
< > = <= >= <>		比較演算子
IS ISNULL NOTNULL		IS TRUE、IS FALSE、IS NULL、IS DISTINCT FROM、その他
NOT	右	論理否定
AND	左	論理積
OR	左	論理和

演算子優先順位の規則は、上記で触れた組み込み演算子と同じ名前を持つユーザ定義演算子にも当てはまります。例えばもし「+」演算子をある独自のデータ型に定義すると、新しい演算子が何をするかにかかわらず、「+」組み込み演算子と同じ優先順位を持つようになります。

次の例のように、OPERATOR構文でスキーマで修飾された演算子名を使用する場合、

```
SELECT 3 OPERATOR(pg_catalog.+) 4;
```

OPERATOR構文は、表 4.2の「その他の演算子」で示されているデフォルトの優先順位を持つとみなされます。これは、OPERATOR()にどの特定の演算子が入る場合でも変わりません。

> **注記**
>
> 9.5より前のPostgreSQLのバージョンでは少し異なる演算子優先順位規則を使っていました。特に<=、>=、<>は一般的な演算子として扱われていました。ISテストは高い優先順位を持つとして使われていました。NOT BETWEENとそれに関係する構文は振る舞いが一貫しておらず、BETWEENではなくNOTの優先順位を持つと見なされる場合がありました。標準SQLにより準拠し、論理的に等しい構文の一貫しない扱いから来る混乱を減らすように、これらの規則は変更されました。ほとんどの場合、これらの変更により振る舞いが変わることはないでしょうし、もし変わっても恐らく「no such operator」で失敗になるくらいでしょう。後者は括弧を追加することで解決できるでしょう。しかしながら、稀に問い合わせがパースエラーを返すことなく振る舞いを変える場合があります。これらの変更が黙って何かを壊してしまったかどうかが心配であれば、設定パラメータoperator_precedence_warningをオンにして、何か警告がログに書き込まれるかを見てください。

4.2. 評価式

評価式は、例えばSELECTコマンドの目的リストとして、INSERTやUPDATEの新しい列の値として、もしくはいくつかのコマンドの検索条件として様々な文脈の中で使われます。評価式の結果は、テーブル式の結果（つまりテーブル）から区別するために、スカラと呼ばれることもあります。したがって、評価式はスカラ式（またはもっと簡単に式）とも呼ばれます。式の構文によって、基本的な部分から算術、論理、集合などの演算を使って値の計算を行うことができます。

評価式は下記のうちのいずれかです。

- 定数あるいはリテラル値
- 列の参照
- 関数定義の本体やプリペアド文における、位置パラメータ参照
- 添字付きの式
- フィールド選択式
- 演算子の呼び出し
- 関数呼び出し

- 集約式
- ウィンドウ関数呼び出し
- 型キャスト
- 照合順序(collation)式
- スカラ副問い合わせ
- 配列コンストラクタ
- 行コンストラクタ
- (副式をグループ化したり優先順位を変更するのに使用される)括弧で囲まれた別の評価式

これ以外にも、式として分類されるけれども一般的な構文規約には従わない、いくつかの構成要素があります。これらは一般的に関数あるいは演算子の意味を持ちます。第9章の該当部分で説明されています。例を挙げるとIS NULL句があります。

4.1.2で既に定数については説明しました。続く節では残りのオプションについて説明します。

4.2.1. 列の参照

列は、下記のような形式で参照することができます。

```
correlation.columnname
```

correlationは、テーブル名(スキーマで修飾されている場合もあります)、あるいはFROM句で定義されたテーブルの別名です。correlationの名前と区切り用のドットは、もし列名が現在の問い合わせで使われる全てのテーブルを通して一意である場合は省略することができます。(第7章も参照してください)。

4.2.2. 位置パラメータ

位置パラメータ参照は、外部からSQL文に渡される値を示すために使用されます。パラメータはSQL関数定義およびプリペアド問い合わせの中で使用されます。また、クライアントライブラリの中には、SQLコマンド文字列とデータ値を分離して指定できる機能をサポートするものもあります。この場合、パラメータは行外データ値を参照するために使用されます。パラメータ参照の形式は以下の通りです。

```
$number
```

例えば、関数 dept の定義が以下のようにされたとします。

```
CREATE FUNCTION dept(text) RETURNS dept
    AS $$ SELECT * FROM dept WHERE name = $1 $$
    LANGUAGE SQL;
```

ここで$1は関数が呼び出される時に最初の関数引数の値を参照します。

4.2.3. 添字

式が配列型の値となる場合、配列値の特定要素は以下のように記述することで抽出できます。

```
expression[subscript]
```

また、隣接する複数の要素(「配列の一部分」)は以下のように記述することで抽出できます。

```
expression[lower_subscript:upper_subscript]
```

(ここで大括弧[ ]は文字通りに記述してください(訳注:これはオプション部分を表す大括弧ではありません)。) 各subscriptはそれ自体が式であり、最も近い整数値へと丸められます。

一般的には、配列expressionは括弧で括らなければなりませんが、添字を付けるそのexpressionが単なる列参照や位置パラメータであった場合、その括弧を省略することができます。また、元の配列が多次元の場合は複数の添字を連結することができます。以下に例を示します。

```
mytable.arraycolumn[4]
mytable.two_d_column[17][34]
$1[10:42]
(arrayfunction(a,b))[42]
```

最後の例では括弧が必要です。配列の詳細は8.15を参照してください。

4.2.4. フィールド選択

式が複合型(行型)の値を生成する場合、行の特定のフィールドは以下のように記述することで抽出できます。

```
expression.fieldname
```

一般的には、行expressionは括弧で括らなければなりません。しかし、選択元となる式が単なるテーブル参照や位置パラメータの場合、括弧を省略することができます。以下に例を示します。

```
mytable.mycolumn
$1.somecolumn
(rowfunction(a,b)).col3
```

(したがって、修飾された列参照は実際のところ、単なるこのフィールド選択構文の特殊な場合です。) 重要となる特殊な場合としては、複合型のテーブル列からフィールドを抽出するときです。

```
(compositecol).somefield
(mytable.compositecol).somefield
```

compositecolがテーブル名でなく列名であること、または2番目の場合のmytableがスキーマ名でなくテーブル名であることを示すため丸括弧が要求されます。

.*を記述することで、複合型の全ての値を問い合わせることが可能です。

```
(compositecol).*
```

この表記はコンテキストに依存して異なった振る舞いをします。詳細は8.16.5を参照してください。

4.2.5. 演算子の呼び出し

演算子の呼び出しには以下の3構文が可能です。

expression operator expression（二項中置演算子）
operator expression（単項前置演算子）
expression operator（単項後置演算子）

ここでoperatorトークンは、4.1.3構文規則に従うもの、もしくはキーワードAND、OR、NOTのいずれか、または以下の形式の修飾された演算子名です。

```
OPERATOR(schema.operatorname)
```

具体的にどんな演算子が存在し、それが単項か二項かどうかは、システムやユーザによってどんな演算子が定義されたかに依存します。第9章にて、組み込み演算子について説明します。

4.2.6. 関数呼び出し

関数呼び出しの構文は、関数名（スキーマ名で修飾されている場合があります）に続けてその引数を丸括弧で囲んで列挙したものです。

```
function_name ([expression [, expression ... ]] )
```

例えば、以下のものは2の平方根を計算します。

```
sqrt(2)
```

組み込み関数の一覧は第9章にあります。他の関数はユーザが追加できます。

あるユーザが他のユーザを信用しないデータベースで問い合わせを発行する場合には、関数呼び出しを書く時に10.3のセキュリティの事前の対策を守ってください。

引数には名前を任意で付与することができます。詳細は4.3を見て下さい。

> **注記**
>
> 複合型の単一引数をとる関数はフィールド選択の構文を使っても呼び出すことができます。反対にフィールド選択を関数形式で記述することもできます。つまり、col(table)やtable.colのどちらを使っても良いということです。この動作は標準SQLにはありませんが、PostgreSQLでは、これにより「計算されたフィールド」のエミュレートをする関数の利用が可能になるため、提供しています。詳しくは8.16.5を参照してください。

4.2.7. 集約式

集約式は、問い合わせによって選択される行に対して集約関数を適用することを表現します。集約関数は、例えば入力の合計や平均などのように、複数の入力を単一の出力値にします。集約式の構文は下記のうちのいずれかです。

```
aggregate_name (expression [ , ... ] [ order_by_clause ] ) [ FILTER ( WHERE filter_clause ) ]
aggregate_name (ALL expression [ , ... ] [ order_by_clause ] ) [ FILTER ( WHERE filter_clause ) ]
aggregate_name (DISTINCT expression [ , ... ] [ order_by_clause ] ) [ FILTER ( WHERE filter_clause
 ) ]
aggregate_name ( * ) [ FILTER ( WHERE filter_clause ) ]
aggregate_name ( [ expression [ , ... ] ] ) WITHIN GROUP ( order_by_clause ) [ FILTER
 ( WHERE filter_clause ) ]
```

ここで、aggregate_nameは事前に定義された集約(スキーマ名で修飾された場合もあります)、expressionはそれ自体に集約式またはウィンドウ関数呼び出しを含まない任意の値評価式です。省略可能なorder_by_clauseとfilter_clauseは後述します。

集約式の最初の構文は、それぞれの入力行に対して1回ずつ集計を呼び出します。ALLはデフォルトなので、2つ目の形式は最初の形式と同じです。3番目の形式は、入力行の中にある式の、全ての重複しない値(複数式では重複しない値集合)の集約を呼び出します。4番目の形式はそれぞれの入力行に対して1回ずつ集約を呼び出します。具体的な入力値が指定されていないため、これは一般的にcount(*)集約関数でのみ役に立ちます。最後の形式は順序集合集約関数で使われるもので、順序集合集約関数については後述します。

ほとんどの集約関数はNULL入力を無視するため、行内の1つ以上の式がNULLを返す行は破棄されます。特記されていない限り、すべての組み込み集約がそのような動作になると想定して良いです。

例えば、count(*)は入力行の合計数を求めます。countはNULLを無視しますので、count(f1)はf1が非NULLである入力行の数を求めます。count(distinct f1)はf1の重複しない非NULL値の数を求めます。

通常、入力行は順序を指定されずに集計関数に与えられます。多くの場合では問題になりません。たとえばminは入力順序に関係なく同一の結果を返します。しかし一部の集約関数(array_aggおよびstring_aggなど)は入力行の順序に依存した結果を返します。こうした集約関数を使用する際は、オプションのorder_by_clauseを使用して必要とする順序を指定できます。order_by_clauseは、7.5で説明する問い合わせレベルのORDER BY句と同じ構文を取りますが、その式は常に単なる式であり、出力列名や序数とすることはできません。以下に例を示します。

```
SELECT array_agg(a ORDER BY b DESC) FROM table;
```

複数の引数を取る集約関数を扱う場合、ORDER BY句はすべての集約引数の後に指定することに注意してください。例えば、

```
SELECT string_agg(a, ',' ORDER BY a) FROM table;
```

であり、

```
SELECT string_agg(a ORDER BY a, ',') FROM table;  -- incorrect
```

ではありません。後者は構文的には有効なものですが、2つのORDER BYキーを持つ単一引数の集約関数の呼び出しを表しています。(2つ目のキーは定数なので役には立ちません。)

order_by_clauseに加えDISTINCTが指定された場合、すべてのORDER BY式が集約関数の通常の引数に一致しなければなりません。つまり、DISTINCTリストに含まれない式でソートすることはできません。

> **注記**
>
> 集計関数においてDISTINCTとORDER BYの両方を指定できる機能はPostgreSQLの拡張です。

上記のように集約の通常の引数リストにORDER BYを置くことは、汎用的で統計的な集約への入力行を整列する時に使いますが、その整列は省略可能です。たいていは集約の計算がその入力行の特定の順序に関してのみ意味を持つために、order_by_clauseが*必要な順序集合集約*と呼ばれる集約関数の副クラスがあります。順序集合集約の典型的な例は順位や百分位数の計算を含みます。順序集合集約では、order_by_clauseは上の構文の最後に示したようにWITHIN GROUP (...)の中に書かれます。order_by_clauseの式は、通常の集約の引数のように入力行1行につき一度評価され、order_by_clauseの要求に従って整列され、集約関数に入力引数として渡されます。(非WITHIN GROUP order_by_clauseではない場合はこれとは異なり、集約関数の引数としては扱われません。) WITHIN GROUPの前に引数の式があれば、order_by_clauseに書かれた*集約引数*と区別するために*直接引数*と呼ばれます。通常の集約引数とは異なり、直接引数は集約の呼び出しの時に一度だけ評価され、入力行1行に一度ではありません。これは、変数がGROUP BYによりグループ化された場合にのみ、その変数を含むことが可能であることを意味します。この制限は直接引数が集約式の中に全くない場合と同じです。直接引数は、典型的には1度の集約計算で1つの値だけが意味がある百分位数のようなもので使われます。直接引数のリストは空でも構いません。この場合、(*)ではなく()と書いてください。(PostgreSQLは実際にどちらの綴りも受け付けますが、後者だけが標準SQLに準拠しています。)

順序集合集約の例は以下の通りです。

```
SELECT percentile_cont(0.5) WITHIN GROUP (ORDER BY income) FROM households;
 percentile_cont
-----------------
           50489
```

これは、テーブルhouseholdsからincome列の50番目の百分位数、すなわち中央値を得ます。ここで0.5は直接引数です。百分位数が行毎に変化する値であったら意味がありません。

FILTERが指定されていれば、filter_clauseが真と評価した入力行のみがウィンドウ関数に渡されます。それ以外の行は破棄されます。例えば、

```
SELECT
    count(*) AS unfiltered,
    count(*) FILTER (WHERE i < 5) AS filtered
FROM generate_series(1,10) AS s(i);
 unfiltered | filtered
------------+----------
         10 |        4
(1 row)
```

定義済みの集約関数は9.21で説明されています。ユーザは他の集約関数を追加することができます。

集約式は、SELECTコマンドの結果リストもしくはHAVING句内でのみ記述することができます。WHEREなどの他の句では許されません。これらの句は集計結果が形成される前に論理的に評価されるためです。

集約式が副問い合わせ（4.2.11と9.23を参照）内に現れた場合、通常、集約は副問い合わせの行全体に対して評価されます。しかし、その集約の引数(と、もしあればfilter_clause)が上位レベルの変数のみを持つ場合は例外です。その場合、集約は最も近い外側のレベルに属し、その問い合わせの行全体に対して評価されます。全体として、その集約式は、その後、その集約を含む副問い合わせでは外部参照となり、その副問い合わせにおける評価に対しては定数として動作します。結果リストもしくはHAVING句にのみ現れるという制約は、その集約が属する問い合わせレベルに関連して適用されます。

4.2.8. ウィンドウ関数呼び出し

ウィンドウ関数呼び出しは、問い合わせにより選択された行のある部分に渡って集約のような機能を実現することを表します。非ウィンドウ集約関数呼び出しと異なり、これは選択された行を1つの行にグループ化することに束縛されず、各行は別途問い合わせ出力に残ります。しかしウィンドウ関数は、ウィンドウ関数呼び出しのグループ化指定（PARTITION BYリスト）に従った、現在の行のグループの一部となる行にすべてアクセスできます。ウィンドウ関数呼び出しの構文は以下のいずれかです。

```
function_name ([expression [, expression ... ]]) [ FILTER ( WHERE filter_clause ) ]
 OVER window_name
function_name ([expression [, expression ... ]]) [ FILTER ( WHERE filter_clause ) ] OVER
 ( window_definition )
function_name ( * ) [ FILTER ( WHERE filter_clause ) ] OVER window_name
function_name ( * ) [ FILTER ( WHERE filter_clause ) ] OVER ( window_definition )
```

ここで、window_definitionは以下の構文になります。

```
[ existing_window_name ]
[ PARTITION BY expression [, ...] ]
[ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST | LAST } ] [, ...] ]
[ frame_clause ]
```

オプションのframe_clauseは次の中の1つです。

```
{ RANGE | ROWS | GROUPS } frame_start [ frame_exclusion ]
{ RANGE | ROWS | GROUPS } BETWEEN frame_start AND frame_end [ frame_exclusion ]
```

ここでframe_startおよびframe_endは以下のいずれかです。

```
UNBOUNDED PRECEDING
offset PRECEDING
CURRENT ROW
offset FOLLOWING
UNBOUNDED FOLLOWING
```

そして、frame_exclusionは以下のいずれかです。

```
EXCLUDE CURRENT ROW
EXCLUDE GROUP
EXCLUDE TIES
EXCLUDE NO OTHERS
```

ここで、expressionはそれ自身ウィンドウ関数呼び出しを含まない任意の値式を表わします。

window_nameは、問い合わせのWINDOW句で定義された名前付きウィンドウ仕様への参照です。あるいはまた、完全なwindow_definitionをWINDOW句で定義された名前付きウィンドウと同じ構文を使って丸括弧の中に書くことができます。詳細はSELECTマニュアルページを見てください。OVER wnameはOVER (wname ...)とは厳密には等価ではないことを指摘しておくのは価値のあることでしょう。後者はウィンドウ定義をコピーしたり修正したりすることを示唆しており、参照されるウィンドウ仕様がフレーム句を含む場合には拒絶されます。

PARTITION BY句は問い合わせの行を*パーティション*に纏め、パーティションはウィンドウ関数により別々に処理されます。PARTITION BYは、その式が常に式であって出力列名や番号ではないという点を除いて、問い合わせレベルのGROUP BY句と同様に動作します。PARTITION BYがなければ、問い合わせで生じる行すべてが一つのパーティションとして扱われます。ORDER BY句はパーティションの行がウィンドウ関数により処理される順序を決定します。問い合わせレベルのORDER BY句と同様に動作しますが、やはり出力列名や番号は使えません。ORDER BYがなければ、行は不定の順序で処理されます。

frame_clauseは、パーティション全体ではなくフレーム上で作動するウィンドウ関数に対して、*ウィンドウフレーム*を構成する行の集合を指定します。ウィンドウフレームは現在のパーティションの部分集合になります。フレームの中の行の集合は、どの行が現在の行であるかによって変わります。フレームはRANGEモードでもROWSモードでも指定できます。どちらの場合でもframe_startからframe_endまでです。frame_endを省略した場合のデフォルトはCURRENT ROWです。

frame_startがUNBOUNDED PRECEDINGならばフレームがパーティションの最初の行から始まること意味し、同様に、frame_endがUNBOUNDED FOLLOWINGならばフレームがパーティションの最後の行で終わること意味します。

RANGEあるいはGROUPSモードでは、frame_startがCURRENT ROWならば、フレームが現在行の最初の*ピア*行(ウィンドウのORDER BY句が現在行と同じ順序とみなす行)から始まることを意味し、一方、frame_endがCURRENT ROWならばフレームが現在行の最後の同等なORDER BYピア行で終わることを意味します。ROWSモードでは、CURRENT ROWは単に現在行を意味します。

offset PRECEDINGとoffset FOLLOWINGフレームオプションでは、offsetは一切の変数、集計関数、あるいはウィンドウ関数を含まない式でなければなりません。offsetの意味はフレームモードに依存します。

- ROWSモードでは、offsetの評価値は非NULL、非負の整数でなければならず、このオプションは現在行の前あるいは後の指定した数の行でフレームが開始あるいは終了することを意味します。
- GROUPSモードでも、offsetの評価値は非NULL、非負の整数でなければならず、このオプションは現在行のピアグループ(peer group)の前あるいは後の指定した数の*ピアグループ*でフレームが開始あるいは終了することを意味します。ここでピアグループは、ORDER BYによる順序付け中で等しい行の集合です。(ウィンドウ定義でGROUPSモードを使うには、ORDER BY句が存在しなければなりません。)
- RANGEモードでは、ORDER BY句が正確に一つの列を指定することがこれらのオプションによって要求されます。offsetは現在行の列の値と、フレーム中の前あるいは後ろの行の値の最大の差を指定します。

offset式のデータ型は、順序付けをしている列のデータ型に依存して変わります。数値型の順序付け列では、典型的には順序付け列と同じですが、日付時間の順序付け列では、intervalになります。たとえば、順序付け列の型がdateあるいはtimestampなら、RANGE BETWEEN '1 day' PRECEDING AND '10 days' FOLLOWINGと書くことができるでしょう。ここでもoffsetは非NULLかつ非負である必要があります。ただし、「非負」の意味はデータ型に依存します。

どの場合でも、フレームの最後までの距離はパーティションの最後までの距離に制限されます。ですからパーティションの最後近くの行では他の場合に比べてフレームには少ない行が含まれるかも知れません。

ROWSとGROUPSモードでは、0 PRECEDINGと0 FOLLOWINGはCURRENT ROWと同じであることに注意してください。データ型固有の意味で「0」が適切ならば、通常RANGEにおいても同様です。

フレームの開始、終了オプションで含まれることになる行であっても、frame_exclusionオプションで現在行周辺の行がフレームに含まれないようにすることができます。EXCLUDE GROUPは、現在行とその順序付ピアをフレームから除外します。EXCLUDE TIESは、現在行そのものを除き、フレームにおける現在行のピアをフレームから除外します。EXCLUDE NO OTHERSは、現在の行あるいはそのピアを除外しないというデフォルトの挙動を明示的に指定するだけです。

デフォルトのフレーム化オプションはRANGE UNBOUNDED PRECEDINGで、RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROWと同じです。ORDER BYがあると、フレームはパーティションの開始から現在行の最後のORDER BYピア行までのすべての行になります。ORDER BYが無い場合は、すべての行が現在行のピアとなるので、パーティションのすべての行がウィンドウフレームに含まれることを意味することになります。

制限は、frame_startをUNBOUNDED FOLLOWINGとすることができない点、frame_endをUNBOUNDED PRECEDINGとすることができない点、および、上記のframe_startとframe_endのオプションのリストで、frame_endの選択がframe_startの選択よりも先に現れるものであってはならない点です。例えば、RANGE BETWEEN CURRENT ROW AND offset PRECEDINGは許されません。しかし、例えば、決してどの行も選択しないとしても、ROWS BETWEEN 7 PRECEDING AND 8 PRECEDINGは許されます。

FILTERが指定されていれば、filter_clauseが真と評価した入力行のみがウィンドウ関数に渡されます。それ以外の行は破棄されます。集約ウィンドウ関数だけがFILTER句を受け付けます。

組み込みウィンドウ関数は表 9.60に記載されています。その他のウィンドウ関数をユーザが追加することが可能です。また、全ての組み込み、またはユーザ定義の、汎用または統計集約関数もウィンドウ関数として使用できます。(順序集合と仮想集合集約は現在のところウィンドウ関数として使用できません。)

*を使用した構文は、例えばcount(*) OVER (PARTITION BY x ORDER BY y)のように、パラメータのない集約関数をウィンドウ関数として呼び出すために使用されます。アスタリスク(*)は習慣的にウィンドウ固有の関数には使われません。ウィンドウ固有の関数は、関数引数リストの中でDISTINCTやORDER BYが使われることを許可しません。

ウィンドウ関数呼び出しは問い合わせのSELECTリストとORDER BY句の中でのみ許可されます。

更なるウィンドウ関数についての情報は3.5、9.22、7.2.5にあります。

4.2.9. 型キャスト

型キャストは、あるデータ型から他のデータ型への変換を指定します。PostgreSQLは型キャストに2つの等価な構文を受け付けます。

```
CAST ( expression AS type )
expression::type
```

CAST構文はSQLに準拠したものです。::を使用する構文は、PostgreSQLで伝統的に使用されている方法です。

キャストが既知の型の評価式に適用された場合、それは実行時型変換を表します。このキャストは、適切な型変換操作が定義されている場合のみ成功します。4.1.2.7で示すように、これと定数のキャストの使用との微妙な違いに注意してください。修飾されていない文字列リテラルに対するキャストは、リテラル定数値の初期に割り当てられる型を表します。ですから、これは(文字列リテラル定数の内容がそのデータ型の入力構文で受け付けられるのであれば)全ての型で成功します。

評価式が生成しなければならない型に曖昧さがない場合(例えばテーブル列への代入時など)、明示的な型キャストは通常は省略することができます。その場合、システムは自動的に型キャストを適用します。しかし、自動キャストは、システムカタログに「暗黙的に適用しても問題なし」と示されている場合にのみ実行されます。その他のキャストは明示的なキャスト構文で呼び出す必要があります。この制限は、知らないうちに変換が実行されてしまうことを防ぐためのものです。

また、関数のような構文を使用して型キャストを指定することもできます。

```
typename ( expression )
```

しかし、これはその型の名前が関数の名前としても有効な場合にのみ動作します。例えば、double precisionはこの方式で使用できませんが、同等のfloat8は使用できます。また、interval、time、timestampという名前は、構文が衝突するため、二重引用符で括った場合にのみこの方式で使用できます。このように、この関数のようなキャスト構文は一貫性がなくなりがちですので、おそらくアプリケーションでは使用すべきではありません。

> **注記**
>
> この関数のような構文は、実際には単なる関数呼び出しです。2つの標準的なキャスト構文のうちの1つが実行時変換で使用されると、この構文は登録済みの関数を内部的に呼び出して変換を実行します。慣習的に、これらの変換関数は自身の出力型と同じ名前を持ち、これにより、「関数のような構文」は背後にある変換用関数を直接呼び出す以上のことを行いません。移植性を持つアプリケーションが依存すべきものでないことは明確です。詳細についてはCREATE CASTを参照してください。

4.2.10. 照合順序式

COLLATE句は式の照合順序規則を上書きします。適用するため次の様に式の後に追記します。

```
expr COLLATE collation
```

ここでcollationは識別子で、スキーマ修飾可能です。COLLATE句は演算子よりも結合優先度が高いです。必要に応じて括弧で囲うことができます。

もし照合順序が何も指定されなければ、データベースシステムは式にある列から照合順序を取得します。もし列に関する照合順序が式になければ、データベースのデフォルトの照合順序を使います。

COLLATE句の主な使われ方が2つあります。1つはORDER BY句での並び替え順序を上書きをするもので、例えば次のようにします。

```
SELECT a, b, c FROM tbl WHERE ... ORDER BY a COLLATE "C";
```

もう一つは、計算結果がロケールに依存する関数や演算子の呼び出しについて、照合順序を上書きするもので、例えば次のようにします。

```
SELECT * FROM tbl WHERE a > 'foo' COLLATE "C";
```

とします。後者の場合、COLLATE句が、処理対象と想定している入力演算子の引数に対して付与されることに注意してください。演算子や関数の呼び出しのどの引数に対してCOLLATE句が付与されるかは問題ではありません。演算子や関数により適用される照合順序は対象となる全ての引数を考慮して引き出され、そして明示的に指定されたCOLLATE句がその他の全ての引数に対しての照合順序を上書きするからです。(しかし、複数の引数に対して一致しないCOLLATE句の付与はエラーとなります。詳細は23.2を参照してください)。このため、前述の例と同じ結果を次の様にして取得することができます。

```
SELECT * FROM tbl WHERE a COLLATE "C" > 'foo';
```

ただし、次の例はエラーになります。

```
SELECT * FROM tbl WHERE (a > 'foo') COLLATE "C";
```

>演算子の結果に対して照合順序を適用しようとしますが、>演算子は照合不可能なデータ型であるbooleanとなるからです。

4.2.11. スカラ副問い合わせ

スカラ副問い合わせは、正確に1行1列を返す、括弧内の通常のSELECT問い合わせです（問い合わせの記述方法については第7章を参照してください）。そのSELECT問い合わせは実行され、返される単一の値はその値の前後の評価式で使用されます。1行を超える行や1列を超える列がスカラ副問い合わせ用の問い合わせとして使用された場合はエラーになります（しかし、ある実行時に、副問い合わせが行を返さない場合はエラーとはなりません。そのスカラ結果はNULLとして扱われます）。副問い合わせは、その周りの問い合わせ内の値を参照することができます。その値は副問い合わせの評価時には定数として扱われます。副問い合わせに関する他の式については9.23も参照してください。

例えば、以下は各州の最大都市の人口を検索します。

```
SELECT name, (SELECT max(pop) FROM cities WHERE cities.state = states.name)
    FROM states;
```

4.2.12. 配列コンストラクタ

配列コンストラクタは、メンバー要素に対する値を用いて配列値を構築する式です。単純な配列コンストラクタの構成は、ARRAYキーワード、左大括弧[、(カンマで区切った)配列要素値用の式のリストで、最後に右大括弧]です。以下に例を示します。

```
SELECT ARRAY[1,2,3+4];
  array
---------
 {1,2,7}
(1 row)
```

デフォルトで配列要素型は、メンバ式の型と同じで、UNIONやCASE構文と同じ規則を使用して決定されます（10.5を参照してください）。これを明示的に配列コンストラクタを希望する型にキャストすることで書き換えることができます。例をあげます。

```
SELECT ARRAY[1,2,22.7]::integer[];
  array
----------
 {1,2,23}
(1 row)
```

これはそれぞれの式を配列要素の型に個別にキャストするのと同じ効果があります。キャストについてより多くは4.2.9を参照してください。

多次元配列値は、配列コンストラクタを入れ子にすることで構築できます。内側のコンストラクタではARRAYキーワードは省略可能です。例えば、以下は同じ結果になります。

```
SELECT ARRAY[ARRAY[1,2], ARRAY[3,4]];
     array
---------------
 {{1,2},{3,4}}
(1 row)

SELECT ARRAY[[1,2],[3,4]];
     array
---------------
 {{1,2},{3,4}}
(1 row)
```

多次元配列は長方形配列でなければなりませんので、同一レベルの内部コンストラクタは同一次元の副配列を生成しなければなりません。外部ARRAYコンストラクタに適用される全てのキャストは自動的に全ての内部コンストラクタに伝播します。

多次元配列コンストラクタの要素は、副ARRAY構文だけでなく、適切な種類の配列を生成するものをとることができます。以下に例を示します。

```
CREATE TABLE arr(f1 int[], f2 int[]);

INSERT INTO arr VALUES (ARRAY[[1,2],[3,4]], ARRAY[[5,6],[7,8]]);

SELECT ARRAY[f1, f2, '{{9,10},{11,12}}'::int[]] FROM arr;
                     array
```

```
-----------------------------------------------
 {{{1,2},{3,4}},{{5,6},{7,8}},{{9,10},{11,12}}}
(1 row)
```

空配列を構築できますが、型を所有しない配列を持つことは不可能なので、空配列を望まれる型に明示的にキャストしなければなりません。例をあげます。

```
SELECT ARRAY[]::integer[];
 array
-------
 {}
(1 row)
```

また、副問い合わせの結果から配列を構成することも可能です。この形式の場合、配列コンストラクタはARRAYキーワードの後に括弧（大括弧ではない）で括られた副問い合わせとして記述されます。以下に例を示します。

```
SELECT ARRAY(SELECT oid FROM pg_proc WHERE proname LIKE 'bytea%');
                              array
------------------------------------------------------------------
 {2011,1954,1948,1952,1951,1244,1950,2005,1949,1953,2006,31,2412}
(1 row)

SELECT ARRAY(SELECT ARRAY[i, i*2] FROM generate_series(1,5) AS a(i));
              array
---------------------------------
 {{1,2},{2,4},{3,6},{4,8},{5,10}}
(1 row)
```

副問い合わせは単一の列を返さなければなりません。副問い合わせの出力列が非配列型であれば、その結果である一次元配列は、副問い合わせの出力列と一致する型を要素型とした、副問い合わせの結果内の各行を要素として持ちます。副問い合わせの出力列が配列型であれば、その結果は、同じ型で1つ次元の高い配列になります。この場合、副問い合わせの列はすべて同じ次元の配列とならなければなりません。そうでないと結果が長方形になりません。

ARRAYで構築された配列値の添字は、常に1から始まります。配列についての詳細は8.15を参照してください。

4.2.13. 行コンストラクタ

行コンストラクタは、そのメンバフィールドに対する値を用いて行値（複合値とも呼ばれます）を構築する式です。行コンストラクタは、ROWキーワード、左括弧、行のフィールド値用の0個以上の式（カンマ区切り）、最後に右括弧からなります。以下に例を示します。

```
SELECT ROW(1,2.5,'this is a test');
```

ROWキーワードは、2つ以上の式がリスト内にある場合は省略することができます。

行コンストラクタにはrowvalue.*構文を含めることができます。これは、SELECTリストの最上位レベルで.*構文が使用された時とまったく同様に、行値の要素の列挙に展開されます(8.16.5参照)。たとえば、テーブルtがf1列とf2列を持つ場合、以下は同一です。

```
SELECT ROW(t.*, 42) FROM t;
SELECT ROW(t.f1, t.f2, 42) FROM t;
```

注記

PostgreSQL 8.2より前では、.*構文は行コンストラクタ内では展開されませんでした。ROW(t.*, 42)と記述すると、1つ目のフィールドにもう一つの行値を持つ、2つのフィールドからなる行が作成されました。たいていの場合、新しい動作はより使いやすくなっています。入れ子状の行値という古い動作が必要であれば、内側の行値には.*を使用せずに、たとえばROW(t, 42)と記述してください。

デフォルトでは、ROW式により作成される値は匿名レコード型になります。必要に応じて、名前付きの複合型、つまりテーブルの行型あるいはCREATE TYPE ASで作成された複合型にキャストすることができます。曖昧性を防止するために明示的なキャストが必要となることもあります。以下に例を示します。

```
CREATE TABLE mytable(f1 int, f2 float, f3 text);

CREATE FUNCTION getf1(mytable) RETURNS int AS 'SELECT $1.f1' LANGUAGE SQL;


-- getf1()が1つしか存在しないためキャスト不要。
SELECT getf1(ROW(1,2.5,'this is a test'));
 getf1
-------
     1
(1 row)

CREATE TYPE myrowtype AS (f1 int, f2 text, f3 numeric);

CREATE FUNCTION getf1(myrowtype) RETURNS int AS 'SELECT $1.f1' LANGUAGE SQL;


-- ここでは、どの関数を呼び出すのかを示すためにキャストが必要。
SELECT getf1(ROW(1,2.5,'this is a test'));
ERROR:  function getf1(record) is not unique

SELECT getf1(ROW(1,2.5,'this is a test')::mytable);
 getf1
-------
     1
(1 row)

SELECT getf1(CAST(ROW(11,'this is a test',2.5) AS myrowtype));
```

```
 getf1
-------
    11
(1 row)
```

行コンストラクタは、複合型のテーブル列に格納する複合型の値を構築するため、あるいは複合型のパラメータを受け付ける関数に渡すために使用することができます。また、以下の例のように、2つの行値を比較することも、IS NULLもしくはIS NOT NULLで行を検査することも可能です。

```
SELECT ROW(1,2.5,'this is a test') = ROW(1, 3, 'not the same');


SELECT ROW(table.*) IS NULL FROM table;  -- すべてがNULLの行を検出します。
```

詳細は9.24を参照してください。行コンストラクタは、9.23で説明するように、副問い合わせと一緒に使用することもできます。

4.2.14. 式の評価規則

副式の評価の順序は定義されていません。特に演算子や関数の入力は、必ずしも左から右などの決まった順序で評価されるわけではありません。

さらに、その式の一部を評価しただけで式の結果を決定できる場合には、他の副式がまったく評価されないこともあります。例えば、

```
SELECT true OR somefunc();
```

では、(おそらく)somefunc()は呼び出されないでしょう。以下の場合も同様です。

```
SELECT somefunc() OR true;
```

これは一部のプログラミング言語に見られる、ブーリアン演算子での左から右への「短絡評価」とは異なることに注意してください。

そのため、副次作用がある関数を複雑な式の一部として使用することは推奨されません。特に、WHERE句およびHAVING句で副次作用や評価順に依存するのは危険です。これらの句は、実行計画を作成する過程で頻繁に再処理されるからです。これらの句のブール式(AND/OR/NOTの組み合わせ)は、ブール代数の規則で許されるあらゆる方式で再編成される可能性があります。

評価の順序を強制することが重要であれば、CASE構文(9.18を参照)を使用できます。例えば、次の式はWHERE句で0除算を避ける方法としては信頼性の低いものです。

```
SELECT ... WHERE x > 0 AND y/x > 1.5;
```

しかし、次のようにすれば安全です。

```
SELECT ... WHERE CASE WHEN x > 0 THEN y/x > 1.5 ELSE false END;
```

このような方法で使用されるCASE構文は最適化を妨げるものなので、必要な場合にのみ使用してください。（特に、この例では、y > 1.5*xと代わりに記述することが問題を回避するより優れた方法です。）

しかしながら、CASEはそのような問題に対する万能薬ではありません。上で示したような方法の限界の1つは、定数副式が早く評価されるのを防げないことです。37.7に記すように、IMMUTABLEと印をつけられた関数と演算子は、実行される時ではなく問い合わせが計画される時に評価されるかもしれません。そのため、例えば

```
SELECT CASE WHEN x > 0 THEN x ELSE 1/0 END FROM tab;
```

は、たとえテーブルのすべての行がx > 0であり、実行時にはELSE節に決して入らないとしても、プランナが定数副式を単純化しようとするためにゼロによる除算での失敗という結果に終わるでしょう。

この特別な例は馬鹿げたものに見えるかもしれませんが、定数を含むことが明らかではない関連する場合が関数の中で実行される問い合わせで起こり得ます。関数の引数とローカル変数は計画作成の都合で定数として問い合わせに入れられることがあるからです。例えば、PL/pgSQL関数の中では、IF-THEN-ELSE文を使って危険な計算を保護する方がCASE式の中で入れ子にするよりもずっと安全です。

同種の別の限界は、その中に含まれる集約式の評価をCASEが防げないことです。なぜなら、SELECTリストやHAVING句の別の式が考慮される前に、集約式が計算されるからです。例えば、以下の問い合わせは対策を施しているように見えるにも関わらずゼロ除算エラーになり得ます。

```
SELECT CASE WHEN min(employees) > 0
            THEN avg(expenses / employees)
       END
    FROM departments;
```

min()とavg()集約は入力行すべてに対して同時に計算されますので、もしemployeesがゼロになる行があれば、min()の結果が検査される機会の前にゼロ除算エラーが起こります。代わりに、まずは問題のある入力行が集約関数に渡されないようにするためにWHEREまたはFILTER句を使ってください。

4.3. 関数呼び出し

PostgreSQLでは名前付きパラメータを持つ関数について、位置表記と名前付け表記のいずれでも呼び出すことが可能です。名前付け表記は、パラメータと引数の関連をより明確・確実にするので、多数のパラメータを持つ関数において特に有用です。位置表記の関数呼び出しでは、関数宣言で定義されたのと同じ並び順で、引数を記述します。名前付け表記では、引数と関数パラメータは名前で対応付けられ、引数はどのような並び順で書いても構いません。それぞれの表記で、10.3に書かれているように、関数の引数の型の効果も考慮してください。

どちらの表記でも、関数定義時にデフォルト値を与えられているパラメータについては、呼び出し時に記述される必要はありません。しかしこれは、名前付け表記で特に有用です。なぜなら、パラメータ群の任意の組み合わせを省略できるからです。一方、位置表記のパラメータは右から左へ省略していくことしかできません。

PostgreSQLでは、名前付け表記と位置表記の混在表記もサポートしています。この場合、位置表記のパラメータが最初に記述され、その後に名前付け表記のパラメータが記述されることになります。

本節の例では、次の関数定義を使って、3通りすべての表記方法について説明します。

```
CREATE FUNCTION concat_lower_or_upper(a text, b text, uppercase boolean DEFAULT false)
RETURNS text
AS
$$
 SELECT CASE
        WHEN $3 THEN UPPER($1 || ' ' || $2)
        ELSE LOWER($1 || ' ' || $2)
        END;
$$
LANGUAGE SQL IMMUTABLE STRICT;
```

concat_lower_or_upper関数は、aとbの指定必須となる2つのパラメータを持ちます。加えて、uppercaseというデフォルトがfalseとなっている省略可能なパラメータを一つ持ちます。aとbで入力された文字列が結合され、uppercaseパラメータにより大文字か小文字に変換されます。他のこの関数定義についての詳細は、ここでは重要ではありません。(詳細は第37章を参照して下さい。)

4.3.1. 位置表記の使用

位置表記は、PostgreSQLの引数を関数に渡す伝統的な仕組みです。例を挙げます。

```
SELECT concat_lower_or_upper('Hello', 'World', true);
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)
```

すべての引数を順番通りに指定します。uppercaseがtrueと指定されていますので、結果は大文字です。別の例を示します。

```
SELECT concat_lower_or_upper('Hello', 'World');
 concat_lower_or_upper
-----------------------
 hello world
(1 row)
```

ここではuppercaseパラメータが省略されていますので、そのデフォルト値であるfalseを受け取ることとなり、結果は小文字になります。位置表記では引数がデフォルト値を持つ限り右側から左の方向で、引数を省略することができます。

4.3.2. 名前付け表記の使用

名前付け表記では、各引数の名前は=>を使用し引数の式と分けて指定されます。例を挙げます。

```
SELECT concat_lower_or_upper(a => 'Hello', b => 'World');
 concat_lower_or_upper
-----------------------
 hello world
(1 row)
```

この場合も、uppercase引数が省略されていますので、暗黙的にfalseに設定されます。名前付け表記の使用の利点の1つとして、引数を任意の順序で指定できる点があります。以下に例を示します。

```
SELECT concat_lower_or_upper(a => 'Hello', b => 'World', uppercase => true);
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)

SELECT concat_lower_or_upper(a => 'Hello', uppercase => true, b => 'World');
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)
```

":="に基づく古い文法は後方互換性のためにサポートされます。

```
SELECT concat_lower_or_upper(a := 'Hello', uppercase := true, b := 'World');
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)
```

4.3.3. 混在表記の利用

混在表記は名前付け表記と位置表記を組み合わせたものです。しかし既に述べたように、名前付けされた引数は位置づけされたパラメータより前に記述することはできません。例を挙げます。

```
SELECT concat_lower_or_upper('Hello', 'World', uppercase => true);
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)
```

上記の問い合わせでは、aとbが位置で指定され、uppercaseは名前で指定されています。この例では文書化の目的以外ほとんど意味がありません。デフォルト値が割り当てられた多くのパラメータを持つ、もっと複雑な関数では、名前付けもしくは混在表記により記述量を大きく減らすことができ、かつ、エラーが紛れ込む可能性を抑えることができます。

注記

名前付けと混在呼び出し表記は集約関数の呼び出しでは現在使用できません（が、集約関数がウィンドウ関数として使われる場合には動作します）。

第5章 データ定義

本章では、データを保持するためのデータベース構造を作成する方法を説明します。リレーショナルデータベースでは生データはテーブルに格納されます。したがって、本章ではテーブルの作成と変更の方法や、テーブルにどのようなデータを格納するかを制御するための機能について重点的に解説します。さらに、テーブルをスキーマに編成する方法、およびテーブルへの権限の割り当てについても説明します。そして最後に、継承、テーブルのパーティショニング、ビュー、関数、およびトリガなど、データの格納に影響する機能について簡単に説明します。

5.1. テーブルの基本

リレーショナルデータベースのテーブルは、紙に書く表によく似ています。テーブルは行と列からできています。列の数と順序は固定されており、それぞれの列に名前が付けられています。行の数は可変です。つまり行の数とは、その時点でどれだけのデータが格納されているのかを示すものです。SQLではテーブル内の行の順序は保証されません。テーブルを読み込むと、明示的に並び替えが要求されない限り、行は不特定な順序で返されます。これについては第7章を参照してください。さらに、SQLでは行に固有の識別子が割り当てられないので、テーブル内にまったく同一の行がいくつも存在することがあり得ます。これは、SQLの基礎をなす数学的モデルの帰結ですが、通常は好ましいことではありません。この問題の対処法については、本章で後述します。

それぞれの列にデータ型があります。データ型によって、列に割り当てられる値が制限されます。また、列に格納されているデータに意味が割り当てられ、データを計算に使用できるようになります。例えば、数値型と宣言された列は任意のテキスト文字列は受け付けません。そして、数値型の列に格納されているデータは算術計算に使用できます。これに対して、文字列型と宣言された列はほとんど全ての種類のデータを受け付けます。しかし、文字列の結合といった演算には使用できますが、算術計算には使用できません。

PostgreSQLには、様々なアプリケーションに対応した多数のデータ型の集合が組み込まれています。またユーザが独自のデータ型を定義することも可能です。組み込みデータ型のほとんどにはわかりやすい名前と意味が付けれられているので、詳しい説明はここでは行わず、第8章で行います。よく使用されるデータ型としては、整数を表すinteger、小数も表すことができるnumeric、文字列を表すtext、日付を表すdate、時刻を表すtime、そして日付と時刻の両方を含むtimestampがあります。

テーブルを作成するには、その名の通りCREATE TABLEコマンドを使用します。このコマンドで最低限指定する必要があるのは、新規テーブル名、列名、各列のデータ型です。例を示します。

```
CREATE TABLE my_first_table (
    first_column text,
    second_column integer
);
```

これで2列からなるmy_first_tableという名前のテーブルが作成されます。最初の列の名前はfirst_columnで、そのデータ型はtextです。2番目の列の名前はsecond_columnで、そのデータ型はintegerです。テーブル名および列名は、4.1.1で説明した識別子の構文に従います。型名も通常は識別子ですが、例外もあります。列リストはカンマで区切り、括弧で囲むことに注意してください。

先ほどの例は、説明が目的であるため現実的ではありません。通常、テーブルおよび列の名前は、どのようなデータが格納されているかわかるような名前にします。以下に、より現実的な例を示します。

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric
);
```

(numeric型は小数を格納することができ、金額を扱う場合はこれが一般的です。)

> **ヒント**
>
> 相関するテーブルを数多く作成する場合は、テーブルと列の命名規則を一貫させるのが賢明です。例えば、テーブル名に単数形あるいは複数形どちらの名詞を使用するかという選択肢があります(これは論者によって好みが分かれています)。

テーブルに含めることができる列の数には制限があります。制限は、列の型に応じて250～1600の間となります。しかし、これほど多くの列を使用することは稀ですし、そのような場合は設計に問題があることも多いのです。

必要のないテーブルができた場合は、DROP TABLEコマンドを使用してそのテーブルを削除できます。例を示します。

```
DROP TABLE my_first_table;
DROP TABLE products;
```

存在しないテーブルを削除しようとすると、エラーになります。もっともテーブルが存在するかどうか関係なくスクリプト全体を動作させることができるように、テーブルを作成する前に、エラーメッセージを無視して無条件に削除操作を行うことは、SQLスクリプトファイルではよく行われることです。(この操作を行いたければ、エラーメッセージの出力を防ぐDROP TABLE IF EXISTSという構文を使用することができます。しかし、これは標準SQLではありません。)

既に存在するテーブルを変更する方法については、本章で後述する5.6を参照してください。

これまでに説明したツールを使用して、十分に機能するテーブルを作成できます。本章の残りでは、テーブル定義に機能を追加して、データの整合性、安全性、利便性を確実にする方法について述べていきます。この時点でテーブルにデータを入力したければ、本章の残りを後回しにして第6章に進んでも構いません。

5.2. デフォルト値

列にはデフォルトの値を割り当てることができます。新しく作成された行のいくつかの列に値が指定されていない場合、そうした空欄にはそれぞれの列のデフォルト値が入ります。データ操作コマンドを使用して、列を(どのような値かを把握する必要なく)デフォルト値に設定するように明示的に要求することもできます。(データ操作コマンドの詳細については第6章を参照してください。)

明示的に宣言されたデフォルト値がない場合は、デフォルト値はNULL値になります。NULL値は不明のデータを表すものとみなすことができるので、通常はこの方法で問題ありません。

テーブル定義では、デフォルト値は列データ型の後に列挙されています。例を示します。

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric DEFAULT 9.99
);
```

デフォルト値を式にすることが可能で、それはデフォルト値が挿入される時はいつでも(テーブルが作成されたときではありません)評価されます。よくある例として、timestamp列が挿入時の時刻に設定されるように、その列はデフォルトのCURRENT_TIMESTAMPを持つことができます。もう1つの例としては、各行に「通番」を割り振る場合です。PostgreSQLでは、典型的に以下のように記述することにより生成されます。

```
CREATE TABLE products (
    product_no integer DEFAULT nextval('products_product_no_seq'),
    ...
);
```

ここで、nextval()関数が、シーケンスオブジェクトから連続した値を生成します(9.17を参照してください)。これは非常によく使われるやり方なので、以下のような特別な短縮記法が用意されています。

```
CREATE TABLE products (
    product_no SERIAL,
    ...
);
```

省略形であるSERIALは8.1.4で詳しく述べられています。

5.3. 生成列

生成列は常に他の列から計算される特別な列です。ですから、これは列におけるテーブルに対するビューのようなものです。生成列には格納と仮想の2種類があります。格納生成列はそれが書かれた(挿入または更新)時に計算され、あたかも通常の列のようにストレージが割当てられます。仮想列にはストレージは割り当てられず、列が読み出された時に計算されます。つまり、仮想生成列はビューに似ており、格納生成列はマテリアライズドビューに似ています。(常に自動的に更新される点は除きます。) 今の所PostgreSQLは格納生成列のみを実装しています。

生成列を作るには、CREATE TABLEでGENERATED ALWAYS AS節を使ってください。例を示します。

```
CREATE TABLE people (
    ...,
    height_cm numeric,
```

```
    height_in numeric GENERATED ALWAYS AS (height_cm / 2.54) STORED
);
```

種類を格納生成列として選択するためにキーワードSTOREDを選択する必要があります。より詳しくはCREATE TABLEをご覧ください。

生成列には直接書き込みができません。INSERTあるいはUPDATEコマンドでは値を生成列には指定できませんが、キーワードDEFAULTが指定できます。

デフォルトを備えた列と生成列の違いを考えてみましょう。列のデフォルトは、他に値が指定されないときに、最初に行が挿入された時に一度だけ評価されます。生成列は、行が変更された時に常に更新され、上書きはできません。デフォルトを備えた列はテーブルの他の列を参照することはできませんが、生成式は通常それを行います。デフォルトを備えた列は揮発性の関数、たとえばrandom()や現在時刻を参照する関数を使用できますが、これは生成列では許されていません。

生成列の定義と生成列を伴うテーブルには以下の制限が適用されます。

- 生成式は不変関数のみが使用でき、副問合せ、あるいは現在の行以外へのいかなる参照も使用できません。
- 生成式はほかの生成列を参照できません。
- 生成式はtableoid以外のシステム列を参照できません。
- 生成列は列デフォルトも識別定義も持てません。
- 生成列はパーティションキーの一部にはなれません。
- 外部テーブルは生成列を持つことができます。詳細はCREATE FOREIGN TABLEをご覧ください。
- 継承の場合：
  - 親列が生成列なら、子列もまた同じ式を用いた生成列でなければなりません。子列の定義ではGENERATED節は親列からコピーされるので、指定しないでください。
  - 多重継承では、一つの親列が生成列なら、すべての親列は同じ式による生成列でなければなりません。
  - 親列が生成列でなければ、子列は生成列として定義されるかもしれませんし、されないかもしれません。

生成列の利用の際には以下の追加の考慮が必要です。

- 生成列は元になる基底列とは別にアクセス権限を維持します。ですから、ある特定のロールが生成列を読み出しつつも、元になる基底列からは読み出さないように調整することが可能です。
- 概念的には、生成列はBEFOREトリガが走った後に更新されます。ですから、BEFOREトリガの中で基底列に加えられた変更は生成列に反映されます。しかし逆に生成列をBEFOREトリガの中でアクセスすることは許されません。

5.4. 制約

データ型は、テーブルに格納するデータの種類を限定するための方法です。しかし、多くのアプリケーションでは、型が提供する制約では精密さに欠けます。例えば、製品の価格が入る列には、おそらく正数のみを受け入れるようにする必要があります。しかし、正数のみを受け入れるという標準のデータ型はありません。また、他の列や行に関連して列データを制約したい場合もあります。例えば、製品の情報が入っているテーブルでは、1つの製品番号についての行が2行以上あってはなりません。

このような問題を解決するため、SQLでは列およびテーブルに対する制約を定義することができます。制約によってテーブル内のデータを自由に制御することができます。制約に違反するデータを列に格納しようとすると、エラーとなります。このことは、デフォルト値として定義された値を格納する場合にも適用されます。

5.4.1. 検査制約

検査制約は最も汎用的な制約の種類です。これを使用して、特定の列の値が論理値の式を満たす（真の値）ように指定できます。例えば、製品価格を必ず正数にするには以下のようにします。

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CHECK (price > 0)
);
```

このように、制約の定義はデフォルト値の定義と同様に、データ型の後にきます。デフォルト値と制約は任意の順序で列挙できます。検査制約はCHECKキーワードの後に続く括弧で囲まれた式で構成されます。検査制約式には、制約される列を含む必要があります。そうしないと、制約はあまり意味のないものになります。

制約に個別に名前を付けることもできます。名前を付けることで、エラーメッセージがわかりやすくなりますし、変更したい制約を参照できるようになります。構文は以下の通りです。

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CONSTRAINT positive_price CHECK (price > 0)
);
```

上記のように、名前付き制約の指定はCONSTRAINTキーワードで始め、これに識別子、制約定義と続きます。（この方法で制約名を指定しない場合は、システムにより名前が付けられます。）

検査制約では複数の列を参照することもできます。例えば、通常価格と割引価格を格納する場合に、必ず割引価格が通常価格よりも低くなるようにしたいとします。

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CHECK (price > 0),
    discounted_price numeric CHECK (discounted_price > 0),
```

```
    CHECK (price > discounted_price)
);
```

最初の2つの制約は上で説明した通りです。3つ目の制約では新しい構文を使っています。これは特定の列に付加されるのではなく、カンマで区切られた列リスト内の別個の項目として現れます。列定義およびこれらの制約定義は、任意の順序で列挙することができます。

最初の2つの制約を列制約と言います。これに対し、3つ目の制約は列定義とは別個に書かれるので、テーブル制約と言います。列制約をテーブル制約として書くことはできますが、その逆はできる場合とできない場合があります。なぜなら列制約は、制約に関連付けられている列のみを参照するためです。（PostgreSQLはこの規則を強制しません。しかし、作成したテーブル定義を他のデータベースシステムでも動作させたい場合はこの規則に従ってください。）上の例は、以下のように書くこともできます。

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric,
    CHECK (price > 0),
    discounted_price numeric,
    CHECK (discounted_price > 0),
    CHECK (price > discounted_price)
);
```

あるいは、次のようにもできます。

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CHECK (price > 0),
    discounted_price numeric,
    CHECK (discounted_price > 0 AND price > discounted_price)
);
```

どのようにするかは好みの問題です。

列制約と同様に、テーブル制約に名前を割り当てることができます。

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric,
    CHECK (price > 0),
    discounted_price numeric,
    CHECK (discounted_price > 0),
    CONSTRAINT valid_discount CHECK (price > discounted_price)
);
```

検査制約では、検査式が真またはNULL値と評価された場合に、条件が満たされることに注意して下さい。ほとんどの式は、演算項目に一つでもNULLがあればNULLと評価されるので、検査制約では制約対象の列にNULL値が入るのを防げません。列がNULL値を含まないようにするために、次節で説明する非NULL制約を使用することができます。

注記

PostgreSQLは、検査対象の新しい行もしくは更新対象行以外のテーブルデータを参照するCHECK制約はサポートしていません。このルールに違反するCHECK制約は単純なテストでは動いたように見えますが、(関連する他の行が後で更新されたことにより)データベースがその制約条件が偽になるような状態にならないことを保証できません。これによってデータベースのダンプと再ロードの失敗が引き起こされるでしょう。最終的なデータベース状態が制約に対して一貫した状態であったとしても、制約を満たす順で行がロードされないことにより再ロードは失敗することがあります。可能ならばUNIQUE、EXCLUDE、FOREIGN KEY制約を使って行あるいはテーブルをまたがる制約を表現してください。

常に一貫性の保障を維持するのではなく、行挿入の際に一回だけの行の検査が必要なら、その実装のためにカスタムトリガが利用できます。(pg_dumpはデータの再ロード後までトリガを再インストールせず、ダンプ/再ロード中は検査が強制されないため、この方法でダンプ/再ロード問題を回避できます。)

注記

PostgreSQLはCHECK制約の条件は不変であると仮定します。つまり同じ入力行に対して常に同じ結果が返るということです。この仮定によりCHECK制約が挿入あるいは更新時にのみ検査され、他のときには検査されないことが正当化されます。(他のテーブルデータを参照しないことによる上述の警告はこの制限の本当に特別な場合です。)

この仮定に反する一般的な例は、CHECK式でユーザ定義関数を参照し、その関数の振る舞いを変更することです。PostgreSQLはこれを禁止はしませんが、今やCHECK制約に違反する行がテーブル中に存在することを通知しません。これによって後でデータベースのダンプと再ロードの失敗を引き起こすでしょう。そのような変更に対処するおすすめの方法は、(ALTER TABLEを使って)制約を削除し、関数定義を調整し、そして制約を再度追加して、それによってテーブル全体の行に対して再チェックを行うことです。

5.4.2. 非NULL制約

非NULL制約は単純に、列がNULL値を取らないことを指定します。構文の例は以下の通りです。

```
CREATE TABLE products (
    product_no integer NOT NULL,
    name text NOT NULL,
    price numeric
```

```
);
```

非NULL制約は常に列制約として記述されます。非NULL制約はCHECK (column_name IS NOT NULL)という検査制約と機能的には同等ですが、PostgreSQLでは、明示的に非NULL制約を作成する方がより効果的です。このように作成された非NULL制約に明示的な名前を付けられないのが欠点です。

もちろん、1つの列に複数の制約を適用することもできます。そのためには、次々と制約を書いていくだけです。

```
CREATE TABLE products (
    product_no integer NOT NULL,
    name text NOT NULL,
    price numeric NOT NULL CHECK (price > 0)
);
```

順序は関係ありません。書かれた順序と検査される順序は必ずしも同じではありません。

NOT NULL制約に対し、逆のパターンであるNULL制約があります。これは、列がNULLでなければならないということではありません。そのような制約は意味がありません。この制約は、列がNULLであってもよいというデフォルトの振舞いを選択するだけのものです。NULL制約は標準SQLには存在しませんので、移植予定のアプリケーションでは使用すべきではありません。(これは、PostgreSQLと他の一部のデータベースシステムとの互換性のために追加された機能に過ぎません。) もっとも、スクリプトファイルでの制約の切り替えが簡単であるという理由でこの機能を歓迎するユーザもいます。例えば、最初に

```
CREATE TABLE products (
    product_no integer NULL,
    name text NULL,
    price numeric NULL
);
```

と書いてから、必要な場所にNOTキーワードを挿入することができます。

ヒント

ほとんどのデータベース設計において、列の多くをNOT NULLとマークする必要があります。

5.4.3. 一意性制約

一意性制約によって、列あるいは列のグループに含まれるデータが、テーブル内の全ての行で一意であることを確実にします。列制約の場合の構文は以下の通りです。

```
CREATE TABLE products (
    product_no integer UNIQUE,
    name text,
```

```
    price numeric
);
```

また、テーブル制約の場合の構文は

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric,
    UNIQUE (product_no)
);
```

となります。

列の集合に対して一意性制約を定義するには、列名をカンマで区切り、表制約として記述します。

```
CREATE TABLE example (
    a integer,
    b integer,
    c integer,
    UNIQUE (a, c)
);
```

これは、指定された列の値の組み合わせがテーブル全体で一意であることを指定しています。しかし、列の片方が一意である必要はありません（通常一意ではありません）。

一意性制約には、通常の方法で名前を割り当てることもできます。

```
CREATE TABLE products (
    product_no integer CONSTRAINT must_be_different UNIQUE,
    name text,
    price numeric
);
```

一意性制約を追加すると、制約で指定された列または列のグループに対して一意的なBツリーのインデックスが自動的に作られます。一部の行だけに適用される一意性の制限を一意性制約として作成することはできませんが、そのような制限を一意な部分インデックスを作成することで実現することは可能です。

一般に、制約の対象となるすべての列について同じ値を持つ行が、テーブル内に2行以上ある場合は、一意性制約違反になります。しかし、この比較では2つのNULL値は決して等価とはみなされません。つまり、一意性制約があったとしても、制約対象の列の少なくとも1つにNULL値を持つ行を複数格納することができるということです。この振舞いは標準SQLに準拠していますが、この規則に従わないSQLデータベースがあることを聞いたことがあります。ですから、移植する予定のアプリケーションを開発する際には注意してください。

5.4.4. 主キー

主キー制約は、列または列のグループがテーブル内の行を一意に識別するものとして利用できることを意味します。これには値が一意で、かつNULLでないことが必要となります。つまり、次の2つのテーブル定義は同じデータを受け入れます。

```
CREATE TABLE products (
    product_no integer UNIQUE NOT NULL,
    name text,
    price numeric
);
```

```
CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
    price numeric
);
```

主キーも複数の列に渡ることがあり、その構文は一意性制約に似ています。

```
CREATE TABLE example (
    a integer,
    b integer,
    c integer,
    PRIMARY KEY (a, c)
);
```

主キーを追加すると、主キーで指定された列または列のグループに対して一意的なBツリーのインデックスが自動的に作られます。また、その列についてNOT NULLの印が強制されます。

1つのテーブルは最大1つの主キーを持つことができます。(一意性制約および非NULL制約には個数の制限はありません。機能的にはほとんど同じものですが、主キーとして識別される制約は1つのみです。)リレーショナルデータベース理論では、全てのテーブルに主キーが1つ必要とされています。この規則はPostgreSQLでは強制されませんが、たいていの場合はこれに従うことが推奨されます。

主キーは文書化、および、クライアントアプリケーションの両方の面で役に立ちます。例えば、行値の変更が可能なGUIアプリケーションが行を一意的に特定するためには、おそらくテーブルの主キーを知る必要があります。他にも主キーが宣言されているときにデータベースシステムがそれを利用する場面がいくつかあります。例えば、外部キーがそのテーブルを参照するとき、主キーがデフォルトの対象列となります。

5.4.5. 外部キー

外部キー制約は、列(または列のグループ)の値が、他のテーブルの行の値と一致しなければならないことを指定します。これによって関連する2つのテーブルの参照整合性が維持されます。

これでまで何度か例に使用したproductsテーブルについて考えてみます。

```
CREATE TABLE products (
```

```
    product_no integer PRIMARY KEY,
    name text,
    price numeric
);
```

また、これらの製品に対する注文を格納するテーブルも作成済みだとしましょう。この注文のordersテーブルには実際に存在する製品の注文のみを格納したいと思っています。そこで、productsテーブルを参照するordersテーブルに外部キー制約を定義します。

```
CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    product_no integer REFERENCES products (product_no),
    quantity integer
);
```

これで、productsテーブルに存在しない非NULLのproduct_no項目を使用して注文を作成することはできなくなります。

このような場合に、ordersテーブルのことを*参照*テーブル、productテーブルのことを*被参照*テーブルと呼びます。同様に、参照列と被参照列もあります。

上記のコマンドは、次のように短縮することもできます。

```
CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    product_no integer REFERENCES products,
    quantity integer
);
```

列リストがないため、被参照テーブルの主キーが被参照列として使用されます。

外部キーでも、列のグループを制約したり参照したりすることもできます。これもまた、テーブル制約の形式で記述する必要があります。以下は、説明のための非現実的な例です。

```
CREATE TABLE t1 (
  a integer PRIMARY KEY,
  b integer,
  c integer,
  FOREIGN KEY (b, c) REFERENCES other_table (c1, c2)
);
```

もちろん、制約される列数および型は、被参照列の数および型と一致しなければなりません。

外部キー制約には、通常の方法で名前を割り当てることもできます。

テーブルは複数の外部キー制約を持つことができます。このことはテーブル間の多対多関係を実装するために使用されます。例えば、製品と注文に関するそれぞれのテーブルがある場合に、複数の製品にまたがる注文を可能にしたいとします（上の例の構造では不可能です）。この場合、次のテーブル構造を使用できます。

```
CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
    price numeric
);

CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    shipping_address text,
    ...
);

CREATE TABLE order_items (
    product_no integer REFERENCES products,
    order_id integer REFERENCES orders,
    quantity integer,
    PRIMARY KEY (product_no, order_id)
);
```

最後のテーブルで、主キーと外部キーが重なっていることに注目してください。

外部キーが製品に関連付けられていない注文の作成を許可しないことは、既に説明した通りです。しかし、ある注文で参照していた製品が、注文後に削除されたらどうなるでしょう。SQLではこのような場合も扱うことができます。直感的に、いくつかのオプションが考えられます。

- 参照される製品の削除を許可しない
- 注文も一緒に削除する
- 他にもありますか？

具体例として、上の例の多対多関係に次のポリシーを実装してみましょう。(order_itemsによって)注文で参照されたままの製品を削除しようしても、この操作を行えないようにします。注文が削除されると、注文項目も削除されます。

```
CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
    price numeric
);

CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    shipping_address text,
    ...
);

CREATE TABLE order_items (
```

```
    product_no integer REFERENCES products ON DELETE RESTRICT,
    order_id integer REFERENCES orders ON DELETE CASCADE,
    quantity integer,
    PRIMARY KEY (product_no, order_id)
);
```

削除の制限およびカスケードという2つは、最も一般的なオプションです。RESTRICTは、被参照行が削除されるのを防ぎます。NO ACTIONは、制約が検査された時に参照行がまだ存在していた場合に、エラーとなることを意味しています。これは、何も指定しない場合のデフォルトの振舞いとなります（これらの本質的な違いは、NO ACTIONでは検査をトランザクション中で後回しにすることができるのに対し、RESTRICTでは後回しにできないということです）。CASCADEは被参照行が削除された時、それを参照する行も同様に削除されることを指定します。他にも2つのオプション、SET NULLとSET DEFAULTがあります。これらは、被参照行が削除された際に、参照行の参照列がそれぞれNULLか各列のデフォルト値に設定されるようになります。これらは制約を守ることを免除することではない、ということに注意してください。例えば、動作にSET DEFAULTを指定したとしても、デフォルト値が外部キー制約を満たさない場合には操作は失敗します。

ON DELETEに似たもので、被参照列が変更（更新）された時に呼び出されるON UPDATEもあります。これらが行えるアクションは同じです。この場合、CASCADEは被参照列の更新後の値が参照行にコピーされることを意味します。

通常、参照行はその参照列のいずれかがnullの場合は外部キー制約を満たす必要がありません。もしMATCH FULLが外部キー宣言に追加された場合、その参照列の全てがnullの場合にのみ参照行は制約を満たすことから逃れることができます(つまりnullと非nullの組み合わせはMATCH FULL制約に違反することが保証されます)。もし参照行が外部キー制約を満たさない可能性を排除したい場合は、参照列をNOT NULLとして宣言してください。

外部キーは主キーであるかまたは一意性制約を構成する列を参照しなければなりません。これは、被参照列は常に(主キーまたは一意性制約の基礎となる)インデックスを持つことを意味します。このため、参照行に一致する行があるかどうかのチェックは効率的です。被参照テーブルからの行のDELETEや被参照行のUPDATEは、古い値と一致する行に対して参照テーブルのスキャンが必要となるので、参照行にもインデックスを付けるのは大抵は良い考えです。これは常に必要という訳ではなく、また、インデックスの方法には多くの選択肢がありますので、外部キー制約の宣言では参照列のインデックスが自動的に作られるということはありません。

データの更新および削除について詳しくは、第6章を参照してください。また、CREATE TABLEのリファレンス文書にある外部キー制約構文の説明も参照してください。

5.4.6. 排他制約

排他制約によって、2つの行に関して指定された列もしくは式を指定された演算子を利用して比較した場合に、少なくとも演算子の比較の1つが偽もしくはnullを返すことを確実にします。構文は以下の通りです。

```
CREATE TABLE circles (
    c circle,
    EXCLUDE USING gist (c WITH &&)
);
```

詳細はCREATE TABLE ... CONSTRAINT ... EXCLUDEを参照して下さい。

排他制約を追加すると、制約宣言で指定された種類のインデックスが自動的に作られます。

5.5. システム列

全てのテーブルには、システムによって暗黙的に定義されたシステム列がいくつかあります。そのため、システム列の名前はユーザ定義列の名前として使うことはできません。(これらの制約は名前がキーワードであるかどうかとは関係ありません。つまり、名前を引用符で囲んでもこの制約を回避することはできません。) システム列については、あまり意識する必要はありません。これらが存在することを知っていれば十分です。

tableoid

この行を含むテーブルのOIDです。この列は特に、継承階層からの選択問い合わせでは便利です(5.10を参照してください)。この列がないと、どのテーブルからその行が来たのかわかりにくいからです。tableoidをpg_classのoid列に結合することでテーブル名を得ることができます。

xmin

この行バージョンの挿入トランザクションの識別情報(トランザクションID)です。(行バージョンとは、行の個別の状態です。行が更新される度に、同一の論理的な行に対する新しい行バージョンが作成されます。)

cmin

挿入トランザクション内の(0から始まる)コマンド識別子です。

xmax

削除トランザクションの識別情報(トランザクションID)です。削除されていない行バージョンではゼロです。可視の行バージョンでこの列が非ゼロの場合があります。これは通常、削除トランザクションがまだコミットされていないこと、または、削除の試行がロールバックされたことを意味しています。

cmax

削除トランザクション内のコマンド識別子、もしくはゼロです。

ctid

テーブル内における、行バージョンの物理的位置を表します。ctidは行バージョンを素早く見つけるために使うことができますが、行のctidはVACUUM FULLにより更新あるいは移動させられると変わります。したがって、ctidは長期の行識別子としては使えません。論理行を識別するためには、主キーを使うべきです。

トランザクション識別子も32ビット量です。長期間使用するデータベースでは、トランザクションIDが一周してしまう可能性があります。これは、適切な保守作業を行うことで、致命的な問題にはなりません。詳細は第24章を参照してください。しかし、長期(10億トランザクション以上)にわたってトランザクションIDの一意性に依存することは賢明ではありません。

コマンド識別子もまた、32ビット量です。このため、単一トランザクション内のコマンド数には2^{32}（40億）個までという制限が発生します。実際、この制限は問題にはなりません。これはSQLコマンド数に対する制限であり、処理される行数に対する制限ではないことに注意してください。また、データベースの内容を実際に変更するコマンドのみがコマンド識別子を消費します。

5.6. テーブルの変更

テーブルの作成後に間違いに気付いたり、あるいはアプリケーションの要件が変わったりした場合には、テーブルをいったん削除して再度作成することができます。しかし、テーブルにデータを入力済みの場合、あるいはそのテーブルが他のデータベースオブジェクト（例えば外部キー制約）によって参照されている場合、これは良い方法ではありません。そのため、PostgreSQL では既存のテーブルに変更を加えるための一連のコマンドが用意されています。テーブル内のデータを変更するという概念ではないことに注意してください。ここでは、テーブルの定義や構造を変更することに焦点を合わせます。

次のことができます。

- 列の追加
- 列の削除
- 制約の追加
- 制約の削除
- デフォルト値の変更
- 列のデータ型の変更
- 列名の変更
- テーブル名の変更

これらの操作は全てALTER TABLEコマンド（本節の説明範囲を超えますので詳細はこちらを参照してください）を使用して行うことができます。

5.6.1. 列の追加

列を追加するには、次のようなコマンドを使用します。

```
ALTER TABLE products ADD COLUMN description text;
```

新しい列にはデフォルト値が初期値として入ります（DEFAULT句を指定しない場合はNULL値が入ります）。

> **ヒント**
>
> PostgreSQL 11から、定数のデフォルト値の列を追加するためにテーブルの各行がALTER TABLE実行時に更新される必要はもうありません。その代わりに、デフォルト値は次回にその行にアクセスされた場合に返され、テーブルが書き換えられた際に適用されるため、ALTER TABLEは巨大なテーブルでも非常に高速になります。
>
> しかしながら、もしデフォルト値に揮発性（例えば、clock_timestamp()）がある場合、各行はALTER TABLE実行時に計算した値に更新される必要があります。潜在的に長時間の更新作業を避けるため、

> 特に列を主にデフォルト以外の値でとにかく埋めたい場合、デフォルトのない列を追加しUPDATEを使用して正しい値を挿入することが望ましいかもしれません。その上で、後述するように期待するデフォルトを追加してください。

次の構文を使用すると、列の制約も同時に定義することができます。

```
ALTER TABLE products ADD COLUMN description text CHECK (description <> '');
```

実際にはCREATE TABLE内の列の記述に使用されている全てのオプションが、ここで使用できます。ただしデフォルト値は与えられている制約を満足するものでなくてはならないことに注意してください。満足しない場合はADDが失敗します。一方で、新規の列に正しく値を入れた後で制約を追加することができます（下記参照）。

5.6.2. 列の削除

列を削除するには、次のようなコマンドを使用します。

```
ALTER TABLE products DROP COLUMN description;
```

列内にある、どんなデータであれ消去します。またその列に関連するテーブルの制約も消去されます。しかし、その列が他のテーブルの外部キー制約として参照されている場合は、PostgreSQLは暗黙のうちに制約を消去したりはしません。CASCADEを追加することにより列に依存する全てを消去することを許可できます。

```
ALTER TABLE products DROP COLUMN description CASCADE;
```

この背後にある一般的な仕組みに関する説明については5.14を参照してください。

5.6.3. 制約の追加

制約を追加するには、テーブル制約の構文が使用されます。

```
ALTER TABLE products ADD CHECK (name <> '');
ALTER TABLE products ADD CONSTRAINT some_name UNIQUE (product_no);
ALTER TABLE products ADD FOREIGN KEY (product_group_id) REFERENCES product_groups;
```

非NULL制約はテーブル制約として記述できないので、追加するには次の構文を使用します。

```
ALTER TABLE products ALTER COLUMN product_no SET NOT NULL;
```

制約は即座に検査されますので、制約を追加する前にテーブル内のデータがこれに従っている必要があります。

5.6.4. 制約の削除

制約を削除するには、その制約名を知る必要があります。自分で名前を付けた場合は簡単です。しかし、自分で名前を付けていない場合はシステム生成の名前が割り当てられているので、それを調べなくてはなりません。それにはpsqlの\d tablenameコマンドを使用すると便利です。他のインタフェースにもテーブルの詳細を調べる方法があるかもしれません。制約名がわかったら、次のコマンドで制約を削除できます。

```
ALTER TABLE products DROP CONSTRAINT some_name;
```

（自動生成された$2といった制約名を扱う場合は、有効な識別子となるように二重引用符で括る必要があることを忘れないでください。）

列の削除の場合と同じく、何か他のものが依存している制約を削除する場合にはCASCADEを付ける必要があります。例えば、外部キー制約は、参照されている列の一意または主キー制約に依存しています。

上記は、非NULL制約以外の全ての制約型に適用できます。非NULL制約を削除するには、次のようにします。

```
ALTER TABLE products ALTER COLUMN product_no DROP NOT NULL;
```

（非NULL制約には名前がないことを想起してください。）

5.6.5. 列のデフォルト値の変更

列に新しいデフォルトを設定するには、以下のようなコマンドを使用します。

```
ALTER TABLE products ALTER COLUMN price SET DEFAULT 7.77;
```

これはテーブル内の既存の行には何も影響を与えないことに注意してください。これは将来のINSERTコマンドのために単純にデフォルトを変えるだけです。

デフォルト値を削除するには次のようにします。

```
ALTER TABLE products ALTER COLUMN price DROP DEFAULT;
```

これは、デフォルトをNULLに設定することと同等です。そのため、定義されていないデフォルト値を削除してもエラーにはなりません。なぜなら NULL値が暗黙的にデフォルトとなっているからです。

5.6.6. 列のデータ型の変更

列を異なるデータ型に変換するには以下のようなコマンドを使用してください。

```
ALTER TABLE products ALTER COLUMN price TYPE numeric(10,2);
```

これは、その列の既存の項目が新しい型に暗黙的キャストにより変換できる場合にのみ成功します。より複雑な変換が必要な場合、古い値から新しい値をどのように計算するかを指定するUSING句を付けることができます。

PostgreSQLは、(もしあれば)列のデフォルト値を新しい型に、同時に、その列に関連する全ての制約も新しい型に変換しようとします。しかし、こうした変換は失敗するかもしれませんし、予想を超えた結果になってしまうかもしれません。型を変更する前にその列に関する制約を全て削除し、後で適切に変更した制約を付け直すことが最善な場合がよくあります。

5.6.7. 列名の変更

列名を変更するには、次のようにします。

```
ALTER TABLE products RENAME COLUMN product_no TO product_number;
```

5.6.8. テーブル名の変更

テーブル名を変更するには、次のようにします。

```
ALTER TABLE products RENAME TO items;
```

5.7. 権限

オブジェクトが作成されると、所有者が割り当てられます。通常、所有者は作成する文を実行したロールです。ほとんどの種類のオブジェクトについて、初期状態では所有者(またはスーパーユーザ)だけがそのオブジェクトを使用できます。他のユーザがこのオブジェクトを使用するには、権限が付与されていなければなりません。

権限にはいくつかの種類があります。すなわちSELECT、INSERT、UPDATE、DELETE、TRUNCATE、REFERENCES、TRIGGER、CREATE、CONNECT、TEMPORARY、EXECUTE、USAGEです。特定のオブジェクトに適用可能な権限は、オブジェクトの型(テーブル、関数など)により変わります。これらの権限の詳細な意味を以下に示します。以降の節および章でもこれらの権限の使用方法についての説明があります。

オブジェクトの変更や削除の権限は所有者に固有のもので、それ自体を許可したり取り消したりはできません (しかし、すべての権限同様、その権限を所有者のロールのメンバーが継承することはできます。21.3をご覧ください)。

たとえば次のように、オブジェクトに対する適切な種類のALTERコマンドにより、あるオブジェクトに新しい所有者を割り当てることができます。

```
ALTER TABLE table_name OWNER TO new_owner;
```

スーパーユーザはいつでも所有者を変更できます。通常のロールは、対象オブジェクトの現在の所有者(または所有者ロールのメンバー)であり、かつ新しい所有者ロールのメンバーである場合に限り、所有者を変更できます。

権限を割り当てるには、GRANTコマンドを使用します。例えば、joeという既存のロールとaccountsという既存のテーブルがある場合、このテーブルを更新する権限を付与するには以下のようにします。

```
GRANT UPDATE ON accounts TO joe;
```

特定の権限名を指定する代わりにALLを指定すると、その種類のオブジェクトに関連する全ての権限が付与されます。

システム内の全てのロールに権限を付与するには、特別な「ロール」名であるPUBLICを使用することができます。また、「グループ」ロールを使用すれば、データベース内に多くのユーザが存在する場合に権限の管理が簡単になります。詳細は第21章を参照してください。

以前与えられた権限を取り消す(revoke)には、それに相応しい名前のREVOKEコマンドを使用します。

```
REVOKE ALL ON accounts FROM PUBLIC;
```

普通はオブジェクトの所有者（またはスーパーユーザ）だけが、オブジェクトにおける権限の付与や剥奪ができます。しかし「with grant option」を付けることで、権限を与えられたユーザが、所有者と同様に他のユーザに権限を付与することが可能になります。もし後になってグラントオプションが剥奪されると、剥奪されたユーザから（直接もしくは権限付与の連鎖により）権限を与えられていたユーザはすべて、その権限が剥奪されます。詳細は、GRANTとREVOKEを参照してください。

オブジェクトの所有者は、所有する通常の権限を削除することを選択できます。たとえば、他のものと同様、自身のためにテーブルを読み取り専用にできます。しかし、所有者は常にすべての付与オプションを持つものとして扱われます。ですから、いつでも自身の権限を再び付与することができます。

可能な権限は以下です。

SELECT

テーブル、ビュー、マテリアライズドビュー、あるいはそれ以外のテーブルのように見えるオブジェクトに対してSELECTをある列、あるいは指定した列（複数可）に許可します。また、COPYの利用を許可します。この権限はUPDATEあるいはDELETEにおいて既存の列を参照する場合にも必要になります。シーケンスにおいてはこの権限はcurrval関数の使用を許可します。ラージオブジェクトにおいてはこの権限はオブジェクトの読み出しを許可します。

INSERT

テーブル、ビューなどに新しい行をINSERTすることを許可します。特定の列だけをINSERTコマンドで指定したい場合に、それらの列に許可することができます。（したがって他の列にはデフォルトが設定されます）。COPY FROMを利用することもできます。

UPDATE

テーブル、ビューなどの列をUPDATEすることを許可します。（実用的には、簡単ではないUPDATEコマンドにはSELECT権限も必要になります。どの行を更新するかを決定したり、列に対して新しい値を計算するためにテーブルの列を参照しなければならないからです。）SELECT ... FOR UPDATEとSELECT ... FOR SHAREはSELECT権限に加えて更にこの権限が必要になります。シーケンスではこの権限はnextvalとsetval関数の利用を許可します。ラージオブジェクトではこの権限はオブジェクトへの書き込みあるいは切り詰めを行うことを許可します。

DELETE

テーブル、ビューなどの列をDELETEすることを許可します。(実用的には、簡単ではないDELETEコマンドにはSELECT権限も必要になります。どの行を削除するかを決定するためにテーブルの列を参照しなければならないからです。)

TRUNCATE

テーブルあるいはビューのTRUNCATEを許可します。

REFERENCES

テーブルあるいはテーブルの特定の列を参照する外部キー制約を作ることを許可します。

TRIGGER

テーブルあるいはビューにトリガを作ることを許可します。

CREATE

データベースに対して、データベース内に新しいスキーマとパブリケーションを作ること、信頼できる拡張をデータベース内に作成することを許可します。

スキーマに対して、スキーマ内に新しいオブジェクトを作ることを許可します。既存のオブジェクトの名前を変えるには、オブジェクトを所有するとともにそのオブジェクトを含むスキーマに対してこの権限を持っていなければなりません。

テーブル空間に対しては、そのテーブル空間内にテーブル、インデックス、一時ファイルを作ることを許可し、そのテーブル空間をデフォルトのテーブル空間として持つデータベースを作ることを許可します。

この権限を剥奪しても既存のオブジェクトの存在、あるいはその配置を変更しないことに注意してください。

CONNECT

権限を与えられた者がデータベースに接続することを許可します。(pg_hba.confが課す制限の検査に加えて)この権限は接続の開始時に検査されます。

TEMPORARY

データベース使用中に一時テーブルを作成することを許可します。

EXECUTE

関数上に実装された演算子を含めて関数あるいはプロシージャの呼び出しを許可します。これは関数とプロシージャに適用される唯一のタイプの権限です。

USAGE

手続き言語に対して、言語内で関数を作るために言語を使用することを許可します。これは手続き言語に適用される唯一のタイプの権限です。

スキーマに対しては、(オブジェクト自身の権限要件が満たされているものと仮定した上で)スキーマ内に含まれるオブジェクトへのアクセスを許可します。本質的に、これは権限を授与されたものがスキーマ内のオブジェクトを「検査」することを許可します。この許可がなくても依然としてオブジェクト名を見る

ことを可能です。たとえば、システムカタログを問い合わせることによってです。また、この許可を剥奪した後でも、既存のセッションはすでにこの検査を実施していると主張するかも知れません。ですからこれはオブジェクトへのアクセスを妨げる完全にセキュアな方法ではありません。

シーケンスに対しては、currvalとnextval関数の利用を許可します。

型とドメインに対しては、テーブル、関数、および他のスキーマオブジェクトを生成する際に型とドメインを使用することを許可します。(たとえば問い合わせ中に表れる型の値のような、すべての型の「利用」をこの権限はコントロールするわけではないことに注意してください。その型に依存するオブジェクトが作られるのを防ぐだけです。この権限の主な目的は、どのユーザがある型への依存関係を作ることができるかを制御し、後で所有者がこの型を変更するのを防ぐためです。)

外部データラッパーに対しては、その外部データラッパーを使って新しいサーバを作ることを許可します。

外部サーバに対しては、そのサーバを使って外部テーブルを作ることを許可します。権限を授与されたものは、そのサーバに結びついたユーザマッピングを作成、変更、削除することができます。

他のコマンドで必要となる権限はそれぞれのコマンドのリファレンスページに列挙されています。

PostgreSQLはあるタイプのオブジェクトが作成された時に、そのオブジェクトに対する権限をデフォルトでPUBLICに付与します。テーブル、テーブルの列、シーケンス、外部データラッパー、外部サーバ、ラージオブジェクト、スキーマ、テーブル空間に対しては、デフォルトではPUBLICに権限を付与しません。他のタイプのオブジェクトに対しては、PUBLICにデフォルトで付与される権限は次のものです。CONNECT、TEMPORARY(データベース内で一時テーブルを作成する権限)、関数とプロシージャに対するEXECUTE権限、言語とデータ型(ドメインを含む)に対するUSAGE権限。もちろんオブジェクトの所有者は、デフォルト、あるいは明示的に与えられた権限をREVOKEできます。(セキュリティを最大限に高めるためには、REVOKEをオブジェクトを作成したのと同じトランザクション内で発行してください。そうすれば他のユーザがそのオブジェクトを使う隙が存在しません。) また、デフォルトの権限設定はALTER DEFAULT PRIVILEGESを使って上書きできます。

表 5.1に、*ACL*(Access Control List)値において権限タイプに使われる1文字の短縮形を示します。psqlの出力、あるいはシステムカタログのACL列を参照することでこれらの文字を見ることができます。

表5.1 ACL短縮形

権限	短縮形	適用可能なオブジェクトタイプ
SELECT	r (「read」)	LARGE OBJECT, SEQUENCE, TABLE(およびテーブルのようなオブジェクト)、テーブルの列
INSERT	a (「append」)	TABLE、テーブルの列
UPDATE	w (「write」)	LARGE OBJECT, SEQUENCE, TABLE, テーブルの列
DELETE	d	TABLE
TRUNCATE	D	TABLE
REFERENCES	x	TABLE、テーブルの列
TRIGGER	t	TABLE
CREATE	C	DATABASE, SCHEMA, TABLESPACE
CONNECT	c	DATABASE
TEMPORARY	T	DATABASE

権限	短縮形	適用可能なオブジェクトタイプ
EXECUTE	X	FUNCTION, PROCEDURE
USAGE	U	DOMAIN, FOREIGN DATA WRAPPER, FOREIGN SERVER, LANGUAGE, SCHEMA, SEQUENCE, TYPE

表 5.2は、前述の短縮形を用いてそれぞれのタイプのSQLオブジェクトで利用可能な権限をまとめています。また、それぞれのオブジェクトタイプの権限設定を調べる際に利用できるpsqlコマンドを示します。

表5.2 アクセス権限のまとめ

オブジェクトタイプ	すべての権限	デフォルトPUBLIC権限	psqlコマンド
DATABASE	CTc	Tc	\l
DOMAIN	U	U	\dD+
FUNCTION or PROCEDURE	X	X	\df+
FOREIGN DATA WRAPPER	U	none	\dew+
FOREIGN SERVER	U	none	\des+
LANGUAGE	U	U	\dL+
LARGE OBJECT	rw	none	
SCHEMA	UC	none	\dn+
SEQUENCE	rwU	none	\dp
TABLE(およびテーブルのようなオブジェクト)	arwdDxt	none	\dp
テーブルの列	arwx	none	\dp
TABLESPACE	C	none	\db+
TYPE	U	U	\dT+

あるオブジェクトに与えられている権限はaclitemエントリのリストとして表示されます。そこでは、aclitemはある権限付与者によって与えられている権限授与者の許可を示しています。たとえば、calvin=r*w/hobbesは、ロールcalvinが許可オプション(*)ありのSELECT(r)と許可オプションなしのUPDATE(w)を持ち、それらがロールhobbesに与えられていることを示します。別の権限付与者によって権限が与えられている同じオブジェクトに対してcalvinも権限を持っている場合は、別のaclitemエントリとして表示されます。aclitemの権限授与者フィールドが空であれば、それはPUBLICを表します。

ユーザmiriamがテーブルmytableを作成し、以下を行う例を考えます。

```
GRANT SELECT ON mytable TO PUBLIC;
GRANT SELECT, UPDATE, INSERT ON mytable TO admin;
GRANT SELECT (col1), UPDATE (col1) ON mytable TO miriam_rw;
```

すると、psqlの\dpコマンドは次のように表示するはずです。

```
=> \dp mytable
                                Access privileges
 Schema |  Name   | Type  |   Access privileges   |   Column privileges   | Policies
```

```
--------+---------+-------+----------------------+----------------------+----------
 public | mytable | table | miriam=arwdDxt/miriam+| col1:                +|
        |         |       | =r/miriam            +|   miriam_rw=rw/miriam |
        |         |       | admin=arw/miriam      |                       |
(1 row)
```

あるオブジェクトに対して「Access privileges」列が空なら、そのオブジェクトがデフォルトの権限を持つことを意味します。（つまり、関連するシステムカタログの権限エントリがNULLだということです。）デフォルト権限は常に所有者の全権限を含み、更に上で説明示したようにオブジェクトタイプ依存のPUBLICに対する権限を持つことができます。オブジェクトに対する初回のGRANTあるいはREVOKEにより、デフォルト権限（たとえばmiriam=arwdDxt/miriam）が設定され、次に特定の要求に従って変更されます。同様に、「Column privileges」に示されるエントリは非デフォルトの権限を持つ列のためだけのものです。（注意：「デフォルト権限」は常にオブジェクトのタイプの組み込みのデフォルト権限を意味します。ALTER DEFAULT PRIVILEGESコマンドによって権限が影響を受けるオブジェクトは常にALTERの影響を含む明示的な権限エントリを伴って示されます。）

所有者の暗黙的な許可オプションはアクセス権限表示では印を付けられないことに注意してください。*は許可オプションが明示的に誰かに許可されたときにのみ現れます。

5.8. 行セキュリティポリシー

GRANTによって利用できるSQL標準の権限システムに加えて、通常の問い合わせでどの行が戻され、データ更新のコマンドでどの行を挿入、更新、削除できるかをユーザ単位で制限する*行セキュリティポリシー*をテーブルに定義できます。この機能は*行単位セキュリティ*としても知られています。デフォルトではテーブルには何もポリシーはなく、SQLの権限システムによってテーブルのアクセス権限があるユーザは、テーブル内のすべての行について同じように、問い合わせや更新をすることができます。

テーブルの行セキュリティが有効の場合（ALTER TABLE ... ENABLE ROW LEVEL SECURITYを使います）、行の検索や行の更新のための通常のテーブルアクセスはすべて、行セキュリティポリシーによって許可される必要があります。（ただし、テーブルの所有者は典型的には行セキュリティポリシーの対象とはなりません。）テーブルにポリシーが存在しない場合は、デフォルト拒否のポリシーが使われて、どの行も見ることも更新することもできなくなります。TRUNCATEやREFERENCESなど、テーブル全体に対する操作は行セキュリティの対象とはなりません。

行セキュリティポリシーは特定のコマンド、特定のロール、あるいはその両方に対して定義できます。ポリシーはALLつまりすべてのコマンドに対して適用、あるいはSELECT、INSERT、UPDATE、DELETEに適用することを指定できます。1つのポリシーを複数のロールに割り当てることができ、通常のロールのメンバ資格と継承の規則が当てはまります。

ポリシーでどの行が可視である、あるいは更新可能であるかを指定するために、ブーリアン値を返す式が必要です。ユーザの問い合わせにあるどの条件や関数よりも前に、この式が各行について評価されます。（この規則の例外は、情報リークがないことが保証されるleakproof関数だけです。行セキュリティの確認の前にこのような関数を適用することをオプティマイザが選択することがあります。）式がtrueを返さない行は処理対象になりません。可視である行と変更可能な行について独立した制御ができるように、別々の式を指定することも可能です。ポリシーの式は問い合わせの一部分として、問い合わせをしているユーザの権限で実行されます。ただし、呼び出しユーザに利用できないデータにアクセスするために、セキュリティ定義関数を使うことができます。

スーパーユーザ、およびBYPASSRLS属性のあるロールは、テーブルへのアクセス時に、常に行セキュリティシステムを無視します。テーブルの所有者も通常は行セキュリティを無視しますが、ALTER TABLE ... FORCE ROW LEVEL SECURITYにより、テーブルの所有者も行セキュリティの対象となることができます。

行セキュリティの有効化、無効化、およびポリシーのテーブルへの追加は、常に、テーブルの所有者のみの権限です。

ポリシーはCREATE POLICYコマンドで作成され、ALTER POLICYコマンドで変更され、DROP POLICYコマンドで削除されます。テーブルの行セキュリティを有効に、あるいは無効にするにはALTER TABLEコマンドを使います。

各ポリシーには名前があり、1つのテーブルに複数のポリシーを定義することができます。ポリシーはテーブルごとに定義されるので、1つのテーブルの各ポリシーは異なる名前でなければなりません。異なるテーブルであれば、同じ名前のポリシーが存在しても構いません。

ある問い合わせに複数のポリシーが適用される場合、(デフォルトの許容(permissive)ポリシーについては)ORまたは(制限(restrictive)ポリシーについては) ANDを使って結合されます。これは、あるロールが、それが属するすべてのロールの権限を合わせ持つのと類似しています。許容ポリシーと制限ポリシーについては以下で更に説明します。

簡単な例として、managersロールのメンバーだけが行にアクセスでき、かつ自分のアカウントの行のみアクセスできるポリシーをaccountリレーション上に作成する方法を以下に示します。

```
CREATE TABLE accounts (manager text, company text, contact_email text);

ALTER TABLE accounts ENABLE ROW LEVEL SECURITY;

CREATE POLICY account_managers ON accounts TO managers
    USING (manager = current_user);
```

上記のポリシーは、上記のUSING句と同じWITH CHECK句を暗黙的に提供するので、制約は、コマンドが選択した行にも適用されますし(ですから、マネージャは、違うマネージャに属する既存の行に対してSELECT、UPDATE、DELETEを発行することはできません)、コマンドが変更した行にも適用されます(ですから、違うマネージャに属する行を、INSERTあるいはUPDATEで作ることはできません)。

ロールが指定されなかった場合、あるいは特別なユーザ名PUBLICが指定された場合、ポリシーはシステム上の全ユーザに適用されます。すべてのユーザがusersテーブルの自分自身の行にだけアクセスできるようにするためには、次の簡単なポリシーが使用できます。

```
CREATE POLICY user_policy ON users
    USING (user_name = current_user);
```

これは前の例と同じように動きます。

テーブルに追加される行に対し、可視である行とは異なるポリシーを使用する場合は、複数のポリシーを組み合わせることができます。組み合わせたポリシーにより、すべてのユーザがusersテーブルのすべての行を見ることができますが、自分自身の行だけしか更新できません。

```
CREATE POLICY user_sel_policy ON users
```

```
    FOR SELECT
    USING (true);
CREATE POLICY user_mod_policy ON users
    USING (user_name = current_user);
```

SELECTコマンドでは、ORを使って2つのポリシーが組み合わされ、すべての行を検索できる効果をもたらします。他のコマンドに対しては、二番目のポリシーだけが適用され、以前と効果は同じです。

行セキュリティはALTER TABLEで無効にすることもできます。行セキュリティを無効にしても、テーブルに定義されているポリシーは削除されず、単に無視されるだけになります。このときはSQL標準の権限システムに従って、すべての行が可視で更新可能になります。

以下のより大きな例で、この機能が実運用の環境で如何にして使えるかを示します。passwdテーブルはUnixのパスワードファイルと同等のものです。

```
-- passwdファイルに基づく簡単な例
CREATE TABLE passwd (
  user_name             text UNIQUE NOT NULL,
  pwhash                text,
  uid                   int  PRIMARY KEY,
  gid                   int  NOT NULL,
  real_name             text NOT NULL,
  home_phone            text,
  extra_info            text,
  home_dir              text NOT NULL,
  shell                 text NOT NULL
);


CREATE ROLE admin;  -- 管理者
CREATE ROLE bob;    -- 一般ユーザ
CREATE ROLE alice;  -- 一般ユーザ



-- テーブルに値を入れる
INSERT INTO passwd VALUES
  ('admin','xxx',0,0,'Admin','111-222-3333',null,'/root','/bin/dash');
INSERT INTO passwd VALUES
  ('bob','xxx',1,1,'Bob','123-456-7890',null,'/home/bob','/bin/zsh');
INSERT INTO passwd VALUES
  ('alice','xxx',2,1,'Alice','098-765-4321',null,'/home/alice','/bin/zsh');


-- テーブルの行単位セキュリティを有効にする
ALTER TABLE passwd ENABLE ROW LEVEL SECURITY;
```

```
-- ポリシーを作成する
-- 管理者はすべての行を見ることができ、
      どんな行でも追加できる
CREATE POLICY admin_all ON passwd TO admin USING (true) WITH CHECK (true);

-- 一般ユーザはすべての行を見ることができる
CREATE POLICY all_view ON passwd FOR SELECT USING (true);

-- 一般ユーザは自身のレコードを更新できるが、

-- 変更できるのは使用するシェルだけに制限する
CREATE POLICY user_mod ON passwd FOR UPDATE
  USING (current_user = user_name)
  WITH CHECK (
    current_user = user_name AND
    shell IN ('/bin/bash','/bin/sh','/bin/dash','/bin/zsh','/bin/tcsh')
  );


-- adminにはすべての通常の権限を付与する
GRANT SELECT, INSERT, UPDATE, DELETE ON passwd TO admin;

-- 一般ユーザは公開列にSELECTでアクセスできるだけとする
GRANT SELECT
  (user_name, uid, gid, real_name, home_phone, extra_info, home_dir, shell)
  ON passwd TO public;

-- 特定の列についてはユーザによる更新を許可する
GRANT UPDATE
  (pwhash, real_name, home_phone, extra_info, shell)
  ON passwd TO public;
```

どんなセキュリティ設定でも同じですが、システムが期待通りに動作していることをテストして確認することが重要です。上の例を利用して、以下ではパーミッションのシステムが適切に動作していることを示します。

```
-- adminはすべての行と列を見ることができる
postgres=> set role admin;
SET
postgres=> table passwd;
 user_name | pwhash | uid | gid | real_name |  home_phone  | extra_info | home_dir    |   shell
-----------+--------+-----+-----+-----------+--------------+------------+-------------
+-----------
 admin     | xxx    |   0 |   0 | Admin     | 111-222-3333 |            | /root       | /bin/dash
```

```
 bob       | xxx    |   1 |   1 | Bob       | 123-456-7890 |            | /home/bob   | /bin/zsh
 alice     | xxx    |   2 |   1 | Alice     | 098-765-4321 |            | /home/alice | /bin/zsh
(3 rows)


-- Test what Alice is able to do
postgres=> set role alice;
SET
postgres=> table passwd;
ERROR:  permission denied for relation passwd
postgres=> select user_name,real_name,home_phone,extra_info,home_dir,shell from passwd;
 user_name | real_name |  home_phone  | extra_info | home_dir    |   shell
-----------+-----------+--------------+------------+-------------+-----------
 admin     | Admin     | 111-222-3333 |            | /root       | /bin/dash
 bob       | Bob       | 123-456-7890 |            | /home/bob   | /bin/zsh
 alice     | Alice     | 098-765-4321 |            | /home/alice | /bin/zsh
(3 rows)

postgres=> update passwd set user_name = 'joe';
ERROR:  permission denied for relation passwd

-- Aliceは自分のreal_nameを変更できるが、
      他は変更できない
postgres=> update passwd set real_name = 'Alice Doe';
UPDATE 1
postgres=> update passwd set real_name = 'John Doe' where user_name = 'admin';
UPDATE 0
postgres=> update passwd set shell = '/bin/xx';
ERROR:  new row violates WITH CHECK OPTION for "passwd"
postgres=> delete from passwd;
ERROR:  permission denied for relation passwd
postgres=> insert into passwd (user_name) values ('xxx');
ERROR:  permission denied for relation passwd

-- Aliceは自分のパスワードを変更できる。
-- RLSにより他の行は更新されないが、何も報告されない。
postgres=> update passwd set pwhash = 'abc';
UPDATE 1
```

ここまでで作成したポリシーはすべて許容ポリシーで、つまり複数のポリシーが適用される場合、それらは論理演算子「OR」を使って結合されるものでした。意図した場合にのみ行へのアクセスが許されるよう許容ポリシーを構築することは可能ですが、許容ポリシーを制限ポリシーと組み合わせることで、より単純にすることが可能です（制限ポリシーはレコードが満たさなければならないポリシーで、論理演算子「AND」を使って結合されます）。上記の例に重ねて、管理者がローカルのUnixソケットを通して接続してpasswdテーブルのレコードにアクセスすることを要求する制限ポリシーを追加してみます。

```
CREATE POLICY admin_local_only ON passwd AS RESTRICTIVE TO admin
    USING (pg_catalog.inet_client_addr() IS NULL);
```

こうすると以下のように、制限ポリシーにより、ネットワーク経由で接続している管理者にはレコードが見えないことがわかります。

```
=> SELECT current_user;
 current_user
--------------
 admin
(1 row)

=> select inet_client_addr();
 inet_client_addr
------------------
 127.0.0.1
(1 row)

=> SELECT current_user;
 current_user
--------------
 admin
(1 row)

=> TABLE passwd;
 user_name | pwhash | uid | gid | real_name | home_phone | extra_info | home_dir | shell
-----------+--------+-----+-----+-----------+------------+------------+----------+-------
(0 rows)

=> UPDATE passwd set pwhash = NULL;
UPDATE 0
```

一意性制約、主キー制約、外部キー制約などの参照整合性確認は、データの整合性を維持するため、常に行セキュリティを無視します。スキーマと行単位セキュリティの開発において、このような参照整合性確認により「カバートチャネル(covert channel)」の情報漏洩が起こらないようにするため、注意が必要です。

状況によっては、行セキュリティが適用されないことを確実にするのが重要になります。例えばバックアップを取るとき、行セキュリティのために、何のエラーや警告もなしに一部の行がバックアップされないとすると、破滅的です。このような状況では、設定パラメータrow_securityを`off`にすることができます。これ自体は行セキュリティを無視するわけではなく、問い合わせの結果がポリシーによって影響を受ける場合にエラーを発生させます。その後でエラーの原因を調査して解決することができます。

上の例では、ポリシーの式はアクセス対象または更新対象の行の現在の値のみを考慮していました。これは最も単純で、しかも効率の良い場合です。可能であれば、行セキュリティの適用はこのように動作するよう設計するのが最善です。ポリシーの決定をするために他の行あるいは他のテーブルを参照する必要がある場合は、ポリシーの式で副SELECTを使う、あるいはSELECTを含む関数を使うことができます。ただし、そのよう

なアクセスは注意深く設計しなければ、情報漏洩を起こすような競合条件を作り出す場合があることに注意して下さい。例えば、以下のテーブル設計を考えます。

```
-- 権限グループの定義
CREATE TABLE groups (group_id int PRIMARY KEY,
                     group_name text NOT NULL);

INSERT INTO groups VALUES
  (1, 'low'),
  (2, 'medium'),
  (5, 'high');


GRANT ALL ON groups TO alice;  -- aliceが管理者
GRANT SELECT ON groups TO public;


-- ユーザの権限レベルの定義
CREATE TABLE users (user_name text PRIMARY KEY,
                    group_id int NOT NULL REFERENCES groups);

INSERT INTO users VALUES
  ('alice', 5),
  ('bob', 2),
  ('mallory', 2);

GRANT ALL ON users TO alice;
GRANT SELECT ON users TO public;


-- 保護される情報を保持するテーブル
CREATE TABLE information (info text,
                          group_id int NOT NULL REFERENCES groups);

INSERT INTO information VALUES
  ('barely secret', 1),
  ('slightly secret', 2),
  ('very secret', 5);

ALTER TABLE information ENABLE ROW LEVEL SECURITY;


-- セキュリティのgroup_idが行のgroup_idより大きいか等しいユーザは
-- その行を見ること、
      更新することが可能
```

```
CREATE POLICY fp_s ON information FOR SELECT
  USING (group_id <= (SELECT group_id FROM users WHERE user_name = current_user));
CREATE POLICY fp_u ON information FOR UPDATE
  USING (group_id <= (SELECT group_id FROM users WHERE user_name = current_user));


-- informationテーブルを保護するのにRLSのみに依存する
GRANT ALL ON information TO public;
```

ここでaliceが「slightly secret」の情報を更新したいが、この行の新しい内容に関してmalloryは信頼すべきでないと判断しました。そこで、彼女は次のようにします。

```
BEGIN;
UPDATE users SET group_id = 1 WHERE user_name = 'mallory';
UPDATE information SET info = 'secret from mallory' WHERE group_id = 2;
COMMIT;
```

これは安全なように見えます。malloryが「secret from mallory」の文字列を見ることができる隙はありません。しかし、ここには競合条件があります。例えば、malloryが同時に以下を実行していたとしましょう。

```
SELECT * FROM information WHERE group_id = 2 FOR UPDATE;
```

ここで彼女のトランザクションがREAD COMMITTEDモードなら、彼女は「secret from mallory」を見ることが可能です。それは彼女のトランザクションが、aliceのトランザクションの直後にinformationの行にアクセスした場合に発生します。それはaliceのトランザクションがコミットされるのを待ってブロックされ、次にFOR UPDATE句があるため、更新後の行の内容をフェッチします。しかし、usersからの暗示的なSELECTでは更新後の行をフェッチ*しません*。なぜなら、その副SELECTにはFOR UPDATEがないため、usersの行は問い合わせの開始時に取得したスナップショットから読まれるからです。そのため、ポリシーの式はmalloryの権限レベルの古い値について検査し、更新後の行を見ることを許してしまいます。

この問題を回避する方法はいくつかあります。一つの簡単な答は行セキュリティポリシーの副SELECTでSELECT ... FOR SHAREを使うことです。しかし、これは影響を受けるユーザに対し、参照先テーブル（この場合はusers）のUPDATE権限を付与する必要があり、望ましくないかもしれません。（しかし、もう一つの行セキュリティポリシーを適用して、彼らが実際にその権限を行使することを防ぐことはできます。また、副SELECTをセキュリティ定義関数内に埋め込むことも可能です。）また、参照先テーブルに行共有ロックが同時に大量に発生するとパフォーマンス問題が起きるかもしれません。特にそのテーブルの更新が多いときは問題になるでしょう。別の解決策で、参照先テーブルの更新が少ない場合に現実的なのは、参照先テーブルの更新時に排他ロックを取得するものです。そうすれば、同時実行のトランザクションが行の古い値を調べることはできません。あるいは、参照先のテーブルの更新をコミットした後、単にすべての同時実行トランザクションが終わるのを待ってから、新しいセキュリティ状況に依存する変更をする、ということもできます。

更なる詳細はCREATE POLICYとALTER TABLEを参照して下さい。

5.9. スキーマ

PostgreSQLデータベースクラスタには、1つ以上の名前付きデータベースが含まれます。ロールおよびいくつかの他のタイプのオブジェクトはクラスタ全体で共有されます。サーバに接続しているクライアント接続は、単一のデータベース、つまり接続要求で指定したデータベース内のデータにしかアクセスできません。

> **注記**
>
> クラスタのユーザは、クラスタ内の全てのデータベースへのアクセス権限を持っているとは限りません。ロール名を共有するということは、例えばjoeという同じロール名を持つ異なるユーザが同じクラスタ内の2つのデータベースに存在することはできないということです。しかし、joeが一部のデータベースにのみアクセスできるようにシステムを構成することはできます。

データベースには、1つ以上の名前付き*スキーマ*が含まれ、スキーマにはテーブルが含まれます。スキーマには、データ型、関数および演算子などの他の名前付きオブジェクトも含まれます。同じオブジェクト名を異なるスキーマで使用しても競合は起こりません。例えば、schema1とmyschemaの両方のスキーマにmytableというテーブルが含まれていても構いません。スキーマはデータベースとは異なり厳格に分離されていないので、ユーザは、権限さえ持っていれば接続しているデータベース内のどのスキーマのオブジェクトにでもアクセスすることができます。

スキーマの使用が好まれる理由はいくつかあります。

- 1つのデータベースを多数のユーザが互いに干渉することなく使用できるようにするため。
- 管理しやすくなるよう、データベースオブジェクトを論理グループに編成するため。
- サードパーティのアプリケーションを別々のスキーマに入れることにより、他のオブジェクトの名前と競合しないようにするため。

スキーマは、入れ子にできないという点を除き、オペレーティングシステムのディレクトリと似ています。

5.9.1. スキーマの作成

スキーマを作成するには、CREATE SCHEMAコマンドを使用します。スキーマに自由に名前を付けます。例を示します。

```
CREATE SCHEMA myschema;
```

スキーマ内にオブジェクトを作成したりこれにアクセスするには、スキーマ名とテーブル名をドットで区切った*修飾名*を書きます。

```
schema.table
```

この方法は、後の章で説明するテーブル変更コマンドやデータアクセスコマンドなど、テーブル名を必要とする場合すべてに使用できます。(話を簡単にするため、テーブルについてのみ述べます。しかし型や関数といった名前付きのオブジェクトの他の種類について同様の考え方が適用できます。)

実際には、より一般的な以下の構文

```
database.schema.table
```

を使用することもできますが、現在ではこの構文は標準SQLに形式的に準拠するためにのみ存在しています。記述されるデータベース名は、接続しているデータベースと同じ名前でなければなりません。

ですから、新しいスキーマにテーブルを作成するには次のようにします。

```
CREATE TABLE myschema.mytable (
 ...
);
```

空のスキーマ(全てのオブジェクトが削除されたスキーマ)を削除するには次のようにします。

```
DROP SCHEMA myschema;
```

スキーマ内の全オブジェクトも含めてスキーマを削除する場合には次のようにします。

```
DROP SCHEMA myschema CASCADE;
```

この背後にある一般的な機構についての詳細は5.14を参照してください。

他のユーザが所有するスキーマを作成したい場合があります(これは他のユーザの活動を明確に定義された名前空間内に制限する方法の1つです)。そのための構文は次の通りです。

```
CREATE SCHEMA schema_name AUTHORIZATION user_name;
```

スキーマ名は省略することもでき、その場合スキーマ名はユーザ名と同じになります。この構文の便利な使用方法は5.9.6に記載されています。

pg_で始まるスキーマ名は、システム上の使用のため予約されており、ユーザが作成することはできません。

5.9.2. publicスキーマ

これまでの節ではスキーマ名を指定せずにテーブルを作成してきました。デフォルトでは、このようなテーブル(および他のオブジェクト)は自動的に「public」という名前のスキーマに入れられます。新しいデータベースには全てこのようなスキーマが含まれています。そのため、以下の2つの構文は同等です。

```
CREATE TABLE products ( ... );
```

および

```
CREATE TABLE public.products ( ... );
```

5.9.3. スキーマ検索パス

修飾名を書くのは手間がかかりますし、どちらにしても、アプリケーションに特定のスキーマ名を書き込まない方が良いことも多いのです。そのため、テーブルは多くの場合、テーブル名しか持たない非修飾名として

参照されます。システムは、検索するスキーマのリストである検索パスに従って、どのテーブルを指しているのかを判別します。検索パスで最初に一致したテーブルが、該当テーブルだと解釈されます。検索パス内に一致するテーブルがないと、データベースの他のスキーマ内に一致するテーブルがある場合でもエラーが報告されます。

同じ名前のオブジェクトを異なるスキーマに作成できる結果、正確に同じオブジェクトを参照する問合せを書く作業が、いつも複雑になります。また、ユーザが悪意を持って、あるいは偶然に他のユーザの問合せの挙動を変える可能性をもたらします。PostgreSQL内部では非修飾名を問合せ中で使うことが一般的なので、search_pathにスキーマを追加することは、CREATEの書き込み権限を持っているすべてのユーザを、実質的に信頼することになります。あなたが通常の問合せを実行する際、あなたのサーチパス内のスキーマにオブジェクトを作成できる悪意のあるユーザは、支配権を奪い、あたかもあなたが実行したように任意のSQL関数を実行できます。

検索パスの最初に列挙されているスキーマは、「現在のスキーマ」と呼ばれます。現在のスキーマは、検索される最初のスキーマであると同時に、スキーマ名を指定せずにCREATE TABLEコマンドでテーブルを作成した場合に新しいテーブルが作成されるスキーマでもあります。

現行の検索パスを示すには次のコマンドを使用します。

```
SHOW search_path;
```

デフォルトの設定では次のように返されます。

```
search_path
--------------
 "$user", public
```

最初の要素は、現行ユーザと同じ名前のスキーマを検索することを指定しています。そのようなスキーマが存在していない場合、この項目は無視されます。2番目の要素は、先ほど説明したpublicスキーマを参照しています。

実存するスキーマのうち、検索パス内で最初に現れるスキーマが、新規オブジェクトが作成されるデフォルトの場所になります。これが、デフォルトでオブジェクトがpublicスキーマに作成される理由です。オブジェクトがスキーマ修飾なしで別の文脈で参照される場合（テーブル変更、データ変更、あるいは問い合わせコマンドなど）、一致するオブジェクトが見つかるまで検索パス内で探索されます。そのためデフォルト構成では、非修飾のアクセスはpublicスキーマしか参照できません。

新しいスキーマをパスに追加するには次のようにします。

```
SET search_path TO myschema,public;
```

（$userはまだ必要ないので、ここでは省略しています。）そして、次のようにしてスキーマ修飾なしでテーブルにアクセスします。

```
DROP TABLE mytable;
```

また、myschemaはパス内の最初の要素なので、新しいオブジェクトはデフォルトでここに作成されます。

以下のように書くこともできます。

```
SET search_path TO myschema;
```

このようにすると、今後は修飾名なしでpublicスキーマにアクセスすることができなくなります。publicスキーマはデフォルトで存在するということ以外に特別な意味はありません。他のスキーマと同様に削除することもできます。

スキーマ検索パスを操作する他の方法については9.26を参照してください。

検索パスはデータ型名、関数名、演算子名についても、テーブル名の場合と同じように機能します。データ型および関数の名前は、テーブル名とまったく同じように修飾することができます。式で修飾演算子名を書く場合には、特別な決まりがあります。それは以下の通りです。

```
OPERATOR(schema.operator)
```

この規則は構文が曖昧になることを防ぐためのものです。以下に例を示します。

```
SELECT 3 OPERATOR(pg_catalog.+) 4;
```

実際の場面ではこのような見づらい構文を書かなくて済むように、演算子についても検索パスが使用されています。

5.9.4. スキーマおよび権限

ユーザは、デフォルトでは所有していないスキーマのオブジェクトをアクセスすることはできません。アクセスするためには、そのスキーマの所有者からスキーマのUSAGE権限を付与してもらわなければなりません。そのスキーマ内のオブジェクトに対して操作を行うには、そのオブジェクトに応じて、さらに追加の権限が必要となる場合があります。

他のユーザのスキーマ内でオブジェクトを作成できるようにすることも可能です。それには、スキーマ上でCREATE権限が付与されていなければなりません。デフォルトでは、public スキーマに関しては全てのユーザがCREATEとUSAGE権限を持っていることに注意してください。つまり、全てのユーザは、そのユーザが接続できる任意のデータベース上のpublicスキーマにオブジェクトを作成できるということです。利用パターンでは、その権限を剥奪することを求めています。

```
REVOKE CREATE ON SCHEMA public FROM PUBLIC;
```

（最初の「public」はスキーマです。2番目の「public」は「全てのユーザ」を意味します。最初のpublicは識別子で、2番目のpublicはキーワードなので、それぞれ小文字、大文字を使用しています。4.1.1のガイドラインを思い出してください。）

5.9.5. システムカタログスキーマ

各データベースには、publicおよびユーザ作成のスキーマの他にpg_catalogスキーマが含まれています。このスキーマにはシステムテーブルと全ての組み込みデータ型、関数および演算子が含まれています。pg_catalogは常に検索パスに含まれています。パスに明示的にリストされていない場合は、パスのスキーマ

を検索する前に暗黙的に検索されます。これにより組み込みの名前が常に検索されることが保証されます。しかし、ユーザ定義の名前で組み込みの名前を上書きする場合は、pg_catalogを明示的にパスの最後に置くことができます。

システムカタログの名前はpg_で始まりますので、このような名前は使用しないのが得策と言えます。今後のバージョンでユーザのテーブルと同じ名前のシステムカタログが定義され、競合する事態を避けるためです。(その結果、デフォルトの検索パスでは、ユーザのテーブル名への非修飾の参照はシステムカタログとして解決されることになります。) システムカタログは今後もpg_で始まる規則に従うので、ユーザがpg_という接頭辞を使わない限り、非修飾のユーザ定義テーブル名がシステムカタログと競合することはありません。

5.9.6. 使用パターン

スキーマは様々な方法でデータの編成に使用できます。セキュアなスキーマの使用パターンは信頼できないユーザが他のユーザの問い合わせの振る舞いを変えるのを防ぎます。データベースがセキュアなスキーマの使用パターンを使わない場合、セキュアにデータベースを問い合わせたいユーザはセッションの開始毎に防御的なアクションを取るようにします。とりわけユーザはsearch_pathに空文字をセットするか、さもなければ非スーパーユーザが書き込めるスキーマをsearch_pathから削除します。デフォルト構成で簡単にサポートできるお勧めの使用パターンがいくつかあります。

- 一般ユーザに、ユーザ個人用のスキーマだけを使わせます。これを実現するには、REVOKE CREATE ON SCHEMA public FROM PUBLICを発行し、個々のユーザにユーザと同じ名前でスキーマを作成してください。デフォルトサーチパスは、ユーザ名として解釈される$userで始まることを思い出してください。ですから、ユーザが各自別々のスキーマを所有しているなら、デフォルトでは自身のスキーマにアクセスします。信頼できないユーザがすでにログインしたデータベースでこのパターンを採用した後には、pg_catalogスキーマ内にあるのと同じ名前のpublicスキーマ内のオブジェクトを監視することを考慮してください。セキュアなスキーマ利用パターンが存在しないような、信用できないユーザがデータベース所有者である場合や、CREATEROLE権限を持っている場合を除き、このパターンはセキュアなスキーマ利用パターンです。

- postgresql.confを変更、あるいはALTER ROLE ALL SET search_path = "$user"を実行することにより、デフォルトサーチパスからからpublicスキーマを削除します。全員がパブリックスキーマに引き続きオブジェクトを作ることができますが、オブジェクトの選択は修飾名によってのみ行われます。修飾されたテーブル名による参照は問題ありませんが、パブリックスキーマ内の関数呼び出しは安全ではないか、あるいは信頼性がありません。パブリックスキーマ内に関数や拡張を作る場合は、最初のパターンを代わりに使ってください。それ以外では、最初のパターン同様、信頼できないユーザがデータベース所有者である場合や、CREATEROLE権限を持っている場合を除き、これはセキュアです。

- デフォルトを維持します。すべてのユーザがpublicスキーマに暗黙的にアクセスします。これはスキーマを考慮しない世界からのスムースな移行を可能にしながら、スキーマがまったく利用できない状況をシミュレートします。しかし、これは決してセキュアなパターンではありません。このパターンは、データベースに一人、あるいは少数のお互いに信頼できるユーザだけが存在する場合にのみ受け入れ可能です。

どのパターンでも、共有のアプリケーション(全員が使うテーブル、サードパーティが提供する追加の関数など)をインストールするには、別のスキーマにアプリケーションを入れてください。他のユーザがアプリケーションにアクセスするために、適切な権限を与えることを忘れないようにしてください。ユーザはスキーマ名で名前を修飾するか、あるいは追加スキーマをサーチパスに入れるかを選択し、これらの追加オブジェクトを参照できます。

5.9.7. 移植性

標準SQLでは、1つのスキーマ内のオブジェクトを異なるユーザが所有するという概念は存在しません。それどころか、実装によっては所有者と異なる名前のスキーマを作成することが許可されていない場合もあります。実際、標準で規定されている基本スキーマサポートのみを実装しているデータベースシステムでは、スキーマという概念とユーザという概念はほとんど同じなのです。そのため、修飾名とはuser_name.table_nameのことであると思っているユーザはたくさんいます。PostgreSQLにおいても、ユーザごとに1つのスキーマを作成すると、このようになります。

また、標準SQLには、publicスキーマという概念もありません。標準に最大限従うためには、publicスキーマは使用すべきではありません。

もちろん、スキーマをまったく実装していなかったり、または、データベース間アクセスを（場合によっては制限付きで）許可することによって名前空間の使用をサポートしているSQLデータベースもあります。このようなシステムで作業する必要がある場合は、スキーマをまったく使わないようにすることで最大限の移植性を実現できます。

5.10. 継承

PostgreSQLは、データベース設計者にとって便利なテーブルの継承を実装しています。（SQL:1999以降は型の継承を定義していますが、ここで述べられている継承とは多くの点で異なっています。）

まず例から始めましょう。市（cities）のデータモデルを作成しようとしていると仮定してください。それぞれの州にはたくさんの市がありますが、州都（capitals）は1つのみです。どの州についても州都を素早く検索したいとします。これは、2つのテーブルを作成することにより実現できます。1つは州都のテーブルで、もう1つは州都ではない市のテーブルです。しかし、州都であるか否かに関わらず、市に対するデータを問い合わせたいときには何が起こるでしょうか？ 継承はこの問題を解決できます。citiesから継承されるcapitalsテーブルを定義するのです。

```
CREATE TABLE cities (
    name            text,
    population      float,
    elevation       int     -- in feet
);

CREATE TABLE capitals (
    state           char(2)
) INHERITS (cities);
```

この場合、capitalsテーブルは、その親テーブルであるcitiesテーブルの列をすべて継承します。州都は1つの追加の列stateを持ち、州を表現します。

PostgreSQLでは、1つのテーブルは、0以上のテーブルから継承することが可能です。また、問い合わせはテーブルのすべての行、またはテーブルのすべての行と継承されたテーブルのすべての行のいずれかを参照できます。後者がデフォルトの動作になります。例えば次の問い合わせは、500フィートより高い標高に位置しているすべての市の名前を、州都を含めて検索します。

```
SELECT name, elevation
    FROM cities
    WHERE elevation > 500;
```

PostgreSQLチュートリアルからのサンプルデータ(2.1を参照してください)に対して、この問い合わせは、以下の結果を出力します。

```
name    | elevation
-----------+-----------
 Las Vegas |      2174
 Mariposa  |      1953
 Madison   |       845
```

一方、次の問い合わせは、州都ではなく500フィートより高い高度に位置しているすべての市を検索します。

```
SELECT name, elevation
    FROM ONLY cities
    WHERE elevation > 500;

   name    | elevation
-----------+-----------
 Las Vegas |      2174
 Mariposa  |      1953
```

ここでONLYキーワードは、問い合わせがcitiesテーブルのみを対象にしcities以下の継承の階層にあるテーブルは対象としないことを意味します。これまで議論したコマンドの多く—SELECT、UPDATEそしてDELETE—がONLYキーワードをサポートしています。

また、明示的に子孫テーブルが含まれていることを示すために、テーブル名の後ろに*を書くこともできます:

```
SELECT name, elevation
    FROM cities*
    WHERE elevation > 500;
```

*の指定は、その動作が常にデフォルトであるため、必要ありません。しかし、この構文はデフォルトが変更可能であった古いリリースとの互換性のためにまだサポートされています。

ある特定の行がどのテーブルからきたものか知りたいという場合もあるでしょう。それぞれのテーブルにはtableoidという、元になったテーブルを示すシステム列があります。

```
SELECT c.tableoid, c.name, c.elevation
FROM cities c
WHERE c.elevation > 500;
```

出力は以下の通りです。

```
tableoid |   name    | elevation
```

```
----------+-----------+-----------
   139793 | Las Vegas |      2174
   139793 | Mariposa  |      1953
   139798 | Madison   |       845
```

（この例をそのまま実行しても、おそらく異なる数値OIDが得られるでしょう。）pg_classと結合することで、テーブルの実際の名前が分かります。

```
SELECT p.relname, c.name, c.elevation
FROM cities c, pg_class p
WHERE c.elevation > 500 AND c.tableoid = p.oid;
```

出力は以下の通りです。

```
relname  |   name    | elevation
----------+-----------+-----------
 cities   | Las Vegas |      2174
 cities   | Mariposa  |      1953
 capitals | Madison   |       845
```

同じ効果を得る別の方法は、別名型regclassを使うことで、これによりテーブルのOIDを記号的に表示します。

```
SELECT c.tableoid::regclass, c.name, c.elevation
FROM cities c
WHERE c.elevation > 500;
```

継承はINSERTまたはCOPYによるデータを、継承の階層にある他のテーブルに自動的に伝播しません。この例では、次のINSERT文は失敗します。

```
INSERT INTO cities (name, population, elevation, state)
VALUES ('Albany', NULL, NULL, 'NY');
```

データが、どうにかしてcapitalsテーブルに入ることを期待するかもしれませんが、そのようにはなりません。INSERTは、いつも指定されたテーブルそれ自体に対してデータを挿入します。ルール（詳細は第40章を参照してください）を使用して挿入を中継できる場合もあります。しかし、ルールを使用しても上記のような場合は解決できません。なぜなら、citiesテーブルにstate列が含まれていないため、ルールが適用される前にコマンドが拒否されてしまうからです。

親テーブル上の検査制約と非NULL制約は、NO INHERIT句によって明示的に指定され無い限り、その子テーブルに自動的に継承されます。他の種類の制約（一意性制約、主キー、外部キー制約）は継承されません。

テーブルは1つ以上の親テーブルから継承可能です。この場合、テーブルは親テーブルで定義された列の和になります。子テーブルで宣言された列は、これらの列に追加されることになります。もし親テーブルに同じ名前の列がある場合、もしくは、親テーブルと子テーブルに同じ名前の列がある場合は、列が「統合」されて子テーブルではただ1つの列となります。統合されるには列は同じデータ型を持っている必要があります。異なるデータ型の場合にはエラーとなります。継承可能な検査制約と非NULL制約は、同じようなやり方で統合

されます。つまり、例えば、列定義のいずれかが非NULL制約の印が付いているならば、統合された列に非NULLという印が付きます。検査制約は、同じ名前を持っている場合に統合され、それらの条件が異なる場合は統合に失敗します。

テーブル継承は、通常、CREATE TABLE文のINHERITS句を使用して、子テーブルを作成する時に確立します。他にも、互換性を持つ方法で定義済みのテーブルに新しく親子関係を付けることも可能です。これにはALTER TABLEのINHERIT形式を使用します。このためには、新しい子テーブルは親テーブルと同じ名前の列を持ち、その列の型は同じデータ型でなければなりません。また、親テーブルと同じ名前、同じ式の検査制約を持っていなければなりません。ALTER TABLEのNO INHERIT形式を使用して、同様に継承関係を子テーブルから取り除くことも可能です。このような継承関係の動的追加、動的削除は、継承関係をテーブル分割（5.11を参照）に使用している場合に有用です。

後で子テーブルとする予定の、互換性を持つテーブルを簡単に作成する方法の1つは、CREATE TABLEでLIKE句を使用することです。これは、元としたテーブルと同じ列を持つテーブルを新しく作成します。新しい子テーブルが必ず親テーブルと一致する制約を持ち、互換性があるものとみなされるように、元となるテーブルでCHECK制約が存在する場合は、LIKEにINCLUDING CONSTRAINTSオプションを指定すべきです。

子テーブルが存在する場合親テーブルを削除することはできません。また、子テーブルでは、親テーブルから継承した列、または検査制約を削除することも変更することもできません。テーブルとそのすべての子テーブルを削除したければ、CASCADEオプションを付けて親テーブルを削除することが簡単な方法です（5.14を参照）。

ALTER TABLEは、列データ定義と検査制約の変更を継承の階層にあるテーブルに伝えます。ここでも、他のテーブルに依存する列の削除はCASCADEオプションを使用したときのみ可能となります。ALTER TABLEは、重複列の統合と拒否について、CREATE TABLE時に適用される規則に従います。

継承された問い合わせは、親テーブルのみアクセス権限を検査します。つまり、例えば、UPDATE権限をcitiesテーブルに付与することは、citiesテーブルを通じてアクセスする場合に、capitalsテーブルにも行の更新権限を付与することを意味します。これによりデータが親テーブルに（も）あるように見えることが保たれます。しかし、capitalsテーブルは、追加権限なしに直接更新することはできません。同様に、親テーブルの行セキュリティポリシー（5.8を参照してください）が、継承された問い合わせの時に子テーブルの行に適用されます。子テーブルのポリシー（あれば）は、問い合わせにて明示的に指定されたテーブルである時にのみ適用されます。そしてこの場合、親テーブルに紐付けられたあらゆるポリシーは無視されます。

外部テーブル（5.12参照）も通常のテーブルと同様、親テーブルあるいは子テーブルとして継承の階層の一部となりえます。外部テーブルが継承の階層の一部となっている場合、外部テーブルがサポートしない操作は、その継承全体でもサポートされません。

5.10.1. 警告

すべてのSQLコマンドが継承階層に対して動作できるとは限らないことに注意してください。データの検索、データの変更、スキーマの変更のために使用されるコマンド（例えばSELECT、UPDATE、DELETE、ALTER TABLEのほとんどの構文が該当しますが、INSERTやALTER TABLE ... RENAMEは含まれません）は通常、デフォルトで子テーブルを含み、また、それを除外するためのONLY記法をサポートします。データベース保守およびチューニング（例えばREINDEX、VACUUM）を行うコマンドは通常、個々の物理テーブルに対してのみ動作し、継承階層に対する再帰をサポートしません。個々のコマンドのそれぞれの動作はそのマニュアルページ（SQLコマンド）に記載されています。

継承機能の重大な制限として、インデックス（一意性制約を含む）、および外部キーは、そのテーブルのみに適用され、それを継承した子テーブルには適用されないことがあります。これは外部キーの参照側、被参照側の両方について当てはまります。したがって、上の例では

- もし、cities.nameをUNIQUEまたはPRIMARY KEYと宣言しても、citiesテーブルの行と重複した行をcapitalsテーブル内に持つことを禁止することにはなりません。さらに、これらの重複した行はデフォルトでcitiesテーブルへの問い合わせで現れるでしょう。事実として、capitalsテーブルはデフォルトで一意性制約を持っていませんし、同一の名前の複数の行を持つことがあり得ます。capitalsテーブルに一意性制約を追加できますが、これはcitiesテーブルと比較して重複を禁止することにはなりません。
- 同じように、cities.name REFERENCESで他のテーブルを参照するようにしても、この制約は自動的にcapitalsに引き継がれるわけではありません。この場合はcapitalsテーブルに同一のREFERENCES制約を手動で追加すれば問題を回避できます。
- 他のテーブルの列にREFERENCES cities(name)を指定すると、他のテーブルが市の名前を持つことはできますが、州都の名前を持つことできません。この場合は良い回避策がありません。

継承の階層に対して実装されていないいくつかの機能は、宣言的パーティショニングでは実装されています。従来の継承がアプリケーションにとって有用であるかどうかを判断する際に十分注意してください。

5.11. テーブルのパーティショニング

PostgreSQLは基本的なテーブルのパーティショニング（分割）をサポートしています。この節では、データベース設計において、なぜそしてどのようにしてパーティショニングを実装するのかを解説します。

5.11.1. 概要

パーティショニングとは、論理的には一つの大きなテーブルであるものを、物理的により小さな部品に分割することを指します。パーティショニングによって得られる利点は以下のようにいくつかあります。

- 特定の条件下で問い合わせのパフォーマンスが劇的に向上することがあります。特にテーブル内のアクセスが集中する行の殆どが単一または少数のパーティションに存在している場合がそうです。パーティショニングはインデックスの先頭にある列の代わりになり、インデックスの大きさを小さくして、インデックスの頻繁に使われる部分がメモリに収まりやすくなるようにします。
- 問い合わせや更新が一つのパーティションの大部分にアクセスする場合、インデックスやランダムアクセスを使用してテーブル全体にまたがる読み取りをする代わりに、そのパーティションへの順次アクセスをすることでパフォーマンスを向上させることができます。
- 一括挿入や削除について、その要件をパーティションの設計に組み込んでいれば、それをパーティションの追加や削除で実現することが可能です。ALTER TABLE DETACH PARTITIONを実行する、あるいは個々のパーティションをDROP TABLEで削除するのは、一括の操作をするよりも遥かに高速です。これらのコマンドはまた、一括のDELETEで引き起こされるVACUUMのオーバーヘッドを完全に回避できます。
- 滅多に使用されないデータを安価で低速なストレージメディアに移行することができます。

この利益は通常、そうしなければテーブルが非常に大きくなる場合にのみ価値があります。テーブルがパーティショニングから利益を得られるかどうかの正確な分岐点はアプリケーションに依存しますが、重要なことはテーブルのサイズがデータベースサーバの物理メモリより大きいことです。

PostgreSQLにはパーティショニングについて以下の形式の組み込み機能があります。

範囲パーティショニング

テーブルはキー列またはキー列の集合で定義される「範囲」にパーティション分割され、異なるパーティションに割り当てられる値の範囲に重なりがないようになります。例えば、日付の範囲によってパーティション分割することもあるでしょうし、特定のビジネスオブジェクトの識別子の範囲によって分割することもあるでしょう。

リストパーティション

各パーティションに現れるキーの値を明示的に列挙することでテーブルをパーティションに分割します。

ハッシュパーティション

各パーティションに対して法と剰余を指定することでテーブルをパーティションに分割します。各パーティションは、パーティションキーのハッシュ値を指定された法で割った際に指定された剰余となる行を保持します。

アプリケーションで上記に列挙されていない他の形式のパーティショニングを使用する必要がある場合は、継承やUNION ALLなどの代替方式を代わりに使うことができます。そのような方式は柔軟性がありますが、組み込みの宣言的パーティショニングによるパフォーマンス上の利益の一部を享受できません。

5.11.2. 宣言的パーティショニング

PostgreSQLはテーブルをパーティションと呼ばれる部品に分割する方法を指定するための方法を提供しています。分割されたテーブルは*パーティションテーブル*と呼ばれます。方法の指定は*パーティション方式*と*パーティションキー*として使用される列あるいは式のリストからなります。

パーティションテーブルに挿入されるすべての行は、パーティションキーの値に基づいて*パーティション*の一つに振り向けられます。各パーティションはその*パーティション境界*によって定義されるデータの部分集合を持ちます。現在サポートされるパーティション方式には範囲、リスト、ハッシュがあります。

*サブパーティショニング*と呼ばれる方法を使って、パーティションそれ自体をパーティションテーブルとして定義することができます。パーティションには、他のパーティションとは別に独自のインデックス、制約、デフォルト値を定義できます。パーティションテーブルおよびパーティションの作成についての更なる詳細についてはCREATE TABLEを参照してください。

通常のテーブルをパーティションテーブルに変更する、およびその逆はできません。しかし、データのある通常のテーブルやパーティションテーブルをパーティションテーブルのパーティションとして追加する、あるいはパーティションテーブルからパーティションを削除し、それを独立したテーブルにすることは可能です。ATTACH PARTITIONおよびDETACH PARTITIONのサブコマンドについての詳細はALTER TABLEを参照してください。

個々のパーティションは継承を背景にパーティションテーブルに紐付けられていますが、宣言的パーティションテーブルもしくはそれらのパーティションでは継承の一般的な機能の一部（後述）を使用することはできません。例えば、パーティションテーブルのパーティションは、そのパーティションテーブル以外の親を持つことができませんし、また一般のテーブルはパーティションテーブルをその親にしてパーティションテーブルから継承することはできません。これはつまり、パーティションテーブルおよびそれらのパーティション

は一般のテーブルと継承によって繋がることができないということです。パーティションテーブルとそのパーティションを構成するパーティションの階層は継承の階層でもあるので、継承におけるすべての通常の規則が5.10で説明したとおりに適用されますが、いくつか例外があります。最も重要な例外を以下に示します。

- パーティションテーブルのCHECK制約とNOT NULL制約はいずれも必ずすべてのパーティションに継承されます。パーティションテーブルでNO INHERITの印を付けたCHECK制約を作ることはできません。
- ONLYを使ってパーティションテーブルについてのみ制約を追加または削除することは、パーティションが存在しない場合はサポートされます。パーティションが存在する時にパーティションテーブルについてのみ制約を追加または削除することはサポートされないため、一度パーティションが存在すれば、ONLYの使用はエラーになります。その代わりに、パーティション自身の制約を追加することや（親テーブルに存在しない場合）削除することが可能です。
- パーティションテーブルは直接データを所有することはないため、TRUNCATE ONLYをパーティションテーブルに対して使用しようとすると、必ずエラーが返されます。
- パーティションは親に存在しない列を持つことができません。パーティションをCREATE TABLEで作成する時に列を指定することはできませんし、作成後にALTER TABLEでパーティションに列を追加することもできません。テーブルをALTER TABLE ... ATTACH PARTITIONでパーティションとして追加できるのは、その列が完全に親と一致している場合のみです。
- 親テーブルの列に存在するNOT NULL制約をパーティションの列から削除することはできません。

通常のテーブルにない制限がいくつかありますが、パーティションを外部テーブルとすることができます。詳細はCREATE FOREIGN TABLEを参照してください。例えば、パーティションテーブルに挿入されるデータは外部テーブルのパーティションには振り向けられません。

行のパーティションキーを更新することは、その行がパーティション境界を満たす異なるパーティションに移動される原因となるかもしれません。

5.11.2.1. 例

大きなアイスクリーム会社のデータベースを構築している場合を考えましょう。その会社は毎日の最高気温、および各地域でのアイスクリームの売上を計測します。概念的には次のようなテーブルが必要です。

```
CREATE TABLE measurement (
    city_id         int not null,
    logdate         date not null,
    peaktemp        int,
    unitsales       int
);
```

このテーブルの主な利用目的は経営層向けにオンラインの報告書を作成することであるため、ほとんどの問い合わせは単に直前の週、月、四半期のデータにアクセスするだけであることがわかっています。保存すべき古いデータの量を削減するため、最近3年分のデータのみを残すことに決めました。各月のはじめに、最も古い月のデータを削除します。この場合、計測テーブルについての様々な要求のすべてを、パーティショニングを使って満たすことができます。

この場合に宣言的パーティショニングを使うには、以下の手順に従います。

1. PARTITION BY句を指定して、measurementテーブルをパーティションテーブルとして作成します。PARTITION BY句にはパーティション方式（この場合はRANGE）とパーティションキーとして使う列のリストを記述します。

```
CREATE TABLE measurement (
    city_id         int not null,
    logdate         date not null,
    peaktemp        int,
    unitsales       int
) PARTITION BY RANGE (logdate);
```

   望むなら、範囲パーティショニングのパーティションキーとして複数の列を使うようにすることもできます。もちろん、こうすると多くの場合、パーティションの数が増え、各パーティションの大きさは小さくなります。一方で、列の数を少なくすると、パーティショニングの基準の粒度が荒くなり、パーティションの数が少なくなります。パーティションテーブルにアクセスする問い合わせで、その条件がこれらの列の一部またはすべてを含む場合、より少ない数のパーティションを走査することになります。例えば、パーティションキーとして列lastnameとfirstnameを（この順で）使用して範囲パーティショニングをしたテーブルを考えてみてください。
2. パーティションを作成します。各パーティションの定義では、親のパーティショニング方式およびパーティションキーに対応する境界を指定しなければなりません。新しいパーティションの値が一つ以上の既存のパーティションの値と重なるような境界を指定するとエラーになることに注意してください。既存のおよびパーティションのどれにも当てはまらないデータを親テーブルに挿入するとエラーになります。この場合、適切なパーティションを手作業で追加しなければなりません。

   こうして作成されたパーティションは、すべての点においてPostgreSQLの通常のテーブル（あるいは場合によっては外部テーブル）と同じです。各パーティション毎に別々にテーブル空間や格納パラメータを指定することもできます。

   各パーティションについて、パーティションの境界条件を定義するテーブル制約を作成する必要はありません。その代わりに、指定した境界条件からパーティション制約が暗黙的に生成され、必要なときにはそれが参照されます。

```
CREATE TABLE measurement_y2006m02 PARTITION OF measurement
    FOR VALUES FROM ('2006-02-01') TO ('2006-03-01');

CREATE TABLE measurement_y2006m03 PARTITION OF measurement
    FOR VALUES FROM ('2006-03-01') TO ('2006-04-01');

...
CREATE TABLE measurement_y2007m11 PARTITION OF measurement
    FOR VALUES FROM ('2007-11-01') TO ('2007-12-01');

CREATE TABLE measurement_y2007m12 PARTITION OF measurement
    FOR VALUES FROM ('2007-12-01') TO ('2008-01-01')
    TABLESPACE fasttablespace;

CREATE TABLE measurement_y2008m01 PARTITION OF measurement
```

```
    FOR VALUES FROM ('2008-01-01') TO ('2008-02-01')
    WITH (parallel_workers = 4)
    TABLESPACE fasttablespace;
```

サブパーティショニングを実装するには、例えば以下のように、個々のパーティションを作成するコマンドでPARTITION BY句を指定してください。

```
CREATE TABLE measurement_y2006m02 PARTITION OF measurement
    FOR VALUES FROM ('2006-02-01') TO ('2006-03-01')
    PARTITION BY RANGE (peaktemp);
```

measurement_y2006m02のパーティションの作成後、measurementに挿入されるデータでmeasurement_y2006m02に振り向けられるもの（あるいはmeasurement_y2006m02に直接挿入されるデータでそのパーティション制約を満たしているもの）はすべて、peaktemp列に基いて更にその下のパーティションの一つにリダイレクトされます。指定するパーティションキーは親のパーティションキーと重なっても構いませんが、サブパーティションの境界を指定するときは、それが受け付けるデータの集合がパーティション自体の境界でできるものの部分集合を構成するように注意してください。システムは本当にそのようになっているかどうか、検査しようとしません。

3. キー列にインデックスを作成し、またその他のインデックスも必要に応じてパーティションテーブル上に作成します。（厳密に言えば、キー列のインデックスは必要なわけではありませんが、ほとんどの場合に役に立つでしょう。）これは各パーティション上に1つのインデックスを自動的に作ります。後から作成もしくは追加したパーティションにもインデックスが含まれます。

```
CREATE INDEX ON measurement (logdate);
```

4. postgresql.confで設定パラメータenable_partition_pruningが無効になっていないことを確認します。これが無効になっていると、問い合わせが期待通りには最適化されません。

上記の例では、毎月、新しいパーティションを作ることになりますから、必要なDDLを自動的に生成するスクリプトを作るのが賢明かもしれません。

5.11.2.2. パーティションの保守

最初にテーブルを定義した時に作成したパーティションの集合は、通常はそのまま静的に残ることを意図したものではありません。古いデータのパーティションを削除し、新しいデータの入った新しいパーティションを定期的に作成したいというのが普通です。パーティショニングのもっとも重要な利点の一つは、パーティショニングがなければ大変なことになるであろうこの作業を、大量のデータを物理的に動かすのではなく、パーティション構造を操作することにより、ほとんど一瞬にして実行できるという、まさにそのことなのです。

古いデータを削除する最も単純な方法は、次のように、不要になったパーティションを削除することです。

```
DROP TABLE measurement_y2006m02;
```

これはすべてのレコードを個別に削除する必要がないため、数百万行のレコードを非常に高速に削除できます。ただし、上記のコマンドは親テーブルについてACCESS EXCLUSIVEロックを取得する必要があることに注意してください。

別の方法で多くの場合に望ましいのは、パーティションテーブルからパーティションを削除する一方で、パーティションそれ自体はテーブルとしてアクセス可能なまま残すことです。

```
ALTER TABLE measurement DETACH PARTITION measurement_y2006m02;
```

こうすると、データを削除する前に、そのデータについて追加の操作が実行できます。例えば、COPY、pg_dumpや類似のツールを使ってデータのバックアップをする好機となることが多いでしょう。また、データを集計してより小さな形式にする、その他のデータ操作を実行する、レポート作成を実行するなどのための好機となるかもしれません。

同様に、新しいデータを扱うために新しいパーティションを追加することができます。上で元のパーティションを作ったのと全く同じように、パーティションテーブル内に空のパーティションを以下のように作成できます。

```
CREATE TABLE measurement_y2008m02 PARTITION OF measurement
    FOR VALUES FROM ('2008-02-01') TO ('2008-03-01')
    TABLESPACE fasttablespace;
```

別の方法として、新しいテーブルをパーティション構造の外部に作成し、その後でそれを適切なパーティションにする方が便利な場合もあります。こうすると、パーティションテーブル内でデータが見えるようになるより前に、データをロードし、確認し、変換することができます。

```
CREATE TABLE measurement_y2008m02
  (LIKE measurement INCLUDING DEFAULTS INCLUDING CONSTRAINTS)
  TABLESPACE fasttablespace;

ALTER TABLE measurement_y2008m02 ADD CONSTRAINT y2008m02
   CHECK ( logdate >= DATE '2008-02-01' AND logdate < DATE '2008-03-01' );

\copy measurement_y2008m02 from 'measurement_y2008m02'
-- possibly some other data preparation work

ALTER TABLE measurement ATTACH PARTITION measurement_y2008m02
    FOR VALUES FROM ('2008-02-01') TO ('2008-03-01' );
```

ATTACH PARTITIONコマンドを実行する前に、マッチするパーティション制約を記述するCHECK制約を、パーティションに追加するテーブルに作成することを推奨します。こうすることで、システムは、そうしなければ必要になる暗示的なパーティション制約を検証するための走査を、省略することができます。このような制約がなければ、そのパーティションにACCESS EXCLUSIVEロック、親テーブルにSHARE UPDATE EXCLUSIVEを保持したままで、パーティション制約を検証するためにテーブルを走査することになります。この制約はATTACH PARTITIONが終わった後に削除するのが望ましいかも知れません。

前述したとおり、パーティションテーブル上にインデックスを作成することが可能であり、それらは階層全体に自動で適用されます。既存のパーティションだけではなく将来作成されるパーティションもインデックス付けされるため、これはとても便利です。一つの制限は、そのようなパーティションのインデックスを作成する場合CONCURRENTLY句を使うことができません。長いロック時間を克服するためには、パーティションテーブルにCREATE INDEX ON ONLYを使うことが可能です。そのようなインデックスは無効とマークされ、パーティ

ションは適用するインデックスを自動で取得しません。パーティション上のインデックスはCONCURRENTLYを使用して個々に作成することができ、後からALTER INDEX .. ATTACH PARTITIONを使用して親のインデックスに*attached*できます。全てのパーティションに対してインデックスがアタッチされた時点で、親のインデックスは、自動で有効とマークされます。例を示します。

```
CREATE INDEX measurement_usls_idx ON ONLY measurement (unitsales);

CREATE INDEX measurement_usls_200602_idx
    ON measurement_y2006m02 (unitsales);
ALTER INDEX measurement_usls_idx
    ATTACH PARTITION measurement_usls_200602_idx;
...
```

この手法は、UNIQUEとPRIMARY KEY制約でも使用できます。制約が作成された際にインデックスは暗黙的に作成されます。例を示します。

```
ALTER TABLE ONLY measurement ADD UNIQUE (city_id, logdate);

ALTER TABLE measurement_y2006m02 ADD UNIQUE (city_id, logdate);
ALTER INDEX measurement_city_id_logdate_key
    ATTACH PARTITION measurement_y2006m02_city_id_logdate_key;
...
```

5.11.2.3. 制限事項

パーティションテーブルには以下の制限事項があります。

- すべてのパーティションにまたがる排他制約を作成する方法はありません。それぞれの末端のパーティションについて別々に制約をつけることしかできません。
- パーティションテーブル上の一意制約(したがって主キーも)は、すべてのパーティションキー列を含んでいなければなりません。PostgreSQLが各パーティションの一意性を個別に強制するために、この制限は存在します。
- BEFORE ROWトリガーは、どのパーティションが新しい行の最終目的地であるかを変更することはできません。
- 一時リレーションと永続的リレーションを同じパーティションツリーに混合することはできません。ですから、パーティション化されたテーブルが永続的なら、パーティションも永続的でなければなりません。同様にパーティション化されたテーブルが一時的なら、パーティションも一時的でなければなりません。一時リレーションを使う場合は、パーティションツリーのすべてのメンバーは同じセッションに由来しなければなりません。

5.11.3. 継承を使用した実装

組み込みの宣言的パーティショニングは、ほとんどの一般的な利用例に適合しますが、もっと柔軟な方式が便利な状況もあります。パーティショニングはテーブルの継承を使用して実装することも可能で、これは宣言的パーティショニングではサポートされない以下のような機能が利用できます。

- 宣言的パーティショニングの場合、パーティションは正確にパーティションテーブルと同じ列の集合を持たなければなりません。一方テーブルの継承では、子テーブルは親テーブルに存在しない追加の列を持つかもしれません。
- テーブルの継承では、複数の継承が可能です。
- 宣言的パーティショニングではリストパーティショニング、範囲パーティショニングとハッシュパーティショニングしかサポートされませんが、テーブルの継承ではユーザが選択した方法に従ってデータを分割することができます。(ただし、制約による除外が子テーブルを効果的に分離できない場合、問い合わせのパフォーマンスが悪くなるかもしれないことに注意してください。)
- 宣言的パーティショニングを使用すると、テーブルの継承を使用する場合に比べて、一部の走査でより強いロックが要求されます。例えば、パーティションテーブルからパーティションを削除するには、親テーブルのACCESS EXCLUSIVEロックを取得する必要がありますが、通常の継承の場合にはSHARE UPDATE EXCLUSIVEロックで十分です。

5.11.3.1. 例

上の非パーティションmeasurementテーブルを使用します。継承を使用したパーティショニングを実装するには、以下の手順に従います。

1. 「マスター」テーブルを作成します。すべての「子」テーブルはこれを継承します。このテーブルにはデータは含まれません。子テーブルに同じように適用されるのでなければ、このテーブルにチェック制約を定義しないでください。このテーブル上にインデックスや一意制約を定義することにも意味はありません。以下の例では、マスターテーブルは最初に定義したのと同じmeasurementテーブルです。
2. いくつかの「子」テーブルを作成し、それぞれマスターテーブルを継承するものにします。通常、これらのテーブルはマスターから継承したものに列を追加しません。宣言的パーティショニングの場合と同じく、これらのテーブルはすべての点で普通のPostgreSQLのテーブル(あるいは外部テーブル)と同じです。

```
CREATE TABLE measurement_y2006m02 () INHERITS (measurement);
CREATE TABLE measurement_y2006m03 () INHERITS (measurement);
...
CREATE TABLE measurement_y2007m11 () INHERITS (measurement);
CREATE TABLE measurement_y2007m12 () INHERITS (measurement);
CREATE TABLE measurement_y2008m01 () INHERITS (measurement);
```

3. 子テーブルに、重なり合わないテーブル制約を追加し、各テーブルに許されるキー値を定義します。

   典型的な例は次のようなものです。

```
CHECK ( x = 1 )
CHECK ( county IN ( 'Oxfordshire', 'Buckinghamshire', 'Warwickshire' ))
CHECK ( outletID >= 100 AND outletID < 200 )
```

   制約により、異なる子テーブルで許されるキー値に重なりがないことが保証されるようにします。よくある誤りは、次のような範囲制約を設定することです。

```
CHECK ( outletID BETWEEN 100 AND 200 )
```

```
CHECK ( outletID BETWEEN 200 AND 300 )
```

キー値200がどちらの子テーブルに属するか明らかではないため、これは誤っています。

その代わりに以下のように子テーブルを作る方が良いでしょう。

```
CREATE TABLE measurement_y2006m02 (
    CHECK ( logdate >= DATE '2006-02-01' AND logdate < DATE '2006-03-01' )
) INHERITS (measurement);

CREATE TABLE measurement_y2006m03 (
    CHECK ( logdate >= DATE '2006-03-01' AND logdate < DATE '2006-04-01' )
) INHERITS (measurement);

...
CREATE TABLE measurement_y2007m11 (
    CHECK ( logdate >= DATE '2007-11-01' AND logdate < DATE '2007-12-01' )
) INHERITS (measurement);

CREATE TABLE measurement_y2007m12 (
    CHECK ( logdate >= DATE '2007-12-01' AND logdate < DATE '2008-01-01' )
) INHERITS (measurement);

CREATE TABLE measurement_y2008m01 (
    CHECK ( logdate >= DATE '2008-01-01' AND logdate < DATE '2008-02-01' )
) INHERITS (measurement);
```

4. 各子テーブルについて、キー列にインデックスを作成し、またその他のインデックスも必要に応じて作成します。

```
CREATE INDEX measurement_y2006m02_logdate ON measurement_y2006m02 (logdate);
CREATE INDEX measurement_y2006m03_logdate ON measurement_y2006m03 (logdate);
CREATE INDEX measurement_y2007m11_logdate ON measurement_y2007m11 (logdate);
CREATE INDEX measurement_y2007m12_logdate ON measurement_y2007m12 (logdate);
CREATE INDEX measurement_y2008m01_logdate ON measurement_y2008m01 (logdate);
```

5. アプリケーションでINSERT INTO measurement ...を実行することができ、そのときにデータが適切な子テーブルにリダイレクトされることが望ましいです。マスターテーブルに適当なトリガー関数を追加することでそのような設定にすることができます。データが最後の子テーブルにしか追加されないなら、次のような非常に単純なトリガー関数を使うことができます。

```
CREATE OR REPLACE FUNCTION measurement_insert_trigger()
RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO measurement_y2008m01 VALUES (NEW.*);
    RETURN NULL;
END;
$$
```

```
LANGUAGE plpgsql;
```

関数を作成した後で、このトリガ関数を呼ぶトリガを作成します。

```
CREATE TRIGGER insert_measurement_trigger
    BEFORE INSERT ON measurement
    FOR EACH ROW EXECUTE FUNCTION measurement_insert_trigger();
```

トリガが常に現在の子テーブルを指すようにするためには、毎月、トリガ関数を再定義しなくてはいけません。しかし、トリガ定義を更新する必要はありません。

データを挿入したら、サーバが行を追加すべき子テーブルを自動的に決定するようにしたいかもしれません。これは以下のようなもっと複雑なトリガ関数を作成することにより可能です。

```
CREATE OR REPLACE FUNCTION measurement_insert_trigger()
RETURNS TRIGGER AS $$
BEGIN
    IF ( NEW.logdate >= DATE '2006-02-01' AND
         NEW.logdate < DATE '2006-03-01' ) THEN
        INSERT INTO measurement_y2006m02 VALUES (NEW.*);
    ELSIF ( NEW.logdate >= DATE '2006-03-01' AND
            NEW.logdate < DATE '2006-04-01' ) THEN
        INSERT INTO measurement_y2006m03 VALUES (NEW.*);
    ...
    ELSIF ( NEW.logdate >= DATE '2008-01-01' AND
            NEW.logdate < DATE '2008-02-01' ) THEN
        INSERT INTO measurement_y2008m01 VALUES (NEW.*);
    ELSE
        RAISE EXCEPTION 'Date out of range.  Fix the measurement_insert_trigger() function!';
    END IF;
    RETURN NULL;
END;
$$
```

```
LANGUAGE plpgsql;
```

トリガ定義は前と同じです。それぞれのIFテストを子テーブルのCHECK制約と正確に一致させなければならないことに注意してください。

この関数は単一月の場合より複雑になりますが、頻繁に更新する必要はありません。なぜなら条件分岐を前もって追加しておくことが可能だからです。

> **注記**
>
> 実際には、ほとんどの挿入が一番新しい子テーブルに入る場合は、その子を最初に検査することが最善です。簡単にするため、この例でのほかの部分と同じ順番でのトリガのテストを示しました。

挿入を適切な子テーブルにリダイレクトする別の方法は、マスターテーブルにトリガーではなくルールを設定することです。例えば次のようにします。

```
CREATE RULE measurement_insert_y2006m02 AS
ON INSERT TO measurement WHERE
    ( logdate >= DATE '2006-02-01' AND logdate < DATE '2006-03-01' )
DO INSTEAD
    INSERT INTO measurement_y2006m02 VALUES (NEW.*);
...
CREATE RULE measurement_insert_y2008m01 AS
ON INSERT TO measurement WHERE
    ( logdate >= DATE '2008-01-01' AND logdate < DATE '2008-02-01' )
DO INSTEAD
    INSERT INTO measurement_y2008m01 VALUES (NEW.*);
```

ルールはトリガーに比べるとかなり大きなオーバーヘッドがありますが、このオーバーヘッドは一つの問い合わせに対して一度だけで行ごとではないので、この方法にも一括挿入の状況では利点があります。ただし、ほとんどの場合はトリガーを使う方法の方が良いパフォーマンスが得られます。

COPYはルールを無視することに注意してください。データの挿入にCOPYを使いたい場合は、マスターではなく正しい子テーブルにコピーする必要があります。トリガーであればCOPYでも起動されるので、トリガーを使う方法であれば通常通りに使用することができます。

ルールを使う方法のもう一つの欠点は、ルールの集合が挿入日付に対応しきれていない場合に、強制的にエラーにする簡単な方法がないことです。この場合、データは警告などを出すことなくマスターテーブルに入ります。

6. 設定パラメータconstraint_exclusionがpostgresql.confで無効にされないようにしてください。他の子テーブルが不要にアクセスされるかもしれません。

以上のように、複雑なテーブルの階層はたくさんのDDLが必要となります。上記の例では、毎月新しい子テーブルを作成することになりますが、必要となるDDLを自動的に生成するスクリプトを書くのが賢明です。

5.11.3.2. 継承パーティショニングの保守

古いデータを高速に削除するには、不要になった子テーブルを単に削除します。

```
DROP TABLE measurement_y2006m02;
```

子テーブルを継承階層テーブルから削除するものの、それ自体をテーブルとしてアクセスできるようにするには、次のようにします。

```
ALTER TABLE measurement_y2006m02 NO INHERIT measurement;
```

新しいデータを扱う新しい子テーブルを追加するには、上で最初の子テーブルを作成したときと同じように空の子テーブルを作成します。

```
CREATE TABLE measurement_y2008m02 (
    CHECK ( logdate >= DATE '2008-02-01' AND logdate < DATE '2008-03-01' )
) INHERITS (measurement);
```

あるいは、新たな子テーブルをテーブル階層に追加する前に作成してデータ投入したい場合もあるでしょう。これは、親テーブルのクエリから見えるようになる前にデータのロード、確認、変換できるでしょう。

```
CREATE TABLE measurement_y2008m02
  (LIKE measurement INCLUDING DEFAULTS INCLUDING CONSTRAINTS);
ALTER TABLE measurement_y2008m02 ADD CONSTRAINT y2008m02
   CHECK ( logdate >= DATE '2008-02-01' AND logdate < DATE '2008-03-01' );
\copy measurement_y2008m02 from 'measurement_y2008m02'

-- その他のデータ準備操作を行うこともあります。
ALTER TABLE measurement_y2008m02 INHERIT measurement;
```

5.11.3.3. 注意事項

継承を使用して実装したパーティショニングには以下の注意事項があります。

- すべてのCHECK制約が相互に排他的であることを自動的に確認する手段はありません。各子テーブルを手作業で作成するよりも、子テーブルを生成し、関連オブジェクトを作成、更新するコードを作成するのが安全でしょう。
- インデックスと外部キー制約は継承上の子ではなく、単一テーブルに適用されます。したがってそれらは警告に気を付ける必要があります。
- ここで示した方法は、行のキー列の値が変わらないか、あるいは、少なくとも他のパーティションへの移動が必要になるような変更はないということを前提としています。そのような変更をしようとするUPDATEはCHECK制約のためにエラーになります。このような場合を処理できる必要があるなら、子テーブルに適切なUPDATEトリガーを設定することもできますが、構造の管理がずっと複雑になります。

- 手作業でVACUUMあるいはANALYZEコマンドを実行している場合、それを個々の子テーブルに対して実行する必要があることを忘れないで下さい。次のようなコマンドはマスターテーブルしか処理しません。

```
ANALYZE measurement;
```

- ON CONFLICT句のあるINSERT文は恐らく期待通りには動作しないでしょう。ON CONFLICTの動作は対象となる指定リレーション上での一意制約違反の場合にのみ発生するもので、その子リレーションの場合には発生しないからです。

- アプリケーションがパーティショニングのスキームについて明示的に意識しているのでなければ、トリガーまたはルールで行を適切な子テーブルに振り向ける必要があります。トリガーを書くのは複雑であり、また宣言的パーティショニングによって内部的に実行されるタプルの振り向けよりずっと遅いでしょう。

5.11.4. パーティション除去

パーティション除去は、宣言的パーティショニングテーブルに対するパフォーマンスを向上させる問い合わせの最適化技術です。例えば、

```
SET enable_partition_pruning = on;                 -- the default
SELECT count(*) FROM measurement WHERE logdate >= DATE '2008-01-01';
```

パーティション除去がなければ、上記の問い合わせはmeasurementテーブルの各パーティションをスキャンするでしょう。パーティション除去が有効になっているとき、プランナはそれぞれのパーティションの定義を検証し、パーティションが問い合わせのWHEREに一致する行を含んでいないためにスキャンされる必要が無いことを証明します。プランナはこれを証明すると、問い合わせ計画からそのパーティションを除外(除去)します。

EXPLAINコマンドと設定パラメータenable_partition_pruning を使用することによって、パーティションの除去をした計画とそうでない計画の違いを明らかにすることを可能とします。この種類のテーブル設定に対する典型的な最適化されない計画は以下のようになります。

```
SET enable_partition_pruning = off;
EXPLAIN SELECT count(*) FROM measurement WHERE logdate >= DATE '2008-01-01';
                                     QUERY PLAN
-------------------------------------------------------------------------------
 Aggregate  (cost=188.76..188.77 rows=1 width=8)
   ->  Append  (cost=0.00..181.05 rows=3085 width=0)
         ->  Seq Scan on measurement_y2006m02  (cost=0.00..33.12 rows=617 width=0)
               Filter: (logdate >= '2008-01-01'::date)
         ->  Seq Scan on measurement_y2006m03  (cost=0.00..33.12 rows=617 width=0)
               Filter: (logdate >= '2008-01-01'::date)
...
         ->  Seq Scan on measurement_y2007m11  (cost=0.00..33.12 rows=617 width=0)
               Filter: (logdate >= '2008-01-01'::date)
         ->  Seq Scan on measurement_y2007m12  (cost=0.00..33.12 rows=617 width=0)
               Filter: (logdate >= '2008-01-01'::date)
         ->  Seq Scan on measurement_y2008m01  (cost=0.00..33.12 rows=617 width=0)
```

```
               Filter: (logdate >= '2008-01-01'::date)
```

一部のパーティション、もしくはすべてのパーティションで、テーブル全体に対するシーケンシャルスキャンではなく、インデックススキャンが使用される可能性があります。しかしここで重要なことは、この問い合わせに対する回答のために古いパーティションをスキャンする必要はまったく無いということです。パーティション除去を有効にしたとき、同じ回答を返す計画で、大幅に安価なものを得ることができます。

```
SET enable_partition_pruning = on;
EXPLAIN SELECT count(*) FROM measurement WHERE logdate >= DATE '2008-01-01';
                                     QUERY PLAN
------------------------------------------------------------------------------------
 Aggregate  (cost=37.75..37.76 rows=1 width=8)
   ->  Seq Scan on measurement_y2008m01  (cost=0.00..33.12 rows=617 width=0)
         Filter: (logdate >= '2008-01-01'::date)
```

パーティション除去はパーティションキーによって暗黙的に定義された制約のみで動作し、インデックスの有無では動作しないことに注意してください。よってキー列のインデックスを定義することは必要ではありません。あるパーティションでインデックスが必要かどうかは、パーティションをスキャンする問い合わせが通常はパーティションの大部分をスキャンするのか、あるいは小さな部分をスキャンするのかによります。インデックスは後者において役立ちますが、前者では役立ちません。

パーティション除去は与えられた問い合わせの計画時だけでなく、問い合わせの実行時にも可能です。問い合わせの計画時、句が値のわからない式を含むときにより多くのパーティションを除去できるため便利です。例えば、PREPARE文中に定義されたパラメータや、副問い合わせから取得される値の利用、ネステッドループ結合の内側でパラメータ化された値の利用です。実行中のパーティション除去は、次のいずれかの時点で可能です。

- 問い合わせ計画の初期化時。パーティション除去は、パラメータの値が分かる実行の初期化段階時に可能です。この段階で除去されたパーティションは、問い合わせのEXPLAINやEXPLAIN ANALYZE中に姿を見せることはないでしょう。EXPLAIN出力中に「Subplans removed」プロパティを観察することによってこの段階で削除されるパーティションの数を特定することが可能です。
- 問い合わせ計画の実行時。パーティション除去では実際に問い合わせの実行をする際にのみ分かる値を用いてパーティションを取り除くことも同様に可能でしょう。これは、副問い合わせからの値やネステッドループ結合でパラメータ化されたような実行時のパラメータからの値を含みます。それらのパラメータの値は問い合わせの実行時に何回も変わるかもしれないため、パーティション除去はパーティション除去に使われる実行パラメータの値が変るたびに行われます。この段階で除去されたパーティションを特定するには、EXPLAIN ANALYZE出力中のloopsプロパティの慎重な調査が必要です。異なるパーティションに対応するサブプランは、それぞれ実行時に除去された回数に応じて異なる値を持っているかもしれません。毎回パーティションが除去される場合、一部は(never executed)と表示されるでしょう。

パーティション除去はenable_partition_pruning設定を使うことにより無効化できます。

> **注記**
>
> 実行時のパーティション除去は現在AppendとMergeAppendノードタイプに対してのみ発生します。ModifyTableノードタイプについてはまだ実装されていません。これはPostgreSQLの将来のリリースで変更される可能性があります。

5.11.5. パーティショニングと制約による除外

制約による除外はパーティション除去と同様に問い合わせ最適化技術です。主に従来の継承方法を使用して実装されたパーティショニングのために使用されると同時に 宣言的パーティショニングを含む他の目的に使うことができます。

各テーブルの名前の付いたCHECK制約を使用すること（一方でパーティション除去は宣言的パーティショニングの場合にのみ存在するテーブルのパーティション境界を使用します）を除いて、制約による除外はパーティション除去と極めて同様な方法で動作します。その他の違いは、制約による除外は計画時にのみ適用され実行時にパーティションの削除を試しません。

制約による除外はCHECK制約を使用しているためパーティション除去と比べて遅いですが、ときどき利点として使うことができます。なぜなら、内部のパーティション境界に加えて宣言的パーティションテーブルにも制約は定義することができるため、制約による除外は問い合わせ計画から追加のパーティションを取り除けるかもしれません。

実のところ、constraint_exclusionのデフォルト（かつ推奨）の設定は、onでもoffでもなく、partitionという中間の設定です。これによりこの技法は、継承パーティションテーブルに対して動作することになる問い合わせのみに適用されるようになります。on設定にすると、プランナは、効果のなさそうな単純な問い合わせを含め、すべての問い合わせでCHECK制約を検証するようになります。

制約による除外には以下の注意事項が適用されます。

- 問い合わせの実行中にも適用できるパーティション除去とは違い、制約による除外は問い合わせ計画時にのみ適用されます。
- 制約による除外は問い合わせのWHERE句が定数(または外部から供給されたパラメータ)を含んでいたときにのみ動作します。例えば、CURRENT_TIMESTAMPのような非immutable関数に対する比較は、関数の結果値がどの子テーブルに該当するかを実行時にプランナが知ることが出来ないため、最適化できません。
- パーティション制約を簡単にしておいてください。そうしないとプランナは、子テーブルを使う必要がないことを立証できないでしょう。前述の例で示したとおり、リスト分割のために簡単な等号条件を使用してください。また範囲分割のために簡単な範囲テストを使用してください。手っ取り早い良い方法は、パーティショニングの制約がパーティション列とB-treeインデックス作成可能な演算子を用いた定数の比較のみを含んでいることです。なぜならパーティションキーにはB-treeでインデックス可能な列だけが使用できるからです。
- 親テーブルのすべての子テーブルのすべての制約は、制約による除外で試験されます。よって子テーブルの数が多くなれば問い合わせ計画の時間がかなり増加します。そのため、従来の継承を基にしたパーティショニングはおそらく100個までの子でうまく動作します。何千もの子テーブルを使用することは避けてください。

5.11.6. 宣言的パーティショニングのベストプラクティス

不十分な設計によってクエリ計画および実行性能に負の影響がでる可能性があるため テーブルのパーティション方法の選択は注意して行う必要があります。

最も重要な設計の決定の一つは、データを分割するための一つまたは複数の列です。大抵最適な選択は、パーティションテーブル上で実行されるクエリのWHERE句に最もよく現れる列または列の組み合わせによっ

て分割することです。パーティションキーと一致し互換性があるWHERE句の項目は、不要なパーティションを取り除く為に使うことができます。しかしながら、PRIMARY KEYもしくはUNIQUE制約の条件により、他の決定を強いられるかもしれません。不要なデータの削除も同様にパーティショニング戦略を計画する際に考えるべき要素です。すべてのパーティションはとても早くデタッチすることができるため、一度に削除される全てのデータが単一のパーティション中に設置されるようにパーティション戦略を設計することが有益かもしれません。

テーブルを分割するパーティションの目標数を選択することもまた需要な決定です。十分なパーティションがないとインデックスは大きくなりデータの局所性が貧しいままであるかもしれず、キャッシュヒット率が低い結果となる可能性があります。しかしながら非常に多くのパーティションにテーブルを分割することもまた問題の原因となります。非常に多くのパーティションは、クエリの計画時間が長くなり、クエリの計画および実行の両方の際にメモリ消費が高くなることを意味します。テーブルを分割する方法を選択するとき、将来に起こる変化を考慮することもまた重要です。例えば、顧客毎に一つのパーティションを用意することを選択し、現在大規模な顧客が少数いる場合、数年以内に小規模な顧客を多数代わりに見つける可能性を含めて考慮します。この場合、LISTによって分割しデータの分割が実用的な数以上に顧客の数が増加しないことを期待するより、HASHによって分割し妥当なパーティション数にすることを選択する方が良いかもしれません。

サブパーティショニングは、他のパーティションより巨大になると想定されるパーティションを更に分割するために役立ちますが、過度なサブパーティショニングは大量のパーティションを容易に引き起こし、前の段落で言及したのと同じ問題の原因となります。

クエリの計画および実行時のパーティショニングのオーバーヘッドを考慮することもまた重要です。典型的なクエリではクエリプランナが少数のパーティションを除いて残り全てのパーティションを除外できるという前提に立てば、クエリプランナは通常最大数千パーティションのパーティション階層を適切に操作することができます。プランナがパーティション除去を行った後に多くのパーティションが残るほど、計画時間は長くなりメモリ消費は高くなります。これはUPDATEとDELETEコマンドに特に当てはまります。大量のパーティションを持っていることについて考慮するもうひとつの理由は、特に多くのセッションが大量のパーティションを参照する場合、ある期間にサーバのメモリ消費が著しく増加するかもしれないことです。その理由は、各パーティションは参照される各セッションのローカルメモリにメタデータを読み込む必要があるためです。

データウェアハウスタイプのワークロードでは、OLTPタイプのワークロードより大量のパーティションを使用するのが当然です。通常、データウェアハウスでは処理時間の大半をクエリ実行に費やすため、クエリ計画時間はあまり問題になりません。2種類のワークロードのいずれかでも、大量のデータを再パーティショニングすることは非常に遅いため、初期に適切な決定を下すことが重要です。計画したワークロードのシミュレーションは、パーティショニング戦略を最適化するためにしばしば役立ちます。多数のパーティションがより少数のパーティションより優れていることや、少数のパーティションが多数のパーティションより優れていることを前提としないでください。

5.12. 外部データ

PostgreSQLはSQL/MED仕様を部分的に実装しており、PostgreSQLの外部にあるデータに対して標準的なSQLクエリでアクセスできます。このようなデータは*外部データ*と呼ばれます。(この用語をデータベース内の制約である外部キーと混同しないように気をつけてください。)

外部データは*外部データラッパ*の手助けによりアクセスされます。外部データラッパは外部データソースと通信できるライブラリであり、接続確立やデータ取得といった詳細を隠蔽します。contribモジュールとして、いくつかの外部データラッパが利用できます; 付録Fを参照してください。その他の種類の外部データラッパ

は外部製品として見つかるでしょう。既存の外部データラッパがあなたの用途に合わない場合、独自のラッパを書くことができます。第56章を参照してください。

外部データにアクセスするには、特定の外部データソースへの接続方法をそれを支える外部データラッパが使用するオプションの組み合わせによって定義する*外部サーバ*オブジェクトを作成する必要があります。その後、外部データの構造を定義する*外部テーブル*を少なくともひとつ作成する必要があります。外部テーブルは通常のテーブルと同様にクエリの中で使用できますが、外部テーブルはPostgreSQLサーバには格納領域を持ちません。外部テーブルが使われるたびに、PostgreSQLは外部ソースからデータを取得することや、更新コマンドの場合には外部ソースへデータを送信することを外部データラッパに依頼します。

外部データへのアクセスは外部データソースからの認証を必要とする場合があります。この情報は、現在のPostgreSQLロールに基づいてユーザ名やパスワードといった追加のデータを提供することができる*ユーザマッピング*によって提供することができます。

追加情報は、CREATE FOREIGN DATA WRAPPER、CREATE SERVER、CREATE USER MAPPING、CREATE FOREIGN TABLE、IMPORT FOREIGN SCHEMAを参照してください。

5.13. その他のデータベースオブジェクト

テーブルにはデータが保持されていますので、リレーショナルデータベース構造ではテーブルが中心オブジェクトとなります。しかし、データベースにはテーブルの他にもオブジェクトが存在します。様々なオブジェクトを作成して、データの使用および管理をより効果的に行うことができます。本章ではこれらのオブジェクトについては説明しませんが、どのようなものがあるかをここに列挙します。

- ビュー
- 関数、プロシージャおよび演算子
- データ型およびドメイン
- トリガおよび書き換えルール

これらのトピックに関する詳細な情報はパート Vにあります。

5.14. 依存関係の追跡

外部キー制約や、ビュー、トリガ、関数などを使ったテーブルが多数含まれるような複雑なデータベース構造を作成すると、ユーザはそれらのオブジェクト間の暗黙的な依存関係のネットワークも作成していることになります。例えば、外部キー制約を持つテーブルは、参照するテーブルに依存しています。

データベース構造全体の整合性を保つため、PostgreSQLは、他のオブジェクトと依存関係にあるオブジェクトの削除を許可しません。例えば、5.4.5で検討したproductsテーブルを削除しようとしても、ordersテーブルがこのテーブルに依存しているので、以下のようなエラーメッセージが現れます。

```
DROP TABLE products;
```

```
ERROR:  cannot drop table products because other objects depend on it
DETAIL:  constraint orders_product_no_fkey on table orders depends on table products
HINT:  Use DROP ... CASCADE to drop the dependent objects too.
```

エラーメッセージには役に立つヒントが含まれています。以下のようにすると、依存する全てのオブジェクトを1つずつ削除する手間を省けます。

```
DROP TABLE products CASCADE;
```

これで、全ての依存オブジェクトが削除され、それらに依存するいかなるオブジェクトも削除されます。この場合、ordersテーブルは削除されずに外部キー制約のみが削除されます。外部キー制約に依存するものが何もないので、処理がそこで停止します。(DROP ... CASCADEが何を行うかを知りたい場合は、CASCADEを指定せずにDROPを実行してDETAIL出力を読んでください)。

PostgreSQLでは、ほぼ全てのDROPコマンドにCASCADEを指定することができます。もちろん、どのような依存関係が存在するかは、オブジェクトの種類によって異なります。また、CASCADEではなくRESTRICTと記述することもできます。これは、他のオブジェクトが依存しているオブジェクトの削除を禁止するというデフォルトの振舞いになります。

> **注記**
>
> 標準SQLでは、DROPコマンドでRESTRICTまたはCASCADEのいずれかを指定する必要があります。実際にこの決まり通りのデータベースシステムはありませんが、デフォルトがRESTRICTであるか、CASCADEであるかは、システムによって異なります。

DROPコマンドで複数のオブジェクトを羅列した場合、CASCADEは、指定されたグループの外部に依存関係が存在する時のみ要求されます。例えば、DROP TABLE tab1, tab2と記述したとき、tab2からtab1への外部キー参照の存在は、CASCADEの指定がコマンド成功に必要とされるということを意味しません。

ユーザ定義関数について、PostgreSQLは引数や結果の型など、関数の外部に可視な属性に関連した依存性については追跡しますが、関数の実体を検査することによってしかわからない依存性は追跡しません。例えば以下の状況を考えてみます。

```
CREATE TYPE rainbow AS ENUM ('red', 'orange', 'yellow',
                             'green', 'blue', 'purple');

CREATE TABLE my_colors (color rainbow, note text);

CREATE FUNCTION get_color_note (rainbow) RETURNS text AS
  'SELECT note FROM my_colors WHERE color = $1'
  LANGUAGE SQL;
```

(SQL言語による関数についての説明は37.5を参照してください。) PostgreSQLは関数get_color_noteが型rainbowに依存することは認識します。例えば、その型を削除すると、関数の引数の型が定義されなくなるため、関数の削除も強制されます。しかし、PostgreSQLはget_color_noteがテーブルmy_colorsに依存するとは考えません。従って、そのテーブルが削除されても関数は削除されません。この方法には不利な点もありますが、同時に利益もあります。テーブルがない状態で関数を実行すればエラーを引き起こしますが、そ

れでも関数はある意味で、有効な状態になっています。そのため、同じ名前の新しいテーブルを作成することで、関数を再び動作させることができます。

第6章 データ操作

前章では、データを保持するためのテーブルやその他の構造の作成方法について説明しました。今度は、テーブルにデータを挿入してみましょう。本章では、テーブルのデータの挿入、更新、削除の方法について説明します。次章ではいよいよ、ずっと見つけられなかったデータをデータベースから抽出する方法について説明します。

6.1. データの挿入

テーブルは、作成時にはデータを含んでいません。データベースを利用価値のあるものにするには、まずデータを挿入する必要があります。概念的には、データは一度に1行ずつ挿入されます。もちろんユーザは複数行を挿入することもできますが、1行未満を挿入することはできません。列の値が一部しかわかっていない場合でも、1行全体を作成しなければなりません。

新規の行を作成するには、INSERTコマンドを使用します。このコマンドは、テーブル名と列の値を必要とします。例えば、第5章のproductsテーブルの例で考えてみましょう。

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric
);
```

行を挿入するためのコマンド例は以下のようになります。

```
INSERT INTO products VALUES (1, 'Cheese', 9.99);
```

データ値は、テーブル内で列が存在する順序に従ってカンマで区切って列挙されています。通常、データ値はリテラル（定数）ですが、スカラ式も使用できます。

上記の構文には、テーブル内の列の順序を知っていなければならないという欠点があります。これを避けるには、列を明示的に列挙する方法があります。例えば、以下の2つのどちらのコマンドでも上記のコマンドと同等の効果が得られます。

```
INSERT INTO products (product_no, name, price) VALUES (1, 'Cheese', 9.99);
INSERT INTO products (name, price, product_no) VALUES ('Cheese', 9.99, 1);
```

多くのユーザは常に列名を列挙する方法が優れていると考えています。

値がわからない列については、省略することができます。省略した列には、デフォルト値が挿入されます。以下に例を示します。

```
INSERT INTO products (product_no, name) VALUES (1, 'Cheese');
INSERT INTO products VALUES (1, 'Cheese');
```

後者はPostgreSQLの拡張機能です。これによって、列には左から順に指定されただけの値が挿入され、残りにはデフォルト値が挿入されます。

明確にするため、列ごと、あるいは行全体についてデフォルト値を明示的に要求することもできます。

```
INSERT INTO products (product_no, name, price) VALUES (1, 'Cheese', DEFAULT);
INSERT INTO products DEFAULT VALUES;
```

単一コマンドで複数行を挿入することができます。

```
INSERT INTO products (product_no, name, price) VALUES
    (1, 'Cheese', 9.99),
    (2, 'Bread', 1.99),
    (3, 'Milk', 2.99);
```

また、問い合わせの結果（0行かも、1行かも、複数行かもしれない）を挿入することもできます。

```
INSERT INTO products (product_no, name, price)
  SELECT product_no, name, price FROM new_products
    WHERE release_date = 'today';
```

これにより、挿入する行を計算するためにSQLの問い合わせ機構（第7章）の全機能が提供されます。

> **ヒント**
>
> 一度に大量のデータを挿入する場合はCOPYコマンドの使用を検討してください。INSERTコマンドほどの柔軟性はありませんが、より効率的です。大量のデータをロードする性能を向上することについて、詳細は14.4を参照してください。

6.2. データの更新

既にデータベースに入っているデータを変更することを「更新(update)する」と言います。個別の行、テーブル内の全ての行、あるいは全ての行のサブセットを更新することができます。各列は、他の列に影響を及ぼすことなく個別に更新することができます。

既存の行の更新を行うにはUPDATEコマンドを使用してください。その際には3つの情報が必要となります。

1. 更新するテーブルと列の名前
2. 更新後の列の値
3. 更新する行

第5章で説明した、一般にSQLでは行に対して一意のIDを指定しないことを思い出してください。従って、どの行を更新するかを直接指定することができない場合があります。その代わりに、更新される行が満たすべき条件を指定します。テーブルに主キーを設定している場合に限り（ユーザが宣言したのかどうかには関係なく）、主キーと一致する条件を選択することで確実に個別の行を指定することができます。グラフィカルなデータベースアクセスツールは、この方法を使用して行を個別に更新することを可能にしています。

例えば、値段が5である全ての商品の値段を10に更新するには、以下のコマンドを使用します。

```
UPDATE products SET price = 10 WHERE price = 5;
```

これによって更新される行の数はゼロであるかもしれませんし、1つ、あるいは多数であるかもしれません。一致する行がない条件を指定して更新しようとしてもエラーにはなりません。

では、上記のコマンドの詳細を見てみましょう。最初はUPDATEキーワードで、これにテーブル名が続きます。いつも通り、テーブル名はスキーマで修飾することもできます。修飾しない場合はパス内から検索されます。次にSETキーワードがあり、これに列名、等号、そして更新後の列値が続きます。更新後の列値は、定数だけでなく任意のスカラ式で表すことができます。例えば、全ての商品の価格を10%上げるには以下のようにします。

```
UPDATE products SET price = price * 1.10;
```

このように、新しい値を表す式で行の中の古い値を参照することもできます。ここでは、WHERE句を省略しました。WHERE句を省略すると、テーブル内の全ての行が更新されます。省略しない場合は、WHERE条件に適合する行のみが更新されます。SET句内の等号が代入を表すのに対し、WHERE句内の等号は比較を表すことに注意してください。ただし、これによって曖昧さが生じることはありません。もちろん、必ずしもWHERE条件が等式でなければならないということはありません。その他にも様々な演算子を使用することができます（第9章を参照）。ただし、式の評価結果は論理値でなければなりません。

UPDATEコマンドのSET句に複数の代入式を列挙して、複数の列を更新することもできます。例を示します。

```
UPDATE mytable SET a = 5, b = 3, c = 1 WHERE a > 0;
```

6.3. データの削除

これまで、テーブルにデータを追加する方法と、データを変更する方法について説明してきました。残っているのは不要になったデータを削除する方法です。データの追加が行単位でしか行えないのと同様、削除の場合も、行全体をテーブルから削除するしかありません。前節で、SQLでは個々の行を直接指定する方法がないということを説明しました。ですから行の削除の場合も、削除対象となる行の条件を指定することでしかできません。テーブルに主キーが設定されている場合は、その行を正確に指定できます。しかし、条件を満たす複数の行、あるいは、テーブル内の全ての行を一度に削除することもできます。

行を削除するには、DELETEコマンドを使用します。構文はUPDATEコマンドによく似ています。例えば、productsテーブルから価格が10である全ての行を削除するには以下のようにします。

```
DELETE FROM products WHERE price = 10;
```

単に次のようにすると、テーブル内の全ての行が削除されますので注意してください！

```
DELETE FROM products;
```

プログラマに対する警告です。

6.4. 更新された行のデータを返す

行が更新されるときに、その行のデータを取得できると便利なことがあります。INSERT、UPDATE、DELETEの各コマンドは、いずれもオプションのRETURNING句によりそれが可能となっています。RETURNINGを使うことで、行を取得するために余分なデータベースへの問い合わせを行うことを避けられ、それ以外の方法で更新された行を確実に特定するのが難しい場合には、これは特に貴重です。

RETURNING句で使用できる項目はSELECTコマンドの出力リスト(7.3参照)と同じです。コマンドの対象となっているテーブルの列の名前、あるいはそれらの列を使った値の式を入れることができます。よく使われる省略記法はRETURNING *で、これは対象テーブルのすべての列を順に返します。

INSERTでは、RETURNINGで利用できるデータは、挿入された通りの行です。単純な挿入では、クライアントが提供したデータを単に繰り返すだけになりますから、あまり役には立ちません。しかし、計算されたデフォルト値に依存しているときは、これは非常に便利なことがあります。例えばserialの列を使って一意識別子を提供している場合、以下のようにRETURNINGによって、新しい行に割り当てられたIDを返すことができます。

```
CREATE TABLE users (firstname text, lastname text, id serial primary key);

INSERT INTO users (firstname, lastname) VALUES ('Joe', 'Cool') RETURNING id;
```

また、RETURNING句はINSERT ... SELECTでも非常に役に立ちます。

UPDATEでは、RETURNINGで利用できるデータは、更新された行の新しい内容です。例を示します。

```
UPDATE products SET price = price * 1.10
  WHERE price <= 99.99
  RETURNING name, price AS new_price;
```

DELETEでは、RETURNINGで利用できるデータは、削除された行の内容です。例を示します。

```
DELETE FROM products
  WHERE obsoletion_date = 'today'
  RETURNING *;
```

対象のテーブルにトリガー(第38章)がある場合、RETURNINGで利用できるデータは、トリガーで更新された行です。従って、トリガーによって計算された列を検査するのもRETURNINGの一般的な利用方法の一つです。

第7章 問い合わせ

前章までで、テーブルを作成し、これにデータを挿入し、さらに挿入したデータを操作する方法について説明しました。本章では、データベースからデータを取り出す方法について説明します。

7.1. 概要

データベースからデータを取り出す処理、または、取り出すためのコマンドを*問い合わせ*と言います。SQLでは、SELECTコマンドを、問い合わせを指定するために使います。SELECTコマンドの一般的な構文は次の通りです。

```
[WITH with_queries] SELECT select_list FROM table_expression [sort_specification]
```

以降の節では、選択リスト、テーブル式、並べ替えの仕様について詳細に説明します。WITH問い合わせは、より進んだ機能のため最後で扱います。

単純な問い合わせの形式は次のようなものです。

```
SELECT * FROM table1;
```

table1というテーブルがあるとして、このコマンドはtable1からすべてのユーザ定義の列を全行取り出します。(検索する方法はクライアントアプリケーションに依存します。クライアントライブラリは、問い合わせ結果から個々の値を抽出する機能を提供する一方、例えばpsqlプログラムでは、アスキーアートで表組を画面上に表示します。) 選択リストの指定における*は、テーブル式が持つすべての列を提供することを意味します。選択リストでは、選択可能な列の一部を選択することも、選択可能な列を使用して計算することもできます。例えば、table1にa、b、cという名前の列がある場合(他の列があっても構いません)、以下のような問い合わせができます。

```
SELECT a, b + c FROM table1;
```

(ここではbおよびcは数値型のデータであると仮定しています。) 詳細については7.3を参照してください。

FROM table1は、単純な形のテーブル式で、読み込むテーブルは1つだけです。一般にテーブル式は基本テーブルや結合そして副問い合わせなどで複雑に構成されることがあります。しかし、以下のように、テーブル式をすべて省略し、SELECTコマンドを電卓として使用することもできます。

```
SELECT 3 * 4;
```

選択リストの式が返す結果が変化する場合、これはさらに有用です。例えば、関数を次のように呼び出すことができます。

```
SELECT random();
```

7.2. テーブル式

テーブル式はテーブルを計算するためのものです。テーブル式にはFROM句が含まれており、その後ろにオプションとしてWHERE句、GROUP BY句、HAVING句を付けることができます。単純なテーブル式は、単にディスク上のいわゆる基本テーブルと呼ばれるテーブルを参照するだけです。しかし複雑な式では、様々な方法で基本テーブルを修正したり、結合させて使用することができます。

テーブル式のオプションWHERE句、GROUP BY句、およびHAVING句は、FROM句で派生したテーブル上に対して次々に変換を実行するパイプラインを指定します。これらの変換によって仮想テーブルが1つ生成されます。そしてこの仮想テーブルの行が選択リストに渡され、問い合わせの出力行が計算されます。

7.2.1. FROM句

FROM句は、カンマで分けられたテーブル参照リストで与えられる1つ以上のテーブルから、1つのテーブルを派生します。

```
FROM table_reference [, table_reference [, ...]]
```

テーブル参照は、テーブル名（スキーマで修飾することもできます）、あるいは、副問い合わせ、JOINによる結合、これらの複雑な組み合わせなどの派生テーブルとすることができます。FROM句に複数のテーブル参照がある場合、クロス結合されます（テーブルの行のデカルト積が形成されます。下記を参照）。FROMリストの結果はWHERE句、GROUP BY句、およびHAVING句での変換対象となる中間的な仮想テーブルになり、最終的にはテーブル式全体の結果となります。

テーブル参照で、テーブルの継承階層の親テーブルの名前を指定すると、テーブル名の前にONLYキーワードがない場合は、テーブル参照はそのテーブルだけでなくその子テーブルに継承されたすべての行を生成します。しかし、この参照は名前を指定したテーブルに現れる列のみを生成し、子テーブルで追加された列は無視されます。

テーブル名の前にONLYを記述する代わりに、テーブル名の後に*を記述して、子テーブルが含まれることを明示的に指定することができます。子テーブルを検索するのが今は常にデフォルトの振る舞いですので、この文法を使う本当の理由はもうありません。しかし、古いリリースとの互換性のためにサポートされています。

7.2.1.1. 結合テーブル

結合テーブルは、2つの（実または派生）テーブルから、指定した結合種類の規則に従って派生したテーブルです。内部結合、外部結合、およびクロス結合が使用可能です。テーブル結合の一般的な構文は次のとおりです

```
T1 join_type T2 [ join_condition ]
```

すべての結合は、互いに結び付けたり、あるいは入れ子にしたりすることができます。T1とT2のどちらか、あるいは両方が、結合テーブルになることがあります。括弧でJOIN句を括ることで結合の順序を制御することができます。括弧がない場合、JOIN句は左から右に入れ子にします。

結合の種類

クロス結合

```
T1 CROSS JOIN T2
```

T1およびT2からのすべての可能な行の組み合わせ（つまりデカルト積）に対し、結合されたテーブルはT1のすべての列の後にT2のすべての列が続く行を含みます。テーブルがそれぞれN行とM行で構成されているとすると、結合されたテーブルの行数は N * M 行となります。

FROM T1 CROSS JOIN T2 は FROM T1 INNER JOIN T2 ON TRUE と同じです（下記を参照）。また FROM T1, T2 とも同じです。

> **注記**
>
> 3つ以上のテーブルが現れた場合、この後者の等価性は厳密には保たれてはいません。なぜなら、JOINはカンマより強固に結合するためです。例えば FROM T1 CROSS JOIN T2 INNER JOIN T3 ON condition は FROM T1, T2 INNER JOIN T3 ON condition と同じではありません。なぜなら最初のケースではconditionがT1を参照できますが、2番目ではできないからです。

限定的な結合

```
T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2 ON boolean_expression
T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2 USING ( join column list )
T1 NATURAL { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2
```

INNERやOUTERは省略可能です。INNERがデフォルトとなります。LEFT、RIGHT、FULLは外部結合を意味します。

結合条件は、ON句かUSING句で指定するか、またはNATURAL記述で暗黙的に指定します。結合条件は、以下で詳しく説明するように、2つの元となるテーブルのどの行が「一致するか」を決めます。

限定的な結合には次のものがあります。

INNER JOIN（内部結合）

T1の各行R1に対して、T2において行R1との結合条件を満たしている各行が、結合されたテーブルに含まれます。

LEFT OUTER JOIN（左外部結合）

まず、内部結合が行われます。その後、T2のどの行との結合条件も満たさないT1の各行については、T2の列をNULL値として結合行が追加されます。したがって、連結されたテーブルは常にT1の行それぞれに少なくとも1つの行があります。

RIGHT OUTER JOIN（右外部結合）

まず、内部結合が行われます。その後、T1のどの行の結合条件も満たさないT2の各行については、T1の列をNULL値として結合行が追加されます。これは左結合の反対です。結果のテーブルは、T2の行が常に入ります。

FULL OUTER JOIN（完全外部結合）

まず、内部結合が行われます。その後、T2のどの行の結合条件も満たさないT1の各行については、T2の列をNULL値として結合行が追加されます。さらに、T1のどの行でも結合条件を満たさないT2の各行に対して、T1の列をNULL値として結合行が追加されます。

ON句は最も汎用的な結合条件であり、WHERE句で使われるものと同じ論理値評価式となります。ON式の評価が真となる場合、T1およびT2の対応する行が一致します。

USING句は、結合の両側で結合列に同じ名前を使っているという特別な状況の利点を活かすことができる省略形です。それは、結合テーブルが共通で持つ列名をカンマで区切ったリストから、それぞれの列の等価性を結合条件として生成します。例えば,T1とT2をUSING (a, b)を使用して結合する場合は、ON T1.a = T2.a AND T1.b = T2.bという結合条件を生成します。

さらに、JOIN USINGの出力は、冗長列を抑制します。マッチした列は両方が同じ値を待つので両方を出力する必要がありません。JOIN ON は T1 からのすべての列と、それに続く T2 からのすべての列を生成します。JOIN USINGは指定された列のペアのそれぞれについて1つの出力(結合リストでの指定順)、続いてT1の残りの列、その後にT2の残りの列を出力します。

最後に、NATURALはUSINGの略記形式で、2つの入力テーブルの両方に含まれているすべての列名で構成されるUSINGリストを形成します。USINGと同様、これらの列は出力テーブルに一度だけ現れます。共通する列が存在しない場合、NATURAL JOINはJOIN ... ON TRUEと同様に動作し、クロス積結合を生成します。

> **注記**
>
> USINGは、リストされている列のみ結合するのでリレーションの列の変更から適度に安全です。NATURALは、USINGよりもかなり危険です。いずれかのリレーションのスキーマ変更により新しくマッチする列名が作られると、結合にその新しい列も使われるようになってしまうからです。

まとめとして、以下のテーブルt1

```
num | name
-----+------
   1 | a
   2 | b
   3 | c
```

および、テーブルt2

```
num | value
-----+-------
   1 | xxx
   3 | yyy
   5 | zzz
```

を想定すると、以下のように様々な結合に関する結果が得られます。

```
=> SELECT * FROM t1 CROSS JOIN t2;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   1 | a    |   3 | yyy
   1 | a    |   5 | zzz
```

```
   2 | b    |   1 | xxx
   2 | b    |   3 | yyy
   2 | b    |   5 | zzz
   3 | c    |   1 | xxx
   3 | c    |   3 | yyy
   3 | c    |   5 | zzz
(9 rows)

=> SELECT * FROM t1 INNER JOIN t2 ON t1.num = t2.num;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   3 | c    |   3 | yyy
(2 rows)

=> SELECT * FROM t1 INNER JOIN t2 USING (num);
 num | name | value
-----+------+-------
   1 | a    | xxx
   3 | c    | yyy
(2 rows)

=> SELECT * FROM t1 NATURAL INNER JOIN t2;
 num | name | value
-----+------+-------
   1 | a    | xxx
   3 | c    | yyy
(2 rows)

=> SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   2 | b    |     |
   3 | c    |   3 | yyy
(3 rows)

=> SELECT * FROM t1 LEFT JOIN t2 USING (num);
 num | name | value
-----+------+-------
   1 | a    | xxx
   2 | b    |
   3 | c    | yyy
(3 rows)

=> SELECT * FROM t1 RIGHT JOIN t2 ON t1.num = t2.num;
```

```
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   3 | c    |   3 | yyy
     |      |   5 | zzz
(3 rows)

=> SELECT * FROM t1 FULL JOIN t2 ON t1.num = t2.num;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   2 | b    |     |
   3 | c    |   3 | yyy
     |      |   5 | zzz
(4 rows)
```

ONで指定される結合条件には、結合に直接関係しない条件も含めることができます。これは一部の問い合わせにおいては便利ですが、使用の際には注意が必要です。例を示します。

```
=> SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num AND t2.value = 'xxx';
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   2 | b    |     |
   3 | c    |     |
(3 rows)
```

WHERE句の中に制約を記述すると異なる結果になることに注意してください。

```
=> SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num WHERE t2.value = 'xxx';
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
(1 row)
```

この理由はON句の中の制約は結合の前に処理され、一方WHERE句の中の制約は結合の後に処理されることによります。これは内部結合には影響がありませんが、外部結合には大きな影響があります。

7.2.1.2. テーブルと列の別名

テーブルや複雑なテーブル参照に一時的な名前を付与し、問い合わせの以降の部分では、その名前を使ってテーブルや複雑なテーブル参照を利用することができます。これをテーブルの別名と呼びます。

テーブルの別名を作成するには以下のようにします。

```
FROM table_reference AS alias
```

もしくは

```
FROM table_reference alias
```

ASキーワードはなくても構わないノイズです。aliasは任意の識別子になります。

テーブルの別名の一般的な適用法は、長いテーブル名に短縮した識別子を割り当てて結合句を読みやすくすることです。例を示します。

```
SELECT * FROM some_very_long_table_name s JOIN another_fairly_long_name a ON s.id = a.num;
```

現在の問い合わせに関しては、別名がテーブル参照をする時の新しい名前になります。問い合わせの他の場所で元々の名前でテーブルを参照することはできなくなります。よって、次の例は有効ではありません。

```

SELECT * FROM my_table AS m WHERE my_table.a > 5;    -- 間違い
```

テーブルの別名は主に表記を簡単にするためにあります。しかし次のように、1つのテーブルが自分自身と結合する場合は、必須となります。

```
SELECT * FROM people AS mother JOIN people AS child ON mother.id = child.mother_id;
```

さらに、テーブル参照が副問い合わせ（7.2.1.3を参照）の場合に別名が必要になります。

括弧は曖昧さをなくすために使われます。次の例では、最初の文で2つ目のmy_tableのインスタンスにbという別名を付与し、一方、2つ目の文では結合結果に対して別名を付与しています。

```
SELECT * FROM my_table AS a CROSS JOIN my_table AS b ...
SELECT * FROM (my_table AS a CROSS JOIN my_table) AS b ...
```

次のような形式でテーブル別名を付けて、テーブル自身と同様にテーブルの列に一時的な名前を付けることができます。

```
FROM table_reference [AS] alias ( column1 [, column2 [, ...]] )
```

もし、実際のテーブルが持つ列よりも少ない数の列の別名が与えられる場合、残りの列は改名されません。この構文は、自己結合あるいは副問い合わせで特に役立ちます。

別名がJOIN句の結果に適用される場合、別名はJOIN内で参照される元々の名を隠します。以下に例を示します。

```
SELECT a.* FROM my_table AS a JOIN your_table AS b ON ...
```

は有効なSQLですが、

```
SELECT a.* FROM (my_table AS a JOIN your_table AS b ON ...) AS c
```

は有効ではありません。テーブルの別名aは、別名cの外側では参照することができません。

7.2.1.3. 副問い合わせ

派生テーブルを指定する副問い合わせは括弧で囲む必要があります。また、(7.2.1.2にあるように)必ずテーブル別名が割り当てられている必要があります。例を示します。

```
FROM (SELECT * FROM table1) AS alias_name
```

この例はFROM table1 AS alias_nameと同じです。副問い合わせがグループ化や集約を含んでいる場合は、単純結合にまとめることはできない、より重要な例が発生します。

また、副問い合わせをVALUESリストとすることもできます。

```
FROM (VALUES ('anne', 'smith'), ('bob', 'jones'), ('joe', 'blow'))
     AS names(first, last)
```

繰り返しますが、テーブルの別名が必要です。VALUESリストの列に別名を付与することは省略することもできますが、付与することを勧めます。7.7を参照してください。

7.2.1.4. テーブル関数

テーブル関数は、基本データ型(スカラ型)、もしくは複合データ型(テーブル行)からなる行の集合を生成する関数です。これらは、問い合わせのFROM句内でテーブル、ビュー、副問い合わせのように使用されます。テーブル関数から返される列は、テーブル、ビュー、副問い合わせの列と同様の手順で、SELECT、JOIN、WHEREの中に含めることができます。

テーブル関数はROWS FROM構文を使用することで、それらの返却列を一緒に組み合わせることもできます。このときの結果の行数は、行数が最大となる関数の結果と同じになり、少ない結果側は多い結果に合わせてnull値で埋められます。

```
function_call [WITH ORDINALITY] [[AS] table_alias [(column_alias [, ... ])]]
ROWS FROM( function_call [, ... ] ) [WITH ORDINALITY] [[AS] table_alias [(column_alias
 [, ... ])]]
```

WITH ORDINALITY句が指定されている場合、関数の結果の列にbigint型の列が追加されます。この列は関数の結果の行を1から数えます。(これは標準SQLの構文UNNEST ... WITH ORDINALITYの一般化です。)デフォルトでは、この序数(ordinal)の列はordinalityになります。しかし別の名前をAS句を使用して別名を割り当てることができます。

特別なテーブル関数UNNESTは、任意の数の配列パラメータで呼ぶことができます。そしてそれは、対応する数の列を返し、あたかもUNNEST(9.19)が各パラメータ毎にROWS FROM構文を使用して結合されているかのようになります。

```
UNNEST( array_expression [, ... ] ) [WITH ORDINALITY] [[AS] table_alias [(column_alias
 [, ... ])]]
```

table_aliasが指定されない場合、テーブル名として関数名が使用されます。ROWS FROM()の場合は最初の関数名が使用されます。

列に別名が提供されない場合、基本データ型を返す関数に対しては、列名も関数名と同じになります。複合型を返す関数の場合は、結果の列は型の個々の属性の名前を取得します。

以下に数例示します。

```
CREATE TABLE foo (fooid int, foosubid int, fooname text);

CREATE FUNCTION getfoo(int) RETURNS SETOF foo AS $$
    SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;

SELECT * FROM getfoo(1) AS t1;

SELECT * FROM foo
    WHERE foosubid IN (
                        SELECT foosubid
                        FROM getfoo(foo.fooid) z
                        WHERE z.fooid = foo.fooid
                      );

CREATE VIEW vw_getfoo AS SELECT * FROM getfoo(1);

SELECT * FROM vw_getfoo;
```

呼び出し方法に応じて異なる列集合を返すテーブル関数を定義することが役に立つ場合があります。これをサポートするために、テーブル関数はOUTパラメータを持たないrecord擬似型を返すものと宣言することができます。こうした関数を問い合わせで使用する場合、システムがその問い合わせをどのように解析し計画を作成すればよいのかが判断できるように、想定した行構造を問い合わせ自身内に指定しなければなりません。この構文は次のようになります。

```
function_call [AS] alias (column_definition [, ... ])
function_call AS [alias] (column_definition [, ... ])
ROWS FROM( ... function_call AS (column_definition [, ... ]) [, ... ] )
```

ROWS FROM()構文を使用しない場合は、column_definitionのリストがFROM項目に取り付けることができる列の別名の代わりとなります。列の定義内の名前は、列の別名として機能します。ROWS FROM()構文を使用する場合は、column_definitionリストを個別に各メンバー関数に添付することができます。またはメンバ関数が1つだけしかなく、かつWITH ORDINALITY句がない場合は、column_definitionリストを、ROWS FROM()の後ろの列別名のリストの場所に書くことができます。

以下の例を考えます。

```
SELECT *
    FROM dblink('dbname=mydb', 'SELECT proname, prosrc FROM pg_proc')
```

```
      AS t1(proname name, prosrc text)
    WHERE proname LIKE 'bytea%';
```

dblink関数(dblinkモジュールの一部)は遠隔問い合わせを実行します。これは任意の問い合わせで使用できるように、recordを返すものと宣言されています。実際の列集合は、パーサが例えば*がどのように展開されるかを理解できるように、呼び出した問い合わせ内で指定されなければなりません。

ROWS FROMを使用した例:

```
SELECT *
FROM ROWS FROM
    (
        json_to_recordset('[{"a":40,"b":"foo"},{"a":"100","b":"bar"}]')
            AS (a INTEGER, b TEXT),
        generate_series(1, 3)
    ) AS x (p, q, s)
ORDER BY p;

  p  |  q  | s
-----+-----+---
  40 | foo | 1
 100 | bar | 2
     |     | 3
```

2つの関数を結合して1つのFROMターゲットにします。json_to_recordset()は、2つの列(最初のintegerと2番目のtext)を返すように指示されます。generate_series()の結果は直接使用されます。ORDER BY句では、列値が整数として並べ替えられます。

7.2.1.5. LATERAL 副問い合わせ

FROMに現れる副問い合わせの前にキーワードLATERALを置くことができます。こうすると、副問い合わせは先行するFROM項目によって提供される列を参照できます。(LATERALがない場合、それぞれの副問い合わせは個別に評価され、従ってその他のFROM項目を相互参照できません。)

FROMに現れるテーブル関数の前にもキーワードLATERALを置くことが可能ですが、関数に対してこのキーワードは省略可能です。どんな場合であっても、関数の引数は先行する FROM項目により提供される列の参照を含むことができます。

LATERAL項目はFROMリストの最上層、またはJOIN木の中で表示することができます。後者の場合、右側にあるJOINの左側のすべての項目を参照することが可能です。

FROM項目がLATERAL相互参照を含む場合の評価は以下のようになります。相互参照される列(複数可)を提供するFROM項目のそれぞれの行、もしくは列を提供する複数のFROM項目の行一式に対し、LATERAL項目は列の行または複数行の一式の値により評価されます。結果行(複数可)は通常のように演算された行と結合されます。元となるテーブル(複数可)の列からそれぞれの行、または行の一式に対し反復されます。

LATERALの些細な例としては以下があげられます。

```
SELECT * FROM foo, LATERAL (SELECT * FROM bar WHERE bar.id = foo.bar_id) ss;
```

上記は以下のより伝統的なやり方と全く同じ結果をもたらしますので特別に有用ではありません。

```
SELECT * FROM foo, bar WHERE bar.id = foo.bar_id;
```

LATERALは、結合される行を計算するために相互参照する列を必須とする場合、第一義的に有用です。一般的な利用方法は、集合を返す関数に対して引数の値を提供することです。例えば、vertices(polygon)が多角形の頂点の組みを返す関数だとして、以下のようにしてテーブルに格納されている多角形の互いに近接する頂点を特定できます。

```
SELECT p1.id, p2.id, v1, v2
FROM polygons p1, polygons p2,
     LATERAL vertices(p1.poly) v1,
     LATERAL vertices(p2.poly) v2
WHERE (v1 <-> v2) < 10 AND p1.id != p2.id;
```

この問い合わせは以下のようにも書くことができます。

```
SELECT p1.id, p2.id, v1, v2
FROM polygons p1 CROSS JOIN LATERAL vertices(p1.poly) v1,
     polygons p2 CROSS JOIN LATERAL vertices(p2.poly) v2
WHERE (v1 <-> v2) < 10 AND p1.id != p2.id;
```

そのほか幾つかの同等の定式化が考えられます。(既に言及したとおり、LATERALキーワードはこの例に於いて必要ではありませんが、明確に示すために使用しました。)

LATERAL副問い合わせはLEFT JOINの対象として、しばしば特に重宝します。たとえLATERAL副問い合わせがそこから行を生成しない場合に於いても元となった行が結果に現れるからです。たとえば、get_product_names()が製造者により生産された製品名を返すとして、テーブル内のいくつかの製造者が現在製品を製造していない場合、それらは何であるかを以下のようにして見つけることができます。

```
SELECT m.name
FROM manufacturers m LEFT JOIN LATERAL get_product_names(m.id) pname ON true
WHERE pname IS NULL;
```

7.2.2. WHERE句

WHERE句の構文は以下の通りです。

```
WHERE search_condition
```

ここで、search_conditionにはboolean型を返すどのような評価式(4.2を参照)も指定できます。

FROM句の処理が終わった後、派生した仮想テーブルの各行は検索条件と照合されます。条件の結果が真の場合、その行は出力されます。そうでない(すなわち結果が偽またはNULLの)場合は、その行は捨てられま

す。一般的に検索条件は、FROM句で生成されたテーブルの最低1列を参照します。これは必須ではありませんが、そうしないとWHERE句はまったく意味がなくなります。

注記

内部結合の結合条件は、WHERE句でもJOIN句でも記述することができます。例えば、以下のテーブル式は等価です。

```
FROM a, b WHERE a.id = b.id AND b.val > 5
```

および

```
FROM a INNER JOIN b ON (a.id = b.id) WHERE b.val > 5
```

また、以下でも同じです。

```
FROM a NATURAL JOIN b WHERE b.val > 5
```

どれを使うかは、主にスタイルの問題です。FROM句のJOIN構文はSQL標準であるにも関わらず、おそらく他のSQLデータベース管理システムへの移植性では劣るでしょう。外部結合については、FROM句以外に選択の余地はありません。外部結合のON句またはUSING句は、WHERE条件とは等しくありません。なぜなら、最終結果での行を除去すると同様に、(一致しない入力行に対する)行の追加となるからです。

WHERE句の例を以下に示します。

```
SELECT ... FROM fdt WHERE c1 > 5

SELECT ... FROM fdt WHERE c1 IN (1, 2, 3)

SELECT ... FROM fdt WHERE c1 IN (SELECT c1 FROM t2)

SELECT ... FROM fdt WHERE c1 IN (SELECT c3 FROM t2 WHERE c2 = fdt.c1 + 10)

SELECT ... FROM fdt WHERE c1 BETWEEN (SELECT c3 FROM t2 WHERE c2 = fdt.c1 + 10) AND 100

SELECT ... FROM fdt WHERE EXISTS (SELECT c1 FROM t2 WHERE c2 > fdt.c1)
```

fdtはFROM句で派生されたテーブルです。WHERE句の検索条件を満たさなかった行は、fdtから削除されます。評価式としてのスカラ副問い合わせの使い方に注目してください。他の問い合わせのように、副問い合わせは複雑なテーブル式を使うことができます。副問い合わせの中でどのようにfdtが参照されるかにも注意してください。c1をfdt.c1のように修飾することは、c1が副問い合わせの入力テーブルから派生した列名でもある時にだけ必要です。列名の修飾は、必須の場合ではなくても、明確にするために役立ちます。この例は、外側の問い合わせの列名の有効範囲を、どのようにして内側の問い合わせまで拡張するかを示します。

7.2.3. GROUP BY句とHAVING句

WHEREフィルタを通した後、派生された入力テーブルをGROUP BY句でグループ化し、また、HAVING句を使用して不要なグループを取り除くことができます。

```
SELECT select_list
    FROM ...
    [WHERE ...]
    GROUP BY grouping_column_reference [, grouping_column_reference]...
```

GROUP BY句は、列挙されたすべての列で同じ値を所有する行をまとめてグループ化するために使用されます。列の列挙順は関係ありません。これは共通する値を持つそれぞれの行の集合をグループ内のすべての行を代表する1つのグループ行にまとめるものです。これは、出力の冗長度を排除したり、それぞれのグループに適用される集約計算を行うためのものです。以下に例を示します。

```
=> SELECT * FROM test1;
 x | y
---+---
 a | 3
 c | 2
 b | 5
 a | 1
(4 rows)

=> SELECT x FROM test1 GROUP BY x;
 x
---
 a
 b
 c
(3 rows)
```

2番目の問い合わせでは、SELECT * FROM test1 GROUP BY xと書くことはできません。各グループに関連付けられる列yの単一の値がないからです。GROUP BYで指定した列はグループごとに単一の値を持つので、選択リストで参照することができます。

一般的に、テーブルがグループ化されている場合、GROUP BYでリストされていない列は集約式を除いて参照することはできません。集約式の例は以下の通りです。

```
=> SELECT x, sum(y) FROM test1 GROUP BY x;
 x | sum
---+-----
 a |   4
 b |   5
 c |   2
(3 rows)
```

上記でsum() は、グループ全体について単一の値を計算する集約関数です。使用可能な集約関数の詳細については、9.21を参照してください。

> **ヒント**
>
> 集約式を使用しないグループ化は、列内の重複しない値の集合を効率良く計算します。これはDISTINCT句（7.3.3を参照）の使用で同じように達成することができます。

別の例を示します。これは各製品の総売上を計算します（全製品の総売上ではありません）。

```
SELECT product_id, p.name, (sum(s.units) * p.price) AS sales
    FROM products p LEFT JOIN sales s USING (product_id)
    GROUP BY product_id, p.name, p.price;
```

この例では、product_id列、p.name列、p.price列は必ずGROUP BY句で指定する必要があります。なぜなら、これらは問い合わせ選択リストの中で使われているからです（ただし、以下を参照）。s.units列はGROUP BYで指定する必要はありません。これは、製品ごとの売上計算の集約式（sum(...)）の中だけで使われるためです。この問い合わせは、各製品に対して製品の全販売に関する合計行が返されます。

productsテーブルが、例えば、product_idが主キーであるように設定されている場合、nameとprice列は製品ID（product_id）に関数依存しており、このため製品IDグループそれぞれに対してどのnameとpriceの値を返すかに関するあいまいさがありませんので、上の例ではproduct_idでグループ化することで十分です。

厳密なSQLでは、GROUP BYは、元となるテーブルの列によってのみグループ化できますが、PostgreSQLでは、選択リストの列によるグループ化もできるように拡張されています。単純な列名の代わりに、評価式でグループ化することもできます。

GROUP BYを使ってグループ化されたテーブルで特定のグループのみ必要な場合、結果から不要なグループを除くのに、WHERE句のようにHAVING句を使うことができます。構文は以下の通りです。

```
SELECT select_list FROM ... [WHERE ...] GROUP BY ... HAVING boolean_expression
```

HAVING句内の式は、グループ化された式とグループ化されてない式（この場合は集約関数が必要になります）の両方を参照することができます。

例を示します。

```
=> SELECT x, sum(y) FROM test1 GROUP BY x HAVING sum(y) > 3;
 x | sum
---+-----
 a |   4
 b |   5
(2 rows)
```

```
=> SELECT x, sum(y) FROM test1 GROUP BY x HAVING x < 'c';
 x | sum
---+-----
 a |   4
 b |   5
(2 rows)
```

次に、より現実的な例を示します。

```
SELECT product_id, p.name, (sum(s.units) * (p.price - p.cost)) AS profit
    FROM products p LEFT JOIN sales s USING (product_id)
    WHERE s.date > CURRENT_DATE - INTERVAL '4 weeks'
    GROUP BY product_id, p.name, p.price, p.cost
    HAVING sum(p.price * s.units) > 5000;
```

上の例で、WHERE句は、グループ化されていない列によって行を選択している(この式では最近の4週間の売上のみが真になります)のに対し、HAVING句は出力を売上高が5000を超えるグループに制限しています。集約式が、問い合わせ内で常に同じである必要がないことに注意してください。

ある問い合わせが集約関数を含んでいるがGROUP BY句がない場合でも、グループ化は依然として行われます。結果は単一グループ行(またはHAVINGで単一行が削除されれば、行が全くなくなるかもしれません)となります。HAVING句を含んでいれば、集約関数呼び出しやGROUP BY句がまったく存在しなくても同じことが言えます。

7.2.4. GROUPING SETS、CUBE、ROLLUP

上述のものよりも複雑なグループ化の操作は、グループ化セットの概念を用いることで可能です。FROM句とWHERE句によって選択されたデータは、指定されたグループ化セットによってそれぞれグループ化され、単純なGROUP BY句と同じように集約計算され、その後結果が返されます。例を示します。

```
=> SELECT * FROM items_sold;
 brand | size | sales
-------+------+-------
 Foo   | L    |  10
 Foo   | M    |  20
 Bar   | M    |  15
 Bar   | L    |  5
(4 rows)

=> SELECT brand, size, sum(sales) FROM items_sold GROUP BY GROUPING SETS ((brand), (size), ());
 brand | size | sum
-------+------+-----
 Foo   |      |  30
 Bar   |      |  20
       | L    |  15
       | M    |  35
```

```
       |      | 50
(5 rows)
```

GROUPING SETSの各サブリストはゼロ個以上の列または式を指定することが出来ます。そして、それが直接GROUP BY句で指定したのと同じように解釈されます。空のグループ化セットは、全行が一つのグループにまで集約されることを意味します（何も入力行が存在しない場合でも出力されます）。これは、上述したGROUP BY句がない集約関数の場合と同様です。

グループ化している列または式の参照は、その列が現われないグループ化セットの結果行ではNULL値に置き換えられます。特定の出力行が、どのグループ化から生じたかを識別するには表 9.59を参照して下さい。

グループ化セットの中で一般的な2種類については、略記法での指定方法が提供されています。

```
ROLLUP ( e1, e2, e3, ... )
```

上の句は、式の指定されたリストと空のリストを含めたリストのすべてのプレフィックスを表します。したがって、以下と同等です。

```
GROUPING SETS (
    ( e1, e2, e3, ... ),
    ...
    ( e1, e2 ),
    ( e1 ),
    ( )
)
```

これは一般に、階層データに対する分析のために使用されます。例えば、部署、部門、全社合計による総給与を出します。

```
CUBE ( e1, e2, ... )
```

上の句は、与えられたリストとその可能な部分集合（サブセット）のすべて（すなわち、べき集合）を表します。したがって

```
CUBE ( a, b, c )
```

は以下と同等です。

```
GROUPING SETS (
    ( a, b, c ),
    ( a, b    ),
    ( a,    c ),
    ( a       ),
    (    b, c ),
    (    b    ),
    (       c ),
    (         )
```

```
)
```

CUBE句やROLLUP句の各要素は、個々の式、または括弧で囲まれた要素のサブリスト、どちらかに出来ます。後者の場合には、サブリストは個々のグループ化セットを生成する目的において一つの単位として扱われます。例えば

```
CUBE ( (a, b), (c, d) )
```

は以下と同等です。

```
GROUPING SETS (
    ( a, b, c, d ),
    ( a, b       ),
    (       c, d ),
    (            )
)
```

そして

```
ROLLUP ( a, (b, c), d )
```

は以下と同等です。

```
GROUPING SETS (
    ( a, b, c, d ),
    ( a, b, c    ),
    ( a          ),
    (            )
)
```

CUBEとROLLUP構文は、GROUP BY句の中で直接使用、またはGROUPING SETS句の中で入れ子に出来ます。GROUPING SETS句が別の内側に入れ子になっている場合、内側の句が外側の句に直接書かれている場合と効果は同じになります。

複数の集約項目がGROUP BY句一つで指定されている場合、グループ化セットの最終的なリストは、個々の項目の外積（クロス積）です。例えば

```
GROUP BY a, CUBE (b, c), GROUPING SETS ((d), (e))
```

は以下と同等です。

```
GROUP BY GROUPING SETS (
    (a, b, c, d), (a, b, c, e),
    (a, b, d),    (a, b, e),
    (a, c, d),    (a, c, e),
    (a, d),       (a, e)
)
```

注記

(a, b)という構文は通常行コンストラクタとして式に認識されます。GROUP BY句の中では、トップレベルの式の場合これは適用されず、(a, b)は上記のような式のリストとして解析されます。何らかの理由で、グループ化式の中で行コンストラクタが必要になった場合は、ROW(a, b)を使用して下さい。

7.2.5. ウィンドウ関数処理

問い合わせがウィンドウ関数(3.5、9.22と4.2.8を参照)を含んでいれば、これらの関数はグループ化、集約およびHAVING条件検索が行われた後に評価されます。つまり、問い合わせが何らかの集約、GROUP BYまたはHAVINGを使用していれば、ウィンドウ関数により見える行はFROM/WHEREでの本来のテーブル行ではなく、グループ行となります。

複数のウィンドウ関数が使用された場合、そのウィンドウ定義にある構文的に同等であるPARTITION BYおよびORDER BY句を持つすべてのウィンドウ関数は、データ全体に渡って単一の実行手順により評価されることが保証されています。したがって、ORDER BYが一意に順序付けを決定しなくても同一の並べ替え順序付けを見ることができます。しかし、異なるPARTITION BYまたはORDER BY仕様を持つ関数の評価については保証されません。(このような場合、並べ替え手順がウィンドウ関数評価の諸手順間で一般的に必要となり、ORDER BYが等価と判断する行の順序付けを保存するような並べ替えは保証されません。)

現時点では、ウィンドウ関数は常に事前に並べ替えられたデータを必要とするので、問い合わせ出力はウィンドウ関数のPARTITION BY/ORDER BY句のどれか1つに従って順序付けされます。とはいえ、これに依存することは薦められません。確実に結果が特定の方法で並べ替えられるようにしたいのであれば、明示的な最上階層のORDER BYを使用します。

7.3. 選択リスト

前節で示したように、SELECTコマンド中のテーブル式は、テーブルやビューの結合、行の抽出、グループ化などにより中間の仮想テーブルを作ります。このテーブルは最終的に選択リストによる処理に渡されます。選択リストは、中間のテーブルのどの列を実際に出力するかを決めます。

7.3.1. 選択リスト項目

テーブル式が生成するすべての列を出力する*が最も簡単な選択リストです。そうでなければ、選択リストは、カンマで区切られた(4.2で定義された)評価式のリストです。例えば、以下のような列名のリストであっても構いません。

```
SELECT a, b, c FROM ...
```

a、b、cという列名は、FROM句で参照されるテーブルの実際の列名か、あるいは7.2.1.2で説明したような列名に対する別名です。グループ化されていなければ、選択リストで使用可能な名前空間はWHERE句と同じです。グループ化されている場合は、HAVING句と同じとなります。

もし、2つ以上のテーブルで同じ名前の列がある場合は、次のように、テーブル名を必ず指定しなければいけません。

```
SELECT tbl1.a, tbl2.a, tbl1.b FROM ...
```

複数のテーブルを使用する場合、特定のテーブルのすべての列を求める方法も便利かもしれません。

```
SELECT tbl1.*, tbl2.a FROM ...
```

table_name.*という指定方法の詳細については、8.16.5を参照してください。

任意の評価式が選択リストで使われる場合、返されるテーブルは、概念的には新たに仮想的な列を追加したものとなります。評価式は、それぞれの結果行で、その列参照を置換した行の値としていったん評価されます。しかし、選択リストの式はFROM句で指定されたテーブル式内の列を参照するものである必要はありません。例えば、定数算術式であっても構いません。

7.3.2. 列ラベル

選択リスト中の項目は、ORDER BY句の中での参照、もしくはクライアントアプリケーションによる表示での使用など、それに続く処理のために名前を割り当てることができます。例を示します。

```
SELECT a AS value, b + c AS sum FROM ...
```

ASを使った出力列名の指定がない場合、システムはデフォルトの列名を割り当てます。単純な列参照では参照された列名となります。関数呼び出しでは関数名となります。複雑な表現についてはシステムが汎用の名前を生成します。

ASキーワードは省略することができますが、新規列名がPostgreSQLキーワード（付録Cを参照）のいかなるものとも一致しない場合のみです。あるキーワードと予測外の一致を防ぐために、列名を二重引用符で囲むことができます。例えば、VALUEはキーワードですのでうまく動作しません。

```
SELECT a value, b + c AS sum FROM ...
```

しかしこれは動きます。

```
SELECT a "value", b + c AS sum FROM ...
```

将来のキーワードの追加に対する保護のため、ASを記述するか、出力列名を二重引用符で囲むかのどちらかを推奨します。

> **注記**
>
> ここでの出力列の名前の指定は、FROM句での名前の指定（7.2.1.2を参照）とは異なります。同じ列の名前を2度変更することができますが、渡されるのは選択リストの中で割り当てられたものです。

7.3.3. DISTINCT

選択リストが処理された後、結果テーブルの重複行を削除の対象にすることもできます。これを指定するためには、SELECTの直後にDISTINCTキーワードを記述します。

```
SELECT DISTINCT select_list ...
```

(DISTINCTの代わりにALLキーワードを使用して、すべての行が保持されるというデフォルトの動作を指定することができます。)

少なくとも1つの列の値が異なる場合、もちろん、それら2行は異なるとみなされます。NULL値同士は、この比較において等しいとみなされます。

また、任意の式を使用して、どの行が別であるかを決定することもできます。

```
SELECT DISTINCT ON (expression [, expression ...]) select_list ...
```

ここでexpressionは、すべての行で評価される任意の評価式です。すべての式が等しくなる行の集合は、重複しているとみなされ、集合の最初の行だけが出力内に保持されます。DISTINCTフィルタに掛けられる行の順序の一意性を保証できるよう十分な数の列で問い合わせを並べ替えない限り、出力される集合の「最初の行」は予想不可能であることに注意してください。(DISTINCT ON処理は、ORDER BYによる並べ替えの後に行われます。)

DISTINCT ON句は標準SQLではありません。さらに、結果が不定となる可能性があるため、好ましくないスタイルとみなされることもあります。GROUP BYとFROM中の副問い合わせをうまく使うことにより、この構文を使わずに済みますが、DISTINCT ON句はしばしば非常に便利な代案となります。

7.4. 問い合わせの結合

2つの問い合わせの結果は、和、積、差の集合演算を使って結合することができます。構文は以下の通りです。

```
query1 UNION [ALL] query2
query1 INTERSECT [ALL] query2
query1 EXCEPT [ALL] query2
```

query1とquery2は、これまで説明した機能をすべて使用することができる問い合わせです。集合演算は入れ子にしたり、繋げたりすることができます。例えば、以下の文を見てみましょう。

```
query1 UNION query2 UNION query3
```

上記の文は以下のように実行されます。

```
(query1 UNION query2) UNION query3
```

UNIONは、query2の結果をquery1の結果に付加します(しかし、この順序で実際に行が返される保証はありません)。さらに、UNION ALLを指定しないと、DISTINCTと同様に、結果から重複している行を削除します。

INTERSECTは、query1の結果とquery2の結果の両方に含まれているすべての行を返します。INTERSECT ALLを使用しないと、重複している行は削除されます。

EXCEPTは、query1の結果には含まれているけれども、query2の結果には含まれていないすべての行を返します。(これが2つの問い合わせの差であると言われることがあります。) この場合も、EXCEPT ALL を使用しないと、重複している行は削除されます。

2つの問い合わせの和、積、差を算出するために、そこの2つの問い合わせは「union互換」でなければいけません。つまり、その問い合わせが同じ数の列を返し、対応する列は互換性のあるデータ型(10.5を参照)でなければなりません。

7.5. 行の並べ替え

ある問い合わせが1つの出力テーブルを生成した後(選択リストの処理が完了した後)、並べ替えることができます。並べ替えが選ばれなかった場合、行は無規則な順序で返されます。そのような場合、実際の順序は、スキャンや結合計画の種類や、ディスク上に格納されている順序に依存します。しかし、当てにしてはいけません。明示的に並べ替え手続きを選択した場合にのみ、特定の出力順序は保証されます。

ORDER BY句は並べ替えの順番を指定します。

```
SELECT select_list
    FROM table_expression
    ORDER BY sort_expression1 [ASC | DESC] [NULLS { FIRST | LAST }]
             [, sort_expression2 [ASC | DESC] [NULLS { FIRST | LAST }] ...]
```

並べ替え式(複数可)は問い合わせの選択リスト内で使用可能な任意の式を取ることができます。以下に例を示します。

```
SELECT a, b FROM table1 ORDER BY a + b, c;
```

複数の式が指定された場合、前の式の値が等しい行を並べ替える際に後の式の値が使用されます。列指定の後にオプションでASCもしくはDESCを付与することで、並べ替えの方向を昇順、降順にするかを設定することができます。ASC順がデフォルトです。昇順では、小さな値を先に出力します。ここでの「小さい」とは、<演算子によって決定されます。同様に降順では>演算子で決定されます。[1]

NULLS FIRSTおよびNULLS LASTオプションを使用して、その並べ替え順においてNULL値を非NULL値の前にするか後にするかを決定することができます。デフォルトでは、NULL値はあたかもすべての非NULL値よりも大きいとみなして並べ替えます。と言うことは、NULLS FIRSTはDESC順序付けのデフォルトで、そうでなければNULLS LASTです。

この順序づけオプションは、並べ替えで使用される各列に個別に適用されることに注意してください。例えば、ORDER BY x, y DESCは、ORDER BY x DESC, y DESCと同じではなく、ORDER BY x ASC, y DESCを意味します。

[1] 実際、PostgreSQLは、ASCとDESCの並べ替え順を決定するために、式のデータ型用のデフォルトのB-tree演算子クラスを使用します。慣習的に、データ型は<と>演算子をこの並べ替え順になるように設定されます。しかし、ユーザ定義データ型の設計者は異なるものを選択することができます。

sort_expressionは以下のように列ラベルもしくは出力列の番号で指定することができます。

```
SELECT a + b AS sum, c FROM table1 ORDER BY sum;
SELECT a, max(b) FROM table1 GROUP BY a ORDER BY 1;
```

両方とも最初の出力列で並べ替えられます。出力列名は単体でなければなりません。つまり式としては使用できないことに注意してください。例えば以下は間違いです。

```
SELECT a + b AS sum, c FROM table1 ORDER BY sum + c;          -- 間違い
```

これは曖昧さを減らすための制限です。ORDER BY項目が単純な名前であっても、出力列名とテーブル式による列と同じ名前となる場合、曖昧さはまだ存在します。この場合、出力列名が使用されます。ASを使用して他のテーブル列の名前と同じ名前に出力列を変名した場合にのみ混乱が発生します。

ORDER BYを、UNION、INTERSECT、EXCEPT組み合わせの結果に適用することができます。しかしこの場合、出力列の名前または番号でのみ並べ替えることができ、式では並べ替えることができません。

7.6. LIMITとOFFSET

LIMITおよびOFFSETを使うことで、問い合わせの実行で生成された行の一部だけを取り出すことができます。

```
SELECT select_list
    FROM table_expression
    [ ORDER BY ... ]
    [ LIMIT { number | ALL } ] [ OFFSET number ]
```

限度(limit)数を指定すると、指定した行数より多くの行が返されることはありません（しかし、問い合わせの結果が指定した行数より少なければ、それより少なくなります）。LIMIT ALLは、LIMIT句を省略した場合と同じです。LIMITの引数がNULLの場合も同様です。

OFFSETは、行を返し始める前に飛ばす行数を指定します。OFFSET 0は、OFFSET句を省略した場合と同じです。OFFSETの引数がNULLの場合も同様です。

OFFSETおよびLIMITの両者が指定された場合、OFFSET分の行を飛ばしてから、返されるLIMIT行を数え始めます。

LIMITを使用する時は、結果の行を一意な順序に制御するORDER BY句を使用することが重要です。ORDER BYを使わなければ、問い合わせの行について予測不能な部分集合を得ることになるでしょう。10番目から20番目の行を問い合わせることもあるでしょうが、どういう並び順での10番目から20番目の行でしょうか？ ORDER BYを指定しなければ、並び順はわかりません。

問い合わせオプティマイザは、問い合わせ計画を生成する時にLIMITを考慮します。そのため、LIMITとOFFSETに指定した値によって、異なった計画（得られる行の順序も異なります）が得られる可能性が高いです。従って、1つの問い合わせ結果から異なる部分集合を選び出すために、異なるLIMIT/OFFSETの値を使用すると、ORDER BYで結果の順序を制御しなければ、一貫しない結果が生じるでしょう。これは不具合

ではありません。ORDER BYを使って順序を制御しない限り、SQLは必ずしも特定の順序で問い合わせの結果を渡さないという特性の必然的な結果です。

OFFSET句で飛ばされる行を、実際にはサーバ内で計算しなければなりません。そのため、大きな値のOFFSETは非効率的になることがあります。

7.7. VALUESリスト

VALUESは、「定数テーブル」を生成する方法を提供します。それは実際にはディスク上に作成して配置することなく、問い合わせで使用することができます。構文を以下に示します。

```
VALUES ( expression [, ...] ) [, ...]
```

括弧で括られた式のリストがそれぞれ、テーブルの行を生成します。リストは同一の要素数(つまり、テーブルの列数)を持たなければなりません。また、各リストで対応する項目のデータ型に互換性がなければなりません。最終的に各列に割り当てられる実際のデータ型は、UNIONと同様の規則に従って決定されます。(10.5を参照してください。)

以下に例を示します。

```
VALUES (1, 'one'), (2, 'two'), (3, 'three');
```

これは、2列3行のテーブルを返します。実質的に、以下と同じです。

```
SELECT 1 AS column1, 'one' AS column2
UNION ALL
SELECT 2, 'two'
UNION ALL
SELECT 3, 'three';
```

デフォルトでは、PostgreSQLはVALUESテーブルの各列にcolumn1、column2といった名前をつけます。標準SQLではこれらの列名は規定されていませんので、データベースシステムの種類によって異なる名前を付与しています。そのため、通常はテーブル別名リストを使用して、以下のようにデフォルトの名前を上書きする方がよいでしょう。

```
=> SELECT * FROM (VALUES (1, 'one'), (2, 'two'), (3, 'three')) AS t (num,letter);
 num | letter
-----+--------
   1 | one
   2 | two
   3 | three
(3 rows)
```

文法的には、VALUESの後に式のリストがあるものは、以下と同様に扱われます。

```
SELECT select_list FROM table_expression
```

そして、SELECTが記述できるところであれば、どこにでも記述することができます。例えば、UNIONの一部として使用することもできますし、sort_specification (ORDER BY、LIMIT、OFFSET)を付けることもできます。VALUESはINSERTコマンドの元データとしてもっとも頻繁に使用されます。次に使用頻度が高いのは副問い合わせとしての使用です。

詳しくはVALUESを参照してください。

7.8. WITH問い合わせ(共通テーブル式)

WITHは、より大規模な問い合わせで使用される補助文を記述する方法を提供します。これらの文は共通テーブル式(Common Table Expressions)またはCTEとよく呼ばれるものであり、1つの問い合わせのために存在する一時テーブルを定義すると考えることができます。WITH句内の補助文はそれぞれSELECT、INSERT、UPDATEまたはDELETEを取ることができます。そしてWITH句自身は、これもSELECT、INSERT、UPDATEまたはDELETEを取ることができる主文に付与されます。

7.8.1. WITH内のSELECT

WITH内のSELECTの基本的な価値は、複雑な問い合わせをより単純な部品に分解することです。以下に例を示します。

```
WITH regional_sales AS (
    SELECT region, SUM(amount) AS total_sales
    FROM orders
    GROUP BY region
), top_regions AS (
    SELECT region
    FROM regional_sales
    WHERE total_sales > (SELECT SUM(total_sales)/10 FROM regional_sales)
)
SELECT region,
       product,
       SUM(quantity) AS product_units,
       SUM(amount) AS product_sales
FROM orders
WHERE region IN (SELECT region FROM top_regions)
GROUP BY region, product;
```

これは販売トップの地域(region)のみから製品ごとの売上高を表示します。WITH句は、regional_sales、top_regionsという名前の2つの補助文を定義します。ここで、regional_salesの出力はtop_regions内で使用され、top_regionsはSELECT主問い合わせで使用されます。この例はWITHなしでも記述できますが、二階層の入れ子の副SELECTを必要とします。この方法に従うほうが多少扱いやすいです。

オプションのRECURSIVE修飾子は、WITHを、単に構文上の利便性の高めるだけでなく標準的なSQLでは不可能な機能を実現させます。RECURSIVEを使用すれば、WITH問い合わせが行った自己の結果を参照できるようになります。1から100までの数を合計する非常に単純な問い合わせは以下のようなものです。

```
WITH RECURSIVE t(n) AS (
    VALUES (1)
  UNION ALL
    SELECT n+1 FROM t WHERE n < 100
)
SELECT sum(n) FROM t;
```

再帰的WITH問い合わせの汎用形式は常に、非再帰的表現(non-recursiveterm)、そしてUNION(またはUNION ALL)、そして再帰的表現(recursive term)です。再帰的表現だけが、その問い合わせ自身の出力への参照を含むことができます。このような問い合わせは以下のように実行されます。

再帰的問い合わせの評価

1. 非再帰的表現を評価します。UNION(ただしUNION ALLは除きます)では、重複行を廃棄します。その再帰的問い合わせの結果の残っているすべての行を盛り込み、同時にそれらを一時作業テーブルに置きます。

2. 作業テーブルが空でないのであれば以下の手順を繰り返します。

    a. 再帰自己参照を作業テーブルの実行中の内容で置換し、再帰的表現を評価します。UNION(ただしUNION ALLは除きます)に対し、重複行と前の結果行と重複する行を破棄します。その再帰的問い合わせの結果の残っているすべての行を盛り込み、同時にそれらを一時中間テーブルに置きます。

    b. 中間テーブルの内容で作業テーブルの内容を差し替え、中間テーブルを空にします。

> **注記**
>
> 厳密には、この手順は反復(iteration)であって再帰(recursion)ではありませんが、RECURSIVEはSQL標準化委員会で選ばれた用語です。

上記の例で、作業テーブルはそれぞれの手順での単なる単一行で、引き続く作業で1から100までの値を獲得します。100番目の作業で、WHERE句による出力が無くなり、問い合わせが終了します。

再帰的問い合わせは階層的、またはツリー構造データに対処するため一般的に使用されます。実用的な例は、直接使用する部品を表すテーブル1つのみが与えられ、そこから製品すべての直接・間接部品を見つける次の問い合わせです。

```
WITH RECURSIVE included_parts(sub_part, part, quantity) AS (
    SELECT sub_part, part, quantity FROM parts WHERE part = 'our_product'
  UNION ALL
    SELECT p.sub_part, p.part, p.quantity
    FROM included_parts pr, parts p
    WHERE p.part = pr.sub_part
)
SELECT sub_part, SUM(quantity) as total_quantity
FROM included_parts
GROUP BY sub_part
```

再帰的問い合わせを扱う場合、問い合わせの再帰部分が最終的にはタプルを返さないようにすることが重要です。そうしなければ、問い合わせが永久にループしてしまうからです。UNION ALLの替わりにUNIONを使用することで、重複する前回の出力行が廃棄され、これを実現できることもあるでしょう。しかし、各周期が完全に重複している行を含まないこともよくあり、そのような場合は、1つまたは少数のフィールドを検査して、同じ場所に既に到達したかどうかを調べる必要があるかもしれません。このような状態を取り扱う標準手法は、既に巡回された値の配列を計算することです。例えば、linkフィールドを使ってテーブルgraphを検索する以下の問い合わせを考えて見ます。

```
WITH RECURSIVE search_graph(id, link, data, depth) AS (
    SELECT g.id, g.link, g.data, 1
    FROM graph g
  UNION ALL
    SELECT g.id, g.link, g.data, sg.depth + 1
    FROM graph g, search_graph sg
    WHERE g.id = sg.link
)
SELECT * FROM search_graph;
```

この問い合わせはlink関係が循環を含んでいればループします。「depth」出力を要求しているので、UNION ALLをUNIONに変えるだけでは、ループを取り除くことができません。その代わり、linkの特定の経路をたどっている間に、同じ行に到達したかどうかを認識する必要があります。このループしやすい問い合わせに、pathとcycleの2列を加えます。

```
WITH RECURSIVE search_graph(id, link, data, depth, path, cycle) AS (
    SELECT g.id, g.link, g.data, 1,
      ARRAY[g.id],
      false
    FROM graph g
  UNION ALL
    SELECT g.id, g.link, g.data, sg.depth + 1,
      path || g.id,
      g.id = ANY(path)
    FROM graph g, search_graph sg
    WHERE g.id = sg.link AND NOT cycle
)
SELECT * FROM search_graph;
```

巡回防止の他に、特定行に到達する際に選ばれた「path」それ自体を表示するため、配列値はしばしば利用価値があります。

循環を認識するために検査するために必要なフィールドが複数存在する一般的な状況では、行の配列を使用します。例えば、フィールドf1とf2を比較する必要があるときは次のようにします。

```
WITH RECURSIVE search_graph(id, link, data, depth, path, cycle) AS (
    SELECT g.id, g.link, g.data, 1,
      ARRAY[ROW(g.f1, g.f2)],
      false
```

```
    FROM graph g
  UNION ALL
    SELECT g.id, g.link, g.data, sg.depth + 1,
      path || ROW(g.f1, g.f2),
      ROW(g.f1, g.f2) = ANY(path)
    FROM graph g, search_graph sg
    WHERE g.id = sg.link AND NOT cycle
)
SELECT * FROM search_graph;
```

ヒント

循環を認識するために検査するために必要なフィールドが1つだけである一般的な場合では、ROW()構文を削除します。これで、複合型配列ではなく単純配列で済むので、効率も上がります。

ヒント

再帰的問い合わせ評価アルゴリズムは、幅優先探索順でその出力を作成します。このようにして作られた「path」列を外側問い合わせでORDER BYすれば、深さ優先探索順の結果の表示が可能です。

ループするかどうか確信が持てない問い合わせをテストする有益な秘訣として、親問い合わせにLIMITを配置します。例えば、以下の問い合わせはLIMITがないと永久にループします。

```
WITH RECURSIVE t(n) AS (
    SELECT 1
  UNION ALL
    SELECT n+1 FROM t
)
SELECT n FROM t LIMIT 100;
```

これが動作するのは、PostgreSQLの実装が、実際に親問い合わせで取り出されるのと同じ数のWITH問い合わせの行のみを評価するからです。この秘訣を実稼動環境で使用することは勧められません。他のシステムでは異なった動作をする可能性があるからです。同時に、もし外部問い合わせを再帰的問い合わせの結果を並べ替えたり、またはそれらを他のテーブルと結合するような書き方をした場合、動作しません。このような場合、外部問い合わせは通常、WITH問い合わせの出力をとにかくすべて取り込もうとするからです。

有用なWITH問い合わせの特性は、親問い合わせ、もしくは兄弟WITH問い合わせによりたとえ1回以上参照されるとしても、通常は親問い合わせ実行で1回だけ評価されることです。したがって、複数の場所で必要な高価な計算は、冗長作業を防止するためWITH問い合わせの中に配置することができます。他にありうる適用としては、望まれない副作用のある関数の多重評価を避けることです。しかし、反対の見方をすれば、オプティマイザが親クエリから複数参照されるWITH問い合わせに制約を押し下げることができないということになります。これは、WITH問い合わせの出力が1つのみに影響する場合、その出力のすべての使用に影響する可能性があるためです。複数参照されるWITH問い合わせは、親問い合わせが後で破棄するであろう行を抑制せずに、書かれた通りに評価されます。（しかし、上で述べたように、問い合わせの参照が限定された数の行のみを要求する場合、評価は早期に停止します。）

しかし、WITH問い合わせが非再帰で副作用がない（つまり、揮発性（volatile）の関数を含まないSELECTである）場合は、親問い合わせに組み込むことができ、2つの問い合わせレベルを同時に最適化できます。デフォルトでは、親問い合わせがWITH問い合わせを1回だけ参照する場合にこれが発生しますが、WITH問い合わせを2回以上参照する場合には発生しません。この決定を上書きするには、MATERIALIZEDを指定してWITH問い合わせの個別の計算を強制するか、NOT MATERIALIZEDを指定して親問い合わせにマージするようにします。後者を選択すると、WITH問い合わせの計算が重複する危険性がありますが、WITH問い合わせを使用するたびにWITH問い合わせのごく一部しか必要としない場合は、全体の節約になります。

これらのルールの簡単な例を次に示します。

```
WITH w AS (
    SELECT * FROM big_table
)
SELECT * FROM w WHERE key = 123;
```

このWITH問い合わせは組み込まれ、次のものと同じ実行計画を生成します。

```
SELECT * FROM big_table WHERE key = 123;
```

特に、keyインデックスがある場合、key = 123を持つ行のみをフェッチするために使用される可能性があります。一方で、

```
WITH w AS (
    SELECT * FROM big_table
)
SELECT * FROM w AS w1 JOIN w AS w2 ON w1.key = w2.ref
WHERE w2.key = 123;
```

このWITH問い合わせでは実体化され、big_tableの一時的なコピーが生成されます。このコピーはインデックスのメリットなしに、それ自体に結合されます。この問い合わせは次のように記述すると、より効率的に実行されます。

```
WITH w AS NOT MATERIALIZED (
    SELECT * FROM big_table
)
SELECT * FROM w AS w1 JOIN w AS w2 ON w1.key = w2.ref
WHERE w2.key = 123;
```

親の問い合わせの制限をbig_tableのスキャンに直接適用することが出来ます。

NOT MATERIALIZEDが望ましくない例を次に示します。

```
WITH w AS (
    SELECT key, very_expensive_function(val) as f FROM some_table
)
SELECT * FROM w AS w1 JOIN w AS w2 ON w1.f = w2.f;
```

ここで、WITH問い合わせを生成すると、very_expensive_functionがテーブルの行毎に1回のみ評価され、2回は評価されないことが保証されます。

上の例ではSELECTを使用するWITHのみを示しています。しかし、同じ方法でINSERT、UPDATE、またはDELETEに対して付与することができます。それぞれの場合において、これは主コマンド内で参照可能な一時テーブルを実質的に提供します。

7.8.2. WITH内のデータ変更文

WITH内でデータ変更文(INSERT、UPDATE、DELETE)を使用することができます。これにより同じ問い合わせ内で複数の異なる操作を行うことができます。

```
WITH moved_rows AS (
    DELETE FROM products
    WHERE
        "date" >= '2010-10-01' AND
        "date" < '2010-11-01'
    RETURNING *
)
INSERT INTO products_log
SELECT * FROM moved_rows;
```

この問い合わせは実質、productsからproducts_logに行を移動します。WITH内のDELETEはproductsから指定した行を削除し、そのRETURNING句により削除した内容を返します。その後、主問い合わせはその出力を読み取り、それをproducts_logに挿入します。

上の例の見事なところは、WITH句がINSERT内の副SELECTではなく、INSERTに付与されていることです。これは、データ更新文は最上位レベルの文に付与されるWITH句内でのみ許されているため必要です。しかし、通常のWITHの可視性規則が適用されますので、副SELECTからWITH文の出力を参照することができます。

上の例で示したように、WITH内のデータ変更文は通常RETURNING句(6.4を参照)を持ちます。問い合わせの残りの部分で参照することができる一時テーブルを形成するのは、RETURNING句の出力の出力であって、データ変更文の対象テーブルではありません。WITH内のデータ変更文がRETURNING句を持たない場合、一時テーブルを形成しませんので、問い合わせの残りの部分で参照することができません。これにもかかわらずこうした文は実行されます。特別有用でもない例を以下に示します。

```
WITH t AS (
    DELETE FROM foo
)
DELETE FROM bar;
```

この例はfooテーブルとbarテーブルからすべての行を削除します。クライアントに報告される影響を受けた行数にはbarから削除された行のみが含まれます。

データ変更文内の再帰的な自己参照は許されません。一部の場合において、再帰的なWITHの出力を参照することで、この制限を回避することができます。以下に例を示します。

```
WITH RECURSIVE included_parts(sub_part, part) AS (
```

```
    SELECT sub_part, part FROM parts WHERE part = 'our_product'
  UNION ALL
    SELECT p.sub_part, p.part
    FROM included_parts pr, parts p
    WHERE p.part = pr.sub_part
)
DELETE FROM parts
  WHERE part IN (SELECT part FROM included_parts);
```

この問い合わせはある製品の直接的な部品と間接的な部品をすべて削除します。

WITH内のデータ変更文は正確に1回のみ実行され、主問い合わせがその出力をすべて(実際にはいずれか)を呼び出したかどうかに関係なく、常に完了します。これが、前節で説明した主問い合わせがその出力を要求した時のみにSELECTの実行が行われるというWITH内のSELECTについての規則と異なることに注意してください。

WITH内の副文はそれぞれと主問い合わせで同時に実行されます。したがってWITH内のデータ変更文を使用する時、指定したデータ変更文が実際に実行される順序は予測できません。すべての文は同じスナップショット(第13章参照)を用いて実行されます。このため互いが対象テーブルに行った影響を「見る」ことはできません。これは、行の更新に関する実際の順序が予測できないという影響を軽減し、RETURNINGデータが別のWITH副文と主問い合わせとの間で変更を伝える唯一の手段であることを意味します。この例を以下に示します。

```
WITH t AS (
    UPDATE products SET price = price * 1.05
    RETURNING *
)
SELECT * FROM products;
```

外側のSELECTはUPDATEの動作前の元々の価格を返します。

```
WITH t AS (
    UPDATE products SET price = price * 1.05
    RETURNING *
)
SELECT * FROM t;
```

一方こちらでは外側のSELECTは更新されたデータを返します。

単一の文で同じ行を2回更新しようとすることはサポートされていません。変更のうちの1つだけが行われますが、どれが実行されるかを確実に予測することは簡単ではありません(場合によっては不可能です)。これはまた、同じ文内ですでに更新された行を削除する場合でも当てはまり、更新のみが実行されます。したがって一般的には単一の文で1つの行を2回変更しようと試みることを避けなければなりません。具体的には主文または同レベルの副文で変更される行と同じ行に影響を与えるWITH副文を記述することは避けてください。こうした文の影響は予測することはできません。

現状、WITH内のデータ変更文の対象として使用されるテーブルはすべて、複数の文に展開される条件付きルール、ALSOルール、INSTEADルールを持ってはなりません。

第8章 データ型

PostgreSQLにはユーザが使用可能な豊富なデータ型が始めから備わっています。CREATE TYPEコマンドでPostgreSQLに対し新しいデータ型を追加できます。

表 8.1に組み込みの汎用データ型をすべて示します。「別名」欄に列挙された代替名称のほとんどは、歴史的な理由によりPostgreSQLの内部で使用されている名前です。他にも、内部で使用されるデータ型、削除予定のデータ型もありますが、ここにはリストされていません。

表8.1 データ型

名称	別名	説明
bigint	int8	8バイト符号付き整数
bigserial	serial8	自動増分8バイト整数
bit [(n)]		固定長ビット列
bit varying [(n)]	varbit [(n)]	可変長ビット列
boolean	bool	論理値(真/偽)
box		平面上の矩形
bytea		バイナリデータ(「バイトの配列(byte array)」)
character [(n)]	char [(n)]	固定長文字列
character varying [(n)]	varchar [(n)]	可変長文字列
cidr		IPv4もしくはIPv6ネットワークアドレス
circle		平面上の円
date		暦の日付(年月日)
double precision	float8	倍精度浮動小数点(8バイト)
inet		IPv4もしくはIPv6ホストアドレス
integer	int, int4	4バイト符号付き整数
interval [fields] [(p)]		時間間隔
json		テキストのJSONデータ
jsonb		バイナリJSON データ, 展開型
line		平面上の無限直線
lseg		平面上の線分
macaddr		MAC(Media Access Control)アドレス
macaddr8		MAC (Media Access Control) アドレス (EUI-64 形式)
money		貨幣金額
numeric [(p, s)]	decimal [(p, s)]	精度の選択可能な高精度数値
path		平面上の幾何学的経路
pg_lsn		PostgreSQLログシーケンス番号

名称	別名	説明
pg_snapshot		ユーザレベルのトランザクションIDスナップショット
point		平面上の幾何学的点
polygon		平面上の閉じた幾何学的経路
real	float4	単精度浮動小数点(4バイト)
smallint	int2	2バイト符号付き整数
smallserial	serial2	自動増分2バイト整数
serial	serial4	自動増分4バイト整数
text		可変長文字列
time [(p)] [without time zone]		時刻(時間帯なし)
time [(p)] with time zone	timetz	時間帯付き時刻
timestamp [(p)] [without time zone]		日付と時刻(時間帯なし)
timestamp [(p)] with time zone	timestamptz	時間帯付き日付と時刻
tsquery		テキスト検索問い合わせ
tsvector		テキスト検索文書
txid_snapshot		ユーザレベルのトランザクションIDスナップショット(廃止予定。pg_snapshotを参照)
uuid		汎用一意識別子
xml		XMLデータ

互換性

次に挙げるデータ型(あるいはその綴り方)はSQLで規定されています。bigint、bit、bit varying、boolean、char、character varying、character、varchar、date、double precision、integer、interval、numeric、decimal、real、smallint、time(時間帯付き、なしの両方)、timestamp(時間帯付き、なしの両方)、xml。

それぞれのデータ型はそのデータ型の入出力関数で決定される外部表現を保有しています。組み込みデータ型の多くには、自明の外部書式があります。とは言っても、経路のようなPostgreSQLに特有な型や、あるいは、日付や時刻データ型のように書式を複数選択できる型がいくつかあります。一部の入出力関数は可逆ではありません。つまり、出力関数による結果は元の入力と比較した場合精度を失う可能性があります。

8.1. 数値データ型

数値データ型には2、4、8バイト整数と、4、8バイト浮動小数点および精度設定が可能な数があります。表 8.2に使用可能な型を列挙します。

表8.2 数値データ型

型名	格納サイズ	説明	範囲
smallint	2バイト	狭範囲の整数	-32768から+32767
integer	4バイト	典型的に使用する整数	-2147483648から+2147483647
bigint	8バイト	広範囲整数	-9223372036854775808から+9223372036854775807
decimal	可変長	ユーザ指定精度、正確	小数点より上は131072桁まで、小数点より下は16383桁まで
numeric	可変長	ユーザ指定精度、正確	小数点より上は131072桁まで、小数点より下は16383桁まで
real	4バイト	可変精度、不正確	6桁精度
double precision	8バイト	可変精度、不正確	15桁精度
smallserial	2バイト	狭範囲自動整数	1から32767
serial	4バイト	自動増分整数	1から2147483647
bigserial	8バイト	広範囲自動増分整数	1から9223372036854775807

数値データ型に対する定数の構文は4.1.2で説明しています。数値データ型には対応する算術演算子と関数の一式が揃っています。詳細は第9章を参照してください。次節でデータ型について詳しく説明します。

8.1.1. 整数データ型

smallint、integer、bigintは各種範囲の整数、つまり小数点以下の端数がない数を保持します。許容範囲から外れた値を保存しようとするとエラーになります。

integer型は数値の範囲、格納サイズおよび性能において最も釣合いが取れていますので、一般的に使用されます。smallint型は通常はディスク容量に制限が付いている場合にのみ使用します。bigint型はintegerの許容範囲では十分ではない場合に使用されるよう設計されています。

SQLでは整数の型としてinteger（またはint）とsmallint、bigintのみを規定しています。int2、int4およびint8は拡張ですが、いくつか他のSQLデータベースシステムでも使われています。

8.1.2. 任意の精度を持つ数

numeric型は、非常に大きな桁数で数値を格納できます。通貨金額やその他正確性が求められる数量を保存する時は特に、この型を推奨します。numericの値での計算は、可能なところ、例えば、足し算、引算、掛け算では、正確な結果（訳注：10進の小数で誤差が生じない、ということ）になります。とは言っても、numericの値に対する計算は整数型、もしくは次節で説明する浮動小数点データ型に比較し非常に遅くなります。

この後の説明では、次の用語を使用します。numericの精度*(precision)*とは数字全体の有効桁数です。すなわち、小数点をはさんでいる両側の桁数の合計です。numericの位取り*(scale)*とは、小数点の右側の小数部分の桁数をいいます。そのため、23.5141という数値の精度は6で位取りは4となります。整数の位取りは、ゼロであるとみなすことができます。

numeric列の数値の最大精度と最大位取りの両方を設定することができます。numeric型の列を宣言するには次の構文を使います。

```
NUMERIC(precision, scale)
```

精度は正数、位取りは0もしくは正数でなければなりません。他の記述方法として、

```
NUMERIC(precision)
```

は位取りが0であることを選択します。精度も位取りも指定せず、

```
NUMERIC
```

と記述すると、実装されている限界の精度まで、いかなる精度あるいは位取りの値も格納できる列が作られます。この類の列は入力値をいかなる特定の位取りにも変換しませんが、宣言された位取りを持つnumeric列は入力値をその位取りに変換します。(標準SQLはデフォルトとして位取り0を要求していて、つまり、整数の精度に変換されます。しかし、この方法はあまり役に立たないと思われます。もし移植性を心配するなら、常に精度と位取りを明示的に設定してください。)

> **注記**
>
> 明示的に型宣言で指定される場合に許される最大精度は1000です。精度の指定がないNUMERICは表 8.2で説明する制限に従います。

格納される値の位取りが宣言された列の位取りより大きかった場合、システムは指定された小数部の桁まで値を丸めます。そして、小数点の左側の桁数が、宣言された精度から宣言された位取りを差し引いた数を超える場合にエラーとなります。

数値は物理的に先頭や末尾に0を付与されることなく格納されます。したがって、列の宣言された精度と位取りは最大であり、固定的に割り当てられていません。(この意味ではnumericはchar(n)よりもvarchar(n)に似ています。) 実際の格納に必要な容量は、10進数4桁のそれぞれのグループに対して2バイトと、3から8バイトのオーバヘッドです。

通常の数値に加え、numeric型は、「非数値」を意味するNaNという特別な値を取ることができます。NaNに対する操作はすべて、別のNaNを生成します。この値をSQLコマンドの定数として記述する場合は、例えばUPDATE table SET x = 'NaN'のように、引用符でくくらなければなりません。入力の際は、NaNという文字列は大文字小文字の区別なく認識されます。

> **注記**
>
> ほとんどの「非数」の概念の実装において、NaNは(NaNを含む)他の数値と等価にならないとみなされています。numeric値をソートできる、また、ツリーを基にしたインデックスで使用できるように、PostgreSQLはNaN同士は等しく、すべてのNaN以外の値よりも大きな値となるものとして扱います。

decimalとnumeric型は等価です。2つのデータ型はともに標準SQLに含まれます。

値を丸める際、numeric型は0から離れるように丸めますが、一方で（ほとんどのマシンでは）realやdouble precision型ではその値に最も近い偶数に丸めます。以下に例を示します。:

```
SELECT x,
  round(x::numeric) AS num_round,
  round(x::double precision) AS dbl_round
FROM generate_series(-3.5, 3.5, 1) as x;
  x   | num_round | dbl_round
------+-----------+-----------
 -3.5 |        -4 |        -4
 -2.5 |        -3 |        -2
 -1.5 |        -2 |        -2
 -0.5 |        -1 |        -0
  0.5 |         1 |         0
  1.5 |         2 |         2
  2.5 |         3 |         2
  3.5 |         4 |         4
(8 rows)
```

8.1.3. 浮動小数点データ型

realとdouble precisionは不正確な（訳注：10進の小数で誤差が生じる、ということ）可変精度の数値データ型です。現在サポートされている全てのプラットフォーム上では、これらのデータ型は、使用しているプロセッサ、オペレーティングシステムおよびコンパイラがサポートしていれば、通常は（それぞれ単精度および倍精度の）バイナリ浮動小数点演算用のIEEE規格754の実装です。

不正確というのは、ある値はそのままで内部形式に変換されずに近似値として保存されるということです。ですから、保存しようとする値と抽出しようとする値の間に多少の差異が認められます。これらのエラーを管理し計算によって補正をどうするかについては、数学とコンピュータ科学の系統すべてに関わることで、以下の点を除き触れません。

- （金銭金額など）正確な記録と計算が必要な時は代わりにnumericを使用してください。
- これらのデータ型で何か重要な件に対し複雑な計算を必要とする時、特に（無限大やアンダーフローのような）境界線におけるある種の振舞いについて信頼を置かなければならないのであれば、実装を注意深く検証しなければなりません。
- 2つの浮動小数点値が等価であるのかどうかの比較は予想通りに行かない時もあります。

現在サポートされている全てのプラットフォームでは、real型は最低6桁の精度を持ち、1E-37から1E+37までの範囲です。double precision型は最低15桁の精度でおよそ1E-307から1E+308までの範囲です。大き過ぎたり小さ過ぎる値はエラーの原因になります。入力値の精度が高過ぎる場合は丸められることがあります。ゼロに限りなく近い値で、しかもゼロと異なる値として表現できない数値はアンダーフローエラーになります。

デフォルトでは、浮動小数の値は最も短い正確な10進数のテキスト形式で出力されます。生成される10進値は、同じバイナリ精度で表現できる他の値よりも、実際に格納されているバイナリ値に近い値になります。(ただし、出力値が2つの表現可能な値の厳密な中間になることはありません。これは、入力ルーチンが

最も近い偶数に丸める規則を適切に考慮しないという広範囲にわたる不具合を避けるためです。) この値はfloat8型の値には最大17桁の10進数、float4型の値には最大9桁の10進数を使用します。

> **注記**
>
> この最も短く正確な出力フォーマットは従来の丸められた形式よりもはるかに速く値を生成します。

PostgreSQLの古いバージョンで生成された出力との互換性を確保し、出力精度を低くするために、代わりにextra_float_digitsパラメータを使用して丸めた10進数の出力を選択することができます。値を0に設定した場合は、以前のデフォルト値である6(float4の場合)か15(float8の場合)の有効桁に丸めた値を戻します。負の値を設定すると、桁数がさらに減少します。たとえば、-2を設定すると、出力はそれぞれ4桁または13桁に丸められます。

0より大きいextra_float_digitsの値は、最短の正確なフォーマットを選択します。

> **注記**
>
> 精密な値を必要とするアプリケーションでは、従来、extra_float_digitsを3に設定して値を取得する必要がありました。バージョン間の互換性を最大にするためには継続してそのように設定する必要があります。

通常の数値に加え、浮動小数点型では以下の特殊な値を取ります。

```
Infinity
-Infinity
NaN
```

これらはそれぞれ、IEEE 754の特殊な値、「無限大」、「負の無限大」、「非数値」を表します。これらの値をSQLコマンドの定数として記述する場合、例えばUPDATE table SET x = '-Infinity'のように引用符でくくる必要があります。入力の際、これらの文字列は大文字小文字の区別なく認識されます。

> **注記**
>
> IEEE754では、NaNは(NaNを含む)他のすべての浮動小数点値と比べた時に不等でなければならないと規定しています。浮動小数点値をソートできる、また、ツリーを基にしたインデックスで使用できるように、PostgreSQLはNaN同士は等しく、すべてのNaN以外の値よりも大きな値となるものとして扱います。

また、PostgreSQLでは不正確な数値型についての標準SQLの表記であるfloatとfloat(p)をサポートしています。ここで、pは$2_{進数}$桁数で最低限、許容可能な精度を指定します。PostgreSQLはfloat(1)からfloat(24)をrealを選択するものとして受け付け、float(25)からfloat(53)をdouble precisionを選択するものとして受け付けます。許容範囲外のpの値はエラーになります。精度指定のないfloatはdouble precisionとして解釈されます。

8.1.4. 連番型

> **注記**
>
> このセクションではPostgreSQL固有の自動増分列の作成方法について記述します。SQL標準の識別列を作成する方法は、CREATE TABLEに記述されています。

smallserial、serialおよびbigserialデータ型は正確にはデータ型ではなく、テーブルの列に一意の識別子を作成する簡便な表記法です（他のデータベースでサポートされるAUTO_INCREMENTプロパティに似ています）。現在の実装では、

```
CREATE TABLE tablename (
    colname SERIAL
);
```

は以下を指定することと同じです。

```
CREATE SEQUENCE tablename_colname_seq AS integer;
CREATE TABLE tablename (
    colname integer NOT NULL DEFAULT nextval('tablename_colname_seq')
);
ALTER SEQUENCE tablename_colname_seq OWNED BY tablename.colname;
```

このように整数列を作成し、その列のデフォルト値が連番ジェネレータから割り当てられるようにしました。また、NOT NULL制約を適用することによって、NULL値が挿入されないようにします。（たいていの場合は、重複する値を間違って挿入しないように、UNIQUE制約またはPRIMARY KEY制約も追加することになるでしょうが、これは自動的には行われません。）最後に、シーケンスは列に「より所有」されるものと印が付きます。したがって、テーブルの列が削除された場合にシーケンスは削除されます。

> **注記**
>
> smallserial、serialおよびbigserialはシーケンスを使って実装されているため、行の削除が行われていなくとも、列に"穴"や連番の抜けが発生するかもしれません。また、テーブルへ正常に挿入されていないにも関わらず、シーケンスの値を"消費してしまう"こともあります。これは、例えば挿入したトランザクションがロールバックされた時に発生することがあります。詳細は9.17のnextval()を参照してください。

serial列にシーケンスの次の値を挿入するには、serial列にそのデフォルト値を割り当てるよう指定してください。これは、INSERT文の列リストからその列を除外する、もしくはDEFAULTキーワードを使用することで行うことができます。

serialとserial4という型の名称は等価です。ともにinteger列を作成します。bigserialとserial8という型の名称もbigint列を作成することを除いて同じ振舞いをします。もしテーブルを使用する期間で2^{31}以上の識別子を使用すると予測される場合、bigserialを使用すべきです。smallserialとserial2という型の名称もまた、smallint列を作成することを除いて同じ振舞いをします。

serial列用に作成されたシーケンスは、それを所有する列が削除された時に自動的に削除されます。列を削除せずにシーケンスを削除することができますが、これにより強制的に列のデフォルト式が削除されます。

8.2. 通貨型

money型は貨幣金額を固定精度の小数点で格納します。表 8.3を参照してください。小数点精度はデータベースのlc_monetary設定で決定されます。この表が示すように範囲は小数点2桁を想定しています。'$1,000.00'などの典型的な通貨書式の他、整数、浮動小数点リテラルなど様々な書式の入力を受け付けます。出力形式は通常は後者となりますが、ロケールによって異なります。

表8.3 通貨型

型名	格納サイズ	説明	範囲
money	8バイト	貨幣金額	-92233720368547758.08 から +92233720368547758.07

このデータ型の出力はロケールにより変動しますので、lc_monetary設定が異なるデータベースにmoneyデータをロードする場合には動作しない可能性があります。この問題を防ぐためには、ダンプを新しいデータベースにリストアする前に、lc_monetaryがダンプを行ったデータベースと同じまたは等価であることを確認してください。

numeric、intそしてbigint型はmoney型にキャストすることができます。real型やdouble precision型は最初にnumeric 型にキャストした後に行なう必要があります。以下に例を示します。

```
SELECT '12.34'::float8::numeric::money;
```

しかしこれは推奨されません。浮動小数点数値は丸め誤差の可能性がありますので貨幣を扱うために使用すべきではありません。

money型の値は精度を落とすことなくnumericにキャストすることができます。他の型への変換では精度を落とす可能性があり、また2段階で行う必要があります。

```
SELECT '52093.89'::money::numeric::float8;
```

money型の値を整数型の値で除算すると、小数部分を0に切り捨てるように実行されます。四捨五入した結果を得るためには、小数部分を持つ値で割り算するか、割り算を行う前にmoney型の値をnumeric型にキャストし、あとでmoney型に戻します。(精度を落とすリスクを避けるため、後者の方が好ましいです。) money型の値を別のmoney型の値で除算すると、結果はdouble precision型(通貨ではなく純粋な数値)になります。除算では通貨の単位は相殺されます。

8.3. 文字型

表8.4 文字型

型名	説明
character varying(n), varchar(n)	上限付き可変長
character(n), char(n)	空白で埋められた固定長
text	制限なし可変長

表 8.4はPostgreSQLで使用可能な汎用文字型を示したものです。

SQLは2つの主要な文字データ型を定義しています。character varying(n)とcharacter(n)です。ここでnは正の整数です。これらのデータ型は2つともn文字長（バイト数ではなく）までの文字列を保存できます。上限を越えた文字列をこれらの型の列に保存しようとするとエラーになります。ただし、上限を超えた部分にある文字がすべて空白の場合はエラーにはならず、文字列の最大長にまで切り詰められます。（この一風変わった例外は標準SQLで要求されています。）もし宣言された上限よりも文字列が短い時はcharacterの値は空白で埋められ、character varyingの値は単にその短い文字列で保存されます。

明示的に値をcharacter varying(n)またはcharacter(n)にキャストした場合、指定長を超えるとエラーなしでn文字まで切り詰められます。（これもまた標準SQLの仕様です。）

char(n)およびvarchar(n)という表記法はそれぞれcharacter(n)とcharacter varying(n)の別名です。長さ指定がないcharacterはcharacter(1)と同じです。character varyingが長さ指定なしで使われた時は、いかなる長さの文字列でも受け付けます。後者はPostgreSQLの拡張です。

さらにPostgreSQLは、いかなる長さの文字列でも格納できるtextをサポートします。text型は標準SQLにはありませんが、多くの他のSQLデータベース管理システムも同様にサポートしています。

character型の値は、指定長nになるまで物理的に空白で埋められ、そのまま格納、表示されます。しかし、最後の空白は、重要ではないものとして扱われ、2つのcharacter型の値を比べる際には無視されます。空白が重要な照合順序では、この挙動は予期しない結果を返す可能性があります。例えば、SELECT 'a '::CHAR(2) collate "C" < E'a\n'::CHAR(2)はCロケールでスペースが改行よりも大きいにも関わらず真を返します。character値を他の文字列型に変換する際は、文字列の終わりの空白は除去されます。character varying型とtext型の値の中や、パターンマッチを行なう際、すなわちLIKEや正規表現では、最後の空白は意味的に重要なものですので、注意してください。

短い文字列（126バイトまで）の保存には、実際の文字列に1バイト加えたサイズが必要です。characterでは空白埋め込み分もこれに含まれます。より長い文字列では1バイトではなく4バイトのオーバーヘッドになります。長い文字列はシステムにより自動的に圧縮されますので、ディスク上の物理的必要容量サイズはより小さくなるかもしれません。また、非常に長い値はより短い列の値への高速アクセスに干渉しないように、バックグラウンドテーブルに格納されます。いずれの場合にあっても保存できる最長の文字列は約1ギガバイトです。（データ型宣言に使われるnに許される最大値はこれより小さいものです。マルチバイト文字符号化方式においては文字数とバイト数はまったく異なっているため、この値の変更は便利ではありません。特定の上限を設けずに長い文字列を保存したい場合は、適当な上限を設けるよりも、textもしくは長さの指定がないcharacter varyingを使用してください。）

> **ヒント**
>
> 空白で埋められる型を使用した場合の保存領域の増加、および、長さ制限付きの列に格納する際に長さを検査するためにいくつか余計なCPUサイクルが加わる点を別にして、これら3つの型の間で性能に関する差異はありません。他の一部のデータベースシステムではcharacter(n)には性能的な優位性がありますが、PostgreSQLではこうした利点はありません。実際には、格納の際に追加のコストがあるため、character(n)は3つの中でもっとも低速です。多くの場合、代わりにtextかcharacter varyingを使うのがお薦めです。

文字列リテラルの構文については4.1.2.1、利用可能な演算子と関数については第9章を参照してください。データベースの文字セットは、テキストの値を格納する時に使用される文字セットを決定します。文字セットのサポートに関する詳細については23.3を参照してください。

例8.1 文字データ型の使用

```
CREATE TABLE test1 (a character(4));
INSERT INTO test1 VALUES ('ok');
SELECT a, char_length(a) FROM test1; -- 1

  a   | char_length
------+-------------
 ok   |           2


CREATE TABLE test2 (b varchar(5));
INSERT INTO test2 VALUES ('ok');
INSERT INTO test2 VALUES ('good      ');
INSERT INTO test2 VALUES ('too long');
ERROR:  value too long for type character varying(5)

INSERT INTO test2 VALUES ('too long'::varchar(5)); -- 明示的な切り捨て
SELECT b, char_length(b) FROM test2;

   b   | char_length
-------+-------------
 ok    |           2
 good  |           5
 too l |           5
```

1　char_length関数は9.4で説明されています。

PostgreSQLには、表 8.5に示すように、この他2つの固定長文字型があります。name型は内部のシステムカタログ内の識別子の格納のために*のみ*存在するもので、一般ユーザによって使用されることを意図していません。現在長さは64バイト（63バイトの利用可能文字と終止文字）と定義されていますが、Cソースコードにある`NAMEDATALEN`定数を使って参照される必要があります。この長さはコンパイル時に設定されます（そのため特別な用途に合わせ調整できます）。デフォルトの最大長は今後のリリースで変更される可能性があります。"char"（二重引用符に注意）は、char(1)とは異なり、1バイトの領域しか使用しません。過度に単純化した列挙型としてシステムカタログで内部的に使用されます。

表8.5 特別な文字データ型

型名	格納サイズ	説明
"char"	1バイト	単一バイト内部データ型
name	64バイト	オブジェクト名用の内部データ型

8.4. バイナリ列データ型

byteaデータ型はバイナリ列の保存を可能にします。表 8.6を参照してください。

表8.6 バイナリ列データ型

型名	格納サイズ	説明
bytea	1または4バイトと実際のバイナリ列の長さ	可変長のバイナリ列

バイナリ列はオクテット（またはバイト）の連続です。バイナリ列は2つの点で文字列と区別されます。1点目は、バイナリ列はゼロの値のオクテットと他の「表示できない」オクテット（通常10進数表記で32から126の範囲外のオクテット）を保存できるということです。文字列ではゼロというオクテットは使用できません。また、データベースで選択している文字セット符号化方式で無効なオクテット値やオクテット値の並びも使用できません。2点目は、バイナリ列を演算すると実際のバイトが処理されるのに対して、文字列の処理はロケール設定に従うということです。まとめると、バイナリ列はプログラマが「バイト列そのもの」と考えるものを格納するのに適し、文字列はテキストを格納するのに適しています。

bytea型は入出力用に2つの書式をサポートします。「hex」書式とPostgreSQLの歴史的な「エスケープ」書式です。入力ではこれらの両方とも常に受け入れられます。出力書式はbytea_output設定パラメータに依存し、デフォルトではhexです。（hex書式はPostgreSQL 9.0から導入されたものであることに注意してください。以前のバージョンや一部のツールではこれを理解しません。）

標準SQLは、BLOBまたはBINARY LARGE OBJECTという、異なるバイナリ列型を定義します。入力書式はbyteaと異なりますが、提供される関数および演算子はほぼ同じです。

8.4.1. byteaのhex書式

「hex」書式ではバイナリデータの各バイトを上位4ビット、下位4ビットの順で2桁の16進数に符号化します。（エスケープ書式と区別するために）文字列全体は\xという並びの後に付けられます。一部の文脈では、先頭のバックスラッシュを二重にしてエスケープさせる必要があるかもしれません(以下を参照 4.1.2.1)。これはエスケープ書式でバックスラッシュを二重にしなければならない場合と同じで、詳細は以下に示します。入力する16進数の桁は大文字でも小文字でも構いません。数字のペアの間に空白文字を入れることができます。（しかし桁の組み合わせの間や先頭の\xの間には入れることはできません。）hex書式は外部のアプリケーションおよびプロトコルの間で広く互換性を持ち、また、エスケープ書式と比べ変換が高速になる傾向があります。このため使用が好まれます。

例

```
SELECT '\xDEADBEEF';
```

8.4.2. byteaのエスケープ書式

「エスケープ」書式はbytea型用の伝統的なPostgreSQLの書式です。これは、バイナリ列をASCII文字の並びとして表現しASCII文字として表現できないバイトは特殊なエスケープシーケンスとして表現するという方式を取ります。アプリケーションの見地から文字として表現されたバイトが有意であれば、この表現は簡便です。しかし現実にはバイナリ列と文字列の間の区別があいまいになりますので、通常は混乱します。また選択されたこのエスケープ機構自体が多少非効率的です。このためこの書式はおそらくほとんどの新しいアプリケーションでは避けるべきでしょう。

エスケープ書式でbytea値を入力する際に、特定の値のオクテットをエスケープする必要があります。なお、すべてのオクテットの値をエスケープすることができます。一般的にあるオクテットをエスケープするには、それをその3桁の8進数に変換し、バックスラッシュを前に付けます。他にもバックスラッシュ自体(10進数表記のオクテットで92)を二重のバックスラッシュとして表現することができます。表 8.7には、エスケープする必要がある文字と、その適用可能な代替エスケープシーケンスを示しています。

表8.7 オクテットをエスケープしたbyteaリテラル

10進オクテット値	説明	エスケープされた入力表現	例	出力表現
0	ゼロオクテット	'\000'	'\000'::bytea;	\x00
39	単一引用符	''''もしくは'\047'	''''::bytea;	\x27
92	バックスラッシュ	'\\'もしくは'\\134'	'\\'::bytea;	\x5c
0から31まで、および127から255まで	「表示できない」オクテット	'\xxx' (8進数)	'\001'::bytea;	\x01

実際には、表示できないオクテットに対するエスケープ要求はロケールの設定に依存して異なります。ロケールの設定によっては、エスケープをしないで済むこともあります。

表 8.7で示したように、シングルクォートが二重に必要な理由は、SQLコマンド中のあらゆる文字列に当てはまるためです。一般的な文字列パーサは最も外側のシングルクォートを消費し、シングルクォートのペアを一つの文字データに減らします。byteaを入力する関数が見るのは単に一つのシングルクォートであり、一個の単純なデータ文字として扱います。しかし、byteaを入力する関数はバックスラッシュを特別なものとして扱い、表 8.7に示されているその他の動作はこの関数で実装されています。

一般的な文字列パーサはバックスラッシュのペアを一つの文字データに減らすため、文脈によってはバックスラッシュは上記に見られるように、重ねる必要があります。4.1.2.1も参照ください。

Byteaオクテットはデフォルトではhex書式で出力されます。bytea_outputをescapeに変えると、「表示できない」オクテットは先頭にバックスラッシュがついた3桁のオクテットの値に変換されます。ほとんどの「表示可能な」オクテットはクライアント文字セットの標準的な表示で出力されます。例:

```
SET bytea_output = 'escape';

SELECT 'abc \153\154\155 \052\251\124'::bytea;
     bytea
----------------
 abc klm *\251T
```

10進数で92(バックスラッシュ)を持つオクテットは出力時に二重になります。詳細は表 8.8を参照してください。

表8.8 bytea出力のエスケープされたオクテット

10進オクテット値	説明	エスケープされた出力表現	例	出力結果
92	バックスラッシュ	\\	'\134'::bytea	\\

10進オクテット値	説明	エスケープされた出力表現	例	出力結果
0から31および127から255	「表示できない」オクテット	\xxx(8進数)	'\001'::bytea;	\001
32から126	「表示できる」オクテット	クライアント文字セットにおける表現	'\176'::bytea;	~

使用するPostgreSQLのフロントエンドによっては、bytea文字列をエスケープまたはアンエスケープする追加的な作業が必要になることがあります。例えば、使用するインタフェースが改行文字や復帰文字を自動的に翻訳してしまう場合、これらの文字もエスケープしなければならないかもしれません。

8.5. 日付/時刻データ型

PostgreSQLでは、表 8.9に示されているSQLの日付と時刻データ型のすべてがサポートされています。これらのデータ型で利用できる演算については9.9で説明します。グレゴリオ暦が導入されるより前の年であっても(B.6参照)、日付はグレゴリオ暦にしたがって計算されます。

表8.9 日付/時刻データ型

型名	格納サイズ	説明	最遠の過去	最遠の未来	精度
timestamp [(p)] [without time zone]	8 バイト	日付と時刻両方(時間帯なし)	4713 BC	294276 AD	1マイクロ秒
timestamp [(p)] with time zone	8バイト	日付と時刻両方、時間帯付き	4713 BC	294276 AD	1マイクロ秒
date	4バイト	日付(時刻なし)	4713 BC	5874897 AD	1日
time [(p)] [without time zone]	8バイト	時刻(日付なし)	00:00:00	24:00:00	1マイクロ秒
time [(p)] with time zone	12 バイト	時刻(日付なし)、時間帯付き	00:00:00+1559	24:00:00-1559	1 マイクロ秒
interval [fields] [(p)]	16バイト	時間間隔	-178000000年	178000000年	1マイクロ秒

> **注記**
>
> 標準SQLでは、単なるtimestampという記述はtimestamp without time zoneと同じであることを要求します。PostgreSQLはこれに準じます。timestamp with time zoneはtimestamptzと省略することができますが、これはPostgreSQLの拡張です。

time、timestampおよびintervalは秒フィールドに保有されている小数点以下の桁数を指定する精度値pをオプションで受け付けます。デフォルトでは、精度についての明示的な限界はありません。pの許容範囲は0から6です。

intervalデータ型には追加のオプションがあり、以下の1つの語句を使用して格納されるフィールドの集合を制約します。

```
YEAR
MONTH
DAY
HOUR
MINUTE
SECOND
YEAR TO MONTH
DAY TO HOUR
DAY TO MINUTE
DAY TO SECOND
HOUR TO MINUTE
HOUR TO SECOND
MINUTE TO SECOND
```

fieldsおよびpが共に指定されると、精度は秒のみに適用されるので、fieldsはSECONDを含まなければならないことに注意してください。

time with time zoneは標準SQLで定義されていますが、その定義は、その有用性を疑問視することになりかねない特性を示しています。ほとんどの場合、date、time、timestamp without time zone、timestamp with time zoneの組み合わせで、すべてのアプリケーションで要求される日付/時刻機能すべてを提供しているはずです。

8.5.1. 日付/時刻の入力

日付と時刻の入力は、ISO 8601、SQL互換、伝統的なPOSTGRES、その他を含むほとんどの適正とみなされる書式を受け付けます。一部の書式では日付の入力における日-月-年の順序が曖昧ですが、これらのフィールドの期待される順序を指定する方式が提供されています。DateStyleパラメータをMDYに設定すれば、月日年という順で解釈され、DMYに設定すれば日月年という順で、YMDに設定すれば年月日という順で解釈されます。

PostgreSQLは日付/時刻入力の取扱いにおいて標準SQLの要求よりも柔軟です。日付/時刻の入力における厳密な構文解析規則と、月および週、そして時間帯を含む使用可能なテキストフィールドに関しては付録Bを参照してください。

日付や時刻リテラルの入力では、テキスト文字列のように、単一引用符で囲む必要があることを思い出してください。詳細は4.1.2.7を参照してください。SQLでは下記の構文が要求されます。

```
type [ (p) ] 'value'
```

ここで、pは秒フィールドの小数点以下の桁数を与えるオプションの精度の指定です。精度はtime、timestampおよびinterval型に対して0から6の範囲で設定できます。値の許容範囲は既に説明しています。定数指定において精度指定がない場合は、リテラル値の精度がデフォルトとして使われます(ただし、6桁を超えることはありません)。

8.5.1.1. 日付

表 8.10はdate型で入力可能なものの一部を示します。

表8.10 日付入力

例	説明
1999-01-08	ISO 8601。すべてのモードで1月8日になります（推奨書式）。
January 8, 1999	すべてのdatestyle入力モードにおいて曖昧さがありません。
1/8/1999	MDYモードでは1月8日、DMYモードでは8月1日。
1/18/1999	MDYモードでは1月18日、他のモードでは拒絶されます。
01/02/03	MDYモードでは2003年1月2日、DMYモードでは2003年2月1日、YMDモードでは2001年2月3日。
1999-Jan-08	すべてのモードで1月8日になります。
Jan-08-1999	すべてのモードで1月8日になります。
08-Jan-1999	すべてのモードで1月8日になります。
99-Jan-08	YMDモードで1月8日、他のモードではエラー。
08-Jan-99	1月8日。ただしYMDモードではエラー。
Jan-08-99	1月8日。ただしYMDモードではエラー。
19990108	ISO 8601。すべてのモードで1999年1月8日になります。
990108	ISO 8601。すべてのモードで1999年1月8日になります。
1999.008	年と年間通算日
J2451187	ユリウス日
January 8, 99 BC	西暦紀元前99年

8.5.1.2. 時刻

ある一日の時刻を表す型はtime [(p)] without time zoneとtime [(p)] with time zoneです。time単独ではtime without time zoneと同じです。

これらの型への有効な入力は、時刻、その後にオプションで時間帯からなります。（表 8.11と表 8.12を参照してください。）time without time zoneへの入力に時間帯が指定された場合、時間帯は警告なく無視されます。また、日付を指定することもできますが、America/New_Yorkのような夏時間規則を含む時間帯名を使用しているのでなければ、それは無視されます。夏時間規則のある時間帯名の場合は、標準と夏時間のどちらを適用するかを決定できるように、日付の指定が必要です。適切な時間帯オフセットはtime with time zone型の値に記録されています。

表8.11 時刻入力

例	説明
04:05:06.789	ISO 8601

例	説明
04:05:06	ISO 8601
04:05	ISO 8601
040506	ISO 8601
04:05 AM	04:05と同じ。AMは値に影響を与えない。
04:05 PM	16:05と同じ。時の入力は12以下でなければなりません。
04:05:06.789-8	ISO 8601
04:05:06-08:00	ISO 8601
04:05-08:00	ISO 8601
040506-08	ISO 8601
04:05:06 PST	省略形による時間帯の指定。
2003-04-12 04:05:06 America/New_York	名前による時間帯の指定。

表8.12 時間帯入力

例	説明
PST	省略形（米国太平洋標準時間）
America/New_York	完全な時間帯名
PST8PDT	POSIX書式の時間帯指定
-8:00	ISO 8601。PST用のオフセット
-800	ISO 8601。PST用のオフセット
-8	ISO 8601。PST用のオフセット
zulu	UTC用の軍事用略記
z	zuluの略記

時間帯の指定方法に関する詳細は8.5.3を参照してください。

8.5.1.3. タイムスタンプ

タイムスタンプ型への有効な入力は、日付と時刻を連結し、さらにその後にオプションで時間帯、その後にオプションでADもしくはBCからなります。（他にAD/BCを時間帯の前に付ける方法もありますが、これは推奨される順序ではありません。）したがって、

```
1999-01-08 04:05:06
```

と

```
1999-01-08 04:05:06 -8:00
```

は有効な値で、ISO 8601に準拠しています。また、広く使用されている

```
January 8 04:05:06 1999 PST
```

という書式もサポートされます。

標準SQLでは、timestamp without time zoneのリテラルとtimestamp with time zoneのリテラルを、時刻の後の「+」もしくは「-」記号と時間帯補正の有無により区別します。そのため、標準に従うと、

```
TIMESTAMP '2004-10-19 10:23:54'
```

はtimestamp without time zoneに、

```
TIMESTAMP '2004-10-19 10:23:54+02'
```

はtimestamp with time zoneになります。PostgreSQLでは、その型を決める前に文字列リテラルの内容を検証しません。そのため上の例はいずれもtimestamp without time zoneとして扱います。リテラルが確実にtimestamp with time zoneとして扱われるようにするには、例えば、

```
TIMESTAMP WITH TIME ZONE '2004-10-19 10:23:54+02'
```

のように正しい明示的な型を指定してください。timestamp without time zoneと決定済みのリテラルでは、PostgreSQLは警告なく時間帯情報をすべて無視します。つまり、結果の値は明示された入力値の日付/時刻フィールドから持ち込まれますが、時間帯の調整はなされません。

timestamp with time zoneについて内部に格納されている値は常にUTCです（協定世界時、歴史的にグリニッジ標準時GMTとして知られています）。時間帯が明示的に指定された入力値は、その時間帯に適したオフセットを使用してUTCに変換されます。入力文字列に時間帯が指定されていない場合は、システムのTimeZoneパラメータに示されている値が時間帯とみなされ、timezone時間帯用のオフセットを使用してUTCに変換されます。

timestamp with time zoneの値が出力されると、この値はUTCから現行のtimezoneに変換され、その時間帯のローカル時間として表示されます。他の時間帯での時間を表示するには、timezoneを変更するか、あるいはAT TIME ZONE構文（9.9.3 を参照）を使用します。

timestamp without time zoneとtimestamp with time zoneの間の変換では、通常timestamp without time zoneの値はtimezoneのローカル時間としてみなされる、または、指定されるものと想定されます。AT TIME ZONEを使用する変換では、異なる時間帯を指定できます。

8.5.1.4. 特別な値

PostgreSQLでは利便性のために、表 8.13に示されているような特別な日付/時刻入力値をサポートしています。infinityと-infinityの値は、特別にシステム内部で表現され、変更されずに表示されます。他のものは、単に簡略化された表記で、読み込まれるときに通常の日付/時刻値に変換されます。（特にnowとその関連文字列は読み込まれるとすぐにその時点の値に変換されます。）これらの値はすべて、SQLコマンドで定数として使う場合は、単一引用符でくくらなければなりません。

表8.13 特別な日付/時刻定数

入力文字列	有効な型	説明
epoch	date, timestamp	1970-01-01 00:00:00+00（Unixシステム時間におけるゼロ）
infinity	date, timestamp	他のすべてのタイムスタンプより将来
-infinity	date, timestamp	他のすべてのタイムスタンプより過去
now	date, time, timestamp	現トランザクションの開始時刻
today	date, timestamp	今日の午前0時
tomorrow	date, timestamp	明日の午前0時
yesterday	date, timestamp	昨日の午前0時
allballs	time	00:00:00.00 UTC

SQL互換の関数である、CURRENT_DATE、CURRENT_TIME、CURRENT_TIMESTAMP、LOCALTIME、LOCALTIMESTAMPも、対応するデータ型の現在の日付または時間の値を取得するために使用できます。（9.9.4を参照してください。）これらはSQL関数であり、データ入力文字列として認識されないことに注意してください。

> **注意**
>
> 入力する文字列としてnow、today、tomorrow及びyesterdayはインタラクティブなSQLコマンドの中で利用する時は良いですが、コマンドが保存され後に実行されるような時には驚く挙動になることがあります。例えば、準備された文、ビューや関数の定義です。文字列は特定の時間の値に変換され、その値が古くなってからもしばらく使い続けられることがあります。このような状況では、代わりにSQL関数を使用してください。例えば、'tomorrow'::dateよりもCURRENT_DATE + 1の方が安全です。

8.5.2. 日付/時刻の出力

日付/時刻型の出力書式は、ISO 8601、SQL（Ingres）、伝統的なPOSTGRES（Unix date書式）またはGermanの4つのいずれかに設定できます。デフォルトはISO書式です。（標準SQLではISO 8601書式の使用が定められています。「SQL」という出力書式名は歴史的な事故です。）表 8.14に各出力書式の例を示します。dateとtimeの書式は、例にあるとおり、それぞれ日付と時刻の部分です。しかし、POSTGRESではISO書式の日付部分のみを出力します。(YMDやMDYの場合12-17-1997を返し、DMYの場合17-12-1997を返します。)

表8.14 日付/時刻の出力形式

様式指定	説明	例
ISO	ISO 8601, 標準SQL	1997-12-17 07:37:16-08
SQL	伝統的な様式	12/17/1997 07:37:16.00 PST
Postgres	独自の様式	Wed Dec 17 07:37:16 1997 PST
German	地域限定様式	17.12.1997 07:37:16.00 PST

> **注記**
>
> ISO 8601の仕様では日付と時刻を区切るために大文字のTを使用します。PostgreSQLは入力ではこの書式を受け付けますが、上記のように出力ではTではなく空白を使用します。これは読みやすさのため、そしてRFC3339や他のデータベースシステムとの整合性を保つためです。

SQLとPOSTGRESでは、DMYフィールド順が指定された場合は月の前に日が現れます。指定がなければ日の前に月が現れます。(この設定が入力値の解釈にどう影響を与えるのかについては8.5.1を参考にしてください)。表 8.15に例を示します。

表8.15 日付の順序の慣習

datestyleの設定	入力の順序	出力例
SQL, DMY	day(日)/month(月)/year(年)	17/12/1997 15:37:16.00 CET
SQL, MDY	month(月)/day(日)/year(年)	12/17/1997 07:37:16.00 PST
Postgres, DMY	day(日)/month(月)/year(年)	Wed 17 Dec 07:37:16 1997 PST

ユーザはSET DATESTYLEコマンド、postgresql.conf構成ファイルのDateStyleパラメータ、そしてサーバかクライアントのPGDATESTYLE環境変数を使用して、日付/時刻の様式を選択することができます。

日付/時刻出力のより柔軟な書式設定方法として、書式設定関数to_char(9.8を参照)を使用することもできます。

8.5.3. 時間帯

時間帯および時間帯の取り決めは地球の幾何学的要素のみでなく政治的決定に影響されます。世界にまたがる時間帯は1900年代に標準化されたようですが、特に夏時間規則の点で、勝手に変更する傾向が続いています。PostgreSQLは歴史的な時間帯ルールについての情報に、広く使われているIANA時間帯データベースを使用します。将来の時間は、ある与えられた時間帯に対する最新の既知のルールが、将来長きに渡りそのまま遵守が継続されるということを前提としています。

PostgreSQLは典型的な使用法については標準SQLへの互換性に対し最大限の努力をしています。しかし、標準SQLには、日付と時刻のデータ型と機能に関する混乱が見受けられます。2つの明らかな問題点を以下に示します。

- date型にはそれに関連する時間帯を持てませんが、time型にはあります。現実の世界において、時間帯のオフセットが夏時間への切り替えにより年間を通じて変化することから、時刻と同様に日付もそれに結び付けられていないと意味がありません。
- デフォルトの時間帯はUTCからの整数定数オフセットとして指定されています。したがってDST(夏時間)への切り替えをまたいで日付/時刻演算を行う場合、夏時間を適用することは不可能です。

このような問題を解決するためには、時間帯を使用する際に日付と時刻の両方を保持できる日付/時刻データ型を使用することを勧めます。time with time zone型の使用はお勧めしません(もっともPostgreSQLでは、旧式のアプリケーションや、標準SQLとの互換性のために、time with time zone型の使用をサポートし

ています)。PostgreSQLは、日付または時刻のみを保持するデータ型のすべては使用中の時間帯であると前提しています。

すべての時間帯付きの日付と時刻はUTCで内部的に保存されます。これらはクライアントに表示される前にTimeZone設定パラメータで指定された時間帯におけるローカル時間に変換されます。

PostgreSQLでは、3つの形式で時間帯を指定することができます。

- America/New_Yorkなどの完全な時間帯名称。認識できる時間帯名称はpg_timezone_namesビューに列挙されています(51.92を参照してください)。PostgreSQLはこの目的のためによく使用されているIANA時間帯データを使用します。したがって、他のソフトウェアでも同じ名前が認識されます。

- PSTなどの時間帯省略形。こうした指定は、単に特定のUTCからのオフセットを定義します。一方、完全な時間帯名称では夏時間遷移規則群も組み込まれます。認識可能な省略形はpg_timezone_abbrevsビューに列挙されています(51.91を参照してください)。TimeZone設定パラメータおよびlog_timezone設定パラメータを時間帯省略形に設定することはできませんが、日付時刻型の入力値やAT TIME ZONE演算子に省略形を使用することができます。

- 時間帯名やその省略形に加え、PostgreSQLは、B.5に記載されているPOSIX様式の時間帯指定を受付けます。このオプションは通常、名前付きのタイムゾーンを使用するよりも好ましくありませんが、適切なIANAタイムゾーンのエントリが利用できない場合は必要になるかもしれません。

一言で言うと、これは省略形と正式名称との差異です。省略形はUTCから固定したオフセットを表わすのに対して、多くの正式名称はローカルの夏時間規定を暗示するので、2つのUTCオフセットがあるかもしれません。例えば2014-06-04 12:00 America/New_Yorkはニューヨークの正午を示しますが、これはこの日について言えば東部夏時間(UTC-4)です。つまり2014-06-04 12:00 EDTはこれと同時刻を示します。しかし、2014-06-04 12:00 ESTは、その日に夏時間が使用されていたかどうかに関わらず、東部標準時間(UTC-5)での正午を示します。

問題を更に複雑にしているのは、一部の管轄は同じ略号を使って、年によって異なるUTCオフセットを表していることです。例えばモスクワではMSKはある年ではUTC+3を意味しますが、別の年ではUTC+4を意味します。PostgreSQLではそのような略号について、指定の日に何を意味していたか(あるいは最も最近にどういう意味だったか)に従って解釈します。しかし、ESTの例にあるように、必ずしもその日付における地方常用時を示しているとは限りません。

すべての場合において、時間帯名や略号は大文字小文字の区別なく認識されます。(これはPostgreSQLの8.2より前のバージョンからの変更です。以前は、文脈によって大文字小文字が区別される場合と、されない場合がありました。)

時間帯名、省略形のどちらもサーバ内に組み込まれるわけではありません。インストールディレクトリの.../share/timezone/および.../share/timezonesets/の下に保存される構成ファイルから取得されます(B.4を参照ください)。

TimeZoneはpostgresql.confファイルや第19章で説明する他の標準的な方法で設定することができます。以下に、いくつか特別な設定方法を示します。

- SQLコマンドSET TIME ZONEはセッションの時間帯を設定します。これはSET TIMEZONE TOの別名ですが、SQL仕様の構文へのより高い互換性があります。

- PGTZ環境変数は、libpqクライアントが接続時にサーバにSET TIME ZONEコマンドを送信するために用いられます。

8.5.4. 時間間隔の入力

interval値は以下の冗長な構文を使って記述されます。

```
[@] quantity unit [quantity unit...] [direction]
```

ここで、quantityは(符号付き)時間量、unit(単位)はmicrosecond、millisecond、second(秒)、minute(分)、hour(時)、day(日)、week(週)、month(月)、year(年)、decade(10年単位)、century(100年単位)、millennium(1000年単位)あるいはこれらの単位の簡略形または複数形です。direction(方向)はagoもしくは空です。アットマーク(@)はオプションで、付けても付けなくても構いません。異なる単位における時間量は適切に符号を考慮して暗黙的に足されます。agoはすべてのフィールドの正負を逆にします。この構文はまた、IntervalStyleがpostgres_verboseに設定されている場合に時間間隔の出力でも使用されます。

日、時、分、および秒の時間量は明示的に単位を指定しなくても構いません。例えば、'1 12:59:10'は'1 day 12 hours 59 min 10 sec'(1日と12時間59分10秒)と解釈されます。また年と月の組み合わせはダッシュを使って指定することができます。例えば、'200-10'は'200 years 10 months'(200年と10か月)と解釈されます。(実際のところ、標準SQLで許されている簡略形はこれらだけです。そしてIntervalStyleがsql_standardに設定されている場合には、これらが出力で使用されます。)

標準の4.4.3.2節の「指定文字付書式」または4.4.3.3節の「代替書式」のどちらかを使用して、時間間隔値はISO 8601時間間隔として書くこともできます。指定文字付の書式は以下のようなものです。

```
P quantity unit [ quantity unit ...] [ T [ quantity unit ...]]
```

文字列はPで始まらなければならず、また、日と時間を区切るTを含めることができます。利用可能な単位の省略形を表 8.16に示します。単位は省略しても構いませんし、任意の順番で指定できますが、1日より小さな単位はTの後に書かなければなりません。特にMの意味はTの前にあるか後にあるかに依存します。

表8.16 ISO 8601における時間間隔単位の省略形

省略形	意味
Y	年
M	月(日付部分における)
W	週
D	日
H	時間
M	分(時刻部分における)
S	秒

別の書式を示します。

```
P [ years-months-days ] [ T hours:minutes:seconds ]
```

上の代替書式では、文字列はPから始まらなければなりません。そして、Tは時間間隔の日付部分と時刻部分とを分割します。値はISO 8601日付と同様の数字で指定されます。

fields指定を使って時間間隔定数を記述する場合、または、fields仕様で定義された時間間隔列に文字列を割り当てる場合、マークされていない時間量の解釈はfieldsに依存します。例えばINTERVAL '1' YEARは1年と解釈され、一方でINTERVAL '1'は1秒と解釈されます。同時に、fields仕様によって許可される最下位フィールドの「右側の」フィールド値は警告なしに破棄されます。例えば、INTERVAL '1 day 2:03:04' HOUR TO MINUTEと書くことで、二番目のフィールドは削除されますが、日付フィールドは削除されません。

標準SQLに従うと、時間間隔値のフィールドはすべて同じ符号を持たなければなりません。このため、先頭の負の符号はすべてのフィールドに適用されます。例えば時間間隔リテラル'-1 2:03:04'の負の符号は、日付部分にも時、分、秒部分にも適用されます。PostgreSQLではフィールドに異なる符号を持たせることができます。また伝統的にテキスト形式表現における各フィールドは独立した符号を持つものとして扱われます。このため、この例では時、分、秒部分は正であるとみなされます。IntervalStyleがsql_standardに設定されている場合、先頭の符号はすべてのフィールドに適用されるものとみなされます（ただし他に符号がない場合のみです）。さもなくば、伝統的なPostgreSQLの解釈が使用されます。あいまいさを防ぐために、負のフィールドがある場合には個別に明示的な符号を付けることを勧めます。

冗長な入力書式、および、より簡略な書式の一部のフィールドでは、フィールド値は小数部分を持つことができます。例えば'1.5 week'や'01:02:03.45'です。こうした入力は格納の際適切な月数、日数、秒数に変換されます。これが月または日数が小数になる結果となる場合、小数部分は1月は30日、1日は24時間という変換規則を使用して、より低い順のフィールドに加えられます。例えば'1.5 month'は1月と15日となります。秒だけが出力において小数を示すことになります。

表 8.17は有効なinterval入力のいくつかの例を示しています。

表8.17 時間間隔入力

例	説明
1-2	標準SQL書式。1年2ヶ月
3 4:05:06	標準SQL書式。3日4時間5分6秒
1 year 2 months 3 days 4 hours 5 minutes 6 seconds	伝統的Postgres書式。1年2月3日4時間5分6秒
P1Y2M3DT4H5M6S	ISO 8601「指定文字付き書式」。意味は上と同じ
P0001-02-03T04:05:06	ISO 8601「代替書式」。意味は上と同じ

内部的にintervalの値は月、日、秒として格納されます。これはひと月の日数は変化し、一日も夏時間では23時間もしくは25時間になりうるからです。秒の桁は小数で格納できますが、月、日の桁は整数です。時間の間隔は通常、文字列定数もしくはtimestampの引き算で生成されるため、この格納方式はほとんどのケースではうまくいきますが、予期しない結果を返すこともあります。例：

```
SELECT EXTRACT(hours from '80 minutes'::interval);
 date_part
-----------
         1

SELECT EXTRACT(days from '80 hours'::interval);
 date_part
-----------
         0
```

justify_days関数とjustify_hours関数は範囲から溢れた日と時間の補正に役立ちます。

8.5.5. 時間間隔の出力

時間間隔型の出力書式は、SET intervalstyleコマンドを使用して、sql_standard、postgres、postgres_verboseまたはiso_8601の4つのうちの1つを設定できます。デフォルトはpostgres書式です。表 8.18はそれぞれの出力形式を示した例です。

sql_standard形式は、時間間隔値が標準制約（構成要素に正負が混在していない年数と月数のみ、または日数と時間のみ）を満足する場合、時間間隔リテラル文字列に対し標準SQLに準拠する出力を作成します。それ以外の場合、出力は、標準的な年数-月数のリテラル文字列の後に日数-時間のリテラル文字列が続いたものになり、正負混在した時間間隔のあいまいさを無くすために明示的な符号が付加されます。

postgres書式の出力は、DateStyleパラメータがISOに設定されたとき、PostgreSQL 8.4より前のリリースと一致します。

postgres_verbose書式の出力は、DateStyleパラメータがISO以外に設定されたとき、PostgreSQL 8.4より前のリリースと一致します。

iso_8601書式の出力はISO 8601 標準の4.4.3.2節に記述の「format with designators（指名付き書式）」に一致します。

表8.18 時間間隔出力形式の例

形式指定	年-月時間間隔	日-時刻時間間隔	混在した時間間隔
sql_standard	1-2	3 4:05:06	-1-2 +3 -4:05:06
postgres	1 year 2 mons	3 days 04:05:06	-1 year -2 mons +3 days -04:05:06
postgres_verbose	@ 1 year 2 mons	@ 3 days 4 hours 5 mins 6 secs	@ 1 year 2 mons -3 days 4 hours 5 mins 6 secs ago
iso_8601	P1Y2M	P3DT4H5M6S	P-1Y-2M3DT-4H-5M-6S

8.6. 論理値データ型

PostgreSQLでは、標準SQLのboolean型が提供されています。表 8.19を参照してください。boolean型はいくつかの状態を取ることができます。「真」もしくは「偽」、そして第3の状態はSQLではNULL値で表現される「不明」の状態です。

表8.19 論理値データ型

名前	格納サイズ	説明
boolean	1バイト	真または偽の状態

論理定数はSQL問い合わせの中で、SQLキーワードのTRUE、FALSEおよびNULLによって表現できます。

booleanのデータ型を入力する関数には次の文字列表現を「真」の状態として使うことができます。

```
true
yes
on
1
```

「偽」の状態には以下の表現が使用できます。

```
false
no
off
0
```

t や nなど、これらの文字列固有の接頭辞も利用できます。先頭または末尾の空白文字は無視され、大文字小文字の区別は関係ありません。

booleanのデータ型を出力する関数は例 8.2にあるように、常にtかfを出力します。

例8.2 boolean型の使用

```
CREATE TABLE test1 (a boolean, b text);
INSERT INTO test1 VALUES (TRUE, 'sic est');
INSERT INTO test1 VALUES (FALSE, 'non est');
SELECT * FROM test1;
 a |    b
---+---------
 t | sic est
 f | non est

SELECT * FROM test1 WHERE a;
 a |    b
---+---------
 t | sic est
```

キーワードであるTRUE とFALSEはSQLクエリの中で論理定数の記述として好ましい(SQL準拠)方式です。しかし、4.1.2.7のリンクで記述されている、以下のような一般的な文字列リテラル定数の構文に従って'yes'::booleanというような文字表現することもできます。

パーサーは自動的にTRUEとFALSEはboolean型と理解しますが、NULLは他のすべての型に存在するため、boolean型と理解しない点に気をつけてください。このため、コンテキストによってはNULL::booleanというように、NULLをbooleanに明確にキャストする必要があります。逆に、解析でリテラルがboolean型でなければならないと推論できるコンテキストでは、文字列リテラルブール値のキャストは省略できます。

8.7. 列挙型

列挙(enum)型は静的、順序付き集合から構成されるデータ型です。これは、多くのプログラミング言語でサポートされているenum型と同じです。列挙型の例として、曜日や個々のデータについての状態値の集合が挙げられます。

8.7.1. 列挙型の宣言

列挙型はCREATE TYPEコマンドを使用して作成されます。以下に例を示します。

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
```

作成後、他のデータ型とほとんど同じように、列挙型をテーブルや関数定義で使用することができます。

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
CREATE TABLE person (
    name text,
    current_mood mood
);
INSERT INTO person VALUES ('Moe', 'happy');
SELECT * FROM person WHERE current_mood = 'happy';
 name | current_mood
------+--------------
 Moe  | happy
(1 row)
```

8.7.2. 順序

列挙型内の値の順序はその型が作成された時に値を列挙した順番になります。列挙型に対して、すべての比較演算子と関連する集約関数がサポートされます。以下に例を示します。

```
INSERT INTO person VALUES ('Larry', 'sad');
INSERT INTO person VALUES ('Curly', 'ok');
SELECT * FROM person WHERE current_mood > 'sad';
 name  | current_mood
-------+--------------
 Moe   | happy
 Curly | ok
(2 rows)

SELECT * FROM person WHERE current_mood > 'sad' ORDER BY current_mood;
 name  | current_mood
-------+--------------
 Curly | ok
 Moe   | happy
(2 rows)

SELECT name
FROM person
WHERE current_mood = (SELECT MIN(current_mood) FROM person);
 name
```

```
-------
 Larry
(1 row)
```

8.7.3. 型の安全性

それぞれの列挙型データ型は別個のもので、他の列挙型と比較することはできません。以下の例を参照してください。

```
CREATE TYPE happiness AS ENUM ('happy', 'very happy', 'ecstatic');
CREATE TABLE holidays (
    num_weeks integer,
    happiness happiness
);
INSERT INTO holidays(num_weeks,happiness) VALUES (4, 'happy');
INSERT INTO holidays(num_weeks,happiness) VALUES (6, 'very happy');
INSERT INTO holidays(num_weeks,happiness) VALUES (8, 'ecstatic');
INSERT INTO holidays(num_weeks,happiness) VALUES (2, 'sad');
ERROR:  invalid input value for enum happiness: "sad"
SELECT person.name, holidays.num_weeks FROM person, holidays
  WHERE person.current_mood = holidays.happiness;
ERROR:  operator does not exist: mood = happiness
```

もし本当に上のようなことが必要ならば、独自の演算子を作成するか、問い合わせに明示的なキャストを付けることで行うことができます。

```
SELECT person.name, holidays.num_weeks FROM person, holidays
  WHERE person.current_mood::text = holidays.happiness::text;
 name | num_weeks
------+-----------
 Moe  |         4
(1 row)
```

8.7.4. 実装の詳細

列挙型のラベルは大文字小文字の違いを意識します。このため、'happy'と'HAPPY'は同じではありません。同様にラベルの中の空白も重要です。

列挙型は主に静的な値のセットを対象としていますが、既存の列挙型に新しい値を加えることや名前を変更することをサポートしています(ALTER TYPEを参照)。ただし、列挙型を削除して再作成せずに、既存の列挙型からラベルを削除することやソート順が変わる値に変更することはできません。

列挙型の値はディスク上では4バイトを占めます。列挙型の値のテキストラベルの長さは、PostgreSQLに組み込まれたNAMEDATALEN設定により制限されます。標準のビルドでは、これは最大63バイトを意味します。

列挙型の内部値からテキスト形式のラベルへの変換は、pg_enumシステムカタログ内に保持されます。このカタログを直接問い合わせることが役に立つ場合があります。

8.8. 幾何データ型

幾何データ型は2次元空間オブジェクトを表現します。表 8.20は、PostgreSQLで使用可能な幾何データ型を列挙したものです。

表8.20 幾何データ型

型名	格納サイズ	説明	表現
point	16バイト	平面における座標点	(x,y)
line	32バイト	無限の直線	{A,B,C}
lseg	32バイト	有限の線分	((x1,y1),(x2,y2))
box	32バイト	矩形	((x1,y1),(x2,y2))
path	16+16nバイト	閉経路(多角形に類似)	((x1,y1),...)
path	16+16nバイト	開経路	[(x1,y1),...]
polygon	40+16nバイト	多角形(閉経路に類似)	((x1,y1),...)
circle	24バイト	円	<(x,y),r>(中心と半径)

拡大縮小、平行移動、回転、交点の算出といった様々な幾何学的操作を行う関数と演算子の集合が豊富に揃っています。このことについては9.11に説明があります。

8.8.1. 座標点

座標点は幾何データ型の基礎となる2次元構成要素です。point型の値は次の構文のいずれかで指定されます。

```
( x , y )
  x , y
```

ここでxとyは、それぞれの座標を浮動小数点数数値で表したものです。

座標点は1番目の構文で出力されます。

8.8.2. 直線

直線は線形方程式Ax + By + C = 0で表現されます。ここでAとBは同時に0になることはありません。line型の値は以下の書式で入出力されます。

```
{ A, B, C }
```

入力のためには以下の書式を代替として使用することもできます。

```
[ ( x1 , y1 ) , ( x2 , y2 ) ]
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
```

ここで(x1,y1)と(x2,y2)はその直線上の2つの異なる点です。

8.8.3. 線分

線分は終点を示す2つの点の組み合わせで表現されます。lseg型の値は以下の構文のいずれかで指定されます。

```
[ ( x1 , y1 ) , ( x2 , y2 ) ]
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
```

ここで、(x1,y1)と(x2,y2)は線分の終端点です。

線分は1番目の構文で出力されます。

8.8.4. 矩形

矩形は、矩形の対角線の両端の座標点の組み合わせで表されます。box型の値は以下の構文のいずれかで指定されます。

```
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
```

ここで(x1,y1)と(x2,y2)は矩形の対角線の両端です。

矩形は最初に示した構文で出力されます。

任意の対角頂点を入力として指定することができます。しかし頂点は右上の頂点を最初に、左下の頂点をその後に格納するよう必要に応じて並べ替えられます。

8.8.5. 経路

経路は接続している座標点のリストで表現されます。経路は最初の座標点と最後の座標点が接続されていないとみなされる開いている状態か、最初の座標点と最後の座標点が接続されているとみなされる閉じた状態かのいずれかです。

path型の値は次の構文のいずれかで指定されます。

```
[ ( x1 , y1 ) , ... , ( xn , yn ) ]
```

```
( ( x1 , y1 ) , ... , ( xn , yn ) )
  ( x1 , y1 ) , ... , ( xn , yn )
  ( x1 , y1   , ... ,   xn , yn )
    x1 , y1   , ... ,   xn , yn
```

ここで、各座標点は、経路を構成する線分の終端点です。大括弧([])は開経路を、括弧(())は閉経路を示します。3番目から4番目の構文のようにもっとも外側の括弧が省略された場合、閉経路と仮定されます。

経路は最初または2番目の適切な構文で出力されます。

8.8.6. 多角形(ポリゴン)

多角形は座標点(多角形の頂点)のリストで表現されます。多角形は閉経路ととても良く似ていますが、異なった方式で格納されると同時にそれぞれ独自のサポート関数群を持っています。

polygon型の値は次の構文のいずれかで指定されます。

```
( ( x1 , y1 ) , ... , ( xn , yn ) )
  ( x1 , y1 ) , ... , ( xn , yn )
  ( x1 , y1   , ... ,   xn , yn )
    x1 , y1   , ... ,   xn , yn
```

各座標点は多角形の境界を構成する線分の終端点です。

多角形は最初の構文で出力されます。

8.8.7. 円

円は中心座標点と半径で表現されます。circle型の値は次の構文のいずれかで指定されます。

```
< ( x , y ) , r >
( ( x , y ) , r )
  ( x , y ) , r
    x , y   , r
```

ここで(x,y)は円の中心点、rは円の半径です。

円は最初の構文で出力されます。

8.9. ネットワークアドレス型

PostgreSQLは、表 8.21に示すように、IPv4アドレス、IPv6アドレス、MACアドレスを格納するデータ型を提供します。ネットワークアドレスを格納するには普通のテキストデータ型の代わりにこれらの型を使うことの方が優れています。なぜなら、これらのデータ型は入力値のエラー検査と専用の演算子と関数を提供しているからです (9.12を参照してください)。

表8.21 ネットワークアドレスデータ型

名前	格納サイズ	説明
cidr	7もしくは19バイト	IPv4、およびIPv6ネットワーク
inet	7もしくは19バイト	IPv4もしくはIPv6ホスト、およびネットワーク
macaddr	6バイト	MACアドレス
macaddr8	8 バイト	MAC アドレス (EUI-64 形式)

inetもしくはcidrをソートする時、IPv4アドレスは常にIPv6よりも前にソートされます。::10.2.3.4や::ffff:10.4.3.2などIPv6アドレス内に埋め込まれた、もしくは関連付けされたIPv4アドレスも同様です。

8.9.1. inet

inet型はIPv4もしくはIPv6ホストアドレスとオプションでそのサブネットを1つのフィールドに保持します。サブネットはホストアドレス内のネットワークアドレスのビット数(「ネットマスク」)により表現されます。ネットマスクが32でアドレスがIPv4の場合、その値はサブネットを示さず、単一ホストを表します。IPv6ではアドレス長は128ビットですので、128ビットが一意なホストアドレスを指定します。ネットワークのみを使用したい場合はinetではなくcidr型を利用してください。

このデータ型に対する入力書式はaddress/yです。ここで、addressはIPv4またはIPv6のアドレス、yはネットマスクのビット数です。/y部分が省略された場合、ネットマスクはIPv4では32、IPv6では128となり、つまり、 その値は単一ホストを表現します。ネットマスクが単一ホストを表す場合、その表示時、/yの部分は抑制されます。

8.9.2. cidr

cidrデータ型はIPv4、IPv6ネットワーク仕様を保持します。入出力書式はCIDR表記(クラスレスアドレッシング)に従います。ネットワークを指定する時の書式はaddress/yで、addressがIPv4もしくはIPv6アドレスで表したネットワークの最下位アドレスで、yはネットマスクのビット数です。yが省略された場合には、従来のクラス付きアドレス番号指定システムに従って計算されますが、入力時に書き込まれたオクテットすべてが含まれるように大きさは確保されます。指定したネットマスクの右側にビットをセットしたネットワークアドレスを指定するとエラーになります。

表 8.22に例をいくつか示します。

表8.22 cidrデータ型入力例

cidr入力	cidr出力	abbrev(cidr)
192.168.100.128/25	192.168.100.128/25	192.168.100.128/25
192.168/24	192.168.0.0/24	192.168.0/24
192.168/25	192.168.0.0/25	192.168.0.0/25
192.168.1	192.168.1.0/24	192.168.1/24
192.168	192.168.0.0/24	192.168.0/24
128.1	128.1.0.0/16	128.1/16

cidr入力	cidr出力	abbrev(cidr)
128	128.0.0.0/16	128.0/16
128.1.2	128.1.2.0/24	128.1.2/24
10.1.2	10.1.2.0/24	10.1.2/24
10.1	10.1.0.0/16	10.1/16
10	10.0.0.0/8	10/8
10.1.2.3/32	10.1.2.3/32	10.1.2.3/32
2001:4f8:3:ba::/64	2001:4f8:3:ba::/64	2001:4f8:3:ba/64
2001:4f8:3:ba:2e0:81ff:fe22:d1f1/128	2001:4f8:3:ba:2e0:81ff:fe22:d1f1/128	2001:4f8:3:ba:2e0:81ff:fe22:d1f1/128
::ffff:1.2.3.0/120	::ffff:1.2.3.0/120	::ffff:1.2.3/120
::ffff:1.2.3.0/128	::ffff:1.2.3.0/128	::ffff:1.2.3.0/128

8.9.3. inetとcidrデータ型の違い

inetデータ型とcidrデータ型との基本的な相違は、inetではネットマスクの右側に0でないビット値を受け付けますが、cidrでは受け付けないことです。例えば、192.168.0.1/24 はinetでは有効ですが、cidrでは有効ではありません。

> **ヒント**
>
> もしinetもしくはcidrの値の出力書式が気に入らないのであれば、関数host、textおよびabbrevを試してください。

8.9.4. macaddr

macaddrデータ型は例えばイーサネットカードのハードウェアアドレスとして知られるMACアドレスを保持します（MACアドレスは他の目的でも使われますが）。入力は以下の形式を受け入れます。

```
'08:00:2b:01:02:03'
'08-00-2b-01-02-03'
'08002b:010203'
'08002b-010203'
'0800.2b01.0203'
'0800-2b01-0203'
'08002b010203'
```

これらの例はすべて同一のアドレスを指定します。aからfまでの桁は大文字小文字どちらでも構いません。出力は常に最初に示された形式となります。

IEEE標準802-2001では、2番目の書式（ハイフンを使用）をMACアドレスの正規の表現と規定しています。また、最初の書式（コロンを使用）をビット反転記法、つまり08-00-2b-01-02-03は01:00:4D:08:04:0Cであると

規定しています。この規約は現在ではほぼ無視され、古びたネットワーク(トークンリングなど)のみに関連するものです。PostgreSQLではビット反転に関する準備をしていません。また、すべての受付け可能な書式では正規のLSB順を使用します。

残る5つの入力書式はどの標準にも属しません。

8.9.5. macaddr8

macaddr8データ型はイーサーネットカードのハードウェアアドレスなどで知られるEUI-64形式でデータを格納します(MACアドレスは他の目的にもよく使用されます。)。このデータ型は6バイト長と8バイト長の両方の長さのMACアドレスを受け入れることがき、8バイト長の形式で格納します。6バイト形式で与えられたMACアドレスは8バイト長の形式では、それぞれ、4番目と5番目のバイトをFFとFEとして格納されます。IPv6はEUI-48から変換後に7番目のビットに1となるべき設定がなされた修正EUI-64形式を使用する点に注意してください。macaddr8_set7bit関数がこの変換生成を提供します。一般的には(バイト境界上での)16進数の対で構成され、任意に':'、'-' もしくは '.'のいずれかの一貫した記号で分割された入力を受け付けます。16進数の桁数は16桁(8 バイト)か12桁(6バイト)のいずれかである必要があります。前後の空白は無視されます。以下の入力形式の例は受け付けられます。

```
'08:00:2b:01:02:03:04:05'
'08-00-2b-01-02-03-04-05'
'08002b:0102030405'
'08002b-0102030405'
'0800.2b01.0203.0405'
'0800-2b01-0203-0405'
'08002b01:02030405'
'08002b0102030405'
```

これらの例は全て同じアドレスを指します。桁には大文字のA からF、小文字のa からfも受付けられます。出力は常に1番目の形式です。

上記の最後の6つの形式は標準ではありません。

従来のEUI-48形式の48ビットのMACアドレスからIPv6のホスト部を含む修正がなされたEUI-64形式へ変更するためには、以下に示すようにmacaddr8_set7bitを使用します。

```
SELECT macaddr8_set7bit('08:00:2b:01:02:03');

    macaddr8_set7bit
-------------------------
 0a:00:2b:ff:fe:01:02:03
(1 row)
```

8.10. ビット列データ型

ビット列とは1と0のビットが連続したものです。ビットマスクを格納したり可視化するために使用されます。SQLのビット型には2つあります。bit(n)とbit varying(n)です。ここでnは正の整数です。

bit型のデータはnで表される長さに正確に一致しなければなりません。この長さより長いか短いビット列を格納しようとするとエラーになります。bit varying型のデータは最大nまでの可変長です。最大長を越えるビット列は受け付けません。長さ指定のないbitデータ型はbit(1)データ型と同一で、長さ指定のないbit varyingデータ型は無限長を意味します。

> **注記**
>
> ビット列の値を明示的にbit(n)にキャストすると、厳密にnビットになるように、切り捨てられるか右側をゼロ詰めされ、エラーにはなりません。同様に、ビット列の値を明示的にbit varying(n)にキャストすると、ビット数がnを超える場合は右側が切り捨てられます。

ビット列定数に関する構文についての情報は4.1.2.5を参照してください。ビット論理演算子とビット列操作関数が利用可能ですが、9.6を参照してください。

例8.3 ビット列データ型の使用

```
CREATE TABLE test (a BIT(3), b BIT VARYING(5));
INSERT INTO test VALUES (B'101', B'00');
INSERT INTO test VALUES (B'10', B'101');

ERROR:  bit string length 2 does not match type bit(3)

INSERT INTO test VALUES (B'10'::bit(3), B'101');
SELECT * FROM test;

  a  |  b
-----+-----
 101 | 00
 100 | 101
```

ビット列の値は8ビット毎に1バイト、さらにビット列長に応じた5または8バイトのオーバーヘッドが必要です。(しかし、文字列に関する8.3で説明したように、長い値は圧縮または行外に移動する可能性があります。)

8.11. テキスト検索に関する型

PostgreSQLは、自然言語の文書の集合を通して検索を行い問い合わせに最も合致する文書を見つける機能である全文検索をサポートするために設計された2つのデータ型を提供します。tsvector型はテキスト検索に最適化された形式で文書を表現します。tsquery型は同様に問い合わせを表現します。第12章ではこの機能を詳しく説明します。また、9.13では、関連する関数や演算子を要約します。

8.11.1. tsvector

tsvectorの値は重複がない語彙素のソート済みリストです。語彙素とは同じ単語の変種をまとめるために正規化された単語です(詳細は第12章を参照)。以下の例に示すようにソートと重複除去は入力の際に自動的になされます。

```
SELECT 'a fat cat sat on a mat and ate a fat rat'::tsvector;
                      tsvector
----------------------------------------------------
 'a' 'and' 'ate' 'cat' 'fat' 'mat' 'on' 'rat' 'sat'
```

空白文字または句読点を含む語彙素を表現するには、引用符でくくってください。

```
SELECT $$the lexeme '    ' contains spaces$$::tsvector;
                 tsvector
-------------------------------------------
 '    ' 'contains' 'lexeme' 'spaces' 'the'
```

（この例と次の例では、リテラル内で引用符記号を二重にしなければならないことによる混乱を防ぐためにドル引用符付け文字列を使用します。）引用符およびバックスラッシュが埋め込まれている場合は、以下のように二重にしなければなりません。

```
SELECT $$the lexeme 'Joe''s' contains a quote$$::tsvector;
                    tsvector
-----------------------------------------------
 'Joe''s' 'a' 'contains' 'lexeme' 'quote' 'the'
```

オプションとして、語彙素に整数の位置を付けることもできます。

```
SELECT 'a:1 fat:2 cat:3 sat:4 on:5 a:6 mat:7 and:8 ate:9 a:10 fat:11 rat:12'::tsvector;
                                  tsvector
-------------------------------------------------------------------------------
 'a':1,6,10 'and':8 'ate':9 'cat':3 'fat':2,11 'mat':7 'on':5 'rat':12 'sat':4
```

位置は通常、元の単語の文書中の位置を示します。位置情報を近接順序に使用することができます。位置の値は1から16383までで、これより大きな値は警告なく16383に設定されます。同一語彙素に対する重複する位置項目は破棄されます。

位置を持つ語彙素はさらに重み付きのラベルを付与することができます。ラベルはA、B、C、Dを取ることができます。Dはデフォルトですので、以下のように出力には現れません。

```
SELECT 'a:1A fat:2B,4C cat:5D'::tsvector;
          tsvector
----------------------------
 'a':1A 'cat':5 'fat':2B,4C
```

典型的に重みは、例えば、表題の単語には本文の単語と異なる印をつけるといった、文書構造を反映させるために使用されます。テキスト検索の順序付け関数は異なる重み印に異なる優先度を割り当てることができます。

tsvector型自体は単語の正規化を行わないことを理解することは重要です。与えられる単語はアプリケーションのために適切に正規化されていると仮定しています。以下に例を示します。

```
SELECT 'The Fat Rats'::tsvector;
      tsvector
--------------------
 'Fat' 'Rats' 'The'
```

ほとんどの英文テキスト検索アプリケーションでは、上の単語は正規化されていないとみなされますが、tsvectorは気にしません。検索用に単語を適切に正規化するために、生の文書テキストは通常to_tsvector経由で渡されます。

```
SELECT to_tsvector('english', 'The Fat Rats');
   to_tsvector
-----------------
 'fat':2 'rat':3
```

これについても、詳細は第12章を参照してください。

8.11.2. tsquery

tsqueryの値には検索される語彙素が格納されます。それらは論理演算子& (論理積)、| (論理和)、!(否定)および語句検索演算子<->(FOLLOWED BY)を組み合わせることができます。FOLLOWED BY演算子には<N>という変化形もあり、Nは2つの検索される語彙素の距離を指定する数値型の定数です。<->と<1>は同じです。

括弧を使用して演算子を強制的にグループ化することができます。括弧が無い場合、! (NOT)が最も強く結合し、<-> (FOLLOWED BY)が次に強く結合します。次いで、& (AND)の結合が強く、| (OR)の結合が最も弱くなります。

以下に例を示します:

```
SELECT 'fat & rat'::tsquery;
    tsquery
---------------
 'fat' & 'rat'

SELECT 'fat & (rat | cat)'::tsquery;
          tsquery
---------------------------
 'fat' & ( 'rat' | 'cat' )

SELECT 'fat & rat & ! cat'::tsquery;
        tsquery
------------------------
 'fat' & 'rat' & !'cat'
```

省略することもできますが、tsquery内の語彙素に1つ以上の重み文字でラベルを付けることができます。こうすると、これらの重みを持つtsvector語彙素のみに一致するように制限することになります。

```
SELECT 'fat:ab & cat'::tsquery;
    tsquery
------------------
 'fat':AB & 'cat'
```

同時に、tsquery内の語彙素は、前方一致を指定するため*でラベルを付けることができます。

```
SELECT 'super:*'::tsquery;
  tsquery
-----------
 'super':*
```

この問い合わせでは「super」で始まるtsvector中の全ての言葉と一致します。

語彙素の引用符規則は前に説明したtsvectorにおける語彙素と同じです。また、tsvector同様、必要な単語の正規化はtsquery型に変換する前に行う必要があります。こうした正規化の実行にはto_tsquery関数が簡便です。

```
SELECT to_tsquery('Fat:ab & Cats');
    to_tsquery
------------------
 'fat':AB & 'cat'
```

to_tsqueryは他の言葉と同じように接頭辞を扱うことに注意してください。以下の比較の例ではtrueを返します。

```
SELECT to_tsvector( 'postgraduate' ) @@ to_tsquery( 'postgres:*' );
 ?column?
----------
 t
```

これはpostgresにはpostgrの語幹を含んでいるためです。

```
SELECT to_tsvector( 'postgraduate' ), to_tsquery( 'postgres:*' );
  to_tsvector  | to_tsquery
---------------+------------
 'postgradu':1 | 'postgr':*
```

これはpostgraduateの語幹の形と一致します。

8.12. UUID型

uuidデータ型は、RFC 4122、ISO/IEC 9834-8:2005および関連する標準に従う、汎用一意識別子（UUID）を格納します。（一部のシステムでは、このデータ型をグローバル一意識別子（GUID）と呼んでいます。）この識別子は、同一のアルゴリズムを使用しても既知の世界上の他の誰かが同一識別子が生成される可能性が

ほとんどないように選択されたアルゴリズムで生成された128ビット量の値です。したがって、分散システムにおいて、これら識別子は、単一データベース内でしか一意にならないシーケンスジェネレータよりも優れた一意性保証を提供します。

UUIDは、小文字の16進数表記桁の並びをいくつかのグループでハイフンで区切って表現されます。具体的には、8桁のグループが1つ、4桁のグループが3つ、次いで、12桁のグループが1つとなり、合計32桁で128ビットを表します。この標準形式のUUIDの例を以下に示します。

```
a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11
```

また、PostgreSQLは入力の別形式として、桁を大文字表記したもの、標準形式を中括弧でくくったもの、いくつかまたはすべてのハイフンを省略したもの、4桁ごとのグループの間の任意の箇所にハイフンを付加したものも受け付けます。以下に例を示します。

```
A0EEBC99-9C0B-4EF8-BB6D-6BB9BD380A11
{a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11}
a0eebc999c0b4ef8bb6d6bb9bd380a11
a0ee-bc99-9c0b-4ef8-bb6d-6bb9-bd38-0a11
{a0eebc99-9c0b4ef8-bb6d6bb9-bd380a11}
```

出力は常に標準形式になります。

PostgreSQLでUUIDを生成する方法は9.14を確認してください。

8.13. XML型

xmlデータ型を使用して、XMLデータを格納することができます。text型のフィールドにXMLデータを格納する方法より、入力された値が整形式かどうかを検査する利点があります。また、型を安全に操作するサポート関数があります。9.15を参照してください。このデータ型を使用するためには、インストレーションが`configure --with-libxml`で構築されていることが必要です。

xml型は、XML標準で定義された整形式の「文書」およびXQueryとXPathデータモデルのより寛容な「文書ノード」[1]を参照して定義される「コンテンツ」フラグメントを格納できます。大雑把に言うと、これは、コンテンツフラグメントが2つ以上の最上位要素や文字ノードを持つことができることを意味します。`xmlvalue IS DOCUMENT`という式を使用して、特定のxml値が完全な文書か単なるコンテンツフラグメントか評価することができます。

xmlデータ型の制限と互換性に関する注意事項は、D.3から確認できます。

8.13.1. XML値の作成

文字データからxml型の値を生成するためには、xmlparse関数を使用してください。

```
XMLPARSE ( { DOCUMENT | CONTENT } value)
```

[1] https://www.w3.org/TR/2010/REC-xpath-datamodel-20101214/#DocumentNode

例:

```
XMLPARSE (DOCUMENT '<?xml version="1.0"?><book><title>Manual</title><chapter>...</chapter></
book>')
XMLPARSE (CONTENT 'abc<foo>bar</foo><bar>foo</bar>')
```

標準SQLに従って文字列をXML値に変換するためにはこの方法しかありませんが、次のようなPostgreSQL固有の構文も使用することができます。

```
xml '<foo>bar</foo>'
'<foo>bar</foo>'::xml
```

xml型では文書型定義(DTD)に対して入力値を検証することは、入力値がDTDを指定していたとしても、行いません。また同様に、現時点ではXML Schemaなどの他のXMLスキーマ言語に対する検証サポートも組み込まれていません。

xmlから文字列値を生成するという逆演算ではxmlserialize関数を使用してください。

```
XMLSERIALIZE ( { DOCUMENT | CONTENT } value AS type )
```

ここで、typeは、character、character varying、text(またはこれらの別名)を取ることができます。この場合も、標準SQLに従ってxmlと文字列型間の変換を行うためにはこの方法しかありません。PostgreSQLでは単に値をキャストすることが可能です。

XMLPARSEやXMLSERIALIZEを使わずに文字列値とxmlとの間をキャストした場合、DOCUMENTかCONTENTかという選択が「XML option」セッション設定パラメータによって決定されます。このパラメータは標準コマンド

```
SET XML OPTION { DOCUMENT | CONTENT };
```

または、よりPostgreSQLらしい構文

```
SET xmloption TO { DOCUMENT | CONTENT };
```

を使用して設定することができます。デフォルトはCONTENTですので、すべての書式のXMLデータを扱うことができます。

8.13.2. 符号化方式の取扱い

クライアント側、サーバ側、および、これらを経由してやり取りされるXMLデータ内部で複数の文字符号化方式を扱う場合には注意が必要です。テキストモードを使用してサーバに問い合わせを渡し、そしてクライアントに問い合わせ結果を渡す場合(これが通常のモードです)、PostgreSQLは、クライアントからサーバ、サーバからクライアントでやり取りされるすべての文字データを受信側の文字符号化方式に変換します。23.3を参照してください。これには上の例のようなXML値の文字列表現も含まれます。これは通常、埋め込まれたencoding宣言は変更されずに、クライアント/サーバ間でやり取りされる間に文字データが他方の符号化方式に変換されてしまうので、XMLデータ内のencodingが無効になる可能性があることを意味します。この動作に対処するため、xml型の入力として表現された文字列に含まれているencoding宣言は無視され、その

内容は常にサーバの現在の符号化方式になっているものと仮定されます。したがって、正しく処理するためには、XMLデータにおける文字列をクライアントの現在の符号化方式で送信しなければなりません。サーバに送信する前に文書を現在のクライアントの符号化方式に変換するか、クライアントの符号化方式を適切に調節するかは、クライアントの責任です。出力ではxml型の値はencoding宣言を持ちません。クライアントはすべてのデータが現在のクライアントの符号化方式であることを前提としなければなりません。

バイナリモードを使用して、問い合わせパラメータをサーバに渡し、そして問い合わせ結果をクライアントに返す場合、符号化方式の変換は行われません。このため状況は異なります。この場合、XMLデータ内のencoding宣言が認識され、もし存在しなければ、データがUTF-8であると仮定されます。(XML標準の要求通りです。PostgreSQLはUTF-16をサポートしていないことに注意してください。) 出力では、データはクライアントの符号化方式を指定したencoding宣言を持ちます。ただし、もしクライアントの符号化方式がUTF-8の場合はencoding宣言は省略されます。

言うまでもありませんが、PostgreSQLを使用したXML処理では、XMLデータの符号化方式、クライアントの符号化方式、サーバの符号化方式が同じ場合にエラーが起こりづらく、より効率的です。XMLデータは内部的にUTF-8として処理されますので、サーバの符号化方式が同一のUTF-8である場合、最も効率が上がります。

> **注意**
>
> サーバ符号化方式がUTF-8でない場合、いくつかのXMLに関係した関数は非ASCIIデータに対して全く機能しないことがあります。これは特にxmltable()とxpath()に対する問題として知られています。

8.13.3. XML値へのアクセス

xmlデータ型は、比較演算子をまったく提供しないというところが他と異なります。これは、XMLデータに対し、よく定義され、誰にとっても有用な比較アルゴリズムが存在しないためです。この結果、xml列を検索値と比べて行を取り出すことはできません。したがって通常XML値には、IDなどの別のキーフィールドを一般的に付属させなければなりません。XML値の比較を行うもうひとつの方法は、文字列に一度変換することです。しかし、文字列比較は有用なXML比較方法といえないことに注意してください。

xmlデータ型用の比較演算子がありませんので、この型の列に直接インデックスを作成することはできません。XMLデータを高速に検索することが望まれるなら、その表現を文字列型にキャストし、それをインデックス付けするか、または、XPath式をインデックス付けするかという対策をとることができます。当然ながら、インデックス付けされた式で検索されるよう実際の問い合わせを調整する必要があります。

PostgreSQLのテキスト検索機能を使用して、XMLデータの全文検索速度をあげることもできます。しかし、PostgreSQL配布物では必要な前処理を未だサポートしていません。

8.14. JSONデータ型

JSONデータ型はJSON(JavaScript Object Notation)データを格納するためのものです。JSONの仕様はRFC 7159[2]に定義されています。このようなデータは、text型として格納することもできますが、JSONデー

[2] https://tools.ietf.org/html/rfc7159

タ型は、それぞれ格納された値がJSONルールに従って有効に施行されるという利点があります。これらのデータ型に格納されたデータのために利用可能な各種JSON固有の関数と演算子もあります。9.16を参照してください。

PostgreSQLには、JSONデータを格納するための2つの型、jsonとjsonbがあります。これらのデータ型に対して効率的な問い合わせメカニズムを実装するために、PostgreSQLは8.14.6で説明されているjsonpathデータ型も提供します。

json型とjsonb型というデータ型は、ほとんど同一の入力値セットを受け入れます。現実的に主要な違いは効率です。jsonデータ型は入力テキストの正確なコピーで格納し、処理関数を実行するたびに再解析する必要があります。jsonbデータ型では、分解されたバイナリ形式で格納されます。格納するときには変換のオーバーヘッドのため少し遅くなりますが、処理するときには、全く再解析が必要とされないので大幅に高速化されます。また jsonb型の重要な利点はインデックスをサポートしていることです。

json型は入力値のコピーを格納しているので、意味的に重要でないトークン間の空白だけでなく、JSONオブジェクト内のキーの順序も維持します。また、JSONオブジェクト内に同じキーと値が複数含まれていてもすべてのキー／値のペアが保持されます。(この処理関数は最後の値1つを処理させるようにすれば済みます。) これとは対照的に、jsonbは空白を保持しません。オブジェクトキーの順序を保持せず、重複したオブジェクトキーを保持しません。重複キーを入力で指定された場合は、最後の値が保持されます。

一般的に、ほとんどのアプリケーションではJSONデータ型としてjsonb型のほうが望ましいでしょう。ただし、オブジェクトキーを従来のような順序であることを仮定する非常に特殊なニーズが存在するような場合は除きます。

RFC 7159は、JSON文字列はUTF8でエンコードすべきと指定しています。従ってデータベースエンコーディングがUTF8でない限り、厳密にはJSON型がJSON仕様に準拠することはできません。データベースのエンコーディングで表現できない文字を直接含めようとすると失敗します。逆に、UTF8で許可されずにデータベースのエンコーディングで許可される文字が許されてしまいます。

RFC 7159 では、JSON文字列はUnicodeエスケープシーケンス \uXXXX を許可するように記述されています。json型の入力関数は、データベースエンコーディング方式に関係なくUnicodeエスケープが許可されています。それは、構文上の正しさ(つまり\uに続けて16進数が4桁)だけをチェックしています。しかし、jsonbの入力関数はより厳しくなります。データベースエンコーディング方式で表現できない文字のUnicodeエスケープを禁止します。jsonb型は\u0000も許可しません。(なぜならPostgreSQLのtext型で表現できないためです。) また、Unicode基本多言語面以外の文字はUnicodeのサロゲートペアに直すことが要求されています。有効なUnicodeエスケープは、同等の単一の文字に変換されて格納されます。これはサロゲートペアを単一の文字に変換する処理も含まれています。

> **注記**
>
> 9.16で説明されているJSONの処理関数の多くは、Unicodeエスケープを通常の文字に変換します。そして、それらの入力はjsonbでないjsonの場合でも記載された同じ種類のエラーになります。json入力関数がこれらのチェックをしないことは歴史的経緯によるものと言えるかもしれませんが、そのために、表現された文字をサポートしないデータベースエンコーディングで、JSON Unicodeエスケープされた文字を単に格納(処理を必要としない場合)できてしまいます。

原文のJSONがjsonb型に変換されるときには、RFC 7159に記載されているプリミティブ型は表 8.23に記されているようにPostgreSQLのネイティブな型に変換されます。そのため、jsonbデータ型には、json型に

なく、また理論上JSONにはないマイナーな制約があります。それは基礎となるデータ型に付随する制限によって表されます。特にjsonb型は、PostgreSQLのnumeric型の範囲外の数を拒否します。このような処理系で定義される制限はRFC 7159で許可されています。しかし、それは IEEE 754 倍精度浮動小数点がJSONのnumberプリミティブ型を表すのが一般的であるように、実際には他の実装でこのような問題が発生することの方がはるかに可能性が高いです(RFC 7159が明示的に予測して、許可しています)。このようなシステムとPostgreSQLで交換フォーマットとしてJSONを使用する場合は、数値精度を失う危険性があることを把握しておく必要があります。

逆に、表に示すようにJSONプリミティブ型の入力フォーマットには、対応するPostgreSQL型と適合しない、いくつかのマイナーな制限があります。

表8.23 JSONプリミティブ型とPostgreSQL型の対応表

JSON プリミティブ型	PostgreSQL型	注釈
string	text	\u0000は許可されません。またそのデータベースエンコーディング方式で利用できない文字を表現するユニコードエスケープも許可されません。
number	numeric	NaN と infinity 値は許可されません
boolean	boolean	小文字のtrue と false という綴りのみ許可されます
null	(none)	SQLのNULLとは概念が異なります

8.14.1. JSONの入出力構文

JSON型の入出力構文の仕様はRFC 7159 に規定されています。

以下は、すべて有効なjson型(または jsonb型)の式です。

```
-- シンプルなスカラ/プリミティブ値
-- プリミティブ値は、
      数値、
      引用符で括られた文字列、
      true、
       false、
      またはnullです。
SELECT '5'::json;


-- 0個以上の要素の配列（要素は同じ型である必要はありません)。
SELECT '[1, 2, "foo", null]'::json;


-- キーと値のペアを含むオブジェクト
-- オブジェクトキーは常に引用符で括られた文字列でなければならないことに注意してください。
SELECT '{"bar": "baz", "balance": 7.77, "active": false}'::json;
```

```
-- 配列とオブジェクトは任意に入れ子にすることができます。
SELECT '{"foo": [true, "bar"], "tags": {"a": 1, "b": null}}'::json;
```

先に述べたようにJSONの値が入力されたときに、その後、追加の処理を行わずに表示する場合、jsonは入力と同じテキストが出力されます、jsonbでは、空白のような意味を持たない情報を保持しません。例を示します。ここでは相違点に注意してください。

```
SELECT '{"bar": "baz", "balance": 7.77, "active":false}'::json;
                      json
-------------------------------------------------
 {"bar": "baz", "balance": 7.77, "active":false}
(1 row)

SELECT '{"bar": "baz", "balance": 7.77, "active":false}'::jsonb;
                      jsonb
--------------------------------------------------
 {"bar": "baz", "active": false, "balance": 7.77}
(1 row)
```

もう一つ注目に値するのは、jsonbでは、数値はnumeric型の動作に応じて表示され、意味を持たない情報を保持しません。実際には数字はE表記なしで表示されることを意味します。例を示します。

```
SELECT '{"reading": 1.230e-5}'::json, '{"reading": 1.230e-5}'::jsonb;
         json          |          jsonb
-----------------------+-------------------------
 {"reading": 1.230e-5} | {"reading": 0.00001230}
(1 row)
```

しかし、この例に見られるようにjsonbは小数の末尾のゼロを保持します。それにも関わらず、等しいかチェックする場合等では、意味的に重要ではありません。

JSONの値の作成と処理に使用可能な組み込み関数と演算子のリストについては、9.16を参照してください。

8.14.2. JSONドキュメントの設計

JSONデータは従来のリレーショナルデータモデルよりもかなり柔軟に表現することができます。そのため、要件が変わりやすい環境では説得力があります。そして、それは同じアプリケーション内で、両方のアプローチが共存し相互に補完することが可能です。しかし、最大の柔軟性が要求されるアプリケーションのためでもJSONドキュメントには、まだいくらかの固定構造を持つことを推奨します。構造は（いくつかのビジネスルールを強制することは宣言的に可能であるが）、一般的に強制されないですが、テーブル内の「ドキュメント」（データ）セットをまとめて予測可能な構造にすることで、簡単に問い合わせを記述することができます。

JSONデータはテーブルに格納するとき、他のデータ型と同一の同時実行制御の対象となります。大きな文章を保存することは実行可能ですが、すべての更新が行レベルロックを取得することに留意してください。更新トランザクション間のロックの競合を減少させるために、管理可能なサイズにJSONドキュメントを制限す

ることを検討してください。理想的には、JSONドキュメントはビジネス・ルール上、独立して変更することができない単位までデータを分割すべきです。

8.14.3. jsonb型用包含演算子と存在演算子

包含演算子のテストはjsonb型の重要な機能です。これらのセットはjson型には全くありません。jsonbドキュメントが、その中に指定する値を含むかどうかをテストします。これらの例は、特に記載がないかぎりtrueを返します。

```
-- 単純なスカラ/プリミティブ値は、
      同一の値が含まれています。
SELECT '"foo"'::jsonb @> '"foo"'::jsonb;


-- 左辺の配列に右辺の配列が含まれています。
SELECT '[1, 2, 3]'::jsonb @> '[1, 3]'::jsonb;


-- 配列要素の順序は重要ではありませんので、
      これもまた真になります。
SELECT '[1, 2, 3]'::jsonb @> '[3, 1]'::jsonb;


-- 配列要素に重複が含まれているかは問題ではありません。
SELECT '[1, 2, 3]'::jsonb @> '[1, 2, 2]'::jsonb;


-- 右辺の単一ペアを持つオブジェクトが左辺のオブジェクト内に含まれています。
SELECT '{"product": "PostgreSQL", "version": 9.4, "jsonb": true}'::jsonb @> '{"version":
 9.4}'::jsonb;


-- 右辺の配列は左辺の配列に含まれま せん、

-- 類似の配列が、
      その中のネストに含まれているにも関わらず。

SELECT '[1, 2, [1, 3]]'::jsonb @> '[1, 3]'::jsonb;  -- falseになる


-- しかし、
      ネストで層を合わせれば含まれるようになります。
SELECT '[1, 2, [1, 3]]'::jsonb @> '[[1, 3]]'::jsonb;
```

```
-- 同様に、
      これも含まれません。

SELECT '{"foo": {"bar": "baz"}}'::jsonb @> '{"bar": "baz"}'::jsonb;  -- falseになる


-- トップレベルのキーと空のオブジェクトが含まれる。
SELECT '{"foo": {"bar": "baz"}}'::jsonb @> '{"foo": {}}'::jsonb;
```

一般原則では、オブジェクトにオブジェクトが含まれているかを判断するには、いくつかの条件に一致しない配列要素とキー／値のペアを含むオブジェクトを捨てた後に構造とデータを一致させる必要があります。しかし、条件に一致するには配列要素の順序は重要ではなく、重複要素は一回のみ有効に評価されることを覚えておく必要があります。

構造が一致しなければならないという一般原則の特別な例外として、配列はプリミティブな値を含めることができます。

```

-- この配列はプリミティブな文字列を含みます。
SELECT '["foo", "bar"]'::jsonb @> '"bar"'::jsonb;


-- この例外は相互的ではありません。 -- これは含まれません。

SELECT '"bar"'::jsonb @> '["bar"]'::jsonb;  -- falseになる
```

jsonb型は、また存在演算子を持ちます。包含の変種です。それは文字列(与えられたtext値)が、jsonb値のオブジェクトキーまたは配列のトップレベルに存在するかどうかをテストします。これらの例は、特に記載がないかぎりtrueを返します。

```

-- 文字列が配列要素に存在する。
SELECT '["foo", "bar", "baz"]'::jsonb ? 'bar';


-- 文字列がオブジェクトキーに存在する。
SELECT '{"foo": "bar"}'::jsonb ? 'foo';


-- オブジェクト値は考慮されません。

SELECT '{"foo": "bar"}'::jsonb ? 'bar';  -- falseになる


-- オブジェクトはトップレベルから一致するように存在する必要があります。

SELECT '{"foo": {"bar": "baz"}}'::jsonb ? 'bar'; -- falseになる
```

```
-- 文字列はJSONプリミティブ文字列と一致させることができます。
SELECT '"foo"'::jsonb ? 'foo';
```

JSONオブジェクトは、関係するキーや要素が多く存在する場合、含むかどうかまたは存在するかどうかのテストに適しています。なぜなら配列とは異なり、リニア検索をする必要がなく、内部的に検索に最適化されています。

ヒント

JSONでは包含がネストされるので、適切な問い合わせではサブオブジェクトの明示的な選択を省略することが出来ます。例を挙げます。doc列にトップレベルのオブジェクトがあります。このオブジェクトには、tagsフィールドが含まれ、このフィールドにサブオブジェクトの配列が多く含まれているとします。以下の問い合わせは、サブオブジェクトが"term":"paris"と"term":"food"の両方を含むエントリを探します。そのときtags配列の外側にある、それらのキーは無視されます。

```
SELECT doc->'site_name' FROM websites
  WHERE doc @> '{"tags":[{"term":"paris"}, {"term":"food"}]}';
```

同じことを達成することは出来ます。例えば、

```
SELECT doc->'site_name' FROM websites
  WHERE doc->'tags' @> '[{"term":"paris"}, {"term":"food"}]';
```

しかし、そのアプローチは柔軟性に欠け、効率も落ちます。

一方、JSONの存在演算子は、ネストしていません。JSONの値の最上位に指定されたキーまたは配列要素のみを探します。

JSONの様々な包含演算子や存在演算子、他のすべてのJSON演算子と関数は 9.16に記載されています。

8.14.4. jsonb インデックス

GINインデックスは、多数のjsonbドキュメント(データ)のキーやキー／値ペアを効率的に検索するときに用いることができます。異なるパフォーマンスと柔軟性のトレードオフを持つ、2つのGIN「演算子クラス」が提供されています。

jsonb型の問い合わせでサポートしているデフォルトのGIN演算子クラスは、トップレベルのキーが存在するかの演算子として?、?&、?|があり、パス／値が存在するかの演算子として@>があります。(これらの演算子の意味の詳細は、表 9.45を参照してください。) この演算子クラスのインデックスを作成する例。

```
CREATE INDEX idxgin ON api USING GIN (jdoc);
```

デフォルトでないGIN演算子クラスは、@>演算子のみサポートするjsonb_path_opsがあります。この演算子クラスのインデックスを作成する例。

```
CREATE INDEX idxginp ON api USING GIN (jdoc jsonb_path_ops);
```

サードパーティのWebサービスから、ドキュメント化されたスキーマ定義を持つJSONドキュメントを取得し、格納するテーブルの例を考えてみましょう。典型的なドキュメントは、次のとおりです。

```
{
    "guid": "9c36adc1-7fb5-4d5b-83b4-90356a46061a",
    "name": "Angela Barton",
    "is_active": true,
    "company": "Magnafone",
    "address": "178 Howard Place, Gulf, Washington, 702",
    "registered": "2009-11-07T08:53:22 +08:00",
    "latitude": 19.793713,
    "longitude": 86.513373,
    "tags": [
        "enim",
        "aliquip",
        "qui"
    ]
}
```

テーブル名 apiにjsonb型でjdocをカラム名として格納します。このカラムにGINインデックスを作成した場合、以下のような問い合わせがインデックスを利用することができます。

```

-- "company"キー が "Magnafone"値であるものを見つける
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc @> '{"company": "Magnafone"}';
```

しかし 次のような問い合わせはインデックスを使用しません。なぜなら、?演算子はインデックス可能ですが、jdocカラムのインデックスが直接適用されていないためです。

```

-- キー "tags" の配列要素に "qui"が含まれているか見つける
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc -> 'tags' ? 'qui';
```

それでも、上記の問い合わせは、式インデックスを適切に使用することでインデックスを使用することができます。一般的な "tags"キーから特定の項目を照会する場合、このようなインデックスを定義すると良いかもしれません。

```
CREATE INDEX idxgintags ON api USING GIN ((jdoc -> 'tags'));
```

さて、WHERE句の jdoc -> 'tags' ? 'qui'は、インデックス式では 、?演算子はインデックス可能として認識されます。(式インデックスに関する詳細情報は11.7を参照してください。)

また、GINインデックスはjsonpathのマッチングを実行する@@演算子と@?演算子をサポートします。

```
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc @@ '$.tags[*] == "qui"';
```

```
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc @@ '$.tags[*] ? (@ == "qui")';
```

GINインデックスは、jsonpath: accessors_chain = constの形式の文を抽出します。アクセサチェーン(Accessors chain)は[*]と[index]アクセサで構成されます。jsonb_opsは、さらに.*と.**アクセサもサポートします。

別のアプローチとして包含を利用する問い合わせがあります。例を示します。

```
-- キー "tags"に 要素"qui"が含まれるかどうか見つける
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc @> '{"tags": ["qui"]}';
```

jdocカラムのシンプルなGINインデックスは、この問い合わせをサポートすることができます。しかし、前の例では、tagsキーの下にあるデータのみをインデックスに格納していたのに対して、そのようなインデックスは、jdocのすべてのキーと値のコピーを保存しますので、注意が必要です。シンプルなインデックスアプローチは(それが全てのキーについての問い合わせをサポートしているため)はるかに柔軟ですが、ターゲット式インデックスは単純なインデックスより小さく、検索のときに高速である可能性が高くなります。

jsonb_path_ops演算子クラスは、@>、@@、@?演算子をサポートしているだけですが、デフォルト演算子クラスのjsonb_opsよりも顕著なパフォーマンス上の利点があります。jsonb_path_opsインデックスは、通常同じデータのjsonb_opsインデックスよりもはるかに小さく、データの中で頻繁に現れるキーを含む場合のような特別な検索には、より良くなります。そのため、デフォルトの演算子クラスよりも検索性能が良くなります。

jsonb_opsとjsonb_path_opsのGINインデックスの技術的差異は、前者はデータのキーと値のための独立したインデックスを作成しますが、後者は、データの値に対してのみインデックスを作成します。[3] 基本的に、jsonb_path_opsインデックス項目は、値とキーのハッシュです。例えば、{"foo": {"bar": "baz"}}のインデックスはハッシュ値にfoo、bar、bazすべてを組み込んで作成されます。したがって、包含問い合わせのためのインデックス検索は、非常に特定の構造を返すようになっています。しかしfooがキーとして表示されるかどうかを調べるには全く方法はありません。一方、jsonb_opsインデックスは個別にはfoo、bar、bazを表す3つのインデックス項目を作成します。その後、包含問い合わせをおこなうには、これらの項目の3つすべてを含む行を探します。GINインデックスは、かなり効率的に検索することができますが、特に3つの索引項目のいずれかで、非常に多数の行が単一の場合に、同等のjsonb_path_ops検索よりも遅くなります。

jsonb_path_opsアプローチの欠点は、{"a": {}}のような、任意の値を含まないJSON構造のためのインデックスエントリを生成しません。このような構造を含むドキュメントの検索が要求された場合、それは、フルインデックススキャンを必要とします。それは非常に遅くなります。そのため、jsonb_path_opsは、多くの場合、そのような検索を実行するのには不適当です。

jsonb型は、btree と hash インデックスもサポートします。これらは通常、JSONドキュメントの完全性をチェックすることが重要な場合のみ有用です。jsonbのためのbtree順序には、興味深いことはほとんどありませんが、しかし、完全さのために次に示します。

```
Object > Array > Boolean > Number > String > Null

Object with n pairs > object with n - 1 pairs
```

[3] この目的のために、「値」という用語は配列の要素を含みますが、JSONの専門用語では、オブジェクト内の値と配列の要素が時々違うことがあります。

```
Array with n elements > array with n - 1 elements
```

ペアの同じ番号を持つオブジェクトは、順に比較されます。

```
key-1, value-1, key-2 ...
```

そのオブジェクトのキーは、その格納順に比較されることに注意してください。短いキーは長いキーの前に格納されているため、特にこれは、次のような直感的でない結果に結果につながるかもしれません。

```
{ "aa": 1, "c": 1} > {"b": 1, "d": 1}
```

同様に、配列と同じ番号を持つ要素を比較する順番。

```
element-1, element-2 ...
```

JSONプリミティブ値は基本的にPostgreSQLデータ型と同じルールで比較されます。文字列は、デフォルトのデータベース照合を使用して比較されます

8.14.5. 変換

異なるプロシージャ言語でjsonb型の変換を実装した追加の拡張が入手可能です。

PL/Perl向けの拡張は、jsonb_plperlとjsonb_plperluと呼ばれます。この拡張を使うとjsonbの値は適したPerlの配列、ハッシュ、スカラにマップされます。

PL/Python向けの拡張は、jsonb_plpythonu、jsonb_plpython2u、jsonb_plpython3uと呼ばれます(PL/Pythonの命名規約については45.1を参照してください)。この拡張を使うとjsonbの値は適したPythonの辞書型、リストやスカラにマップされます。

上記の拡張のうち、jsonb_plperlは「trusted」と見なされます。つまり、現在のデータベースに対してCREATE権限を持つ非スーパーユーザがインストールできます。残りはインストールするのにスーパーユーザ権限が必要です。

8.14.6. jsonpath型

jsonpath型は、PostgreSQLでJSONデータの効率的な問い合わせをするために、SQL/JSONパス言語のサポートを実装しています。構文解析されたSQL/JSONパス式のバイナリ表現を提供し、SQL/JSON問い合わせ関数でさらに処理するために、パスエンジンがJSONデータから取得する項目を指定します。

SQL/JSONパス述部および演算子のセマンティクスは、SQLに準拠しています。同時に、JSONデータを処理する自然な方法を提供するために、SQL/JSONのパス構文ではいくつかのJavaScript規則を使用します。

- ドット(.)は、メンバアクセスに使用されます。
- 大括弧([])は配列アクセスに使用されます。
- 1から始まる通常のSQL配列とは異なり、SQL/JSON配列は0スタートです。

SQL/JSONパス式は通常、SQL問い合わせでSQL文字列リテラルとして記述されるため、一重引用符で囲む必要があり、値内で必要な一重引用符は二重にする必要があります (4.1.2.1を参照)。一部の形式のパス式では、文字列リテラルを含める必要があります。これらの埋め込み文字列リテラルは二重引用符で囲む必要があり、バックスラッシュエスケープを使用してハード・タイプ文字を表すことができます。特に、埋め込み文字列リテラル内で二重引用符を記述する方法は\"であり、バックスラッシュを記述する必要がある場合は\\と書く必要があります。その他の特別なバックスラッシュ構文には、以下のJSON文字列で認識されるものが含まれます。さまざまなASCII制御用文字の\b、\f、\n、\r、\t、\v、および4つの16進数のコードポイントで識別されるUnicode文字用の\uNNNNです。バックスラッシュ構文には、JSONでは許されない2つのケースも含まれています。\xNNは2桁の16進数だけで記述された文字コードの場合で、\u{N...}は、1〜6桁の16進数で記述された文字コードの場合です。

パスの式は、次のようなパス要素のシーケンスで構成されます。

- JSONプリミティブ型のパスリテラル。ユニコードテキスト、数値、true、false、又はnullです。
- パス変数表 8.24。
- アクセサ演算子表 8.25。
- jsonpath演算子とメソッド9.16.2.2。
- 括弧。フィルタ式を提供したり、パス評価の順序を定義するために使用できます。

jsonpath式を使用したSQL/JSON問い合わせ関数の詳細は、9.16.2を参照してください。

表8.24 jsonpath変数

変数	説明
$	問い合わせ対象(*context item*)のJSON値を表す変数。
$varname	名前付き変数。その値はいくつかのJSON処理関数のパラメーターvarsで設定できます。詳細は表 9.47を参照してください。
@	フィルター式のパス評価の結果を表す変数。

表8.25 jsonpath Accessors

アクセサ演算子	説明
.key ."$varname"	指定されたキーを持つオブジェクトメンバを返すメンバアクセサ。キー名が$で始まる名前付き変数に一致する場合、または識別子のJavaScriptルールを満たさない場合は、文字列リテラルとするため二重引用符で囲む必要があります。
.*	現在のオブジェクトの最上位レベルになるすべてのメンバの値を返すワイルドカードメンバアクセサ。
.**	現在のオブジェクトのJSON階層のすべてのレベルを処理し、ネストされたレベルに関わらず全てのメンバ値を返す再帰的なワイルドカードメンバアクセサ。これはSQL/JSON標準のPostgreSQLの拡張です。
.**{level} .**{start_level to end_level}	.**と似ていますが、JSON階層の指定したレベルだけを選びます。ネストレベルは整数で指定します。レベル0は現在のオブジェクトに対応します。最下位のネストレベルにアクセスするのに、lastキーワードが使用できます。これはSQL/JSON標準のPostgreSQLの拡張です。

アクセサ演算子	説明
[subscript, ...]	配列要素アクセサ。subscriptは、indexまたはstart_indexからend_indexまでの2つの形式で指定できます。最初の形式は、インデックスによって単一の配列要素を返します。2番目の形式は、指定されたstart_indexとend_indexに対応する要素を含む、インデックスの範囲による配列スライスを返します。 指定されたindexには、整数だけでなく、自動的に整数にキャストされる単一の数値を返す式を指定できます。インデックス0は最初の配列要素に対応します。また、lastキーワードを使用して最後の配列要素を指定することもできます。これは、長さが不明な配列の処理に役立ちます。
[*]	全ての配列の要素を返すワイルドカード配列要素アクセサ。

8.15. 配列

PostgreSQLではテーブルの列を可変長多次元配列として定義できます。あらゆる組み込み型あるいはユーザ定義の基本型、列挙型、複合型、範囲型そしてドメインの配列も作成可能です。

8.15.1. 配列型の宣言

実際に配列の使い方を説明するために、次のテーブルを作成します。

```
CREATE TABLE sal_emp (
    name            text,
    pay_by_quarter  integer[],
    schedule        text[][]
);
```

見ておわかりのように配列データ型は配列要素のデータ型の名前に大括弧（[]）を付けて指定します。このコマンドはtext型文字列（name）、従業員の四半期の給与を保存するinteger型の一次元配列（pay_by_quarter）、そして従業員の週間スケジュールを保存するtext型の二次元配列（schedule）の列を持つsal_empという名前のテーブルを作成します。

CREATE TABLE構文で指定する配列の正確な大きさを指定することができます。

```
CREATE TABLE tictactoe (
    squares   integer[3][3]
);
```

とは言っても現在の実装では指定された配列の大きさの制限を無視します。つまり、長さの指定がない配列と同じ振舞いをします。

現在の実装では次元数の宣言も強制していません。特定の要素型の配列はすべて大きさあるいは次元数とは無関係に同じ型とみなされます。ですからCREATE TABLEで配列の大きさや次元数を宣言することは、単なる説明です。実行時の動作に影響を及ぼしません。

SQLに準拠し、ARRAYキーワードを使用したもう1つの構文を一次元配列に使うことができます。pay_by_quarterを次のように定義することもできます。

```
pay_by_quarter  integer ARRAY[4],
```

または、もし配列の大きさが指定されない場合は次のようになります。

```
pay_by_quarter  integer ARRAY,
```

しかし、前で触れたようにPostgreSQLはどんな場合でも大きさの制限を強要しません。

8.15.2. 配列の値の入力

リテラル定数として配列の値を書き込むには、その要素の値を中括弧で囲み、それぞれの要素の値をカンマで区切ります（C言語を知っているならば、構造体を初期化するための構文のようなものと考えてください）。要素の値を二重引用符でくくることもでき、カンマもしくは中括弧がある時は必ずそのように書かなければなりません（詳細は以下に出てきます）。したがって配列定数の一般的書式は次のようになります。

```
'{ val1 delim val2 delim ... }'
```

ここでdelimはそのpg_type項目に記録されている型の区切り文字です。PostgreSQL配布物で提供されている標準データ型の内、セミコロン(;)を使用するbox型を除き、すべてはカンマ(,)を使います。それぞれのvalは配列要素の型の定数か副配列です。配列定数の例を以下に示します。

```
'{{1,2,3},{4,5,6},{7,8,9}}'
```

この定数は整数の3つの副配列を持っている二次元3×3の配列です。

配列定数の要素をNULLとするためには、その要素値にNULLと記載してください。(NULLを大文字で書いても小文字で書いても構いません。)「NULL」という文字列値を指定したければ、二重引用符でくくって記載しなければなりません。

(この種の配列定数は実際4.1.2.7で説明されている一般型定数の特別の場合に過ぎません。この定数は元々文字列として扱われていて配列入力ルーチンに渡されます。明示的な型指定が必要かもしれません。)

では、INSERT文をいくつか紹介します。

```
INSERT INTO sal_emp
    VALUES ('Bill',
    '{10000, 10000, 10000, 10000}',
    '{{"meeting", "lunch"}, {"training", "presentation"}}');

INSERT INTO sal_emp
    VALUES ('Carol',
    '{20000, 25000, 25000, 25000}',
    '{{"breakfast", "consulting"}, {"meeting", "lunch"}}');
```

上に記載した2つの挿入文の結果は次のようになります。

```
SELECT * FROM sal_emp;
 name  |      pay_by_quarter       |                 schedule
-------+---------------------------+-----------------------------------------
 Bill  | {10000,10000,10000,10000} | {{meeting,lunch},{training,presentation}}
 Carol | {20000,25000,25000,25000} | {{breakfast,consulting},{meeting,lunch}}
(2 rows)
```

多次元配列では、各次元の範囲を合わせなければなりません。一致しないと以下のようにエラーが発生します。

```
INSERT INTO sal_emp
    VALUES ('Bill',
    '{10000, 10000, 10000, 10000}',
    '{{"meeting", "lunch"}, {"meeting"}}');
ERROR:  multidimensional arrays must have array expressions with matching dimensions
```

ARRAY生成子構文も使えます。

```
INSERT INTO sal_emp
    VALUES ('Bill',
    ARRAY[10000, 10000, 10000, 10000],
    ARRAY[['meeting', 'lunch'], ['training', 'presentation']]);

INSERT INTO sal_emp
    VALUES ('Carol',
    ARRAY[20000, 25000, 25000, 25000],
    ARRAY[['breakfast', 'consulting'], ['meeting', 'lunch']]);
```

配列要素は通常のSQL定数もしくは演算式であることに注意してください。例えば文字列リテラルは配列リテラルと同様、二重引用符ではなく単一引用符でくくられます。ARRAY生成子構文は4.2.12により詳しい説明があります。

8.15.3. 配列へのアクセス

ではテーブルに対していくつかの問い合わせを行ってみましょう。初めに、配列の単一要素にアクセスする方法を示します。この問い合わせは第2四半期に給与が更新された従業員の名前を抽出します。

```
SELECT name FROM sal_emp WHERE pay_by_quarter[1] <> pay_by_quarter[2];

 name
-------
 Carol
(1 row)
```

配列の添字番号は大括弧で囲んで記述されます。デフォルトでPostgreSQLは配列に対し「1始まり」の振り番規定を採用しています。つまり要素がn個ある配列はarray[1]で始まり、array[n]で終わります。

次の問い合わせは全ての従業員の第3四半期の給与を抽出します。

```
SELECT pay_by_quarter[3] FROM sal_emp;

 pay_by_quarter
----------------
          10000
          25000
(2 rows)
```

また、配列や副配列の任意の縦方向の部分を切り出すこともできます。一次元以上の配列についてその一部を表現するには、lower-bound:upper-boundと記述します。例えばこの問い合わせはBillのその週の初めの2日に最初何が予定されているかを抽出します。

```
SELECT schedule[1:2][1:1] FROM sal_emp WHERE name = 'Bill';

        schedule
------------------------
 {{meeting},{training}}
(1 row)
```

任意の次元を部分として、つまりコロンを含めて記述すると、すべての次元が部分として扱われます。単一の番号のみ(コロンを持たない)を持つ次元はすべて、1から指定番号までと扱われます。例えば、[2]は以下の例のように [1:2]と扱われます。

```
SELECT schedule[1:2][2] FROM sal_emp WHERE name = 'Bill';

                schedule
-----------------------------------------
 {{meeting,lunch},{training,presentation}}
(1 row)
```

切り出しのない場合と混乱を避けるため、すべての次元に対し切り出し構文を使用することが最善です。例えば、[2][1:1]ではなく、[1:2][1:1]のようにします。

切り出し指定子のlower-bound、upper-boundは省略可能です。省略された上限または下限は、配列の添字の上限または下限で置き換えられます。例えば、

```
SELECT schedule[:2][2:] FROM sal_emp WHERE name = 'Bill';

        schedule
------------------------
 {{lunch},{presentation}}
(1 row)

SELECT schedule[:][1:1] FROM sal_emp WHERE name = 'Bill';
```

```
        schedule
-----------------------
 {{meeting},{training}}
(1 row)
```

配列自体がNULLもしくはその添字式がNULLとなる場合、配列添字式はNULLを返します。また、配列の範囲を超える添字の場合もNULLが返されます（この場合はエラーになりません）。例えば、scheduleが現在[1:3][1:2]次元であれば、schedule[3][3]の参照はNULLとなります。同様にして、添字として間違った値を指定して配列を参照した場合もエラーではなく、NULLが返されます。

同様に、部分配列式も配列自体がNULLもしくはその添字式がNULLとなる場合にNULLを返します。しかし、現在の配列範囲を完全に超えた部分配列を選択する場合では、部分配列式はNULLではなく空の（0次元）の配列を返します。（これは切り出しなしの動作に一致せず、歴史的理由で行われるものです。）要求された部分配列が配列の範囲に重なる場合、NULLを返さずに、警告なく重複部分だけに減少させます。

array_dims関数で任意の配列値の現在の次元を取り出せます。

```
SELECT array_dims(schedule) FROM sal_emp WHERE name = 'Carol';

 array_dims
------------
 [1:2][1:2]
(1 row)
```

array_dims関数はtext型で結果を返します。人間が結果を見るためには便利ですが、プログラムにとって都合がよくありません。次元はarray_upperとarray_lowerでも抽出することができ、それぞれ特定の配列の次元の上限と下限を返します。

```
SELECT array_upper(schedule, 1) FROM sal_emp WHERE name = 'Carol';

 array_upper
-------------
           2
(1 row)
```

array_lengthは指定された配列次元の長さを返します。

```
SELECT array_length(schedule, 1) FROM sal_emp WHERE name = 'Carol';

 array_length
--------------
            2
(1 row)
```

cardinalityは配列の全次元に渡る要素の総数を返します。実質的にunnestの呼び出しで生成される行の数です。

```
SELECT cardinality(schedule) FROM sal_emp WHERE name = 'Carol';
```

```
 cardinality
-------------
           4
(1 row)
```

8.15.4. 配列の変更

配列の値を全て置き換えることができます。

```
UPDATE sal_emp SET pay_by_quarter = '{25000,25000,27000,27000}'
    WHERE name = 'Carol';
```

もしくはARRAY演算構文を用いて次のように書きます。

```
UPDATE sal_emp SET pay_by_quarter = ARRAY[25000,25000,27000,27000]
    WHERE name = 'Carol';
```

配列の1つの要素を更新することも可能です。

```
UPDATE sal_emp SET pay_by_quarter[4] = 15000
    WHERE name = 'Bill';
```

あるいは一部分の更新も可能です。

```
UPDATE sal_emp SET pay_by_quarter[1:2] = '{27000,27000}'
    WHERE name = 'Carol';
```

lower-boundやupper-boundが省略された切り出し構文も使用可能ですが、NULLや0次元でない配列の値を更新する場合に限ります(さもなければ、置き換えるべき添字の上限、下限が存在しません)。

保存されている配列の値は、存在しない要素に代入することで拡張することができます。過去に存在した位置と新しく代入された位置との間はNULLで埋められます。例えば、現在配列myarrayの要素数が4の場合、myarray[6]を割り当てる更新の後6要素を持つことなり、myarray[5]はNULLを含みます。現在、こうした方法での拡張は、1次元配列でのみ許されます。多次元配列では行うことができません。

添字指定の代入で1始まり以外の添字がある配列を作れます。例えば添字が-2から7までの値を持つ配列をarray[-2:7]で指定できます。

新規の配列の値は連結演算子||を用いて作成することもできます。

```
SELECT ARRAY[1,2] || ARRAY[3,4];
 ?column?
-----------
 {1,2,3,4}
(1 row)
```

```
SELECT ARRAY[5,6] || ARRAY[[1,2],[3,4]];
      ?column?
---------------------
 {{5,6},{1,2},{3,4}}
(1 row)
```

連結演算子を使うと、一次元配列の最初もしくは最後に1つの要素を押し込むことができます。さらには2つのN-次元配列もしくはN-次元配列とN+1-次元配列にも対応しています。

1つの要素が1次元配列の先頭や末尾に押し込まれた時、結果は配列演算項目と同じ下限添字を持つ配列となります。以下に例を示します。

```
SELECT array_dims(1 || '[0:1]={2,3}'::int[]);
 array_dims
------------
 [0:2]
(1 row)

SELECT array_dims(ARRAY[1,2] || 3);
 array_dims
------------
 [1:3]
(1 row)
```

等しい次元を持った2つの配列が連結された場合、結果は左側演算項目の外側の次元の下限添字を引き継ぎます。結果は右側被演算子のすべての要素に左側被演算子が続いた配列となります。例を挙げます。

```
SELECT array_dims(ARRAY[1,2] || ARRAY[3,4,5]);
 array_dims
------------
 [1:5]
(1 row)

SELECT array_dims(ARRAY[[1,2],[3,4]] || ARRAY[[5,6],[7,8],[9,0]]);
 array_dims
------------
 [1:5][1:2]
(1 row)
```

N-次元配列がN+1-次元配列の最初または最後に押し込まれると、結果は上記と似通った要素配列になります。それぞれのN-次元副配列は本質的にN+1-次元配列の外側の次元の要素となります。例を挙げます。

```
SELECT array_dims(ARRAY[1,2] || ARRAY[[3,4],[5,6]]);
 array_dims
------------
 [1:3][1:2]
```

```
(1 row)
```

配列はarray_prepend、array_append、もしくはarray_catを使って構築することもできます。初めの2つは一次元配列にしか対応していませんが、array_catは多次元配列でも使えます。例を挙げます。

```
SELECT array_prepend(1, ARRAY[2,3]);
 array_prepend
---------------
 {1,2,3}
(1 row)

SELECT array_append(ARRAY[1,2], 3);
 array_append
--------------
 {1,2,3}
(1 row)

SELECT array_cat(ARRAY[1,2], ARRAY[3,4]);
 array_cat
-----------
 {1,2,3,4}
(1 row)

SELECT array_cat(ARRAY[[1,2],[3,4]], ARRAY[5,6]);
      array_cat
---------------------
 {{1,2},{3,4},{5,6}}
(1 row)

SELECT array_cat(ARRAY[5,6], ARRAY[[1,2],[3,4]]);
      array_cat
---------------------
 {{5,6},{1,2},{3,4}}
```

単純な状況では、上で説明した連結演算子はそれぞれの関数を直接実行することよりも望ましいです。とは言っても、連結演算子は3つの場合すべてに対応するようオーバーロードされていますので、その関数の1つを使うとあいまいさを避けるのに役立つ場合があります。例えば、以下のような状況を考えてください。

```
SELECT ARRAY[1, 2] || '{3, 4}';  -- 型指定のないリテラルは配列と見なされる
 ?column?
-----------
 {1,2,3,4}


SELECT ARRAY[1, 2] || '7';                -- これも同様
```

```
ERROR:  malformed array literal: "7"


SELECT ARRAY[1, 2] || NULL;                -- 修飾されていないNULLも同様
 ?column?
----------
 {1,2}
(1 row)


SELECT array_append(ARRAY[1, 2], NULL);    -- これがやりたかった事かも
 array_append
--------------
 {1,2,NULL}
```

上の例では、パーサは連結演算子の一方の側に整数の配列を見つけ、もう一方の側に型の決まらない定数を見つけます。パーサが定数の型を解決するのに使う発見的手法は、演算子のもう一方の入力と同じ型(この場合には整数の配列)だと仮定することです。そのため、連結演算子はarray_appendではなく、array_catと推定されます。これが誤った選択である場合には、定数を配列の要素の型にキャストすることで直せるかもしれません。ですが、array_appendを明示的に使うのが好ましい解決法であるかもしれません。

8.15.5. 配列内の検索

配列内のある値を検索するにはそれぞれの値が検証されなければなりません。もし配列の大きさがわかっているならば手作業でも検索できます。例を挙げます。

```
SELECT * FROM sal_emp WHERE pay_by_quarter[1] = 10000 OR
                            pay_by_quarter[2] = 10000 OR
                            pay_by_quarter[3] = 10000 OR
                            pay_by_quarter[4] = 10000;
```

とは言ってもこの方法では大きい配列では大変な作業となりますし、配列の大きさが不明な場合この方法は使えません。代わりになる方法が9.24で説明されています。上の問い合わせは以下のように書くことができます。

```
SELECT * FROM sal_emp WHERE 10000 = ANY (pay_by_quarter);
```

さらに配列で行の値が全て10000に等しいものを見つけることもできます。

```
SELECT * FROM sal_emp WHERE 10000 = ALL (pay_by_quarter);
```

代わりとして、generate_subscripts関数を使うことができます。以下はその例です。

```
SELECT * FROM
   (SELECT pay_by_quarter,
```

```
           generate_subscripts(pay_by_quarter, 1) AS s
      FROM sal_emp) AS foo
 WHERE pay_by_quarter[s] = 10000;
```

この関数は表 9.62に記載されています。

&&演算子を使って配列を検索することもできます。この演算子は左辺が右辺と重なるかどうかを調べます。例えば、

```
SELECT * FROM sal_emp WHERE pay_by_quarter && ARRAY[10000];
```

この演算子やその他の配列の演算子は9.19により詳しく書かれています。11.2に書いてあるように、適切なインデックスにより高速化されます。

関数array_positionやarray_positionsを使って、配列内の特定の値を検索することもできます。前者は配列内で初めてその値が現れる添字を返し、後者は配列内でその値が現れる添字すべての配列を返します。例えば、以下の通りです。

```
SELECT array_position(ARRAY['sun','mon','tue','wed','thu','fri','sat'], 'mon');
 array_position
----------------
              2
(1 row)

SELECT array_positions(ARRAY[1, 4, 3, 1, 3, 4, 2, 1], 1);
 array_positions
-----------------
 {1,4,8}
(1 row)
```

ヒント

配列は集合ではありません。特定の配列要素に検索をかけることはデータベース設計が誤っている可能性があります。配列の要素とみなされるそれぞれの項目を行に持つ別のテーブルを使うことを検討してください。この方が検索がより簡単になり要素数が大きくなっても規模的拡張性があります。

8.15.6. 配列の入出力構文

配列の値の外部表現は配列の要素の型に対するI/O変換ルールに基づいて解釈された項目と配列の構造を示す装飾項目で構成されています。装飾は配列の値を中括弧({と})で囲んだものと次の項目との間を区切り文字で区切ったものです。区切り文字は通常カンマ(,)ですが他の文字でも構いません。配列の要素の型typdelimを設定することで決まります。PostgreSQL配布物における標準のデータ型の中でセミコロン(;)を使うbox型を除いて、すべてはカンマを使います。多次元配列ではそれぞれの次元(行、面、立体など)はそれ自身の階層において中括弧、同じ階層の中括弧でくくられた次の塊との間に区切り文字が書かれていなければなりません。

空の文字列や中括弧や区切り文字、二重引用符、バックスラッシュ、空白、NULLという単語が含まれていると、配列出力処理は要素の値を二重引用符でくくります。要素の値に組み込まれている二重引用符とバックスラッシュはバックスラッシュでエスケープされます。数値データ型に対しては二重引用符が出現しないと想定するのが安全ですが、テキストデータ型の場合引用符がある場合とない場合に対処できるようにしておくべきです。

デフォルトでは配列の次元の下限インデックス値は1に設定されています。他の下限値を持つ配列を表現したければ、配列定数を作成する前に明示的に配列添字範囲を指定することで実現できます。修飾項目はそれぞれの配列次元の上限と下限をコロン(:)で区切って前後を大括弧([])でくくった形式になっています。代入演算子(=)の後に配列次元修飾項目が続きます。例を示します。

```
SELECT f1[1][-2][3] AS e1, f1[1][-1][5] AS e2
 FROM (SELECT '[1:1][-2:-1][3:5]={{{1,2,3},{4,5,6}}}'::int[] AS f1) AS ss;

 e1 | e2
----+----
  1 |  6
(1 row)
```

1とは異なる下限を持つ場合にのみ、配列出力関数はその結果に明示的な次元を含めます。

要素に指定された値がNULL(またはその亜種)の場合、要素はNULLとして扱われます。引用符やバックスラッシュがあると、これは無効となり、「NULL」という文字列リテラルを入力することができます。また、8.2以前のPostgreSQLとの後方互換性のため、array_nulls設定パラメータをoffにして、NULLをNULLとして認識しないようにすることができます。

前に示したように配列に値を書き込む場合は独立した配列要素を二重引用符でくくります。配列値パーサが配列要素値によって混乱を来さないように必ずこの形式を守ってください。例えば、中括弧、カンマ(もしくはデータ型の区切り文字)、二重引用符、バックスラッシュもしくは前後に付いた空白を含む要素は必ず二重引用符でくくらなければなりません。空文字列やNULLという単語自体も同様に引用符でくくらなければなりません。二重引用符もしくはバックスラッシュを引用符付きの配列要素に付け加えたい場合、その直前にバックスラッシュを付けます。別の方法として、配列構文とみなされかねない全てのデータ文字を保護するために、引用符を使用しないでバックスラッシュでエスケープしても構いません。

括弧の右側もしくは左側それぞれの前と後に空白を追加することができます。同様に独立した項目の文字列の前後に空白を付け加えることもできます。これらすべての場合において空白は無視されます。とは言っても二重引用符で囲まれた要素の中の空白、もしくは要素の空白文字以外により両側がくくられているものは無視されません。

> **ヒント**
>
> SQLコマンドの中で配列値を書く時、配列リテラル構文よりもARRAY生成子構文(4.2.12を参照)の方が往々にして扱いやすい場合があります。

8.16. 複合型

複合型は、行もしくはレコードの構造を表現します。本質的には、これは単なるフィールド名とそのデータ型のリストです。PostgreSQLでは、単純な型において使用される方法と多くは同じ方法で複合型を使用できます。例えば、テーブルの列は複合型の型のものとして宣言することができます。

8.16.1. 複合型の宣言

複合型の宣言の例を以下に2つ示します。

```
CREATE TYPE complex AS (
    r       double precision,
    i       double precision
);

CREATE TYPE inventory_item AS (
    name            text,
    supplier_id     integer,
    price           numeric
);
```

この構文は、フィールド名とその型のみを指定できるという点を除き、CREATE TABLEと同等です。現在は、制約(NOT NULLなど)を含めることはできません。ASキーワードが重要であることに注意してください。これがないと、システムはCREATE TYPEの意味を異なって解釈し、おかしな構文エラーを引き起こします。

定義済みの型を使用して、以下のようにテーブルや関数を生成することができます。

```
CREATE TABLE on_hand (
    item      inventory_item,
    count     integer
);

INSERT INTO on_hand VALUES (ROW('fuzzy dice', 42, 1.99), 1000);
```

また、関数においては以下のように利用できます。

```
CREATE FUNCTION price_extension(inventory_item, integer) RETURNS numeric
AS 'SELECT $1.price * $2' LANGUAGE SQL;

SELECT price_extension(item, 10) FROM on_hand;
```

テーブルを生成する時には、テーブルの行型を表現するために、テーブル名と同じ名前の複合型も自動的に生成されます。例えば、以下のように

```
CREATE TABLE inventory_item (
    name            text,
    supplier_id     integer REFERENCES suppliers,
```

```
    price           numeric CHECK (price > 0)
);
```

テーブルを作成すると、上述のものと同じinventory_itemという複合型が副次的に作成され、同様に使用することができるようになります。しかし、現在の実装には、次のような重要な制限があることに注意してください。複合型には制約が関連付けられませんので、テーブル定義に含まれる制約は、テーブルの外部に作成される複合型には適用されません。(これを回避するためには、複合型を含むドメインを作成し、ドメインのCHECK制約として望みの制約を適用します。)

8.16.2. 複合型の値の構成

複合型をリテラル定数として記述するには、フィールド値をカンマで区切り、それらを括弧で括ります。フィールド値を二重引用符で括ることができ、また、値にカンマや括弧を含む場合は二重引用符で括らなければなりません (より詳細については後で説明します)。したがって、複合型の定数の一般的な書式は以下のようになります。

```
'( val1 , val2 , ... )'
```

以下に例を示します。

```
'("fuzzy dice",42,1.99)'
```

これは、上述のinventory_item型の値として有効なものです。フィールドをNULLにするには、リスト中の該当位置を空にします。例えば、以下の定数は3番目のフィールドにNULLを指定しています。

```
'("fuzzy dice",42,)'
```

NULLではなく空文字列にしたいのであれば、以下のように引用符を二重に記述します。

```
'("",42,)'
```

これにより、最初のフィールドは非NULLの空文字列に、3番目のフィールドはNULLになります。

(実際には、こうした定数は4.1.2.7で説明した、一般的な型の定数の特殊な場合に過ぎません。定数はまず、文字列として扱われ、複合型の入力変換処理に渡されます。定数をどの型に変換するかを示すため、明示的な型指定が必要になることもあります。)

また、ROW式構文も、複合値を生成する際に使用することができます。複数の階層に渡る引用符について考慮する必要がないため、おそらくほとんどの場合、これは文字列リテラル構文よりも簡単に使用できます。上記において、既にこの方法を使用しています。

```
ROW('fuzzy dice', 42, 1.99)
ROW('', 42, NULL)
```

式の中に2つ以上のフィールドがある場合には、ROWキーワードは実際には省略することができます。ですので、以下のように簡略化することができます。

```
('fuzzy dice', 42, 1.99)
('', 42, NULL)
```

ROW構文については4.2.13でより詳細に説明します。

8.16.3. 複合型へのアクセス

複合型の列のフィールドにアクセスするには、テーブル名からフィールドを選択する場合とほぼ同様に、ドットとフィールド名を記述します。実際、テーブル名からの選択とかなり似ていますので、パーサを混乱させないように括弧を使用しなければならないことがしばしばあります。例えば、on_handというテーブルの例からサブフィールドを選択しようとした場合、以下のように書くかもしれません。

```
SELECT item.name FROM on_hand WHERE item.price > 9.99;
```

これは、SQLの構文規則に従ってitemがon_handの列名ではなくテーブル名として解釈されるため、動作しません。以下のように記述しなければなりません。

```
SELECT (item).name FROM on_hand WHERE (item).price > 9.99;
```

また、テーブル名も使用しなければならない場合（例えば複数テーブルに対する問い合わせ）、以下のようになります。

```
SELECT (on_hand.item).name FROM on_hand WHERE (on_hand.item).price > 9.99;
```

これで、括弧で括られたオブジェクトは正しくitem列への参照として解釈され、サブフィールドはそこから選択できるようになります。

似たような構文上の問題は、複合型からフィールドを選択する時、常に発生します。例えば、複合型の値を返す関数の結果から1つだけフィールドを選択する場合、以下のように記述しなければなりません。

```
SELECT (my_func(...)).field FROM ...
```

追加の括弧がないと、これは構文エラーを生成します。

8.16.5でより詳細に説明する通り、*という特別なフィールド名は「すべてのフィールド」を意味します。

8.16.4. 複合型の変更

複合型の列への挿入と更新についての適切な構文の例をいくつか示します。まず、列全体を挿入、更新する例です。

```
INSERT INTO mytab (complex_col) VALUES((1.1,2.2));

UPDATE mytab SET complex_col = ROW(1.1,2.2) WHERE ...;
```

最初の例ではROWを省略し、2番目の例ではROWを使用しています。どちらの方法でも行うことができます。

以下のようにして、複合型の列の個々のサブフィールドを更新することができます。

```
UPDATE mytab SET complex_col.r = (complex_col).r + 1 WHERE ...;
```

ここで、SET直後の列名の周りに括弧を記述する必要がないこと（実際には記述できないこと）、しかし、等号の右で同じ列を参照する場合には括弧が必要なことに注意してください。

また、INSERTの対象としてサブフィールドを指定することもできます。

```
INSERT INTO mytab (complex_col.r, complex_col.i) VALUES(1.1, 2.2);
```

列のサブフィールド全ての値を与えていなければ、残りのサブフィールドはNULL値になります。

8.16.5. 問い合わせでの複合型の使用

問い合わせ内での複合型に関連して様々な特別な構文規則や動作があります。これらの規則により便利なショートカットが提供されますが、その背後にある論理を知らないと混乱を招くかもしれません。

PostgreSQLでは、問い合わせでのテーブル名（または別名）の参照は、実質的にはテーブルの現在行の複合型の値への参照と同じになります。例えば、前に示したinventory_itemというテーブルがあるとして、次のように記述することができます。

```
SELECT c FROM inventory_item c;
```

この問い合わせは単一の複合型の値の列を生成するので、出力は以下のようになります。

```
c
-----------------------
 ("fuzzy dice",42,1.99)
(1 row)
```

ただし、単純な名前はテーブル名より先に列名に対してマッチさせられるので、この例は問い合わせのテーブルにcという名前の列がないから動作したに過ぎないことに注意してください。

通常のtable_name.column_nameという列名修飾の構文は、フィールド選択をテーブルの現在行の複合型の値に対して適用していると考えることもできます。（効率の問題から、実際にはそのような実装にはなっていません。）

```
SELECT c.* FROM inventory_item c;
```

上記のSQLについて、標準SQLではテーブルの内容が別々の列に展開されて、次のような結果になることを定めています。

```
name    | supplier_id | price
```

```
------------+-------------+-------
 fuzzy dice |          42 |  1.99
(1 row)
```

つまりこれは、問い合わせが以下であったかのように動作するということです。

```
SELECT c.name, c.supplier_id, c.price FROM inventory_item c;
```

PostgreSQLでは、この展開の動作をすべての複合型の値の式に適用します。ただし、前に説明したように、.*をつける値が単純なテーブル名でないときは、必ずそれを括弧で括る必要があります。例えば、myfunc()が列a、b、cからなる複合型を返す関数だとすると、次の2つの問い合わせは同じ結果を返します。

```
SELECT (myfunc(x)).* FROM some_table;
SELECT (myfunc(x)).a, (myfunc(x)).b, (myfunc(x)).c FROM some_table;
```

ヒント

PostgreSQLでは、上の1番目の構文を2番目の構文に実際に変換することで列の展開を処理します。従って、この例ではどちらの構文を使ってもmyfunc()は各行に対して3回ずつ呼び出されます。それが高価な関数でそのような事態を避けたいなら、次のような問い合わせにすることもできます。

```
SELECT m.* FROM some_table, LATERAL myfunc(x) AS m;
```

LATERAL FROM項目の中に関数を置くと、関数は1行につき2度以上は呼び出されません。m.*はまだm.a, m.b, m.cに展開されますが、その変数はFROM項目の出力の単なる参照です。(LATERALキーワードはここでは省略可能ですが、関数がsome_tableからxを入手していることを明確にするために書きました。)

composite_value.*の構文は、それがSELECTの出力リスト、INSERT/UPDATE/DELETEのRETURNINGリスト、VALUES句あるいは行コンストラクタの最上位に記述された場合、この種の列展開がされます。それ以外の場合(これらの構文の内側に入れ子になっている場合を含みます)は、複合型の値に.*を付加しても、値は変わりません。なぜなら、それは「すべての列」を意味するため、同じ複合型の値が繰り返し生成されるからです。例えば、somefunc()が複合型の値の引数をとるとして、以下の問い合わせは同じです。

```
SELECT somefunc(c.*) FROM inventory_item c;
SELECT somefunc(c) FROM inventory_item c;
```

どちらの場合もinventory_itemの現在行が単一の複合型の値の引数として関数に渡されます。このような場合に.*は何もしませんが、それをつけることにより、複合型の値であることを意図しているのが明確になるので、つけるのは良い習慣です。特に、パーサがc.*のcを列名ではなくテーブル名あるいは別名を参照するものとみなす一方、.*がないとcがテーブル名なのか列名なのか明らかではなく、実際には、cという名前の列があれば列名としての解釈が優先されてしまいます。

これらの考え方を示す別の例をあげると、以下の3つの問い合わせは同じ意味になります。

```
SELECT * FROM inventory_item c ORDER BY c;
```

```
SELECT * FROM inventory_item c ORDER BY c.*;
SELECT * FROM inventory_item c ORDER BY ROW(c.*);
```

これらのORDER BY句はすべて行の複合型の値を指定しており、9.24.6で説明される規則に従って行を並べ替えた結果になります。ただし、inventory_itemにcという名前の列がある場合は、最初の例はその列によってのみ並べ替えられるので、他の2つとは異なるものになります。以前に示したのと同じ列名であるとしたら、以下の問い合わせも上記のものと同じになります。

```
SELECT * FROM inventory_item c ORDER BY ROW(c.name, c.supplier_id, c.price);
SELECT * FROM inventory_item c ORDER BY (c.name, c.supplier_id, c.price);
```

（最後の例はキーワードROWを省略した行コンストラクタを使用しています。）

複合型の値に関連したもう一つの特別な構文的動作は、複合型の値のフィールドを取り出す時に関数的記法を使用できることです。これを簡単に説明するなら、field(table)という記法とtable.fieldという記法は相互に交換可能です。例えば、以下の問い合わせは同等です。

```
SELECT c.name FROM inventory_item c WHERE c.price > 1000;
SELECT name(c) FROM inventory_item c WHERE price(c) > 1000;
```

さらに、複合型の引数を1つだけとる関数があるとして、それをどちらの記法でも呼び出すことができます。以下の問い合わせはすべて同等です。

```
SELECT somefunc(c) FROM inventory_item c;
SELECT somefunc(c.*) FROM inventory_item c;
SELECT c.somefunc FROM inventory_item c;
```

この関数的記法とフィールド記法の同等性により、複合型に対する関数を使用して「計算されたフィールド」を実装することができます。　上の最後の問い合わせを使用するアプリケーションは、somefuncがテーブルの真の列ではないことを直接には意識する必要がありません。

> **ヒント**
>
> このような動作になるため、複合型の引数を一つだけとる関数に、その複合型に含まれるフィールドと同じ名前をつけることは賢明ではありません。曖昧なときには、フィールド名の構文が使われていれば、フィールド名の解釈が選ばれ、関数呼び出しの構文が使われていれば、関数が選ばれます。しかしながら、11より前のPostgreSQLのバージョンでは、呼び出し構文が関数呼び出しとしてしか扱えない場合を除いて、常にフィールド名の解釈を選んでいました。関数としての解釈を強制する一つの方法は、関数名をスキーマ修飾する、つまりschema.func(compositevalue)とすることです。

8.16.6. 複合型の入出力構文

複合型の外部テキスト表現は、個々のフィールド用のI/O変換規則に従って解釈される項目群と、複合構造を意味する修飾から構成されます。この修飾は、値全体を括る括弧（(および)）と隣接した項目間のカンマ（,）で構成されます。括弧の外側の空白文字は無視されますが、括弧の内部ではフィールド値の一部とみな

されます。ただし、空白に意味があるかないかについては、そのフィールドのデータ型用の入力変換規則に従います。例えば、

```
'(  42)'
```

括弧内の空白文字は、そのフィールド型が整数の場合は無視されますが、テキストの場合は無視されません。

前述の通り、複合型の値を記述する時には、個々のフィールド値を二重引用符で括ることができます。もし、フィールド値が複合型値用のパーサを混乱させる場合には、これは必須です。具体的には、括弧、カンマ、二重引用符、バックスラッシュを含むフィールドの場合、二重引用符で括る必要があります。引用符で括った複合型のフィールド値内に二重引用符やバックスラッシュが存在する場合、その前にバックスラッシュを付けてください（また、引用符で括った複合型のフィールド値内に二重の引用符の組み合わせがあると、これは二重引用符を表す文字として解釈されます。これは、SQLリテラル文字列内の単一引用符の規則と同じです）。そのままでは複合型に対する構文として解釈されてしまう、全てのデータ文字を保護する他の方法として、引用符付けをせずにバックスラッシュによるエスケープを使用することができます。

完全な空フィールド値（カンマや括弧の間にまったく文字がないもの）はNULLを表します。NULLではなく空文字列を値として記述するには "" と記述してください。

複合型の出力処理では、もしフィールド値が空文字列の場合や括弧、カンマ、二重引用符、バックスラッシュ、空白文字を含む場合には、そのフィールド値を二重引用符で括って出力します（空白文字に対するこの処理は重要ではありませんが、可読性を高めます）。フィールド値内に埋め込まれた二重引用符やバックスラッシュは二重化されます。

注記

SQLコマンド内部に記述したものは、まず文字列リテラルとして、その後、複合型として解釈されることを覚えておいてください。これは必要なバックスラッシュの数を倍にします（エスケープ文字列構文が使用されることを仮定しています）。例えば、複合型の値の中に二重引用符とバックスラッシュを持つtextフィールドに挿入するには、以下のように書かなければなりません。

```
INSERT ... VALUES ('("\"\\")');
```

文字列リテラルプロセッサが第1レベルのバックスラッシュを取り除くため、複合型値のパーサに渡されるものは ("\"\\") のようになります。そして、textデータ型の入力関数に渡される文字列は"\になります（もし、例えばbyteaといった、その入力関数もバックスラッシュを特別に扱うデータ型を扱っている場合、1つのバックスラッシュを複合型のフィールドに格納するためにコマンド内に8個ものバックスラッシュが必要になります）。ドル引用符付け（4.1.2.4を参照）を使用して、このバックスラッシュの二重化を防ぐことができます。

ヒント

SQLコマンド内に複合型の値を書く時、通常、ROW生成構文の方が複合型のリテラル構文より作業が簡単です。ROWによる記述では、複合型のメンバ以外の記述方法と同じ方法で個々のフィールド値を記述することができます。

8.17. 範囲型

範囲型は、ある要素型(その範囲の派生元型と呼ばれます)の値の範囲を表わすデータ型です。例えば、timestampの範囲は、会議室が予約されている時間の範囲を表すのに使うことができるでしょう。この場合、データ型はtsrange(「timestamp range」の短縮)で、timestampが派生元型となります。派生元型には完全な順序がなければなりません。これは、要素の値が範囲の前、中間、後のどこにあるのか明確に定義されている必要があるからです。

範囲型は、一つの範囲内の多くの要素の値を表現できる、また、範囲の重なりなどの概念が明確に表現できる、などの理由で便利です。スケジューリングのために時刻と日付の範囲を使うのがもっとも簡単な例ですが、価格の範囲、機器による測定値の範囲などといったものにも利用できるでしょう。

8.17.1. 組み込みの範囲型

PostgreSQLには、以下の組み込みの範囲型があります。

- int4range — integerの範囲
- int8range — bigintの範囲
- numrange — numericの範囲
- tsrange — timestamp without time zoneの範囲
- tstzrange — timestamp with time zoneの範囲
- daterange — dateの範囲

この他にも、独自の範囲型を定義することができます。詳しくはCREATE TYPEを参照してください。

8.17.2. 例

```
CREATE TABLE reservation (room int, during tsrange);
INSERT INTO reservation VALUES
    (1108, '[2010-01-01 14:30, 2010-01-01 15:30)');


-- 含有
SELECT int4range(10, 20) @> 3;


-- 重なり
SELECT numrange(11.1, 22.2) && numrange(20.0, 30.0);
```

```
-- 上限の取得
SELECT upper(int8range(15, 25));


-- 共通部分の計算
SELECT int4range(10, 20) * int4range(15, 25);


-- 範囲は空か
SELECT isempty(numrange(1, 5));
```

範囲型についての演算子と関数の完全なリストについては、表 9.53と表 9.54を参照してください。

8.17.3. 閉じた境界と開いた境界

空でない範囲には必ず2つの境界、つまり下限値と上限値があります。これらの値の間にある値はすべてその範囲に含まれます。閉じた境界とは、その境界値自体が範囲に含まれることを意味し、開いた境界とは、その境界値が範囲に含まれないことを意味します。

範囲を文字列の形式で表すとき、閉じた下限値は「[」で、開いた下限値は「(」で表します。同様に、閉じた上限値は「]」で、開いた上限値は「)」で表します。(詳しくは 8.17.5を参照してください。)

関数lower_incおよびupper_incはそれぞれ、範囲の下限値と上限値が閉じているかどうかを検査します。

8.17.4. 無限の(境界のない)範囲

例えば、(,3]のように、範囲の下限値は省略することができ、このとき、上限値より小さいすべての値はその範囲に含まれることになります。同じように、範囲の上限値も省略することができ、このときは、下限値より大きいすべての値がその範囲に含まれることになります。下限値と上限値が両方とも省略されたときは、その要素型のすべての値がその範囲に含まれるとみなされます。省略された閉じた境界は自動的に開いた境界に変換されます。例えば、[,]は(,)に変換されます。省略された値を+/-無限大と考えることができますが、特殊な範囲型の値であり、いかなる範囲の要素型の+/-無限大の値も越えていると考えられます。

「無限大」の概念がある要素型では、それを明示的な境界値として使用できます。例えば、timestampの範囲で[today,infinity)は特殊なtimestamp値、infinityを含みませんが、一方、[today,infinity]は[today,)や[today,]と同じようにinfinityを含みます。

関数lower_infおよびupper_infはそれぞれ範囲の下限値と上限値が無限大かどうかを検査します。

8.17.5. 範囲の入出力

範囲値の入力は、以下の形式の一つに従わなければなりません。

```
(lower-bound,upper-bound)
```

```
(lower-bound,upper-bound]
[lower-bound,upper-bound)
[lower-bound,upper-bound]
empty
```

前にも述べたとおり、丸括弧と大括弧は下限値と上限値が開いているか閉じているかを表します。最後の形式がemptyであることに注意してください。これは空の範囲(範囲に含まれる値が1つもない)を表します。

lower-boundは、その派生元型の有効な入力値となる文字列か、あるいは省略して下限値がないことを指定するかのいずれかです。同様に、upper-boundは、その派生元型の有効な入力値となる文字列か、あるいは省略して上限値がないことを指定するかのいずれかです。

境界値は"(二重引用符)で括ることができます。これは特に境界値が丸括弧、大括弧、カンマ、二重引用符、あるいはバックスラッシュを含んでいる場合に必要となります。そうしなければ、これらの文字は範囲の構文の一部とみなされてしまうからです。二重引用符あるいはバックスラッシュを引用符で括られた境界値の中に入れるには、その直前にバックスラッシュを入れてください。(また、SQLの文字列リテラルと同じように、二重引用符で括られた境界値の中で二重引用符を2つ続けることで1つの二重引用符を表すこともできます。)あるいは、引用符で括る代わりに、範囲の構文の一部とみなされるすべての文字をバックスラッシュでエスケープする、ということもできます。なお、境界値として空文字列を指定するには""と書いてください。何も書かないと、境界値が無限大であることになってしまいます。

境界値の前後に空白文字を入れることができますが、括弧内にある空白文字はすべて下限値あるいは上限値の一部とみなされます。(このことは、要素型によっては重要かもしれませんし、重要でないかもしれません。)

> **注記**
>
> これらの規則は、複合型のリテラルにフィールド値を記述する時と非常によく似ています。詳細な解説は8.16.6を参照してください。

例:

```
-- 3を含み、
      7を含まない。その間の数はすべて含まれる
SELECT '[3,7)'::int4range;


-- 3も7も含まないが、
      その間の数はすべて含まれる
SELECT '(3,7)'::int4range;


-- 1つの値、
      4だけを含む
SELECT '[4,4]'::int4range;
```

```
-- 含まれる点は何もない('empty'に正規化される)
SELECT '[4,4)'::int4range;
```

8.17.6. 範囲の生成

範囲型には、その範囲型と同じ名前のコンストラクタ関数があります。コンストラクタ関数を使うと、境界値の指定で余計な引用を使わずに済むので、リテラルの定数で範囲を記述するよりも便利なことが多いでしょう。コンストラクタ関数は2つ、または3つの引数をとります。引数が2つの形式では、(閉じた下限値, 開いた上限値)という標準的な形式の範囲を生成します。引数が3つの形式では、3番目の引数で指定した形式の境界の範囲を生成します。3番目の引数は、以下の文字列のいずれかでなければなりません。「()」、「(]」、「[)」、または「[]」。例えば、

```
-- 完全な形式では、
      下限値、
      上限値、
      そして境界が閉じているか開いているかを
-- 示す文字列の引数を指定する
SELECT numrange(1.0, 14.0, '(]');


-- 3番目の引数が省略されると、
     '[)'を指定したのと同じになる
SELECT numrange(1.0, 14.0);


-- ここでは'(]'を指定しているが、
     int8rangeは離散的な範囲型(下記参照)なので
-- 正規化された形式に変換されて表示される
SELECT int8range(1, 14, '(]');


-- 境界値にNULLを指定すると、範囲の上限、あるいは下限がないことになる
SELECT numrange(NULL, 2.2);
```

8.17.7. 離散的な範囲型

離散的な範囲とは、integerやdateのように明確に定義された「ステップ」のある要素型の範囲のことです。このような型において、2つの要素の間に有効な値が1つもないとき、その2つの要素は隣接している、と言います。これは連続的な範囲と対照的です。連続的な範囲では、任意の2つの値について、それらの間に別の値を見つけることが、いつでも(あるいは、ほとんどいつでも)可能です。例えば、numeric型やtimestamp型の範囲は連続的です。(timestampの精度は限界があるので、理論的には離散的として取り扱うことも可能ですが、ステップの大きさについて関心がないのが普通ですから、連続的であると考える方が良いでしょう。)

離散的な範囲型に関するもう1つの考え方は、各要素の値について、「次」あるいは「前」の値というのものが明確に考えられるか、ということです。これを知っていれば、範囲の境界の閉じた表現、あるいは開いた表現について、その値の次、あるいは前の値を使って、表現を変換することができます。例えば、整数の範囲型[4,8]と(3,9)は同じ値の集合を意味しますが、これがnumericの範囲型であったならそうではありません。

離散的な範囲型はその要素型で使いたいステップのサイズを認識する正規化関数を持つべきです。正規化関数は同等な値の範囲型を、同一の表現に、特に、閉じた境界、開いた境界について一定の形式に変換します。正規化関数が指定されない場合、異なる形式の範囲は必ず等しくないものとして扱われます。これは例え、それらが現実的に同じ値の集合であったとしても、等しくないとされます。

組み込みの範囲型であるint4range、int8range、およびdaterangeはいずれも閉じた下限値と開いた上限値、つまり[)の正規化形式を使います。しかし、ユーザ定義の範囲型はこれとは別の方式を使うことができます。

8.17.8. 新しい範囲型の定義

独自の範囲型を定義することもできます。もっともありそうな理由は、組み込みの範囲型では、その派生元型についての範囲型が提供されていない、ということでしょう。例えば、float8を派生元型とする新しい範囲型を定義するには次のようにします。

```
CREATE TYPE floatrange AS RANGE (
    subtype = float8,
    subtype_diff = float8mi
);

SELECT '[1.234, 5.678]'::floatrange;
```

float8には意味のある「ステップ」がないので、この例では正規化関数を定義していません。

独自の範囲型を定義すると、派生元型とは異なるB-tree演算子クラスや照合順を指定でき、どの値が指定の範囲に入るかを決定するソート順を変更することもできます。

派生元型が、連続的ではなく離散的な値を持つと考えられる場合は、CREATE TYPEコマンドでcanonical(正規化)関数を指定する必要があります。正規化関数は、範囲の値を入力として受け取り、それと同等な範囲の値を返さなければなりません。戻り値は、入力値とは異なる境界値と形式になっているかもしれません。同じ値の集合を表す範囲、例えば、整数の範囲である[1, 7]と[1, 8)の正規化出力は、同一である必要があります。異なる形式の同等な値が、いつでも同じ形式の同じ値に変換されるのであれば、正規化出力の形式は何であってもかまいません。正規化関数は、閉じた境界、開いた境界の形式を調整するだけではありません。派生元型が格納できるよりも大きなサイズのステップを使いたい場合は境界値を丸めることもあります。例えばtimestampの範囲型をステップのサイズを1時間として定義することができます。このとき、正規化関数は1時間の倍数になっていない境界値を丸める必要があります。あるいは、その代わりにエラーを投げることもできます。

また、GiSTまたはSP-GiSTインデックスと一緒に使われる範囲型は、派生元型の差分、つまりsubtype_diff関数を定義すべきです。(そのインデックスはsubtype_diffがなくても機能しますが、差分関数が提供されている時に比べると、あまり効果的でないことが多いでしょう。) 派生元型の差分関数は、2

つの派生元型の入力値をとり、その差分(つまり、X引くY)をfloat8型の値として返します。上の例では、通常のfloat8のマイナス演算子が呼び出す関数float8miを使うことができますが、それ以外の派生元型では何らかの型変換が必要となるでしょう。差分をいかにして数字で表現するかについて、創造的な発想も必要になるかもしれません。可能な限りにおいて、subtype_diff関数は、選択した演算子クラスと照合順が示唆するソート順と矛盾しないようにすべき、つまり、ソート順で、1番目の引数が2番目の引数より上に来る場合は、必ず差分関数の結果は正になるべきです。

subtype_diff関数の単純化されすぎていない例を以下に示します。

```
CREATE FUNCTION time_subtype_diff(x time, y time) RETURNS float8 AS
'SELECT EXTRACT(EPOCH FROM (x - y))' LANGUAGE sql STRICT IMMUTABLE;

CREATE TYPE timerange AS RANGE (
    subtype = time,
    subtype_diff = time_subtype_diff
);

SELECT '[11:10, 23:00]'::timerange;
```

範囲型の作成について、より詳細な情報はCREATE TYPEを参照してください。

8.17.9. インデックス

範囲型の列にGiSTおよびSP-GiSTインデックスを作成することができます。例えば、GiSTインデックスを作成するには、

```
CREATE INDEX reservation_idx ON reservation USING GIST (during);
```

GiSTあるいはSP-GiSTインデックスがあると、以下の範囲演算子を含む検索を高速に実行できます。=、&&、<@、@>、<<、>>、-|-、&<、および&> (より詳細な情報は表 9.53を参照してください)。

さらに、範囲型の列にB-treeおよびハッシュインデックスを作ることもできます。これらのインデックスについては、基本的に、等値演算のみが有効な範囲の演算です。範囲の値についてB-treeのソート順が、<および>演算子について定義されていますが、現実にはこの順序はあまり意味がなく、有効ではありません。範囲型のB-treeとハッシュのサポートは実際にインデックスを作ることよりも、むしろ、検索時に内部的にソートやハッシュをできるようにするのが主な目的です。

8.17.10. 範囲の制約

UNIQUEはスカラー値には自然な制約ですが、範囲型には通常は適当ではありません。代わりに排他(exclude)制約を使うことの方が適切なことが多いです(CREATE TABLE ... CONSTRAINT ... EXCLUDEを参照してください)。排他制約により、範囲型について「重なりがない」などといった制約を指定することができます。例えば、

```
CREATE TABLE reservation (
```

```
    during tsrange,
    EXCLUDE USING GIST (during WITH &&)
);
```

この制約は、テーブル上で重なりのある値が同時に存在することを防ぎます。

```
INSERT INTO reservation VALUES
    ('[2010-01-01 11:30, 2010-01-01 15:00)');
INSERT 0 1

INSERT INTO reservation VALUES
    ('[2010-01-01 14:45, 2010-01-01 15:45)');
ERROR:  conflicting key value violates exclusion constraint "reservation_during_excl"
DETAIL:  Key (during)=(["2010-01-01 14:45:00","2010-01-01 15:45:00")) conflicts
with existing key (during)=(["2010-01-01 11:30:00","2010-01-01 15:00:00")).
```

btree_gistの拡張を使って通常のスカラーのデータ型について排他制約を定義することができます。これをさらに範囲の排他と組み合わせることで大きな柔軟性を得ることができます。例えば、btree_gistをインストールした時、次の制約は範囲の重なりについて、会議室の部屋番号も同じ時にのみ拒絶します。

```
CREATE EXTENSION btree_gist;
CREATE TABLE room_reservation (
    room text,
    during tsrange,
    EXCLUDE USING GIST (room WITH =, during WITH &&)
);

INSERT INTO room_reservation VALUES
    ('123A', '[2010-01-01 14:00, 2010-01-01 15:00)');
INSERT 0 1

INSERT INTO room_reservation VALUES
    ('123A', '[2010-01-01 14:30, 2010-01-01 15:30)');
ERROR:  conflicting key value violates exclusion constraint "room_reservation_room_during_excl"
DETAIL:  Key (room, during)=(123A, ["2010-01-01 14:30:00","2010-01-01 15:30:00")) conflicts
with existing key (room, during)=(123A, ["2010-01-01 14:00:00","2010-01-01 15:00:00")).

INSERT INTO room_reservation VALUES
    ('123B', '[2010-01-01 14:30, 2010-01-01 15:30)');
INSERT 0 1
```

8.18. ドメイン型

ドメインは他の基となる型を元にしたユーザ定義のデータ型です。オプションとして基となる型が許可する型のサブセットの有効な値を制限する制約を持つことができます。他は基となる型のように振る舞います。—例

えば、基となる型に適用できる演算子や関数はドメイン型でも動作します。ビルトインもしくはユーザが定義した基本型や列挙型、配列型、複合化型、範囲型もしくは他のドメインが基となる型になれます。

例として正の整数のみを許容する整数型のドメインを作成します。

```
CREATE DOMAIN posint AS integer CHECK (VALUE > 0);
CREATE TABLE mytable (id posint);
INSERT INTO mytable VALUES(1);   -- works
INSERT INTO mytable VALUES(-1);  -- fails
```

基となる型の演算子や関数にドメインの値が適用されると、ドメインは自動的に基となる型にダウンキャストされます。このため、例えば、mytable.id - 1の結果はposintではなく、integer型として考えられます。ドメイン制約の再チェックが発生するのでposint型にキャストするために(mytable.id - 1)::posintと記述することができます。このケースでは、式にidの値として1が与えられると結果はエラーになるでしょう。明確なキャストを書かずにドメイン型の変数やフィールドに基となる型の値を代入することが許容されていますが、ドメインの制約はチェックされます。

より詳細な情報はCREATE DOMAINを確認ください。

8.19. オブジェクト識別子データ型

オブジェクト識別子(OID)はPostgreSQLの内部で様々なシステムテーブルの主キーとして使用されます。oidデータ型はオブジェクト識別子を表します。oidには別名型もいくつかあります。reg何とかと名付けられたoidの様々なエイリアスの型は表 8.26からその概要を見ることができます。

oidデータ型は現在、符号なし4バイト整数として実装されています。このため、大きなデータベース内でデータベース単位での一意性や個別の大きなテーブルで一意性を提供するためには十分な大きさではありません。

oidデータ型自体は、比較以外の演算はほとんど行いません。しかし、整数としてキャストすることもでき、その場合標準の整数演算子を使用して操作することができます。(これを行うと、符号付きと符号なしの間で混乱が起きかねないことに注意してください。)

OIDの別名データ型は、専用の入出力ルーチン以外には演算を行いません。これらのルーチンでは、oid型が使用するような未加工の数値ではなく、システムオブジェクト用のシンボル名を受け入れたり表示したりできます。別名データ型により、オブジェクトのOID値の検索が簡単になります。例えば、mytableテーブルに関連したpg_attribute行を確認するには、以下のように記述することができます。

```
SELECT * FROM pg_attribute WHERE attrelid = 'mytable'::regclass;
```

次のように記述する必要はありません。

```
SELECT * FROM pg_attribute
  WHERE attrelid = (SELECT oid FROM pg_class WHERE relname = 'mytable');
```

後者もそう悪くないように見えますが、これは過度に単純化されています。異なるスキーマにmytableテーブルが複数ある場合には、正しいOIDを選択するために、より複雑なSELECTが必要となります。regclass入力

変換ではスキーマパスの設定に従ってテーブル検索を扱いますので、自動的に「正しい検索」を行います。同様に、テーブルのOIDをregclassにキャストすることは、数値のOIDのシンボル表示に便利です。

表8.26 オブジェクト識別子データ型

型名	参照	説明	値の例
oid	すべて	数値オブジェクト識別子	564182
regclass	pg_class	リレーション名	pg_type
regcollation	pg_collation	照合名	"POSIX"
regconfig	pg_ts_config	テキスト検索設定	english
regdictionary	pg_ts_dict	テキスト検索辞書	simple
regnamespace	pg_namespace	名前空間名	pg_catalog
regoper	pg_operator	演算子名	+
regoperator	pg_operator	引数の型を持つ演算子	*(integer,integer) or -(NONE,integer)
regproc	pg_proc	関数名	sum
regprocedure	pg_proc	引数の型を持つ関数	sum(int4)
regrole	pg_authid	ロール名	smithee
regtype	pg_type	データ型の名前	integer

名前空間でグループ化されたオブジェクトのOID別名型はすべてスキーマ修飾名を受け入れ、出力時にスキーマ修飾名を表示します。ただし、現在の検索パスでオブジェクトが見つけられなければ、修飾せずに出力します。regprocとregoper別名型は、一意な(オーバーロードしていない)名前のみを入力として受け入れるため、これらの使用には限度があります。ほとんどの場合、regprocedureまたはregoperatorを使用するのが適切です。regoperatorの場合、単項演算子は未使用のオペランドをNONEと記述することによって指定されます。

ほとんどのOID別名型のさらなる属性は依存性の作成です。これらの型の1つの定数が格納された式内に存在する場合(列のデフォルト式やビューなど)、参照されるオブジェクトへの依存性を生成します。例えば、列がnextval('my_seq'::regclass)というデフォルト式を持つ場合、PostgreSQLはデフォルト式がmy_seqシーケンスに依存することを理解します。システムは先にこのデフォルト式が削除されない限り、このシーケンスを削除させません。プロパティの唯一の例外はregroleです。このような式では、この型の定数は使用できません。

> **注記**
>
> OID別名型はトランザクション分離規則に完全には従いません。プランナも単なる定数として扱いますので、次善の計画になるかもしれません。

システムが使用するもう1つの識別子の型はxid、すなわちトランザクション(略してxact)識別子です。これはxminシステム列およびxmaxシステム列のデータ型です。トランザクション識別子は32ビット長です。文脈によっては64bitに変形したxid8が使われます。xidの値と違い xid8の値は厳密に単調増加し、データベースクラスタのライフタイムの中で再利用されることはありません。

システムが使用する3つ目の識別子はcid、すなわちコマンド識別子です。これはcminシステム列およびcmaxシステム列のデータ型です。コマンド識別子も32ビット長です。

システムが使用する最後の識別子はtid、すなわちタプル識別子（行識別子）です。これはctidシステム列のデータ型です。タプルIDはテーブル内の行の物理的位置を識別するための組（ブロック番号、ブロック内のタプルインデックス）です。

（システム列の詳細は5.5で説明します。）

8.20. pg_lsn 型

pg_lsn型はWALの位置を示すLSN(Log Sequence Number)データを格納するために使用します。この型はXLogRecPtrを示すPostgreSQLの内部的なシステムの型です。

内部的にはLSNは64bit整数型で、WALストリームのバイト位置を表現します。LSNは例えば、16/B374D848のように2つのスラッシュで分けられた8桁の16進数で表示されます。pg_lsnは例えば、=や>などの標準の比較演算子をサポートしています。2つのLSNは-演算子を使い引き算することも可能で、結果はこれらの2つのWALの位置のbytes差分です。

8.21. 疑似データ型

PostgreSQL型システムには、疑似データ型と総称される特殊用途のエントリが多数含まれます。疑似データ型は列データ型としては使用できませんが、関数の引数や結果データ型を宣言するために使用できます。これらの使用可能な疑似データ型は、ある関数の振舞いが、特定のSQLデータ型の値を単に取得したり返したりする操作に対応していない場合に便利です。表 8.27に既存の疑似データ型を列挙します。

表8.27 疑似データ型

型名	説明
any	関数がどのような入力データ型でも受け入れることを示します。
anyelement	関数がどのような入力データ型でも受け入れることを示します(37.2.5を参照)。
anyarray	関数がどのような配列データ型でも受け入れることを示します（37.2.5を参照してください）。
anynonarray	関数がどのような非配列データ型でも受け入れることを示します（37.2.5を参照してください）。
anyenum	関数が何らかの列挙データ型を受け入れることを示します（37.2.5および8.7を参照してください）。
anyrange	関数が範囲データ型を受け入れることを示します(37.2.5と 8.17を参照してください)。
anycompatible	関数が複数の引数を一般的なデータ型に自動的に昇格させるどのようなデータ型でも受け入れることを示します(37.2.5)を参照してください)。
anycompatiblearray	関数が複数の引数を一般的なデータ型に自動的に昇格させるどのような配列のデータ型でも受け入れることを示します(37.2.5を参照してください)。

型名	説明
anycompatiblenonarray	関数が複数の引数を一般的なデータ型に自動的に昇格させるどのような非配列のデータ型でも受け入れることを示します(37.2.5を参照ください)。
anycompatiblerange	関数が複数の引数を一般的なデータ型に自動的に昇格させるどのような範囲データ型でも受け入れることを示します(37.2.5と8.17を参照ください)。
cstring	関数がヌル終端のC文字列を受け入れる、もしくは返すことを示します。
internal	関数がサーバ内部用データ型を受け入れる、もしくは返すことを示します。
language_handler	手続き言語呼び出しハンドラはlanguage_handlerを返すものとして宣言されます。
fdw_handler	外部データラッパハンドラはfdw_handlerを返すものとして宣言されます。
table_am_handler	テーブルアクセスメソッドのハンドラはtable_am_handlerを返すものとして宣言されます。
index_am_handler	インデックスアクセスメソッドのハンドラは index_am_handlerを返すものとして宣言されます。
tsm_handler	テーブルサンプリング方式のハンドラはtsm_handlerを返すものとして宣言されます。
record	未指定の行型の引数を取る、あるいは返す関数を指定します。
trigger	トリガ関数はtriggerを返すものとして宣言されます。
event_trigger	イベントトリガ関数はevent_triggerを返すものとして宣言されます。
pg_ddl_command	イベントトリガが使用できるDDLコマンドの表現を指定します。
void	関数が値を返さないことを示します。
unknown	未解決の型を特定します。例えば、修飾されていない文字列リテラルのような型です。

C言語で作成された関数(それが組み込みか動的にロードされるかに関係なく)は、これらの疑似データ型のどれでも受け入れたり返したりするように宣言することができます。引数型として疑似データ型が使用されても関数が安全に機能するように、関数の作成時に気を付ける必要があります。

手続き型言語で作成された関数では、実装する言語によって許可された疑似データ型のみを使用できます。現在、ほとんどの手続き型言語では疑似データ型を引数型として使用することが原則として禁止されており、結果型としてのvoidとrecord(および関数がトリガまたはイベントトリガとして使用される場合のtriggerまたはevent_trigger)のみが許可されています。また、一部の関数は、多様な疑似型を使用する多様関数をサポートしています。これについては、前述の37.2.5で詳細に説明されています。

internal疑似データ型は、データベースシステムによって内部的にのみ呼び出される関数を宣言する場合に使用され、SQL問い合わせでの直接呼び出しには使用できません。関数に少なくとも1つのinternal型の引数があると、これをSQLから呼び出すことはできません。この制限の影響からデータ型の安全性を保持するためには、次のコーディング規則に従うことが重要です。internal引数が少なくとも1つある場合を除き、internalを返すと宣言される関数を作成すべきではありません。

第9章 関数と演算子

PostgreSQLは組み込みデータ型に対して数多くの関数と演算子を用意しています。この章ではそのほとんどについて説明しますが、特殊用途の関数はマニュアルの関連する節に記載しています。また、パート Vで解説しているように、ユーザは独自の関数と演算子を定義することもできます。psqlの\dfコマンドと\doコマンドはそれぞれ全ての使用可能な関数と演算子をリストするのに使用されます。

この章全体で関数と演算子の引数と返り値のデータ型の記述は以下のようになります。

```
repeat ( text, integer ) → text
```

つまり関数repeatは、一つのテキスト型と一つの整数型の引数を取り、テキスト型の結果を返します。また、右矢印を使ってある例の結果を示します。ですから、以下のようになります。

```
repeat('Pg', 4) → PgPgPgPg
```

もし移植性が気になるのであれば、最も基本的な算術および比較演算子と、いくつかの明示的に印を付けた関数を除き、本章で説明する大多数の関数と演算子は、標準SQLで規定されていない点に注意してください。この拡張機能のいくつかは、他のSQLデータベース管理システムにも備わっており、多くの場合この機能には各種実装間で互換性と整合性があります。

9.1. 論理演算子

通常の論理演算子が使用できます。

```
boolean AND boolean → boolean
boolean OR boolean → boolean
NOT boolean → boolean
```

SQLはtrue、false、そして「不明」を意味するnullの3値の論理システムを使用します。以下の真理値表を参照してください。

a	b	a AND b	a OR b
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	NULL	NULL	TRUE
FALSE	FALSE	FALSE	FALSE
FALSE	NULL	FALSE	NULL
NULL	NULL	NULL	NULL

a	NOT a
TRUE	FALSE

a	NOT a
FALSE	TRUE
NULL	NULL

AND演算子とOR演算子は可換です。つまり、結果に影響を与えることなく左右のオペランドを交換することができます。(しかし、左オペランドが右オペランドよりも先に評価されるという保証はありません。副式の評価順についてのより詳細は4.2.14を参照してください。)

9.2. 比較関数および演算子

表 9.1に示すように、通常の比較演算子が使用可能です。

表9.1 比較演算子

演算子	説明
datatype < datatype → boolean	小なり
datatype > datatype → boolean	大なり
datatype <= datatype → boolean	等しいかそれ以下
datatype >= datatype → boolean	等しいかそれ以上
datatype = datatype → boolean	等しい
datatype <> datatype → boolean	等しくない
datatype != datatype → boolean	等しくない

注記

<>がSQL標準における「等しくない」の記法です。!=はその別名で、構文解析のごく初期に<>に変換されます。ですから!=演算子と<>演算子に異なる処理を行わせる実装はできません。

これらの比較演算子は、数値、文字列、日付、時刻データ型などの自然な順序付けを持つすべての組み込みデータ型に用意されています。更に、要素となるデータ型が比較可能なら、配列、複合データ型、範囲は比較可能です。

通常関連性のあるデータ型も比較することができます。たとえばinteger >bigintも可能です。ある場合にはこれらの比較は「型をまたがる」比較演算子で直接実装されています。そうした演算子がなければ、パーサはより一般的ではない型をより一般的な型に変換して後者の比較演算子に適用します。

上で示したように、全ての比較演算子は二項演算子で、boolean型の値を返します。ですから1 < 2 < 3のような式は(ブール値と3を比較する<演算子がないので)無効です。下で示すBETWEEN述語を使って範囲検査を行ってください。

表 9.2に示すように、比較述語がいくつかあります。これらは演算子と同様に振る舞いますが、標準SQLによって強制される特別の構文があります。

表9.2 比較述語

述語 説明 例
datatype BETWEEN datatype AND datatype → boolean 間にある(範囲の端点を含む)。 2 BETWEEN 1 AND 3 → t 2 BETWEEN 3 AND 1 → f
datatype NOT BETWEEN datatype AND datatype → boolean 間にない(BETWEENの否定)。 2 NOT BETWEEN 1 AND 3 → f
datatype BETWEEN SYMMETRIC datatype AND datatype → boolean 2つの端点値をソートした上で、間にある。 2 BETWEEN SYMMETRIC 3 AND 1 → t
datatype NOT BETWEEN SYMMETRIC datatype AND datatype → boolean 2つの端点値をソートした上で、間にない。 2 NOT BETWEEN SYMMETRIC 3 AND 1 → f
datatype IS DISTINCT FROM datatype → boolean NULLを比較可能な値とした上で、等しくない。 1 IS DISTINCT FROM NULL → t (NULLではなく) NULL IS DISTINCT FROM NULL → f (NULLではなく)
datatype IS NOT DISTINCT FROM datatype → boolean NULLを比較可能な値とした上で、等しい。 1 IS NOT DISTINCT FROM NULL → f (NULLではなく) NULL IS NOT DISTINCT FROM NULL → t (NULLではなく)
datatype IS NULL → boolean 値がNULLかどうか検査する。 1.5 IS NULL → f
datatype IS NOT NULL → boolean 値がNULLではないかどうか検査する。 'null' IS NOT NULL → t
datatype ISNULL → boolean 値がNULLかどうか検査する。(非標準構文)
datatype NOTNULL → boolean 値がNULLではないかどうか検査する。(非標準構文)
boolean IS TRUE → boolean 論理式の結果が真となるかどうか検査する。 true IS TRUE → t NULL::boolean IS TRUE → f (NULLではなく)
boolean IS NOT TRUE → boolean 論理式の結果が偽または不明となるかどうか検査する。 true IS NOT TRUE → f

述語 説明 例
NULL::boolean IS NOT TRUE → t (NULLではなく)
boolean IS FALSE → boolean 論理式の結果が偽となるかどうか検査する。 true IS FALSE → f NULL::boolean IS FALSE → f (NULLではなく)
boolean IS NOT FALSE → boolean 論理式の結果が真または不明となるかどうか検査する。 true IS NOT FALSE → t NULL::boolean IS NOT FALSE → t (NULLではなく)
boolean IS UNKNOWN → boolean 論理式の結果が不明となるかどうか検査する。 true IS UNKNOWN → f NULL::boolean IS UNKNOWN → t (NULLではなく)
boolean IS NOT UNKNOWN → boolean 論理式の結果が真または偽となるかどうか検査する。 true IS NOT UNKNOWN → t NULL::boolean IS NOT UNKNOWN → f (NULLではなく)

BETWEEN述語は範囲の検査を次のように単純にします。

```
a BETWEEN x AND y
```

は

```
a >= x AND a <= y
```

と同じです。BETWEENは範囲内に含まれるとして端点値を扱うことに注意してください。BETWEEN SYMMETRICは、ANDの左側の引数が右側の引数より小さいか、もしくは等しいという必要性が無い点を除きBETWEENと同様です。この条件を満たしていない場合、2つの引数は自動的に交換されますので、常に空ではない範囲となります。

BETWEENの変種は通常の比較演算子を使って実装されており、比較可能なすべてのデータ型に対して使用できます。

> **注記**
>
> BETWEEN構文中でANDを使用すると、ANDを論理演算子として使うこととの曖昧さが生じます。これを解決するために、BETWEEN句の第2引数としては限定された式の種類のみが利用できます。BETWEEN中で複雑な副式を使用する必要がある場合は、副式を括弧で囲んでください。

入力のどちらかがNULLの場合、通常の比較演算子は真や偽ではなく(「不明」を意味する)nullを生成します。例えば7 = NULLはnullになります。7 <> NULLも同様です。この動作が適切でない場合は、IS [NOT] DISTINCT FROM述語を使用してください。

```
a IS DISTINCT FROM b
a IS NOT DISTINCT FROM b
```

非NULLの入力では、IS DISTINCT FROMは<>演算子と同じです。しかし、入力がどちらもNULLの場合、これは偽を返し、片方の入力のみがNULLの場合は真を返します。同様に、IS NOT DISTINCT FROMは非NULL入力では=と同じですが、両方の入力がNULLであれば真を、片方のみがNULLの場合は偽を返します。このように、これらの述語はNULLを「不明な値」ではなく、通常の値かのように動作します。

値がNULLかNULLでないかを検証するには次の述語を使います。

```
expression IS NULL
expression IS NOT NULL
```

あるいは、これと同等の、非標準の述語も使えます。

```
expression ISNULL
expression NOTNULL
```

NULLとNULLとは「等しい」関係にはありませんので、expression = NULLと記述してはいけません（NULL値は不明の値を表しているため、不明な値同士が同じかどうかは識別できません）。

> **ヒント**
>
> アプリケーションによっては、expression = NULLが、expressionがNULL値と評価されるのであれば真を返すことを期待することがあります。こうしたアプリケーションは標準SQLに従うように改修することを強く推奨します。しかし、それができなければtransform_null_equalsを使用することで対応することができます。これを有効にした場合、PostgreSQLはx = NULL句をx IS NULLに変換します。

expressionが行値の場合、行式自体がNULLまたは、行のフィールドすべてがNULLの場合にIS NULLは真となります。一方IS NOT NULLは、行式自体が非NULLかつ、行のフィールドすべてが非NULLの場合に真となります。この動作により、IS NULLおよびIS NOT NULLは行値評価式に対し常に反対の結果を返すわけではありません。特に、NULLと非NULLの値の両方を含む行値式はどちらの試験でも偽を返します。場合によっては、row IS DISTINCT FROM NULLあるいはrow IS NOT DISTINCT FROM NULLと記述する方が望ましいことがあるでしょう。これらは単に行全体の値がNULLかどうかを検査し、行のフィールドについての追加的検査を全く行わないからです。

論理値も次の述語で検証できます。

```
boolean_expression IS TRUE
boolean_expression IS NOT TRUE
boolean_expression IS FALSE
boolean_expression IS NOT FALSE
boolean_expression IS UNKNOWN
boolean_expression IS NOT UNKNOWN
```

これらは、常に真か偽を返し、演算項目がNULLであってもNULL値を返すことはありません。NULL値が入力されると、「不明」という論理値として扱われます。IS UNKNOWNとIS NOT UNKNOWNが、入力式が論理値型でなければならないという点を除き、それぞれ実質的にIS NULLとIS NOT NULLと同じであることに注意してください。

表 9.3に示すように、比較に関連した関数がいくつか使用可能です。

表9.3 比較関数

関数 説明 例
num_nonnulls (VARIADIC "any") → integer 非NULLの引数の数を返す。 num_nonnulls(1, NULL, 2) → 2
num_nulls (VARIADIC "any") → integer NULL引数の数を返す。 num_nulls(1, NULL, 2) → 1

9.3. 算術関数と演算子

PostgreSQLの数多くの型に対する算術演算子が用意されています。標準算術表現法が存在しない型（例えば、日付/時刻データ型）については、後続する節で実際の動作を説明します。

表 9.4は標準の数値型で使用可能な算術演算子を示しています。特に説明がない限り、numeric_typeを受け付けると表示されている演算子はすべてのsmallint、integer、bigint、numeric、real、double precisionデータ型で利用可能です。integral_typeを受け付けると表示されている演算子はすべてのsmallint、integer、bigintデータ型で利用可能です。特に説明がない限り、それぞれの演算子は引数と同じデータ型を返します。integer + numericのように、複数の引数データ型が使われる呼び出しは、このリストの後で現れる型を使って解決されます。

表9.4 算術演算子

演算子 説明 例
numeric_type + numeric_type → numeric_type 和 2 + 3 → 5
+ numeric_type → numeric_type 単項和（演算なし） + 3.5 → 3.5
numeric_type - numeric_type → numeric_type 差 2 - 3 → -1
- numeric_type → numeric_type

演算子 説明 例
否定 - (-4) → 4
numeric_type * numeric_type → numeric_type 積 2 * 3 → 6
numeric_type / numeric_type → numeric_type 商(整数型では、除算によってゼロへ余りが切り捨てられます) 5.0 / 2 → 2.5000000000000000 5 / 2 → 2 (-5) / 2 → -2
numeric_type % numeric_type → numeric_type 剰余(余り)。smallint、integer、bigint、numericで利用可能 5 % 4 → 1
numeric ^ numeric → numeric double precision ^ double precision → double precision 累乗(典型的な数学的用法とは違って、^は左から右に適用されます) 2 ^ 3 → 8 2 ^ 3 ^ 3 → 512
|/ double precision → double precision 平方根 |/ 25.0 → 5
||/ double precision → double precision 立方根 ||/ 64.0 → 4
bigint ! → numeric 階乗 (削除予定です。代わりにfactorial()を使ってください。) 5 ! → 120
!! bigint → numeric 前置演算子としての階乗(削除予定です。代わりにfactorial()を使ってください。) !! 5 → 120
@ numeric_type → numeric_type 絶対値 @ -5.0 → 5
integral_type & integral_type → integral_type ビットごとのAND 91 & 15 → 11
integral_type | integral_type → integral_type ビットごとのOR 32 | 3 → 35

演算子 説明 例
integer_type # integer_type → integer_type ビットごとの排他的論理和 17 # 5 → 20
~ integer_type → integer_type ビットごとのNOT ~1 → -2
integer_type << integer → integer_type ビットごとの左シフト 1 << 4 → 16
integer_type >> integer → integer_type ビットごとの右シフト 8 >> 2 → 2

表 9.5に使用可能な算術関数を示します。これら関数の多くは、異なる引数型を持つ複数の形で提供されています。特に記述がある場合を除き、すべての形式の関数はその引数と同じデータ型を返します。複数の型をまたがる場合は上記の演算子のところで説明したのと同じ方法で解決されます。double precisionデータに対する関数のほとんどはホストシステムのCライブラリの上層に実装されています。このため、精度と境界近くの場合の振舞いはホストシステムに依存して変わります。

表9.5 算術関数

関数 説明 例
abs (numeric_type) → numeric_type 絶対値 abs(-17.4) → 17.4
cbrt (double precision) → double precision 立方根 cbrt(64.0) → 4
ceil (numeric) → numeric ceil (double precision) → double precision 引数より大きいか等しく、引数に最も近い整数 ceil(42.2) → 43 ceil(-42.8) → -42
ceiling (numeric) → numeric ceiling (double precision) → double precision 引数より大きいか等しく、引数に最も近い整数（ceilと同じ） ceiling(95.3) → 96
degrees (double precision) → double precision ラディアンを度に変換

関数 説明 例
degrees(0.5) → 28.64788975654116
div (y numeric, x numeric) → numeric y/xの整数商（0に向かって切り捨て） div(9,4) → 2
exp (numeric) → numeric exp (double precision) → double precision 指数（eを底とする指定のべき乗） exp(1.0) → 2.7182818284590452
factorial (bigint) → numeric 階乗 factorial(5) → 120
floor (numeric) → numeric floor (double precision) → double precision 引数より小さいか等しく、引数に最も近い整数 floor(42.8) → 42 floor(-42.8) → -43
gcd (numeric_type, numeric_type) → numeric_type 最大公約数（余りなく入力を割る最大の正の整数）。両方の入力が0なら0を返す。integer、bigint、numericで利用可能 gcd(1071, 462) → 21
lcm (numeric_type, numeric_type) → numeric_type 最小公倍数（両方の入力の整数倍となる最小の厳密な正の数）。両方の入力が0なら0を返す。integer、bigint、numericで利用可能 lcm(1071, 462) → 23562
ln (numeric) → numeric ln (double precision) → double precision 自然対数 ln(2.0) → 0.6931471805599453
log (numeric) → numeric log (double precision) → double precision 10を底とした対数 log(100) → 2
log10 (numeric) → numeric log10 (double precision) → double precision 10を底とした対数（logと同じ） log10(1000) → 3
log (b numeric, x numeric) → numeric bを底としたxの対数 log(2.0, 64.0) → 6.0000000000
min_scale (numeric) → integer

関数 説明 例
与えられた値を正確に表現するのに必要な最小の桁数（小数点以下の10進の桁数） min_scale(8.4100) → 2
mod (y numeric_type, x numeric_type) → numeric_type y/xの剰余。smallint、integer、bigint、numericで利用可能 mod(9,4) → 1
pi ( ) → double precision πの近似値 pi() → 3.141592653589793
power (a numeric, b numeric) → numeric power (a double precision, b double precision) → double precision aのb乗 power(9, 3) → 729
radians (double precision) → double precision 度をラディアンに変換 radians(45.0) → 0.7853981633974483
round (numeric) → numeric round (double precision) → double precision 最も近い整数への丸め round(42.4) → 42
round (v numeric, s integer) → numeric vを小数点第s位までに丸め round(42.4382, 2) → 42.44
scale (numeric) → integer 引数の位取り（小数点以下の10進の桁数） scale(8.4100) → 4
sign (numeric) → numeric sign (double precision) → double precision 引数の符号(-1, 0,あるいは +1) sign(-8.4) → -1
sqrt (numeric) → numeric sqrt (double precision) → double precision 平方根 sqrt(2) → 1.4142135623730951
trim_scale (numeric) → numeric 後方のゼロを削除することにより値の桁数（小数点以下の10進桁数）を減じる trim_scale(8.4100) → 8.41
trunc (numeric) → numeric trunc (double precision) → double precision 整数へ切り捨て（ゼロに向かって）

関数 説明 例
trunc(42.8) → 42 trunc(-42.8) → -42
trunc (v numeric, s integer) → numeric vを小数点以下s桁で切り捨て trunc(42.4382, 2) → 42.43
width_bucket (operand numeric, low numeric, high numeric, count integer) → integer width_bucket (operand double precision, low double precision, high double precision, count integer) → integer lowからhighまでの範囲に広がる等幅でバケット数countのヒストグラムにおいて、operandが割り当てられるバケット番号を返す。範囲外の入力値に対しては0またはcount+1を返す。 width_bucket(5.35, 0.024, 10.06, 5) → 3
width_bucket (operand anyelement, thresholds anyarray) → integer バケットの最小値を示す配列が与えられた時に、operandが割り当てられるバケット番号を返す。最初の最小値よりも小さい入力値に対しては0を返す。operandと配列要素は標準の比較演算子を持つ型であればどのような型でも構いません。thresholds配列はソートされていなければならず、小さいものが最初です。さもなければ予想外の結果となるでしょう。 width_bucket(now(), array['yesterday', 'today', 'tomorrow']::timestamptz[]) → 2

表 9.6に乱数を生成する関数を示します。

表9.6 乱数関数

関数 説明 例
random ( ) → double precision 0.0 <= x < 1.0の範囲の乱数値を返す random() → 0.897124072839091
setseed (double precision) → void 今後のrandom()呼び出しで使用されるシード(種)を設定する。引数は-1.0から1.0までの境界を含む値でなければなりません。 setseed(0.12345)

random()関数は単純な線形合同法を使用しています。高速ですが、暗号用途には適していません。より安全な代替物としてpgcryptoモジュールを参照してください。setseed()が呼び出されると、現在のセッション内での以後の一連のrandom()の呼び出し結果はsetseed()を同じ引数で再実行することによって再現可能となります。

使用可能な三角関数を表 9.7に示します。それぞれの三角関数には、角度の単位をラディアンにするものと度にするものの2種類があります。

表9.7 三角関数

関数 説明 例
acos (double precision) → double precision 逆余弦関数、結果はラディアン acos(1) → 0
acosd (double precision) → double precision 逆余弦関数、結果は度 acosd(0.5) → 60
asin (double precision) → double precision 逆正弦関数、結果はラディアン asin(1) → 1.5707963267948966
asind (double precision) → double precision 逆正弦関数、結果は度 asind(0.5) → 30
atan (double precision) → double precision 逆正接関数、結果はラディアン atan(1) → 0.7853981633974483
atand (double precision) → double precision 逆正接関数、結果は度 atand(1) → 45
atan2 (y double precision, x double precision) → double precision y/xの逆正接関数、結果はラディアン atan2(1,0) → 1.5707963267948966
atan2d (y double precision, x double precision) → double precision y/xの逆正接関数、結果は度 atan2d(1,0) → 90
cos (double precision) → double precision 余弦関数、引数はラディアン cos(0) → 1
cosd (double precision) → double precision 余弦関数、引数は度 cosd(60) → 0.5
cot (double precision) → double precision 余接関数、引数はラディアン cot(0.5) → 1.830487721712452
cotd (double precision) → double precision 余接関数、引数は度 cotd(45) → 1
sin (double precision) → double precision 正弦関数、結果はラディアン

関数 説明 例
sin(1) → 0.8414709848078965
sind (double precision) → double precision 正弦関数、結果は度 sind(30) → 0.5
tan (double precision) → double precision 正接関数、引数はラディアン tan(1) → 1.5574077246549023
tand (double precision) → double precision 正接関数、引数は度 tand(45) → 1

> **注記**
>
> 度単位の角度を扱う別の方法は、前に示した単位変換関数radians()とdegrees()を使うことです。しかし、角度を使う方法の方が、sind(30)のような特別な場合の丸め誤差を避けられるため、推奨されます。

表 9.8に利用可能な双曲線関数を示します。

表9.8 双曲線関数

関数 説明 例
sinh (double precision) → double precision 双曲線正弦 sinh(1) → 1.1752011936438014
cosh (double precision) → double precision 双曲線余弦 cosh(0) → 1
tanh (double precision) → double precision 双曲線正接 tanh(1) → 0.7615941559557649
asinh (double precision) → double precision 逆双曲線正弦 asinh(1) → 0.881373587019543
acosh (double precision) → double precision 逆双曲線余弦 acosh(1) → 0
atanh (double precision) → double precision 逆双曲線正接

関数 説明 例
atanh(0.5) → 0.5493061443340548

9.4. 文字列関数と演算子

本節では文字列の値の調査や操作のための関数と演算子について説明します。ここでの文字列とはcharacterデータ型、character varyingデータ型、およびtextデータ型の値を含みます。補足説明のない限り、下記に挙げている全ての関数はtext型を受付、また戻り値型として返すように宣言されています。それらはcharacter varyingデータ型も同じように受け付けます。character型の値は関数あるいは演算子に適用される前にtextに変換され、character値の末尾の空白が削除されることになります。

SQLでは引数の区切りにカンマではなくキーワードを使用する文字列関数をいくつか定義しています。詳細については表 9.9を参照してください。またPostgreSQLは、これらの関数に対して通常の関数呼び出し構文を使用するバージョンを提供します（表 9.10を参照してください）。

> **注記**
>
> PostgreSQLの8.3より前において、これらの関数はいくつかの非文字列データ型の値を警告なしに受け付けたのは、それらデータ型を暗黙的にtext型に型変換していたことによります。この強制的な変換は、頻繁に予期しない動作の原因となったので削除されました。しかし、文字列連結演算子（||）は表 9.9で示されるように、少なくともひとつの入力が文字列型であれば、依然として非文字列入力を受け付けます。その他の場合、以前と同じ動作が必要なら、textへの明示的な変換を行ってください。

表9.9 SQL文字列関数と演算子

関数/演算子 説明 例
text || text → text 2つの文字列を結合します。 'Post' || 'greSQL' → PostgreSQL
text || anynonarray → text anynonarray || text → text 非文字列の入力をテキストに変換したのちに2つの文字列を結合します。（非文字列の入力は配列型であってはいけません。配列の||演算子との間で曖昧性が生じるからです。配列のテキストあるいは類似のものを結合する場合は明示的にtextにキャストしてください。） 'Value: ' || 42 → Value: 42
text IS [NOT] [form] NORMALIZED → boolean 文字列が指定したUnicode正規形の範囲かどうかをチェックします。オプションのformキーワードは正規形を指定します。NFC（デフォルトです）、NFD、NFKCあるいはNFKDです。この式はサーバエンコーディングがUTF8のときだけ使用できます。この式を用いた正規形のチェックは、しばしばすでに正規化されている可能性のある文字列を正規化するよりも高速であることに注意してください。

関数/演算子 説明 例
U&'\0061\0308bc' IS NFD NORMALIZED → t
bit_length (text) → integer 文字列中のビット数を返します（octet_lengthの8倍です。） bit_length('jose') → 32
char_length (text) → integer character_length (text) → integer 文字列中の文字数を返します。 char_length('jos é ') → 4
lower (text) → text データベースの照合順のルールに従い、文字列をすべて小文字に変換します。 lower('TOM') → tom
normalize (text [, form]) → text 文字列を指定したUnicode正規形に変換します。オプションのformキーワードは正規形を指定します。NFC (デフォルトです)、NFD、NFKCあるいはNFKDです。この式はサーバエンコーディングがUTF8のときだけ使用できます。 normalize(U&'\0061\0308bc', NFC) → U&'\00E4bc'
octet_length (text) → integer 文字列のバイト数を返します。 octet_length('jos é ') → 5（サーバーエンコーディングがUTF8の場合）
octet_length (character) → integer 文字列のバイト数を返します。このバージョンの関数は直接character型を受け付けるので、末尾の空白を削除しません。 octet_length('abc '::character(4)) → 4
overlay (string text PLACING newsubstring text FROM start integer [FOR count integer]) → text stringのstart文字目からcount文字をnewsubstringで置き換えます。countを省略するとnewsubstringの長さがデフォルトになります。 overlay('Txxxxas' placing 'hom' from 2 for 4) → Thomas
position (substring text IN string text) → integer string中のsubstringで指定する文字列開始位置を返します。0ならその文字列は存在しません。 position('om' in 'Thomas') → 3
substring (string text [FROM start integer] [FOR count integer]) → text startが指定されていればstart番目の文字で始まるstringの部分文字列を返します。countが指定されていればcount数の文字を取り出します。少なくともstartかcountのどちらかを指定してください。 substring('Thomas' from 2 for 3) → hom substring('Thomas' from 3) → omas substring('Thomas' for 2) → Th
substring (string text FROM pattern text) → text POSIX正規表現にマッチする部分文字列を返します。9.7.3を参照してください。 substring('Thomas' from '...$') → mas
substring (string text FROM pattern text FOR escape text) → text SQL正規表現にマッチする部分文字列を返します。9.7.2を参照してください。

関数/演算子 説明 例
substring('Thomas' from '%#"o_a#"_' for '#') → oma
trim([LEADING | TRAILING | BOTH] [characters text] FROM string text) → text characters(空白一文字がデフォルトです)に現れる文字のみを含む最長の文字列をstringの最初、最後、あるいは両方(BOTHがデフォルトです)から取り除きます。 trim(both 'xyz' from 'yxTomxx') → Tom
trim([LEADING | TRAILING | BOTH] [FROM] string text [, characters text]) → text これはtrim()の非標準構文です。 trim(both from 'yxTomxx', 'xyz') → Tom
upper(text) → text データベースの照合順のルールに従い、文字列をすべて大文字に変換します。 upper('tom') → TOM

この他、表 9.10に列挙する文字列操作関数が使えます。そのいくつかは、表 9.9で列挙した標準SQLの文字列関数を実装するため、内部的に使用されます。

表9.10 その他の文字列関数

関数 説明 例
ascii(text) → integer 引数の最初の文字の数値コードを返します。UTF8符号化方式ではその文字のUnicodeコードポイントを返します。その他のマルチバイト符号化方式の場合、引数はASCII文字でなくてはなりません。 ascii('x') → 120
btrim(string text [, characters text]) → text characters(空白一文字がデフォルトです)に現れる文字のみを含む最長の文字列をstringの最初と最後から取り除きます。 btrim('xyxtrimyyx', 'xyz') → trim
chr(integer) → text 与えられたコードの文字を返します。UTF8符号化方式では、引数はUnicodeコードポイントと見なされます。その他のマルチバイト符号化方式の場合、引数は指定のASCII文字でなくてはなりません。chr(0)は禁止されています。テキストデータ型はその文字を格納できないからです。 chr(65) → A
concat(val1 "any" [, val2 "any" [, ...]]) → text 引数をテキスト形式にしたものを結合します。NULL引数は無視されます。 concat('abcde', 2, NULL, 22) → abcde222
concat_ws(sep text, val1 "any" [, val2 "any" [, ...]]) → text 最初の引数以外をセパレータとともに結合します。最初の引数はセパレータ文字列として使われ、NULLにすべきではありません。それ以外のNULLの引数は無視されます。 concat_ws(',', 'abcde', 2, NULL, 22) → abcde,2,22
format(formatstr text [, formatarg "any" [, ...]]) → text

関数 説明 例
引数の書式をフォーマット文字列に従って整形します。9.4.1を参照してください。この関数はC言語関数のsprintfと似ています。 format('Hello %s, %1$s', 'World') → Hello World, World
initcap (text) → text それぞれの単語の第一文字を大文字に、残りは小文字に変換します。ここで単語とは、英数字以外の文字で区切られた、英数字からなる文字の並びのことです。 initcap('hi THOMAS') → Hi Thomas
left (string text, n integer) → text 文字列の先頭からn文字を返します。nが負数の場合、文字列の末尾から|n|文字を切り取った文字列を返します。 left('abcde', 2) → ab
length (text) → integer 文字列内の文字数を返します。 length('jose') → 4
lpad (string text, length integer [, fill text]) → text 文字fill(デフォルトは空白文字)を文字列の前に追加して、stringをlengthの長さにします。stringが既にlengthの長さを超えている場合は(右側が)切り捨てられます。 lpad('hi', 5, 'xy') → xyxhi
ltrim (string text [, characters text]) → text stringの最初から、characters(デフォルトは空白文字)で指定された文字だけを有する最長の文字列を削除します。 ltrim('zzzytest', 'xyz') → test
md5 (text) → text 引数のMD5ハッシュ計算し、16進数で結果を返します。 md5('abc') → 900150983cd24fb0d6963f7d28e17f72
parse_ident (qualified_identifier text [, strict_mode boolean DEFAULT true]) → text[] qualified_identifierを識別子の配列に分割し、個々の識別子に引用符があればそれを削除します。デフォルトでは、最後の識別子の後に続く余分な文字はエラーとされますが、2番目のパラメータがfalseの場合は、そのような余分な文字は無視されます。(この動作は、関数のようなオブジェクトに対して名前を解析するときに便利でしょう。) この関数は、長すぎる識別子を切り詰めないことに注意してください。切り詰めが必要なときは、その結果をname[]にキャストすることができます。 parse_ident('"SomeSchema".someTable') → {SomeSchema,sometable}
pg_client_encoding ( ) → name 現在のクライアントの符号化方式の名前を返します。 pg_client_encoding() → UTF8
quote_ident (text) → text 与えられた文字列を、SQL問い合わせ文字列で識別子として使用できるように、適切な引用符を付けて返します。引用符は、必要な場合(すなわち、文字列に識別子として使用できない文字が含まれる場合や、大文字変換される場合)にのみ追加されます。埋め込まれた引用符は、適切に二重化されます。例 42.1も参照してください。 quote_ident('Foo bar') → "Foo bar"
quote_literal (text) → text 与えられた文字列を、SQL問い合わせ文字列で文字リテラルとして使用できるように、適切な引用符を付けて返します。埋め込まれた単一引用符およびバックスラッシュは、適切に二重化されます。quote_literalはNULL入力に対してNULLを返す

関数

説明

例

ことに注意してください。引数がNULLとなる可能性がある場合、よりquote_nullableの方がしばしば適しています。例 42.1も参照してください。

quote_literal(E'O\'Reilly') → 'O''Reilly'

quote_literal (anyelement) → text

与えられた値をテキストに変換し、そしてリテラルとして引用符付けします。埋め込まれた単一引用符とバックスラッシュは適切に二重化されます。

quote_literal(42.5) → '42.5'

quote_nullable (text) → text

与えられた文字列を、SQL問い合わせ文字列で文字列リテラルとして使用できるように、適切な引用符を付けて返します。また、引数がNULLの場合、NULLを返します。埋め込まれた単一引用符およびバックスラッシュは適切に二重化されます。例 42.1も参照してください。

quote_nullable(NULL) → NULL

quote_nullable (anyelement) → text

与えられた値をテキストに変換し、そしてリテラルとして引用符付けします。引数がNULLの場合はNULLを返します。埋め込まれた単一引用符とバックスラッシュは適切に二重化されます。

quote_nullable(42.5) → '42.5'

regexp_match (string text, pattern text [, flags text]) → text[]

stringに対してPOSIX正規表現でマッチし、捕捉された最初の部分文字列を返します。より詳細は9.7.3を参照してください。

regexp_match('foobarbequebaz', '(bar)(beque)') → {bar,beque}

regexp_matches (string text, pattern text [, flags text]) → setof text[]

stringに対してPOSIX正規表現でマッチし、捕捉された部分文字列を返します。より詳細は9.7.3を参照してください。

regexp_matches('foobarbequebaz', 'ba.', 'g') →

```
{bar}
 {baz}
```

regexp_replace (string text, pattern text, replacement text [, flags text]) → text

POSIX正規表現に一致する部分文字列を置換します。より詳細は9.7.3を参照してください。

regexp_replace('Thomas', '.[mN]a.', 'M') → ThM

regexp_split_to_array (string text, pattern text [, flags text]) → text[]

POSIX正規表現を区切り文字に使ってstringを分割します。詳しくは9.7.3を参照ください。

regexp_split_to_array('hello world', '\s+') → {hello,world}

regexp_split_to_table (string text, pattern text [, flags text]) → setof text

POSIX正規表現を区切り文字に使ってstringを分割します。詳しくは9.7.3を参照ください。

regexp_split_to_table('hello world', '\s+') →

```
hello
 world
```

repeat (string text, number integer) → text

指定されたnumberの数だけstringを繰り返します。

関数 説明 例
repeat('Pg', 4) → PgPgPgPg
replace (string text, from text, to text) → text stringに出現する全てのfrom部分文字列をto部分文字列に置換します。 replace('abcdefabcdef', 'cd', 'XX') → abXXefabXXef
reverse (text) → text 文字列中の文字を逆順にします。 reverse('abcde') → edcba
right (string text, n integer)) → text 文字列の末尾からn文字を返します。nが負数の場合は、文字列の先頭から|n|文字だけ切り取った文字列を返します。 right('abcde', 2) → de
rpad (string text, length integer [, fill text])) → text 文字fill(デフォルトは空白文字)を文字列に追加して、stringをlengthの長さにします。stringが既にlengthの長さを超えている場合は切り捨てられます。 rpad('hi', 5, 'xy') → hixyx
rtrim (string text [, characters text]) → text stringの末尾から、characters(デフォルトは空白文字)で指定された文字のみを有する最長の文字列を削除します。 rtrim('testxxzx', 'xyz') → test
split_part (string text, delimiter text, n integer) → text stringをdelimiterで分割し、その結果からn番目のフィールド(1から始まるように数える)を返します。 split_part('abc~@~def~@~ghi', '~@~', 2) → def
strpos (string text, substring text) → integer string中の指定したsubstringの開始位置を返します。substringが存在しなければゼロを返します。(position(substring in string)と同じですが、引数の順序が逆であることに注意してください。) strpos('high', 'ig') → 2
substr (string text, start integer [, count integer]) → text stringのstart番目の文字から始まり、指定されている場合はcount文字だけ連続したが部分文字列を取り出します(substring(string from from for count)と同じです)。 substr('alphabet', 3) → phabet substr('alphabet', 3, 2) → ph
starts_with (string text, prefix text) → boolean stringがprefixで始まっていれば真を返します。 starts_with('alphabet', 'alph') → t
to_ascii (string text) → text to_ascii (string text, encoding name) → text to_ascii (string text, encoding integer) → text stringを他の名前あるいは数で指定される符号化方式から、ASCIIに変換します。encodingが省略されるとデータベースの符号化方式を指定したと見なします(これは実用的には唯一有用なケースです。) この変換は主にアクセントを削除するのが目的です。LATIN1、LATIN2、LATIN9、WIN1250符号化方式からの変換のみをサポートします。(他のより柔軟な解決方法としては、unaccentモジュールを参照してください。) to_ascii('Karél') → Karel

関数 説明 例
to_hex (integer) → text to_hex (bigint) → text 数を同等の16進数表現に変換します。 to_hex(2147483647) → 7fffffff
translate (string text, from text, to text) → text from集合内の文字と一致するstringにある全ての文字は、to集合内のそれに対応する文字に置き換えられます。もしfrom がtoより長い場合、fromで指定される余分な文字に一致するものは削除されます。 translate('12345', '143', 'ax') → a2x5

concat、concat_wsおよびformat関数はVariadicです。従って、キーワードVARIADICで標しをつけられた配列のように、値を連結またはフォーマットした形で受け渡すことが可能です（37.5.5を参照してください）。配列の要素は関数に対して分割された通常の引数のように扱われます。もしvariadic配列引数がNULLであれば、concatおよびconcat_wsはNULLを返しますが、formatはNULLを要素を持たない配列と扱います。

9.21内のstring_agg集約関数と、文字列とbytea型を変換するための表 9.13内の関数も参照してください。

9.4.1. format

関数formatは、C関数のsprintf同様の形式で、フォーマット文字列に従ってフォーマットされた出力を生成します。

```
format(formatstr text [, formatarg "any" [, ...] ])
```

formatstrは結果がどのようにフォーマットされるかを指定するフォーマット文字列です。フォーマット指示子が使用されている箇所を除き、フォーマット文字列のテキストは結果に直接コピーされます。フォーマット指示子は文字列中のプレースホルダとして振舞い、その後に引き続く関数引数がどのようにフォーマットされ、どのように結果に挿入されるかを定義します。それぞれのformatarg引数はそのデータ型に対する通常の出力規定に従ってテキストに変換され、その後フォーマット指示子に従って、結果文字列に挿入されます。

フォーマット指示子は%文字で始まり、以下の形式をとります。

```
%[position][flags][width]type
```

ここで要素フィールドとは以下になっています。

position (省略可能)

n$の形式の文字列で、nは出力する引数のインデックスです。インデックス1はformatstrの後の最初の引数です。positionが省略されると、一連の中の次の引数がデフォルトとして使用されます。

flags (省略可能)

フォーマット指示子の出力がどのようにフォーマットされるかを制御する追加の任意の要素です。現在、サポートされているflagはマイナス記号(-)のみで、フォーマット指示子の出力が左詰めになるようにします。これはwidthフィールドが同時に指定されていない場合は効果がありません。

width (省略可能)

フォーマット指示子の出力を表示する最小文字数を指定します。出力は、幅を満たすのに必要な空白が左または右(flagの-による)に埋め込まれます。幅が小さすぎても出力が切り詰められることはなく、単に無視されます。幅は次のいずれかでも指定できます。それらは、正の整数、幅としての次の関数引数として使用する星印 (*)、またはn番目の関数引数を幅として使用する*n$という形式の文字列です。

幅を関数引数から取得する場合、その引数はフォーマット指示子の値に使用される引数より先に消費されます。幅の引数が負の場合、フィールド長abs(width)の範囲内で結果は(あたかもflagで-が指定されたように)左詰めになります。

type (必須)

フォーマット指示子の出力を生成するのに使用されるフォーマット変換の型。以下の型がサポートされています。

- sは引数の値を単純文字列にフォーマットします。NULL値は空文字列として扱われます。
- Iは、引数をSQLの識別子として取り扱い、必要ならそれを二重引用符で括ります。NULL値はエラーです(quote_identと同等です)。
- Lは引数値をSQLリテラルとして引用符が付けられます。NULL値は引用符無しの文字列NULLとなります(quote_nullableと同等です)。

上記で説明したフォーマット指示子に加え、特別の並びの%%がリテラル%文字を出力するために使用することもできます。

基本的なフォーマット変換の例を幾つか下記に紹介します。

```
SELECT format('Hello %s', 'World');
Result: Hello World

SELECT format('Testing %s, %s, %s, %%', 'one', 'two', 'three');
Result: Testing one, two, three, %

SELECT format('INSERT INTO %I VALUES(%L)', 'Foo bar', E'O\'Reilly');
Result: INSERT INTO "Foo bar" VALUES('O''Reilly')

SELECT format('INSERT INTO %I VALUES(%L)', 'locations', 'C:\Program Files');
Result: INSERT INTO locations VALUES('C:\Program Files')
```

widthフィールドとflagの-を使用した例を以下に示します。

```
SELECT format('|%10s|', 'foo');
Result: |       foo|

SELECT format('|%-10s|', 'foo');
Result: |foo       |
```

```
SELECT format('|%*s|', 10, 'foo');
Result: |       foo|

SELECT format('|%*s|', -10, 'foo');
Result: |foo       |

SELECT format('|%-*s|', 10, 'foo');
Result: |foo       |

SELECT format('|%-*s|', -10, 'foo');
Result: |foo       |
```

以下の例はpositionフィールドの使い方を示しています。

```
SELECT format('Testing %3$s, %2$s, %1$s', 'one', 'two', 'three');
Result: Testing three, two, one

SELECT format('|%*2$s|', 'foo', 10, 'bar');
Result: |       bar|

SELECT format('|%1$*2$s|', 'foo', 10, 'bar');
Result: |       foo|
```

標準C関数sprintfとは違って、PostgreSQLのformat関数は、同一のフォーマット文字列の中でpositionフィールドがあるフォーマット指示子と、それがないフォーマット指示子の混在を許容します。positionフィールドが無いフォーマット指示子は常に最終の引数が消費された後に次の引数を使用します。さらに、format関数はフォーマット文字列で使用されるべき全ての関数引数を要求しません。例を示します。

```
SELECT format('Testing %3$s, %2$s, %s', 'one', 'two', 'three');
Result: Testing three, two, three
```

%I および %Lのフォーマット指示子は特に動的SQL命令を安全に構築する場合に便利です。例 42.1を参照してください。

9.5. バイナリ文字列関数と演算子

本節ではバイナリ文字列、すなわちbytea型の値を調べたり操作するための関数と演算子について説明します。これらの多くは前節で説明されているテキスト文字列関数と、目的と構文という意味で同じです。

SQLでは、引数の区切りにカンマではなくキーワードを使う文字列関数を、いくつか定義しています。詳細は表 9.11を参照してください。またPostgreSQLは、これらの関数に対して通常の関数呼び出し構文を使用するバージョンを提供します（表 9.12を参照してください）。

表9.11 SQLバイナリ文字列関数と演算子

関数/演算子 説明 例
bytea || bytea → bytea 2つのバイナリ文字列を結合します。 '\x123456'::bytea || '\x789a00bcde'::bytea → \x123456789a00bcde
bit_length (bytea) → integer 文字列中のビット数を返します(octet_lengthの8倍)。 bit_length('\x123456'::bytea) → 24
octet_length (bytea) → integer バイナリ文字列中のバイト数を返します。 octet_length('\x123456'::bytea) → 3
overlay (bytes bytea PLACING newsubstring bytea FROM start integer [FOR count integer]) → bytea bytesのstart番目のバイトからcountバイトをnewsubstringで置き換えます。countを省略するとnewsubstringの長さがデフォルトになります。 overlay('\x1234567890'::bytea placing '\002\003'::bytea from 2 for 3) → \x12020390
position (substring bytea IN bytes bytea) → integer bytes中のsubstringで指定する文字列開始位置を返します。その文字列が存在しなければ0を返します。 position('\x5678'::bytea in '\x1234567890'::bytea) → 3
substring (bytes bytea [FROM start integer] [FOR count integer]) → bytea startが指定されていればstart番目の文字で始まるbytesの部分文字列を返します。countが指定されていればcount数の文字で停止します。少なくともstartかcountのどちらかを指定してください。 substring('\x1234567890'::bytea from 3 for 2) → \x5678
trim ([BOTH] bytesremoved bytea FROM bytes bytea) → bytea bytesremovedに現れるバイトのみを含む最長の文字列をbytesの最初と最後から取り除きます。 trim('\x9012'::bytea from '\x1234567890'::bytea) → \x345678
trim ([BOTH] [FROM] bytes bytea, bytesremoved bytea) → bytea これはtrim()の非標準構文です。 trim(both from '\x1234567890'::bytea, '\x9012'::bytea) → \x345678

この他、表 9.12に列挙するバイナリ文字列操作関数が使えます。そのいくつかは、表 9.11で列挙した標準SQLの文字列関数を実装するため、内部的に使用されます。

表9.12 その他のバイナリ文字列関数

関数 説明 例
btrim (bytes bytea, bytesremoved bytea) → bytea bytesremovedに現れるバイトのみを含む最長のバイトをbytesの最初と最後から取り除きます。 btrim('\x1234567890'::bytea, '\x9012'::bytea) → \x345678
get_bit (bytes bytea, n bigint) → integer バイナリ文字列のn番目のビットを取り出します。

関数 説明 例
get_bit('\x1234567890'::bytea, 30) → 1
get_byte (bytes bytea, n integer) → integer バイナリ文字列のn番目のバイトを取り出します。 get_byte('\x1234567890'::bytea, 4) → 144
length (bytea) → integer バイナリ文字列のバイト数を返します。 length('\x1234567890'::bytea) → 5
length (bytes bytea, encoding name) → integer 与えられたencodingのテキストであると見なしてバイナリ文字列中の文字数を返します。 length('jose'::bytea, 'UTF8') → 4
md5 (bytea) → text バイナリ文字列のMD5ハッシュ計算し、16進数で結果を返します。 md5('Th\000omas'::bytea) → 8ab2d3c9689aaf18b4958c334c82d8b1
set_bit (bytes bytea, n bigint, newvalue integer) → bytea バイナリ文字列のn番目のビットをnewvalueにします。 set_bit('\x1234567890'::bytea, 30, 0) → \x1234563890
set_byte (bytes bytea, n integer, newvalue integer) → bytea バイナリ文字列のn番目のバイトをnewvalueにします。 set_byte('\x1234567890'::bytea, 4, 64) → \x1234567840
sha224 (bytea) → bytea バイナリ文字列のSHA-224 ハッシュを計算します。 sha224('abc'::bytea) → \x23097d223405d8228642a477bda255b32aadbce4bda0b3f7e36c9da7
sha256 (bytea) → bytea バイナリ文字列のSHA-256 ハッシュを計算します。 sha256('abc'::bytea) → \xba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad
sha384 (bytea) → bytea バイナリ文字列のSHA-384 ハッシュを計算します。 sha384('abc'::bytea) → \xcb00753f45a35e8bb5a03d699ac65007272c32ab0eded1631a8b605a43ff5bed8086072ba1e7cc2358baeca134c825a7
sha512 (bytea) → bytea バイナリ文字列のSHA-512 ハッシュを計算します。 sha512('abc'::bytea) → \xddaf35a193617abacc417349ae20413112e6fa4e89a97ea20a9eeee64b55d39a2192992a274fc1a836ba3c23a3feebbd454d4423643ce80e2a9ac94fa54ca49f
substr (bytes bytea, start integer [, count integer]) → bytea start番目の文字で始まるbytesの部分文字列を取り出します。countが指定されていればcount数バイトを取り出します。(substring(bytes from start for count)と同じです。) substr('\x1234567890'::bytea, 3, 2) → \x5678

get_byteとset_byteはバイナリ文字列の先頭バイトを0バイトとして数えます。get_bitとset_bitは各バイト内で右からビットを数えます。例えばビット0は先頭バイトの最下位ビットとなり、ビット15は第二バイトの最上位ビットとなります。

歴史的な理由により、md5は16進のエンコード値を返すのに対し、SHA-2関数はbyteaを返します。両者の間の変換を行うには、関数encodeとdecodeを使ってください。たとえば、16進のエンコードのテキスト表現を得るには、encode(sha256('abc'),'hex')、byteaを得るにはdecode(md5('abc'), 'hex')としてください。

異なる文字集合(文字符号化方式)間で文字列を変換する関数と、テキスト形式の任意のバイナリデータを表現する関数を表 9.13で示します。引数あるいは結果のtext型はデータベースのデフォルト文字符号化方式で表現され、bytea型の引数あるいは結果は別の引数で指定する文字符号化方式名で表現されます。

表9.13 テキスト/バイナリ文字列変換関数

関数 説明 例
convert (bytes bytea, src_encoding name, dest_encoding name) → bytea 文字符号化方式src_encodingのバイナリ文字列で表現したテキストを文字符号化方式dest_encodingのバイナリ文字列に変換します。(利用可能な変換は23.3.4を参照してください。) convert('text_in_utf8', 'UTF8', 'LATIN1') → \x746578745f696e5f75746638
convert_from (bytes bytea, src_encoding name) → text 文字符号化方式src_encodingのバイナリ文字列で表現したテキストをデータベース文字符号化方式のテキストに変換します。(利用可能な変換は23.3.4を参照してください。) convert_from('text_in_utf8', 'UTF8') → text_in_utf8
convert_to (string text, dest_encoding name) → bytea text文字列(データベース文字符号化方式で表現)を文字符号化方式dest_encodingのバイナリ文字列に変換します。(利用可能な変換は23.3.4を参照してください。) convert_to('some_text', 'UTF8') → \x736f6d655f74657874
encode (bytes bytea, format text) → text バイナリデータをテキスト表現形式に符号化します。サポートされているformat値は、base64、escape、hexです。 encode('123\000\001', 'base64') → MTIzAAE=
decode (string text, format text) → bytea テキスト表現からバイナリデータに復号します。format値はencodeと同じです。 decode('MTIzAAE=', 'base64') → \x3132330001

encodeとdecode関数は以下のテキスト形式をサポートしています。

base64

base64形式はRFC 2045 6.8節[1]のものです。RFCに従い、符号化された行は76文字に分割されます。しかし、MIME CRLF行端指示子ではなくて単に改行が行端として使われます。decode関数はキャリッジ・リターン、改行、空白、タブ文字を無視します。行端文字が不正な場合を含み、decodeに不正なbase64のデータが与えられるとエラーが生じます。

[1] https://tools.ietf.org/html/rfc2045#section-6.8

escape

escape形式はゼロバイトとハイビットがセットされたバイトを8進エスケープシーケンス(\nnn)に変換し、バックスラッシュを二重化します。他のバイト値は文字通りに表現されます。バックスラッシュの後が二番目のバックスラッシュあるいは3つの8進数のどちらでもなければ、decode関数はエラーを生じます。他のバイト値はそのまま受け付けます。

hex

hex形式はデータの各々の4ビットを、それぞれのバイトの上位桁を最初にして、0からfの16進数で表現します。encode関数はa-fの16進数を小文字で出力します。最小のデータ単位は8ビットなので、encodeが返す文字数は常に偶数です。decode関数はa-fの文字が大文字でも小文字でも受け付けます。奇数の文字数を含み、不正な16進データを与えられるとエラーが生じます。

9.21内の集約関数string_aggと34.4内のラージオブジェクト関数も参照してください。

9.6. ビット文字列関数と演算子

本節ではbit型とbit varying型の値であるビット文字列を調べたり操作するための関数と演算子について説明します。(この表ではbit型だけが言及されていますが、bit varying型も同じように使用できます。)ビット文字列は表 9.1で示す通常の比較演算子および表 9.14で言及している演算子もサポートします。

表9.14 ビット文字列演算子

演算子 説明 例
bit || bit → bit 結合 B'10001' || B'011' → 10001011
bit & bit → bit ビット単位のAND(入力は同じ長さでなければなりません) B'10001' & B'01101' → 00001
bit | bit → bit ビット単位のOR(入力は同じ長さでなければなりません) B'10001' | B'01101' → 11101

bit # bit → bit
ビット単位の排他的論理和(入力は同じ長さでなければなりません)
B'10001' # B'01101' → 11100

~ bit → bit ビット単位の否定 ~ B'10001' → 01110
bit << integer → bit ビット単位の左シフト(文字列長は保存されます) B'10001' << 3 → 01000
bit >> integer → bit

演算子 説明 例	
ビット単位の右シフト（文字列長は保存されます） B'10001' >> 2 → 00100	

バイナリ文字列で利用可能な関数のいくつかは、表 9.15で示すようにビット文字列でも利用可能です。

表9.15 ビット文字列関数

関数 説明 例
bit_length (bit) → integer ビット文字列中のビット数を返します。 bit_length(B'10111') → 5
length (bit) → integer ビット文字列中のビット数を返します。 length(B'10111') → 5
octet_length (bit) → integer ビット文字列中のバイト数を返します。 octet_length(B'1011111011') → 2
overlay (bits bit PLACING newsubstring bit FROM start integer [FOR count integer]) → bit bitsのstart番目のビットからcountビットをnewsubstringで置き換えます。countを省略するとnewsubstringの長さがデフォルトになります。 overlay(B'01010101010101010' placing B'11111' from 2 for 3) → 0111110101010101010
position (substring bit IN bits bit) → integer bits中のsubstringで指定する文字列開始位置を返します。その文字列が存在しなければ0を返します。 position(B'010' in B'000001101011') → 8
substring (bits bit [FROM start integer] [FOR count integer]) → bit start番目の文字で始まるbitsの部分文字列を取り出します。countが指定されていればcount数ビットを取り出します。少なくともstartかcountのどちらかを指定してください。 substring(B'110010111111' from 3 for 2) → 00
get_bit (bits bit, n integer) → integer ビット文字列のn番目のビットを取り出します。文字列の最初(一番左)のビットを0として数えます。 get_bit(B'101010101010101010', 6) → 1
set_bit (bits bit, n integer, newvalue integer) → bit ビット文字列のn番目のビットをnewvalueにします。文字列の最初(一番左)のビットを0として数えます。 set_bit(B'101010101010101010', 6, 0) → 101010001010101010

さらに、bit型から整数値にキャストすることも整数からbit型にキャストすることも可能です。整数からbit(n)にキャストすると最右端のnビットがコピーされます。その整数より文字列幅が広いビットにキャストすると左のビットが符号拡張されます。以下に例を示します。

```
44::bit(10)                     0000101100
```

```
44::bit(3)                  100
cast(-44 as bit(12))        111111010100
'1110'::bit(4)::integer     14
```

単に「bit」にキャストすることはbit(1)にキャストすることを意味することに注意してください。つまり、単に整数の最下位ビットのみが渡されることになります。

9.7. パターンマッチ

PostgreSQLには、パターンマッチを行うに際して3つの異なった手法があります。伝統的なSQLのLIKE演算子、これより新しいSIMILAR TO演算子（SQL:1999で追加されました）、およびPOSIX様式の正規表現です。基本の「この文字列はこのパターンに一致するか?」を別としても、一致した部分文字列を取り出したり置換したり、そして一致部分で文字列を分割する関数が用意されています。

> **ヒント**
>
> 上記の手法では検索できないようなパターンマッチが必要な場合は、PerlもしくはTclでユーザ定義関数を作成することを検討してください。

> **注意**
>
> ほとんどの正規表現検索はとても速く実行されますが、正規表現は処理するのに任意の時間とメモリを使う可能性があります。悪意のあるソースから正規表現検索パターンを受け取ることに用心してください。そうしなければならないのであれば、文のタイムアウトを強制するのが賢明です。
>
> SIMILAR TOがPOSIX書式の正規表現と同じ多くの機能を提供するので、SIMILAR TOパターンを使う検索は同様のセキュリティ問題を抱えています。
>
> LIKE検索は、他の2つの方法よりずっと単純ですので、悪意があるかもしれないパターンのソースで使うのにはより安全です。

この3種類のパターンマッチング演算子はどれも非決定的照合順序をサポートしていません。必要なら、この制限事項に対応するために別の照合順序を式に適用してください。

9.7.1. LIKE

```
string LIKE pattern [ESCAPE escape-character]
string NOT LIKE pattern [ESCAPE escape-character]
```

LIKE式は供給されたpatternにstringが一致すれば真を返します。（想像される通り、NOT LIKE式はLIKE式が真を返す場合には偽を返し、その逆もまた同じです。同等の式としてNOT (string LIKE pattern)とも表現できます。）

patternがパーセント記号もしくはアンダースコアを含んでいない場合patternは自身の文字列そのものです。この場合LIKE式は等号演算子のように振舞います。patternの中にあるアンダースコア(_)は任意の一文字との一致を意味し、パーセント記号(%)は0文字以上の並びとの一致を意味します。

例:

```
'abc' LIKE 'abc'    true
'abc' LIKE 'a%'     true
'abc' LIKE '_b_'    true
'abc' LIKE 'c'      false
```

LIKEによるパターン一致は常に文字列全体に対して行われます。従って、文字列内の任意位置における並びと一致させたい場合には、パーセント記号を先頭と末尾に付ける必要があります。

他の文字の一致に使用するのではなく、アンダースコアやパーセント記号そのものを一致させたい場合には、patternの中のそれぞれのアンダースコアとパーセント記号の前にエスケープ文字を付けなければなりません。デフォルトのエスケープ文字はバックスラッシュですが、ESCAPE句で他の文字を指定することができます。エスケープ文字そのものを一致させるにはエスケープ文字を2つ書きます。

> **注記**
>
> standard_conforming_stringsパラメータをoffにしている場合、リテラル文字列定数に記述するバックスラッシュを二重にする必要があります。詳細は4.1.2.1を参照してください。

同時にESCAPE ''と記述することでエスケープ文字を選択しないことも可能です。これにより、事実上エスケープ機構が働かなくなります。つまり、パターン内のアンダースコアおよびパーセント記号の特別な意味を解除することはできなくなります。

SQL標準によれば、ESCAPEを省略することは(デフォルトがバックスラッシュとなるのではなく)エスケープ文字が存在しないことを意味します。長さゼロのESCAPEは使用できません。ですからこの点でPostgreSQLは少し非標準な振る舞いをします。

現在のロケールに従って大文字小文字を区別しない一致を行うのであれば、LIKEの代わりにILIKEキーワードを使うことができます。これは標準SQLではなく、PostgreSQLの拡張です。

~~演算子はLIKE式と等価で、~~*はILIKEに対応します。またNOT LIKEおよびNOT ILIKEを表す!~~および!~~*演算子があります。これら全ての演算子はPostgreSQL固有のものです。パーサは実際にはLIKEなどをこれらの演算子に変換するため、こうした演算子名はEXPLAINの出力などで見ることができます。

LIKE、ILIKE、NOT LIKE、NOT ILIKE句は一般にPostgreSQLの構文上は演算子として扱われます。たとえば、式 演算子 ANY(副問い合わせ)構文で使用できます。しかし、ESCAPE句はこれには含むことはできません。状況によっては背後の演算子名を代わりに使わなければならない場合もあります。

単に文字列の先頭からの開始が必要なだけのケースであれば、接頭辞演算子^@とそれに対応するstarts_with関数もあります。

9.7.2. SIMILAR TO正規表現

```
string SIMILAR TO pattern [ESCAPE escape-character]
string NOT SIMILAR TO pattern [ESCAPE escape-character]
```

SIMILAR TO演算子は、そのパターンが与えられた文字列に一致するかどうかにより、真もしくは偽を返します。これは、標準SQLの正規表現定義を使用してパターンを解釈するという点以外は、LIKEに類似しています。SQLの正規表現は、LIKE表記と一般的な(POSIX)正規表現の表記とを混ぜ合わせたようなものになっています。

LIKEと同様、SIMILAR TO演算子は、そのパターンが文字列全体に一致した場合のみ真を返します。これは、パターンが文字列の一部分であっても一致する、一般的な正規表現の動作とは異なっています。また、LIKEと同様、SIMILAR TOでは、%および_を、それぞれ任意の文字列および任意の単一文字を意味するワイルドカード文字として使用します(これらは、POSIX正規表現での.*および.に相当します)。

LIKEから取り入れた上記の機能に加え、SIMILAR TOでは、以下のようにPOSIX正規表現から取り入れたパターンマッチメタ文字もサポートしています。

- |は、二者択一(2つの選択肢のうちいずれか)を意味します。
- *は、直前の項目の0回以上の繰り返しを意味します。
- +は、直前の項目の1回以上の繰り返しを意味します。
- ?は、直前の項目の0回もしくは1回の繰り返しを意味します。
- {m}は、直前の項目の正確なm回の繰り返しを意味します。
- {m,}は、直前の項目のm回以上の繰り返しを意味します。
- {m,n}は、直前の項目のm回以上かつn回以下の繰り返しを意味します。
- 丸括弧()は、項目を1つの論理項目にグループ化することができます。
- 大括弧式[...]は、POSIX正規表現と同様に文字クラスを指定します。

SIMILAR TOではピリオド(.)はメタ文字ではないことに注意してください。

LIKEと同様、バックスラッシュは全てのメタ文字の特殊な意味を無効にします。異なるエスケープ文字をESCAPEで指定することもできますし、ESCAPE ''と書くことにより、エスケープ機能を無効にすることもできます。

SQL標準によれば、ESCAPEは(デフォルトがバックスラッシュとなるのではなく)エスケープ文字が存在しないことを意味します。長さゼロのESCAPEは使用できません。ですからこの点でPostgreSQLは少し非標準な振る舞いをします。

他の非標準の拡張としては、エスケープ文字に続く文字あるいは数字を用いてPOSIX正規表現で定義されたエスケープシーケンスへのアクセスを提供するというのがあります。以下の表 9.20、表 9.21、表 9.22を参照してください。

例を示します。

```
'abc' SIMILAR TO 'abc'          true
'abc' SIMILAR TO 'a'            false
```

```
'abc' SIMILAR TO '%(b|d)%'      true
'abc' SIMILAR TO '(b|c)%'       false
'-abc-' SIMILAR TO '%\mabc\M%'  true
'xabcy' SIMILAR TO '%\mabc\M%'  false
```

3つのパラメータを持つsubstring関数を使用して、SQL正規表現パターンに一致する部分文字列を取り出すことができます。SQL99の構文にしたがって、この関数は次のように書くことができます。

```
substring(string from pattern for escape-character)
```

あるいは単なる3引数関数として次のように書くこともできます。

```
substring(string, pattern, escape-character)
```

SIMILAR TOと同様、指定したパターンがデータ文字列全体に一致する必要があります。一致しない場合、関数は失敗し、NULLを返します。マッチするデータのうちの対象とする部分文字列に対応するパターンの部分を示すために、エスケープ文字の後に二重引用符(")を繋げたものを2つパターンに含める必要があります。マッチが成功すると、これらの印で括られたパターンの一部に一致するテキストが返されます。

エスケープ文字と二重引用符による区切りは実際にはsubstringのパターン引数を3つの独立した正規表現に分割します。たとえば3つのセクションのどこかに置いた垂直線(|)はそのセクションにしか影響を及ぼしません。また、どのパターンにデータ文字列がマッチするかについて曖昧さがある場合は、最初と3番目の正規表現は、可能な最大のテキストではなく、最小のテキストにマッチするものとして定義されます。(POSIX用語では、最初と3番目の正規表現は非貪欲(non-greedy)に強制されます。)

SQL標準への拡張として、PostgreSQLは、二重引用符による区切りが一個だけ存在することを許容し、その場合は3番目の正規表現が空として扱われます。あるいは、二重引用符による区切りがないことも許容し、その場合は最初と3番目の正規表現は空として扱われます。

例:#"を使用して返される文字列を区切ります。

```
substring('foobar' from '%#"o_b#"%' for '#')   oob
substring('foobar' from '#"o_b#"%' for '#')    NULL
```

9.7.3. POSIX正規表現

表 9.16に、POSIX正規表現を使ったパターンマッチに使用可能な演算子を列挙します。

表9.16 正規表現マッチ演算子

演算子 説明 例
text ~ text → boolean 文字列が正規表現にマッチ、大文字小文字の区別あり 'thomas' ~ '.*thom.*' → t
text ~* text → boolean

演算子 説明 例	
	文字列が正規表現にマッチ、大文字小文字の区別なし 'thomas' ~* '.*Thom.*' → t
text !~ text	text → boolean 文字列が正規表現にマッチしない、大文字小文字の区別あり 'thomas' !~ '.*thomas.*' → f
text !~* text	text → boolean 文字列が正規表現にマッチしない、大文字小文字の区別なし 'thomas' !~* '.*vadim.*' → t

POSIX正規表現は、パターンマッチという意味合いでは、LIKEおよびSIMILAR TO演算子よりもさらに強力です。egrep、sed、あるいはawkのような多くのUnixツールはここで解説しているのと類似したパターンマッチ言語を使用しています。

正規表現とは文字列の集合(正規集合)の簡略された定義である文字が連なっているものです。ある文字列が正規表現で記述された正規集合の要素になっていれば、その文字列は正規表現にマッチしていると呼ばれます。LIKEと同様、正規表現言語で特殊文字とされているもの以外、パターン文字は文字列と完全にマッチされます。とは言っても、正規表現はLIKE関数が使用するのとは異なる特殊文字を使用します。LIKE関数のパターンと違って正規表現は、明示的に正規表現が文字列の最初または最後からと位置指定されていない限り文字列内のどの位置でもマッチを行えます。

例:

```
'abc' ~ 'abc'    true
'abc' ~ '^a'     true
'abc' ~ '(b|d)'  true
'abc' ~ '^(b|c)' false
```

POSIXパターン言語について以下により詳しく説明します。

2つのパラメータを持つsubstring関数、substring(string from pattern)を使用して、POSIX正規表現パターンにマッチする部分文字列を取り出すことができます。この関数は、マッチするものがない場合にはNULLを返し、ある場合はパターンにマッチしたテキストの一部を返します。しかし、丸括弧を持つパターンの場合、最初の丸括弧内部分正規表現(左丸括弧が最初に来るもの)にマッチするテキストの一部が返されます。この例外を起こさずにパターン中に丸括弧を使用したいのであれば、常に正規表現全体を丸括弧で囲むことができます。パターン内の抽出対象の部分文字列より前に丸括弧が必要な場合、後述の捕捉されない丸括弧を参照してください。

例:

```
substring('foobar' from 'o.b')     oob
substring('foobar' from 'o(.)b')   o
```

regexp_replace関数は、POSIX正規表現パターンにマッチする部分文字列を新規テキストと置換します。構文は、regexp_replace(source, pattern, replacement [, flags])です。patternにマッチしない場合は、

source文字列がそのまま返されます。マッチすると、マッチ部分文字列をreplacement文字列で置換したsource文字列が返されます。replacement文字列に\n(nは1から9までの数字)を入れて、パターン内のn番目の丸括弧つき部分表現にマッチする元の部分文字列を挿入することができます。また、\&を入れて、パターン全体とマッチする部分文字列を挿入することもできます。置換テキスト内にバックスラッシュそのものを挿入する必要がある時は\\と記述します。flagsパラメータは、関数の動作を変更するゼロもしくはそれ以上の1文字フラグを含むオプションのテキスト文字列です。フラグiは大文字小文字を区別しないマッチを指定する一方、フラグgは、最初にマッチしたもののみではなく、それぞれマッチした部分文字列の置換を指定します。有効なフラグは(gを除く)表 9.24に記述されています。

例:

```
regexp_replace('foobarbaz', 'b..', 'X')
                                   fooXbaz
regexp_replace('foobarbaz', 'b..', 'X', 'g')
                                   fooXX
regexp_replace('foobarbaz', 'b(..)', 'X\1Y', 'g')
                                   fooXarYXazY
```

regexp_match関数はPOSIX正規表現パターンを文字列にマッチさせた結果、捕捉された最初の部分文字列のテキスト配列を返します。regexp_match(string, pattern [, flags])の構文になります。マッチするものがなければ、結果はNULLとなります。マッチする部分があり、かつpatternが丸括弧で括られた部分文字列を含まない場合、結果はパターン全体にマッチする部分文字列を含む単一要素のテキスト配列となります。マッチする部分があり、かつpatternが丸括弧で括られた部分文字列を含む場合、結果はテキスト配列で、そのn番目の要素はpatternのn番目に丸括弧で括られた部分文字列にマッチする部分文字列となります(「捕捉されない」丸括弧は数えません。詳細は以下を参照してください)。flagsパラメータは、関数の動作を変更するゼロもしくは複数の単一文字フラグを含むオプションのテキスト文字列です。有効なフラグは表 9.24に記載されています。

例を示します。

```
SELECT regexp_match('foobarbequebaz', 'bar.*que');
 regexp_match
--------------
 {barbeque}
(1 row)

SELECT regexp_match('foobarbequebaz', '(bar)(beque)');
 regexp_match
--------------
 {bar,beque}
(1 row)
```

マッチするときはマッチする部分文字列全体、マッチしないときはNULLを返したいというよくあるケースは、以下のように書くことができます。

```
SELECT (regexp_match('foobarbequebaz', 'bar.*que'))[1];
 regexp_match
```

```
--------------
 barbeque
(1 row)
```

regexp_matches関数はPOSIX正規表現パターンを文字列にマッチさせた結果、捕捉された部分文字列のテキスト配列の集合を返します。構文はregexp_matchと同じです。この関数は、マッチするものがないときは行を返しませんが、マッチするものがあり、gフラグが指定されていないときは1行だけ、マッチするものがN個あり、gフラグが指定されているときはN行を返します。返される各行は上でregexp_matchについて説明したのと全く同じで、マッチする部分文字列全体、または丸括弧で括られた部分文字列にマッチする部分文字列を含むテキスト配列です。regexp_matchesは表 9.24に示すすべてのフラグに加え、最初のマッチだけでなくすべてのマッチを返すgを受け付けます。

例を示します。

```
SELECT regexp_matches('foo', 'not there');
 regexp_matches
----------------
(0 rows)

SELECT regexp_matches('foobarbequebazilbarfbonk', '(b[^b]+)(b[^b]+)', 'g');
 regexp_matches
----------------
 {bar,beque}
 {bazil,barf}
(2 rows)
```

ヒント

最初にマッチするものだけが必要なときはregexp_match()を使う方がより簡単で効率的ですから、regexp_matches()はほとんどの場合gフラグを指定して使われるでしょう。しかし、regexp_match()はPostgreSQLのバージョン10以上でのみ利用できます。古いバージョンを使う時によくある手法は、以下の例のように、副SELECTの中にregexp_matches()の呼び出しを入れることです。

```
SELECT col1, (SELECT regexp_matches(col2, '(bar)(beque)')) FROM tab;
```

これはregexp_match()と同じく、マッチするものがあればテキスト配列を生成し、マッチしなければNULLとなります。副SELECTを使わなければ、マッチするものがないテーブル行については問い合わせの出力が生成されず、多くの場合に期待される動作と異なります。

regexp_split_to_table関数はPOSIX正規表現パターンを区切り文字として使用し、文字列を分割します。regexp_split_to_table(string, pattern [, flags])の構文になります。patternにマッチしない場合、関数はstringを返します。少なくともひとつのマッチがあれば、それぞれのマッチに対して関数は最後のマッチの終わり(あるいは文字列の始め)から最初のマッチまでのテキストを返します。もはやマッチしなくなると最後のマッチの終わりから文字列の最後までテキストを返します。flagsパラメータ

は、関数の動作を変更するゼロもしくは複数の単一文字フラグを含むオプションのテキスト文字列です。regexp_split_to_tableは表 9.24で記載されているフラグをサポートします。

regexp_split_to_array関数は、regexp_split_to_arrayがその結果をtext配列で返すことを除いて、regexp_split_to_tableと同じ動作をします。regexp_split_to_array(string, pattern [, flags])の構文になります。パラメータはregexp_split_to_tableのものと同じです。

例:

```
SELECT foo FROM regexp_split_to_table('the quick brown fox jumps over the lazy dog', '\s+') AS
 foo;
  foo
-------
 the
 quick
 brown
 fox
 jumps
 over
 the
 lazy
 dog
(9 rows)

SELECT regexp_split_to_array('the quick brown fox jumps over the lazy dog', '\s+');
              regexp_split_to_array
-----------------------------------------------
 {the,quick,brown,fox,jumps,over,the,lazy,dog}
(1 row)

SELECT foo FROM regexp_split_to_table('the quick brown fox', '\s*') AS foo;
 foo
-----
 t
 h
 e
 q
 u
 i
 c
 k
 b
 r
 o
 w
 n
 f
```

```
 o
 x
(16 rows)
```

最後の例が明らかにしているように、regexp分割関数は文字列の最初あるいは終わり、もしくは前のマッチの直後に発生する長さを持たないマッチを無視します。regexp_matchおよびregexp_matchesで実装されたregexpマッチの厳格な定義にこれは相容れませんが、実務上は最も使い勝手の良い動作です。Perlのような他のソフトウェアシステムも似たような定義を使用します。

9.7.3.1. 正規表現の詳細

PostgreSQLの正規表現はHenry Spencerにより書かれたソフトウェアパッケージを使用して実装されています。以下に説明する正規表現の多くの部分は同氏のマニュアルから一字一句複製したものです。

POSIX 1003.2の定義によると、正規表現(RE)には2つの形式があるとされます。*拡張*REもしくはERE(大まかにいってegrepに代表されるもの)、および*基本*REもしくはBRE(大まかにいってedに代表されるもの)です。PostgreSQLは両方の形式をサポートし、さらに、POSIX標準にはないけれどもPerlやTclなどのプログラミング言語で利用できることから広く使用されるようになった、いくつかの拡張もサポートしています。本書では、非POSIX拡張を使用したREを*高度な*REもしくはAREと呼びます。AREはEREの正確な上位セットですが、BREとは複数の記法上の非互換な点があります(さらに非常に多くの制限が課されています)。まず、AREとERE形式について説明し、そして、AREにのみ適用される機能の注意を、さらにBREとの違いについて説明します。

> **注記**
>
> PostgreSQLは常に、まず正規表現はARE規則に従うと推測します。しかし、REパターンの前に、9.7.3.4に記載されているような*埋め込みオプション*を追加することにより、より限られたERE、あるいはBRE規則を選択することができます。これは、POSIX1003.2の規則を正確に期待しているアプリケーションとの互換性に関して有用です。

正規表現は|で区切られた、1つまたは複数の*ブランチ*として定義されます。ブランチのいずれか1つにマッチすればマッチしたことになります。

ブランチはゼロ個以上の*量化アトム*もしくは*制約*の連結です。最初のものにマッチに、次に第2番目のものにマッチを、というふうにマッチします。なお、空のブランチは空文字列にマッチします。

量化アトムとは、単一の*量指定子*が後ろに付く*アトム*のことです。量指定子がないと、アトムにマッチするものがマッチしたことになります。量指定子がある場合、アトムとのマッチが何回あるかでマッチしたことになります。*アトム*は、表 9.17に示したもののいずれかを取ることができます。表 9.18に設定可能な量指定子とその意味を示します。

*制約*は空文字に、特定の条件に合う場合のみにマッチします。アトムを使用できるところには制約を使用することができます。ただしその後に量指定子を付けることはできません。単純な制約を表 9.19に示します。後で他のいくつかの制約を説明します。

表9.17 正規表現のアトム

アトム	説明
(re)	(ここでreは任意の正規表現で、)reとのマッチに適合するもです。マッチは可能である報告用と意味づけられます。
(?:re)	上と同じ。ただし、マッチは報告用と意味づけられません。(「捕捉されない」括弧の集合)(AREのみ)
.	任意の1文字にマッチします。
[chars]	ブラケット式。charsのいずれか1つにマッチします (詳細は9.7.3.2を参照してください)。
\k	(ここでkは英数字以外です。)普通の文字として指定した文字にマッチします。例えば、\\はバックスラッシュ文字です。
\c	ここでcは英数字です (おそらく他の文字が後に続きます)。エスケープです。9.7.3.3を参照してください (AREのみ、EREとBREではこれはcにマッチします)。
{	直後に数字以外がある場合、左中括弧{にマッチします。直後に数字が続く場合、bound(後述)の始まりです。
x	ここでxは他に意味を持たない1文字です。xにマッチします。

REはバックスラッシュ\を終端とすることはできません。

> **注記**
>
> もしstandard_conforming_stringsパラメータをoffにしていた場合、リテラル文字列定数に記述するバックスラッシュは2倍必要となります。詳細は4.1.2.1を参照してください。

表9.18 正規表現量指定子

量指定子	マッチ
*	アトムの0個以上複数の並びにマッチ
+	アトムの1個以上複数の並びにマッチ
?	アトムの0個または1個の並びにマッチ
{m}	アトムの正確にm個の並びにマッチ
{m,}	アトムのm個以上の並びにマッチ
{m,n}	アトムのm個以上n以下の並びにマッチ。mはnを超えることはできません。
*?	*の最短マッチを行うバージョン
+?	+の最短マッチを行うバージョン
??	?の最短マッチを行うバージョン
{m}?	{m}の最短マッチを行うバージョン
{m,}?	{m,}の最短マッチを行うバージョン

量指定子	マッチ
{m,n}?	{m,n}の最短マッチを行うバージョン

{...}を使用する形式はバウンドとして知られています。バウンド内のmとnという数は符号なし10進整数であり、0以上255以下の値を取ることができます。

最短マッチを行う量指定子(AREのみで使用可能)は、対応する通常の(欲張りの)ものと同じものにマッチしますが、最大のマッチではなく最小のマッチを取ります。詳細は9.7.3.5を参照してください。

> **注記**
>
> 量指定子の直後に量指定子を続けることはできません。例えば**は無効です。量指定子から式や副式を始めることはできず、また、^や|の直後に付けることもできません。

表9.19 正規表現制約

制約	説明
^	文字列の先頭にマッチ
$	文字列の末尾にマッチ
(?=re)	先行肯定検索は、reにマッチする部分文字列が始まる任意の場所にマッチします(AREのみ)。
(?!re)	先行否定検索は、reにマッチしない部分文字列が始まる任意の場所にマッチします(AREのみ)。
(?<=re)	後方肯定検索はreにマッチする部分文字列が終わる任意の場所にマッチします(AREのみ)。
(?<!re)	後方否定検索reにマッチしない部分文字列が終わる任意の場所にマッチします(AREのみ)。

先行検索制約および後方検索制約には後方参照(9.7.3.3を参照)を含めることはできません。また、その中の括弧は全て取り込むものではないとみなされます。

9.7.3.2. ブラケット式

ブラケット式とは、[]内の文字のリストです。通常これはそのリスト内の任意の1文字にマッチします(しかし、以降を参照してください)。リストが^から始まる場合、そのリストの残りにはない任意の1文字にマッチします。リスト内の2文字が-で区切られていた場合、これは2つ(を含む)の間にある文字範囲全体を表す省略形となります。例えば、ASCIIにおける[0-9]は全ての数字にマッチします。例えばa-c-eといった、終端を共有する2つの範囲は不正です。範囲は並びの照合順に非常に依存しています。ですので、移植予定のプログラムではこれに依存してはなりません。

このリストに]そのものを含めるには、それを先頭文字(もしそれが使用されれば^の後)にしてください。-そのものを含めるには、それを先頭もしくは末尾の文字とするか、範囲の2番目の終端としてください。-を範囲の最初の終端で使用するには、[.と.]でそれを囲み、照合要素(後述)にしてください。これら文字と、[(次

段落を参照）のなんらかの組み合わせ、およびエスケープ（AREのみ）を例外として、他の全ての特殊文字はブラケット式内では特殊な意味を持ちません。特に、\はEREとBRE規則に従う場合は特別でなくなります。しかし、AREでは（エスケープの始まりとして）特別な意味を持ちます。

ブラケット式内に、照合要素（文字、単一文字であるかのように照合する複数文字の並び、もしくはそれぞれの照合並びの名前）が[.と.]の間にあると、その照合要素の文字の並びを意味します。この並びはブラケット式のリストの一要素として取り扱われます。このことにより、ブラケット式は要素を照合する複数文字を含むブラケット式を1文字以上にマッチさせることができます。例えば、照合並びがch照合要素を含む場合、正規表現[[.ch.]]*cはchchccという文字の最初の5文字にマッチします。

> **注記**
>
> 今のところ、PostgreSQLは複数文字照合要素をサポートしません。この情報は将来の振舞いの可能性を説明したものです。

ブラケット式内の[=と=]の間に照合要素は*同値クラス*です。全ての照合要素の文字の並びが自身を含むものと等価であることを示します。（他に等価な照合要素がある場合、[.と.]で囲まれたかのように扱われます。）例えば、[[=o=]]、[[=^=]]および[o^]が全て同意語であれば、oと^は同値クラスのメンバです。同値クラスは範囲の終端にはなりません。

ブラケット式内では、[:と:]の間にある文字クラスの名称は、そのクラスに属する全ての文字のリストを意味します。文字クラスは範囲の終端位置としては使用できません。POSIX標準は以下の文字クラス名を定義しています。alnum（文字と数字）、alpha（文字）、blank（空白とタブ）、cntrl（制御文字）、digit（数字）、graph（空白以外の印字可能文字）、lower（小文字）、print（空白を含む印字可能文字）、punct（句読点）、space（空白）、upper（大文字）、xdigit（16進数）です。これらの標準文字クラスの振る舞いは7-bit ASCII集合の範囲であれば一般にどのプラットフォームでも同じです。与えられた非ASCII文字がこれらの文字クラスに属すると考えられるかどうかは、正規表現関数または演算子（23.2参照）で使用される*照合順*、あるいはデフォルトとしてはデータベースのLC_CTYPEロケール（23.1)の設定によります。非ASCII文字の分類は、たとえ似たような名前のロケールであってもプラットフォームによって異なることがありえます。（ただしCロケールでは、すべての非ASCII文字はこれらのクラスのどれにも所属しないものとされます。）これらの標準クラスに加え、PostgreSQLでは7-bit ASCII集合を正確に含むascii文字クラスが定義されています。

ブラケット式には2つの特殊な場合があります。[[:<:]]と[[:>:]]というブラケット式は、先頭と終端の単語がそれぞれ空文字であることにマッチする制約です。単語は、単語文字が前後に付かない単語文字の並びとして定義されます。単語文字とは（上述のPOSIX文字クラスで定義されているように）1つのalnum文字またはアンダースコアです。これは、POSIX 1003.2との互換性はありますが、そこでは定義されていない式です。ですので、他システムへ移植予定のソフトウェアでの使用には注意が必要です。通常後述の制約エスケープの方がよく使われます。これはもはや標準ではありませんが、入力しやすいものです。

9.7.3.3. 正規表現エスケープ

*エスケープ*とは、\から始まり英数字がその後に続く特殊な並びです。エスケープには、文字エントリ、クラス省略、制約エスケープ、後方参照といった様々な変種があります。\の後に英数字が続くけれども、有効なエスケープを構成しない並びはAREでは不正です。EREにはエスケープはありません。ブラケット式の外側では、\の後に英数字が続く並びは単に普通の文字としてその文字を意味します。ブラケット式の内側では、\は普通の文字です。（後者はEREとARE間の非互換性の1つです。）

文字エントリエスケープは非印字文字やRE内でその他の不便な文字の指定を簡略化するために存在します。これらを表 9.20に示します。

クラス省略エスケープは、あるよく使用される文字クラスの省略形を提供します。これらを表 9.21に示します。

制約エスケープは、指定した条件に合う場合に空文字にマッチする制約をエスケープとして表したものです。これらを表 9.22に示します。

後方参照(\n)は、直前に括弧で囲まれた副式によってマッチされた、n番目の同一文字列にマッチします(表 9.23を参照してください)。例えば、([bc])\1はbbもしくはccにマッチしますが、bcやcbにはマッチしません。REでは副式全体は後方参照の前になければなりません。副式は開括弧の順番で番号付けされます。取り込まない括弧は副式を定義しません。

表9.20 正規表現文字エントリエスケープ

エスケープ	説明
\a	C言語と同じ警報(ベル)文字
\b	C言語と同じバックスペース
\B	バックスラッシュの必要な二重化回数を減らすためのバックスラッシュ(\)の同義語
\cX	(ここでXは任意の文字で)その下位5ビットがXと同一、その他のビットが0となる文字
\e	照合順名がESCとなる文字、それに失敗したら、033という8進数値を持つ文字。
\f	C言語と同じ改ページ
\n	C言語と同じ改行
\r	C言語と同じ復帰
\t	C言語と同じ水平タブ
\uwxyz	(ここでwxyzは正確に4桁の16進数で)その16進数での値が0xwxyzという文字
\Ustuvwxyz	(ここでstuvwxyzは正確に8桁の16進数で)その16進数での値が0xstuvwxyzという文字
\v	C言語と同じ垂直タブ
\xhhh	(ここでhhhは任意の16進数の並びで)その文字の16進数値が0xhhhとなる文字(使用される16進数の桁数にかかわらず単一の文字)
\0	その値が0(NULLバイト)となる文字
\xy	(ここでxyは正確に2桁の8進数で、後方参照ではない)その値が0xyとなる文字
\xyz	(ここでxyzは正確に3桁の8進数で、後方参照ではない)その値が0xyzとなる文字

16進数の桁とは0-9、a-f、A-Fです。8進数の桁とは0-7です。

ASCIIの範囲(0-127)外の値を指定した数字のエントリエスケープは、その意味がデータベースエンコーディングに依存します。エンコーディングがUTF-8の場合、エスケープ値はユニコード符号位置に相当します。例えば、\u1234は文字U+1234を意味します。その他のマルチバイトエンコーディングでは、文字エントリエスケープはたいてい文字のバイト値の連結を指定します。エスケープ値がデータベースエンコーディングでのいかなる正当な文字にも対応しない場合、エラーは起こりませんが、いかなるデータにもマッチしません。

この文字エントリエスケープは常に普通の文字と解釈されます。例えば、\135はASCIIの]となり、\135はブラケット式の終端にはなりません。

表9.21 正規表現クラス省略エスケープ

エスケープ	説明
\d	[[:digit:]]
\s	[[:space:]]
\w	[[:alnum:]_] (アンダースコアが含まれることに注意)
\D	[^[:digit:]]
\S	[^[:space:]]
\W	[^[:alnum:]_] (アンダースコアが含まれることに注意)

ブラケット式内では、\d、\s、および\wはその外側の大括弧を失い、\D、\Sおよび\Wは不正です。(ですから、例えば[a-c\d]は[a-c[:digit:]]と同じになります。また、[a-c\D]は[a-c^[:digit:]]と同じになり、不正です。)

表9.22 正規表現制約エスケープ

エスケープ	説明
\A	文字列の先頭にのみマッチします(^との違いについては9.7.3.5を参照してください)。
\m	単語の先頭にのみマッチします。
\M	単語の末尾にのみマッチします。
\y	単語の先頭もしくは末尾にのみマッチします。
\Y	単語の先頭もしくは末尾以外の場所にのみマッチします。
\Z	文字列の末尾にのみマッチします($との違いについては9.7.3.5を参照してください)。

単語は前述の[[:<:]]と[[:>:]]の規定通りに定義されます。ブラケット式内では制約エスケープは不正です。

表9.23 正規表現後方参照

エスケープ	説明
\m	(ここでmは非ゼロの数です。)副式のm番目への後方参照
\mnn	(ここでmは非ゼロの数です。nnでさらに桁を指定します。mnn10進数値は取り込み括弧の数よりも多くてはなりません。)副式のmnn番目への後方参照

> **注記**
>
> 8進数の文字エントリエスケープと後方参照の間には曖昧性があります。上でヒントとして示したようにこれは以下の発見的手法で解決されます。先頭の0は常に8進数エスケープを示します。その後に数字が続かない単一の非ゼロ数字は常に後方参照として解釈されます。ゼロから始まらない複数数字の並びは、適切な副式の後にあれば(つまり、その番号が後方参照用の範囲内にあれば)後方参照として解釈されます。さもなくば、8進数として解釈されます。

9.7.3.4. 正規表現メタ構文

上述の主構文の他に、特殊な形式や雑多な構文的な機能が使用可能です。

REは、2つの特殊な*決定子*前置詞のどちらかから始まります。REが***:から始まるものであれば、REの残りはAREと解釈されます。(PostgreSQLはREをAREとして推測するため、通常は影響を受けません。ただし、正規表現関数に対して`flags`パラメータを指定されたEREやBREモードでは影響を受けます。) REが***=から始まるものであれば、REの残りは、全ての文字を普通の文字とみなしたリテラル文字列と解釈されます。

AREは*埋め込みオプション*から始められます。(?xyz)という並びで残りのREに影響するオプションを指定します(ここで`xyz`は1つ以上の英字です)。このオプションは、事前に決定されたオプションを上書きします。— 特に、正規表現演算子、もしくは正規表現関数に与えられた`flags`パラメータにより示される大文字小文字の区別を上書きします。使用可能なオプション文字を表 9.24に示します。これらの同じオプション文字が、正規表現関数の`flags`パラメータで使用されることに注意して下さい。

表9.24 ARE埋め込みオプション文字

オプション	説明
`b`	残りのREはBRE
`c`	大文字小文字を区別するマッチ(演算子で規定される大文字小文字の区別よりこの指定が優先されます)。
`e`	残りのREはERE
`i`	大文字小文字を区別しないマッチ(9.7.3.5を参照)(演算子で規定される大文字小文字の区別よりこの指定が優先されます)。
`m`	nの歴史的な同義語
`n`	改行を区別するマッチ(9.7.3.5を参照)
`p`	部分的な改行を区別するマッチ(9.7.3.5を参照)
`q`	残りのREはリテラル(「引用符付けされた」)文字列、全て普通の文字
`s`	改行を区別しないマッチ(デフォルト)
`t`	厳しめの構文(デフォルト、後述)
`w`	部分的な改行区別の逆(「ワイアード」)マッチ(9.7.3.5を参照)
`x`	拡張構文(後述)

埋め込みオプションはその並びの終端)で有効になります。AREの先頭(もし***:決定子があればその後)でのみ利用可能です。

全ての文字が意味を持つ、通常の（厳しめの）RE構文に加え、x埋め込みオプションを指定することで利用できる拡張構文があります。拡張構文では、RE内の空白文字は無視され、#とその後の改行（もしくはREの終端）の間の全ての文字も同様です。これにより、段落付けや複雑なREのコメント付けが可能になります。基本規則に対して3つの例外があります。

- 直前に\が付いた空白文字もしくは#は保持されます。
- ブラケット式内の空白文字もしくは#は保持されます。
- (?:などの複数文字シンボルでは、空白文字とコメントは不正です。

ここでの空白文字とは、空白、タブ、改行、スペース文字クラスに属する文字です。

最後に、AREのブラケット式の外側では、(?#ttt)という並びはコメントになります（ここでtttは）を含まない任意のテキストです）。繰り返しになりますが、これは(?:などの複数文字シンボルの文字間では使用できません。こうしたコメントは実用性というより歴史的所産です。そのため、この使用は勧めません。代わりに拡張構文を使用してください。

初めに***=決定子が指定され、ユーザの入力がREではなくリテラルとして扱われる場合、これらのメタ構文拡張は使用できません。

9.7.3.5. 正規表現マッチ規則

REが文字列の中の1つ以上の部分文字列とマッチする場合において、REは最初にマッチが始まった部分文字列とマッチします。その位置からまた1つ以上の部分文字列とマッチした際は、正規表現は最短マッチを行わない（欲張り型）か最短マッチを行う（非欲張り型）かによって、最長マッチもしくは最短マッチの文字列のどちらかにマッチします

REが最長マッチかどうかは以下の規則によって決まります。

- ほとんどのアトムおよび全ての式は欲張り属性を持ちません（これらは変動する量のテキストにまったくマッチしないからです）。
- REを括弧で括ることは欲張りかどうかを変更しません。
- {m}もしくは{m}?といった固定繰り返し数の量指定子を持つ量指定付きアトムは、アトム自身と同一の欲張りさを持ちます（まったく持たない可能性もあります）。
- 他の通常の量指定子（{m,n}、mとnが等しい場合も含みます）を持つ量指定付きアトムは欲張り型です（最長マッチを使用します）。
- 他の非欲張り型量指定子（{m,n}?、mとnが等しい場合も含みます）を持つ量指定付きアトムは非欲張り型です（最短マッチを使用します）。
- 最上位レベルの|演算子を持たないREであるブランチは、最初の欲張り属性を持つ量指定付きアトムと同一の欲張り属性を持ちます。
- |演算子で接続された2つ以上のブランチからなるREは常に欲張り型です。

上の規則は、個々の量指定付きアトムだけではなく、量指定付きアトムを複数含むブランチやRE全体の欲張り属性に関連します。つまり、ブランチやRE全体が全体として最長または最短の部分文字列にマッチするという方法でマッチ処理が行われます。全体のマッチの長さが決まると、特定の部分式にマッチする部分がそ

の部分式の欲張り属性によって決まります。この時、RE内でより前にある部分式が後にある部分式よりも高い優先度を持ちます。

この意味の例を示します。

```
SELECT SUBSTRING('XY1234Z', 'Y*([0-9]{1,3})');
Result: 123
SELECT SUBSTRING('XY1234Z', 'Y*?([0-9]{1,3})');
Result: 1
```

最初の例では、Y*が欲張り型であるため、REは全体として欲張り型です。マッチはYの位置から始まり、そこから可能な限り最長の文字列にマッチします。つまりY123となります。出力は括弧で括られた部分、つまり123となります。2番目の例では、Y*?が非欲張り型のため、REは全体として非欲張り型です。マッチはYの位置から始まり、そこから可能な限り最短の文字列にマッチします。つまりY1となります。部分式[0-9]{1,3}は欲張り型ですが、決定されたマッチする全体の長さを変更することはできません。したがって、強制的に1にマッチすることになります。

まとめると、REが欲張り型部分式と非欲張り型部分式の両方を持つ場合、全体のマッチ長はRE全体に割り当てられる属性に応じて、最長マッチ長か最短マッチ長のどちらかになります。部分式に割り当てられた属性は、部分式の中でどれだけの量をその部分式の中で「消費」できるかのみに影響します。

{1,1}および{1,1}?量指定子を副式もしくはRE全体に使用して、それぞれ、欲張りか欲張りでないかを強制することが可能です。RE全体に対してはその要素から推論されるものと異なる欲張りさの属性が必要な場合に、これは便利です。例として、数字をいくつか含む文字列を数字とその前後の部分に分けようとしているとします。次のようにしてみるかもしれません。

```
SELECT regexp_match('abc01234xyz', '(.*)(\d+)(.*)');
Result: {abc0123,4,xyz}
```

上手くいきませんでした。最初の.*が欲張りで、可能なだけ「消費」してしまい、\d+は最後の可能な場所で最後の数字にマッチします。欲張りでなくすることで直そうとするかもしれません。

```
SELECT regexp_match('abc01234xyz', '(.*?)(\d+)(.*)');
Result: {abc,0,""}
```

またもや上手くいきませんでした。今度は、REが全体として欲張りでなくなってしまい、できる限り早く全体に渡るマッチを終わらせてしまうからです。RE全体として欲張りにすることで欲しいものが得られます。

```
SELECT regexp_match('abc01234xyz', '(?:(.*?)(\d+)(.*)){1,1}');
Result: {abc,01234,xyz}
```

REの全体に渡る欲張りさをその要素の欲張りさと別に制御すれば、可変長のパターンを非常に柔軟に扱えます。

マッチが長いか短いかを判断する時には、マッチの長さは照合要素ではなく文字列で測られます。空文字列はまったくマッチする要素がない文字列よりも長いと考えられます。例えば、bb*はabbbcの真中の3文字とマッチし、(week|wee)(night|knights)はweeknightsの全ての10文字とマッチし、abcに対して(.*).*がマッチされると、括弧内の部分正規表現は3つの文字全てにマッチし、bcに対して(a*)*がマッチされると、全体のREと括弧内の正規表現は空文字列にマッチします。

もし大文字小文字を区別しないマッチが指定されると、アルファベット文字の大文字小文字の区別がまったくなくなったのと同じ効果を与えます。ブラケット式の外側にアルファベットの大文字小文字が混ざった通常の文字が出てきた場合、例えば、xが[xX]となるように大文字小文字ともにブラケット式に実質的に転換されます。ブラケット式の中に現れた時は、(例えば)[x]が[xX]となり、また[^x]が[^xX]となるように、全ての大文字小文字それぞれの対がブラケット式に追加されます。

改行を区別するマッチが指定されると、.と^を使用するブラケット式は(REが明示的に調整されていたとしてもマッチが改行をまたがらないようにするために)改行文字にマッチしなくなります。また、^と$はそれぞれ改行直後と直前の空文字列にマッチし、さらに、それぞれ文字列の先頭と末尾にマッチします。しかし、AREエスケープの\Aと\Zは、継続して、文字列の先頭と末尾*のみ*にマッチします。

部分的に改行を区別するマッチが指定されると、.とブラケット式は改行を区別するマッチを行うようになりますが、^と$は変更されません。

部分的に改行を区別する逆マッチが指定されると、^と$は改行を区別するマッチを行うようになりますが、.とブラケット式は変更されません。これはあまり有用ではありません。対称性のために提供されています。

9.7.3.6. 制限と互換性

本実装ではREの長さに関する制限はありません。しかし、移植性を高めたいプログラムでは、256バイトを超えるREを使用すべきではありません。POSIX互換の実装ではそうしたREでは混乱する可能性があります。

AREの機能のうち、POSIX EREと実質的な非互換性があるのは、\がブラケット式の内側で特殊な意味を失わないという点のみです。他の全てのARE機能は、POSIX EREでは不正、未定義、未指定な効果となる構文を使用しています。決定子の***構文などはBREおよびEREのPOSIX構文にはありません。

多くのARE拡張はPerlから拝借したものです。しかし、いくつかは整理され、Perlの拡張のいくつかは存在しません。注意すべき非互換性には、\b、\B、改行の取り扱いに関する特殊な措置の欠落、改行を区別するマッチに影響する点について補足したブラケット式の追加、括弧と先行・後方検索制約内の後方参照についての制限、最長/最短(最初にマッチするではなく)マッチのセマンティクスがあります。

PostgreSQLリリース7.4より前で認知された、AREとERE構文間で大きな非互換が2つあります。

- AREでは、\の後に英数字が続くものはエスケープもしくはエラーとなります。以前のリリースでは、これは単に、英数字を記述する他の方法でした。これは、大きな問題にはならないはずです。以前のリリースではこうした並びを記述する理由がないからです。
- AREでは、\は[]内でも特別な文字です。したがって、ブラケット式では\を\\と記述しなければなりません。

9.7.3.7. 基本正規表現

BREはEREといくつかの面において異なります。BREにおいては、|、+、?は普通の文字であり、それらの機能と等価なものはありません。バウンドの区切りは\{と\}であり、{と}自身は普通の文字です。副式を入れ子にするための括弧は\(と\)であり、(と)自身は普通の文字です。^は、REの先頭にある場合や括弧内の副式の先頭の場合を除き、普通の文字です。$は、REの末尾にある場合や括弧内の副式の末尾の場合を除き、普通の文字です。また、*はREの先頭にある場合や括弧内の副式の先頭にある場合には普通の文字になります(その前に^が付いている可能性もあります)。最後に、1桁の後方参照を使用することができ、また、BREにおいては、\<と\>はそれぞれ[[:<:]]と[[:>:]]と同義です。

9.7.3.8. XQueryとの違い(LIKE_REGEX)

SQL:2008以降、SQL標準にはXQuery正規表現標準によるパターンマッチングを行うLIKE_REGEX演算子が含まれています。PostgreSQLは今の所この演算子を実装していませんが、regexp_match()を使ってよく似た振る舞いを得ることができます。XQueryの正規表現は上で述べたARE構文に非常に近いからです。

既存のPOSIXベースの正規表現機能とXQueryの正規表現の主な違いには以下のものが含まれます。

- XQueryの文字クラス減算はサポートされていません。この機能の例としては、[a-z-[aeiou]]のようにして英語の子音のみにマッチさせるというのがあります。
- XQueryの文字クラス短縮形\c、\C、\i、\Iはサポートされていません。
- \p{UnicodeProperty}あるいはその逆である\P{UnicodeProperty}を使ったXQueryの文字クラス要素はサポートされていません。
- POSIXは有効なロケール(演算子あるいは関数のCOLLATE節で制御できます)にしたがい、\w(表 9.21参照)のような文字クラスを解釈します。XQueryはこれらのクラスをUnicodeの文字属性を参照してこれらのクラスを決定します。ですからUnicodeルールに従うロケールを使用してのみ同等の振る舞いを得ることができます。
- SQL標準(XQuery自身ではなく)はPOSIXが提供するより多様な「newline」の亜種を提供しようとしています。上で述べた改行に敏感なマッチオプションはASCII NL(\n)だけを改行として考慮します。しかしSQLはCR (\r)、CRLF (\r\n)(Windowsスタイルの改行)、LINE SEPARATOR (U+2028)のようなUnicodeのみの文字も改行として扱うことを求めています。とりわけ、SQLにおいては、.と\sは\r\nを2文字ではなく、1文字として数える必要があります。
- 表 9.20で示す文字エントリエスケープのうち、XQueryは\n、\r、\tだけをサポートしています。
- XQueryはブラケット式内の文字クラスとして[:name:]構文をサポートしていません。
- XQueryには先行検索制約および後方検索制約がありませんし、表 9.22に記述された制約エスケープもありません。
- 9.7.3.4に記述されたメタ構文形式はXQueryには存在しません。
- XQueryで定義された正規表現フラグ文字はPOSIX(表 9.24)のオプション文字に関連していますが、同じではありません。iとqオプションは同じように振る舞いますが、その他は違います。
  - XQueryのs(ピリオドが改行にマッチすることを許容する)とm(^と$が改行位置でマッチすることを許容する)フラグは、POSIXのn、p、wフラグと同じ挙動を提供しますが、POSIXのsとmフラグの挙動とは一致しません。ピリオドが改行にマッチするのはPOSIXではデフォルトの挙動ですが、XQueryではそうでないことに留意してください。
  - XQueryのx(パターン中の空白を無視する)フラグはPOSIXの拡張モードフラグとは著しく異なります。POSIXのxフラグは#でパターン中のコメントを始めることもできます。POSIXはバックスラッシュ以降の空白文字を無視しません。

9.8. データ型書式設定関数

PostgreSQLの書式設定関数は多彩なデータ型（日付/時刻データ型、整数データ型、浮動小数点数データ型、数値データ型）を整形された文字列に変換したり、整形された文字列を特定のデータ型に変換する強力なツールの一式を提供しています。表 9.25にこれらを列挙しています。これら関数は共通の呼び出し規約を踏襲しています。最初の引数は整形される値で2番目の引数は入力書式または出力書式を定義するテンプレートです。

表9.25 書式設定関数

関数 説明 例
to_char (timestamp, text) → text to_char (timestamp with time zone, text) → text 与えられた書式設定にしたがってタイムスタンプを文字列に変換します。 to_char(timestamp '2002-04-20 17:31:12.66', 'HH12:MI:SS') → 05:31:12
to_char (interval, text) → text 与えられた書式設定にしたがって時間間隔を文字列に変換します。 to_char(interval '15h 2m 12s', 'HH24:MI:SS') → 15:02:12
to_char (numeric_type, text) → text 与えられた書式設定にしたがって数値を文字列に変換します。integer、bigint、numeric、real、double precisionで利用可能です。 to_char(125, '999') → 125 to_char(125.8::real, '999D9') → 125.8 to_char(-125.8, '999D99S') → 125.80-
to_date (text, text) → date 与えられた書式設定にしたがって文字列を日付に変換します。 to_date('05 Dec 2000', 'DD Mon YYYY') → 2000-12-05
to_number (text, text) → numeric 与えられた書式設定にしたがって文字列を数値に変換します。 to_number('12,454.8-', '99G999D9S') → -12454.8
to_timestamp (text, text) → timestamp with time zone 与えられた書式設定にしたがって文字列をタイムスタンプに変換します。（表 9.32のto_timestamp(double precision)もご覧ください。） to_timestamp('05 Dec 2000', 'DD Mon YYYY') → 2000-12-05 00:00:00-05

ヒント

to_timestampとto_dateは、単純なキャストでは変換できない入力フォーマットを処理するために存在します。ほとんどの標準的日付および時刻のフォーマットに対しては、入力文字列を必要なデータ型に単純にキャストすれば動作し、その方がずっと簡単です。同様に、to_numberも標準的な数値表現に対しては不要です。

to_char用の出力テンプレート文字列には、値に基づいて認識され、適切に整形されたデータで置き換えられるパターンがあります。テンプレートパターンではない全てのテキストは単にそのままコピーされます。同様に、（その他の関数用の）入力テンプレート文字列では、テンプレートパターンは入力されたデータ文字列で

供給される値を特定します。テンプレート文字列中にテンプレートパターンではない文字があれば、(テンプレート文字列の文字と同じかどうかにかかわらず)入力文字列データ中の該当文字は単にスキップされます。

表 9.26に、日付/時刻型の値の書式に使用可能なテンプレートパターンを示します。

表9.26 日付/時刻型の書式テンプレートパターン

パターン	説明
HH	時 (01–12)
HH12	時 (01–12)
HH24	時 (00–23)
MI	分 (00–59)
SS	秒 (00–59)
MS	ミリ秒 (000–999)
US	マイクロ秒 (000000–999999)
FF1	10分の1秒 (0–9)
FF2	100分の1秒 (00–99)
FF3	ミリ秒 (000–999)
FF4	10分の1ミリ秒 (0000–9999)
FF5	100分の1ミリ秒 (00000–99999)
FF6	マイクロ秒 (000000–999999)
SSSS, SSSSS	深夜0時からの秒数 (0–86399)
AM、am、PMまたはpm	午前/午後の指定(ピリオドなし)
A.M.、a.m.、P.M.またはp.m.	午前/午後の指定(ピリオドあり)
Y,YYY	コンマ付き年(4桁以上)
YYYY	年(4桁以上)
YYY	年の下3桁
YY	年の下2桁
Y	年の下1桁
IYYY	ISO 8601週番号年(4桁以上)
IYY	ISO 8601週番号年の下3桁
IY	ISO 8601週番号年の下2桁
I	ISO 8601週番号年の下1桁
BC、bc、AD、またはad	紀元前後の指定(ピリオドなし)
B.C.、b.c.、A.D.、またはa.d.	紀元前後の指定(ピリオド付き)
MONTH	大文字での完全な月名(9文字になるように空白文字を埋める)
Month	大文字で書き始める完全な月名(9文字になるように空白文字を埋める)

パターン	説明
month	小文字での完全な月名（9文字になるように空白文字を埋める）
MON	大文字での短縮形の月名（英語では3文字、現地語化された場合は可変長）
Mon	大文字で書き始める短縮形の月名（英語では3文字。現地語化された場合は可変長）
mon	小文字での短縮形の月名（英語では3文字。現地語化された場合は可変長）
MM	月番号(01–12)
DAY	大文字での完全な曜日名（9文字になるように空白文字を埋める）
Day	大文字で書き始める完全な曜日名（9文字になるように空白文字を埋める）
day	小文字での完全な曜日名（9文字になるように空白文字を埋める）
DY	短縮形の大文字での短縮形の曜日名（英語では3文字。現地語化された場合は可変長）
Dy	大文字で書き始める短縮形の曜日名（英語では3文字。現地語化された場合は可変長）
dy	小文字での短縮形の曜日名（英語では3文字。現地語化された場合は可変長）
DDD	通年の日にち番号 (001–366)
IDDD	ISO 8601週番号年の日にち番号（001–371：通年 第1日は最初のISO週の月曜日）
DD	月内の日にち番号 (01–31)
D	曜日番号、日曜日（1）から土曜日（7）まで
ID	ISO 8601の曜日番号、月曜日（1）から日曜日（7）まで
W	月中の週番号 (1–5)（その月の初日がある週が第1週）
WW	年間を通じた週番号 (1–53)（元日のある週が第1週）
IW	ISO 8601週番号年の年間を通じた週番号 (01–53;新年の最初の木曜日がある週が第1週）
CC	世紀（2桁。21世紀は2001-01-01から開始）
J	ユリウス日（UTC紀元前4714年11月24日午前零時からの整数による通算経過日）
Q	四半期
RM	大文字ローマ数字による月（I–XII、Iは1月）
rm	小文字ローマ数字による月（(i–xii、iは1月）
TZ	大文字による時間帯省略名（to_char内でのみサポートされる）
tz	小文字による時間帯省略名（to_char内でのみサポートされる）
TZH	time-zoneの時間

パターン	説明
TZM	time-zoneの分
OF	UTCからの時間帯オフセット(to_char内でのみサポートされる)

どのようなテンプレートパターンに対しても、その振舞いを変更するために修飾子を適用できます。例えば、FMMonthはFM修飾子の付いたMonthパターンです。表 9.27に、日付/時刻書式の修飾子パターンを示します。

表9.27 日付/時刻書式用のテンプレートパターン修飾子

修飾子	説明	例
FM接頭辞	字詰めモード(先頭の0、およびを空白のパディングを無効)	FMMonth
TH接尾辞	DDTH、例えば12TH	
th接尾辞	DDth、例えば12th	
FX接頭辞	固定書式のグローバルオプション(使用上の注意事項を参照)	FX Month DD Day
TM接頭辞	翻訳モード(lc_timeに基づき、現地語化された曜日、月名を使います)	TMMonth
SP接尾辞	スペルモード(未実装)	DDSP

日付/時刻型書式の使用上の注意事項は次のとおりです。

- FMは、先頭にはゼロ、末尾には空白を追加してパターンを固定長にする機能を無効にします。PostgreSQLでは、FMはその次に記述されたものだけを変更します。一方Oracleでは、FMはそれに続く全ての記述に対して影響し、FM修飾詞を繰り返すと、ゼロや空白を埋めるモードのオンとオフが切り替わります。

- FMが指定されているかどうかに関わらずTMは末尾の空白を抑止します。

- to_timestampとto_dateは入力中の大文字小文字の区別を無視します。例えばMON、Mon、monはすべて同じ文字列として受け付けます。TM修飾子を使うと関数の入力照合順のルールにしたがって大文字小文字の変換が行われます。(23.2参照。)

- FXオプションが使用されていない限り、to_timestampとto_dateは入力文字列内最初の連続した空白と、日付と時間の値の周辺の複数の空白を無視します。例えば、to_timestamp('2000 JUN', 'YYYY MON')とto_timestamp('2000 - JUN', 'YYYY-MON')は動作しますが、to_timestamp('2000 JUN','FXYYYY MON')はエラーを返します。後者のto_timestampは単一のスペースだけがあることを期待するからです。FXはテンプレートの第1項目として指定される必要があります。

- FXオプションが使用されていない限り、to_timestampとto_dateのテンプレート文字列中の区切り文字(空白あるいは記号文字(訳注:原文は"non-letter/non-digit character"))は入力文字中のすべての単一の区切り文字とマッチするか、あるいはマッチしない場合はスキップします。たとえば、to_timestamp('2000JUN', 'YYYY///MON')とto_timestamp('2000/JUN', 'YYYY MON')は動作しますが、to_timestamp('2000//JUN', 'YYYY/MON')は入力文字列中の区切り文字の数がテンプレート中の区切り文字の数を上回っているため、エラーを返します。

FXが指定されていると、テンプレート文字列中の区切り文字は正確に入力文字列中の一文字とマッチします。しかし、入力文字列の文字はテンプレート文字列中の区切り文字と一致する必要はないことに注意してください。たとえば、to_timestamp('2000/JUN', 'FXYYYY MON')は動作しますが、to_timestamp('2000/JUN', 'FXYYYY MON')はテンプレート文字列中の二番目の空白が入力文字列中の文字Jを消費するため、エラーを返します。

- TZHテンプレートパターンは符号付きの数字とマッチします。FXオプションが無い場合、マイナス符号は曖昧で、区切り文字として解釈されるかも知れません。この曖昧さは次のようにして解消されます。テンプレート文字列中のTZHの前の区切り文字の数が入力文字列中のマイナス符号の前の区切り文字の数よりも少なければ、そのマイナス符号はTZHの一部として解釈されます。そうでない場合、マイナス記号が値の区切り記号と見なされます。たとえば、to_timestamp('2000 -10', 'YYYY TZH')では-10がTZHにマッチしますが、to_timestamp('2000 -10', 'YYYY TZH')では10がTZHにマッチします。

- to_charテンプレートには、通常のテキストを入れることができ、それはそのまま出力されます。部分文字列を二重引用符で括ることで、部分文字列にテンプレートパターンがあったとしても、強制的にリテラルテキストとして解釈させることができます。例えば、'"Hello Year "YYYY'ではYYYYは年データに置換されてしまいますが、Year内のYは置換されません。to_date、to_number、to_timestampでは、二重引用符で括られた文字の数だけ入力された文字をスキップします。例えば"XX"は2文字の入力文字（それがXXであるかどうかにかかわらず)をスキップします。

> **ヒント**
>
> PostgreSQL 12より前では、記号文字（訳注：原文は"non-letter or non-digit"）を使って入力文字列中の任意のテキストをスキップすることが可能でした。たとえば、to_timestamp('2000y6m1d', 'yyyy-MM-DD')は動作しました。現在は、この目的のために非記号文字（訳注：原文は"letter characters"）だけを使うことができます。たとえば、to_timestamp('2000y6m1d', 'yyyytMMtDDt')とto_timestamp('2000y6m1d', 'yyyy"y"MM"m"DD"d"')は、y、m、dをスキップします。

- 出力に二重引用符を付けたい場合、'\"YYYY Month\"'のようにその前にバックスラッシュを付けなければなりません。バックスラッシュは、二重引用符の外側では特別扱いされません。二重引用符の内側では、バックスラッシュによって次の文字が何であれ文字通りに扱われるようになります。（しかし、次の文字が二重引用符であるか、あるいは別のバックスラッシュでない限り、これは特別な効果をもたらしません。）

- to_timestampにおいてto_date、YYYの様に4桁未満の年書式が指定され、かつ与えられる年が4桁未満だった場合、年は2020年に最も近くなるよう調整されます。例えば、95の場合は1995年になります。

- to_timestampおよびto_dateにおいて負の年はBCを表します。負の年と明示的なBCフィールドの両方を記述すると、再びADになります。すべての形のゼロ年はBC 1として扱われます。

- to_timestampおよびto_dateにおいてYYYY変換は、5桁以上の年数値を処理するときに制限事項があります。このような場合、YYYYの後に数字以外の文字またはテンプレートを使わなければなりません。そうしないと年は常に4桁と解釈されます。例えば（20000年として）、to_date('200001131', 'YYYYMMDD')は4桁の年と解釈されるので、代わりにto_date('20000-1131', 'YYYY-MMDD')またはto_date('20000Nov31', 'YYYYMonDD')のように数字でない区切り文字を使用してください。

- to_timestampおよびto_dateにおいてYYY、YYYY、もしくはY,YYYフィールドが存在するとCC(世紀)フィールドは受け入れられますが、無視されます。CCがYYもしくはYと共に使用されると、結果は指定された世紀のその年として計算されます。世紀が指定され、年が指定されないときは、その世紀の最初の年と想定されます。

- to_timestampおよびto_dateにおいて、曜日の名前や数字(DAY、Dおよび関連したフィールドの型)は受け付けられますが、結果を計算するという目的においては無視されます。同じことは四半期(Q)フィールドにも当てはまります。

- to_timestampおよびto_dateにおいて、ISO 8601週番号日は(グレゴリオ暦の日付とは異なって)以下の2つの方法のうちのひとつで指定できます。

  - 年、通年の週番号、曜日番号。例えば、to_date('2006-42-4', 'IYYY-IW-ID')は、日付2006-10-19を返します。曜日番号を省略した場合、1(月曜日)と想定されます。

  - 年と通年の日付番号。例えば、to_date('2006-291', 'IYYY-IDDD')も2006-10-19を返します。

  ISO 8601週番号とグレゴリオ暦日のフィールドを混在して使用して日付を構築する試みは無意味なことで、エラーの原因になります。ISO 8601週番号年の文脈では、「月」、あるいは「月内の日付番号」は意味を持ちません。グレゴリオ暦の年の文脈では、ISO週番号は意味を持ちません。

  > **注意**
  >
  > to_dateはグレゴリオとISO週番号日のフィールドの混在を拒否しますが、to_charはそうではありません。YYYY-MM-DD (IYYY-IDDD)のような出力書式指定が有用な場合があるからです。しかし、IYYY-MM-DDのような書き方は避けてください。年の初めの近くで驚くべき結果になるでしょう。(より詳細な情報は9.9.1を参照してください。)

- to_timestampにおいて、ミリ秒(MS)およびマイクロ秒(US)フィールドは小数点の後の秒の桁として使用されます。例えば、to_timestamp('12.3', 'SS.MS')は3ミリ秒ではなく300ミリ秒です。なぜなら変換においてこれは12 + 0.3秒と計算されるからです。従ってSS.MS書式に対して入力値12.3、12.30、12.300は同じミリ秒数を指定することになります。3ミリ秒が必要な場合には12:003のようにしなければなりません。この時、変換において12 + 0.003 = 12.003秒と計算します。

  もう少し複雑な例を挙げます。to_timestamp('15:12:02.020.001230', 'HH24:MI:SS.MS.US')は15時間12分と2秒+20ミリ秒+1230マイクロ秒 = 2.021230秒です。

- to_char(..., 'ID')の曜日番号付けはextract(isodow from ...)関数に一致しますが、to_char(..., 'D')の曜日番号付けはextract(dow from ...)の曜日番号付けに一致しません。

- to_char(interval)関数は、HHとHH12を12時間の時計に表示されるように整形します。例えば0時間と36時間はいずれも12として出力します。一方HH24は時間の値をそのまま出力し、intervalの値であれば23を超えることも可能です。

表 9.28に、数値の書式設定に使用可能なテンプレートパターンを示します。

表9.28 数値書式用のテンプレートパターン

パターン	説明
9	数字の位置(必要ないときは表示しない)

パターン	説明
0	数字の位置（必要ないときでも表示する）
.（ピリオド）	小数点
,（コンマ）	千単位で区切る符号
PR	負の値の角括弧表示
S	符号付き値（ロケールを使用）
L	通貨記号（ロケールを使用）
D	小数点（ロケールを使用）
G	グループ区切り文字（ロケールを使用）
MI	（数値 < 0であれば）指定位置にマイナス記号
PL	（数値 > 0であれば）指定位置にプラス記号
SG	指定された位置にプラス/マイナス記号
RN	ローマ数字（入力は1～3999）
THまたはth	序数接尾辞
V	n 桁シフト（注意事項を参照）
EEEE	科学技術表記法用の指数

数値型書式の使用上の注意事項は次のとおりです。

- 0は、それが先頭あるいは末尾のゼロであっても必ず表示する数字の位置を指定します。9も数字の位置を指定しますが、先頭のゼロであればそれは空白で置換され、また末尾のゼロで字詰めモードが指定されているときは削除されます。（to_number()では、これら2つのパターン文字は同じ意味になります。）

- パターン文字S、L、D、Gはそれぞれ現在のロケールで定義された符号、通貨記号、小数点、3桁区切り文字を表します（lc_monetaryおよびlc_numericを参照）。パターン文字のピリオドとカンマはいずれもその文字そのものを表し、ロケールとは関係なく小数点と3桁区切り文字の意味を持ちます。

- to_char()のパターンで符号について明示的な条件付けがない場合、符号のために一桁が予約され、それは数に繋げられます（すぐ左側に置かれます）。Sがいくつかの9のすぐ左に置かれた場合、同様に数に繋げられます。

- SG、PL、またはMIで整形された符号は、数値と関連付けられません。例えば、to_char(-12, 'MI9999')は'- 12'となる一方、to_char(-12, 'S9999')は' -12'となります。（Oracleの実装では9の前にMIが置かれてはならず、9の後にMIが置かれることを要求しています。）

- THはゼロ未満の値と小数は変換しません

- PL、SG、およびTHはPostgreSQLの拡張です。

- to_numberにおいて、LあるいはTHのように非データテンプレートが使われた場合には、それがデータ文字（すなわち、数字、符号、10進小数点あるいはカンマ）でない限りテンプレートパターンにマッチするかどうかにかかわらず、該当する数分だけの入力文字がスキップされます。

- Vをto_charにつけると、入力値を10^n倍します。ここでnはVに続く桁数です。Vをto_numberにつけると、同じように割り算をします。to_charおよびto_numberは、小数点とVとの混在をサポートしません（例えば、99.9V99 とはできません）。

- EEEE(科学技術表記)は、桁と小数点のパターンを除き、他の書式パターンや修飾子と組み合わせて使うことはできず、また必ず書式文字列の最後に位置しなければなりません(例えば、9.99EEEEは正しい表記となります)。

すべてのテンプレートについて、その動作を変えるために、いくつかの修飾子を適用できます。例えば、FM99.99はFM修飾子が付いた99.99パターンです。表 9.29に、数値の書式用の修飾子パターンを示します。

表9.29 数値の書式用テンプレートパターン修飾子

修飾子	説明	例
FM添え字	字詰めモード（末尾の0と空白の埋め字を無効にする）	FM99.99
TH添え字	大文字による序数添え字	999TH
th添え字	子文字による序数添え字	999th

表 9.30に、to_char関数を使用した例をいくつか示します。

表9.30 to_charの例

式	結果
to_char(current_timestamp, 'Day, DD HH12:MI:SS')	'Tuesday , 06 05:39:18'
to_char(current_timestamp, 'FMDay, FMDD HH12:MI:SS')	'Tuesday, 6 05:39:18'
to_char(-0.1, '99.99')	' -.10'
to_char(-0.1, 'FM9.99')	'-.1'
to_char(-0.1, 'FM90.99')	'-0.1'
to_char(0.1, '0.9')	' 0.1'
to_char(12, '9990999.9')	' 0012.0'
to_char(12, 'FM9990999.9')	'0012.'
to_char(485, '999')	' 485'
to_char(-485, '999')	'-485'
to_char(485, '9 9 9')	' 4 8 5'
to_char(1485, '9,999')	' 1,485'
to_char(1485, '9G999')	' 1 485'
to_char(148.5, '999.999')	' 148.500'
to_char(148.5, 'FM999.999')	'148.5'
to_char(148.5, 'FM999.990')	'148.500'
to_char(148.5, '999D999')	' 148,500'
to_char(3148.5, '9G999D999')	' 3 148,500'

式	結果
to_char(-485, '999S')	'485-'
to_char(-485, '999MI')	'485-'
to_char(485, '999MI')	'485 '
to_char(485, 'FM999MI')	'485'
to_char(485, 'PL999')	'+485'
to_char(485, 'SG999')	'+485'
to_char(-485, 'SG999')	'-485'
to_char(-485, '9SG99')	'4-85'
to_char(-485, '999PR')	'<485>'
to_char(485, 'L999')	'DM 485'
to_char(485, 'RN')	' CDLXXXV'
to_char(485, 'FMRN')	'CDLXXXV'
to_char(5.2, 'FMRN')	'V'
to_char(482, '999th')	' 482nd'
to_char(485, '"Good number:"999')	'Good number: 485'
to_char(485.8, '"Pre:"999" Post:" .999')	'Pre: 485 Post: .800'
to_char(12, '99V999')	' 12000'
to_char(12.4, '99V999')	' 12400'
to_char(12.45, '99V9')	' 125'
to_char(0.0004859, '9.99EEEE')	' 4.86e-04'

9.9. 日付/時刻関数と演算子

表 9.32は、日付/時刻型の値の処理で使用可能な関数を示しています。詳細は、以下の副節で説明します。表 9.31は、(+、*等の)基本的な算術演算子の振舞いを説明しています。書式設定関数については9.8を参照してください。8.5を参照して、日付/時刻データ型についての背景となっている情報に精通していなければなりません。

以下のtimeもしくはtimestamp型の入力を受け取る関数および演算子は全て、実際には2つの種類があります。1つはtime with time zone型またはtimestamp with time zone型を取るもので、もう1つはtime without time zone型もしくはtimestamp without time zone型を取るものです。簡略化のため、これらの種類の違いは個別に示していません。また、+と*演算子は可換な2項をとります(例えばdate + integerとinteger + date)。こうした組み合わせは片方のみ示します。

表9.31 日付/時刻演算子

演算子 説明 例
date + integer → date 日付に日数を加算 date '2001-09-28' + 7 → 2001-10-05
date + interval → timestamp 時刻間隔を日付に加算 date '2001-09-28' + interval '1 hour' → 2001-09-28 01:00:00
date + time → timestamp 日付に時刻を加算 date '2001-09-28' + time '03:00' → 2001-09-28 03:00:00
interval + interval → interval 時間間隔を加算 interval '1 day' + interval '1 hour' → 1 day 01:00:00
timestamp + interval → timestamp 時間間隔をタイムスタンプに加算 timestamp '2001-09-28 01:00' + interval '23 hours' → 2001-09-29 00:00:00
time + interval → time 時間間隔を時分に加算 time '01:00' + interval '3 hours' → 04:00:00
- interval → interval 時間間隔の符号を反転 - interval '23 hours' → -23:00:00
date - date → integer 日付を減算し、経過日数を返す date '2001-10-01' - date '2001-09-28' → 3
date - integer → date 日付から日数を減算 date '2001-10-01' - 7 → 2001-09-24
date - interval → timestamp 日付から時間間隔を減算 date '2001-09-28' - interval '1 hour' → 2001-09-27 23:00:00
time - time → interval 時分を減算 time '05:00' - time '03:00' → 02:00:00
time - interval → time 時分から時刻間隔を減算 time '05:00' - interval '2 hours' → 03:00:00
timestamp - interval → timestamp タイムスタンプから時刻間隔を減算

演算子 説明 例
timestamp '2001-09-28 23:00' - interval '23 hours' → 2001-09-28 00:00:00
interval - interval → interval 時間間隔を減算 interval '1 day' - interval '1 hour' → 1 day -01:00:00
timestamp - timestamp → interval タイムスタンプを減算(justify_hours()と同様に24時間間隔を日数に変換) timestamp '2001-09-29 03:00' - timestamp '2001-07-27 12:00' → 63 days 15:00:00
interval * double precision → interval 時間間隔にスカラーを乗算 interval '1 second' * 900 → 00:15:00 interval '1 day' * 21 → 21 days interval '1 hour' * 3.5 → 03:30:00
interval / double precision → interval 時間間隔をスカラーで除算 interval '1 hour' / 1.5 → 00:40:00

表9.32 日付/時刻関数演算子

関数 説明 例
age (timestamp, timestamp) → interval 引数間の減算。日数だけでなく年と月を使用した「言葉による」結果を生成 age(timestamp '2001-04-10', timestamp '1957-06-13') → 43 years 9 mons 27 days
age (timestamp) → interval current_date(午前零時時点)から引数を減算 age(timestamp '1957-06-13') → 62 years 6 mons 10 days
clock_timestamp ( ) → timestamp with time zone 現在の日付と時刻(文実行中に変化する)。9.9.4を参照。 clock_timestamp() → 2019-12-23 14:39:53.662522-05
current_date → date 現在の日付。9.9.4を参照 current_date → 2019-12-23
current_time → time with time zone 現在の時刻。9.9.4を参照。 current_time → 14:39:53.662522-05
current_time (integer) → time with time zone 精度を限定した現在の時刻。9.9.4を参照。 current_time(2) → 14:39:53.66-05
current_timestamp → timestamp with time zone

関数 説明 例
現在の日付と時刻（現在のトランザクションの開始時）。9.9.4を参照。 current_timestamp → 2019-12-23 14:39:53.662522-05
current_timestamp (integer) → timestamp with time zone 精度を限定した現在の日付と時刻（現在のトランザクションの開始時）。9.9.4を参照。 current_timestamp(0) → 2019-12-23 14:39:53-05
date_part (text, timestamp) → double precision タイムスタンプの部分フィールドの取得（extractと同じ）。9.9.1を参照。 date_part('hour', timestamp '2001-02-16 20:38:40') → 20
date_part (text, interval) → double precision 時間間隔の部分フィールドの取得（extractと同じ）。9.9.1を参照。 date_part('month', interval '2 years 3 months') → 3
date_trunc (text, timestamp) → timestamp 指定された精度で切り捨て。9.9.2参照。 date_trunc('hour', timestamp '2001-02-16 20:38:40') → 2001-02-16 20:00:00
date_trunc (text, timestamp with time zone, text) → timestamp with time zone 指定された時間帯において指定された精度で切り捨て。9.9.2参照 date_trunc('day', timestamptz '2001-02-16 20:38:40+00', 'Australia/Sydney') → 2001-02-16 13:00:00+00
date_trunc (text, interval) → interval 指定された精度で切り捨て。9.9.2参照。 date_trunc('hour', interval '2 days 3 hours 40 minutes') → 2 days 03:00:00
extract (field from timestamp) → double precision タイムスタンプの部分フィールドの取得。9.9.1を参照。 extract(hour from timestamp '2001-02-16 20:38:40') → 20
extract (field from interval) → double precision 時間間隔の部分フィールドの取得。9.9.1を参照。 extract(month from interval '2 years 3 months') → 3
isfinite (date) → boolean 日付が有限（+/-無限でない）かどうかの検査 isfinite(date '2001-02-16') → true
isfinite (timestamp) → boolean タイムスタンプが有限（+/-無限でない）かどうかの検査 isfinite(timestamp 'infinity') → false
isfinite (interval) → boolean 時間間隔が有限かどうかの検査（今の所常に真） isfinite(interval '4 hours') → true
justify_days (interval) → interval 30日周期が1月を表すように時間間隔を調整 justify_days(interval '35 days') → 1 mon 5 days
justify_hours (interval) → interval

関数 説明 例
24時間を1日とする時間間隔の調整 justify_hours(interval '27 hours') → 1 day 03:00:00
justify_interval (interval) → interval justify_daysおよびjustify_hoursを使用し、さらに符号による調整を行っての時間間隔の調整 justify_interval(interval '1 mon -1 hour') → 29 days 23:00:00
localtime → time 現在の時刻。9.9.4を参照。 localtime → 14:39:53.662522
localtime (integer) → time 精度を限定した現在の時刻。9.9.4を参照。 localtime(0) → 14:39:53
localtimestamp → timestamp 現在の日付と時刻(現在のトランザクションの開始時)。9.9.4を参照。 localtimestamp → 2019-12-23 14:39:53.662522
localtimestamp (integer) → timestamp 精度を限定した現在の日付と時刻(現在のトランザクションの開始時)。9.9.4を参照。 localtimestamp(2) → 2019-12-23 14:39:53.66
make_date (year int, month int, day int) → date 年、月、日フィールドから日付を作成 make_date(2013, 7, 15) → 2013-07-15
make_interval ([years int [, months int [, weeks int [, days int [, hours int [, mins int [, secs double precision]]]]]]]) → in terval 年、月、週、日、時間、分、秒フィールドから時間間隔を作成 make_interval(days => 10) → 10 days
make_time (hour int, min int, sec double precision) → time 時、分、秒フィールドから時刻を作成 make_time(8, 15, 23.5) → 08:15:23.5
make_timestamp (year int, month int, day int, hour int, min int, sec double precision) → timestamp 年、月、日、時、分、秒フィールドから時刻を作成 make_timestamp(2013, 7, 15, 8, 15, 23.5) → 2013-07-15 08:15:23.5
make_timestamptz (year int, month int, day int, hour int, min int, sec double precision [, timezone text]) → timestamp wi th time zone 年、月、日、時、分、秒フィールドから時間帯付きの時刻を作成。timezoneが指定されていなければ、現在の時間帯が使われる 。 make_timestamptz(2013, 7, 15, 8, 15, 23.5) → 2013-07-15 08:15:23.5+01
now ( ) → timestamp with time zone 現在の日付と時刻(現在のトランザクションの開始時)。9.9.4を参照。 now() → 2019-12-23 14:39:53.662522-05
statement_timestamp ( ) → timestamp with time zone

関数 説明 例
現在の日付と時刻(現在の文の開始時)。9.9.4を参照。 statement_timestamp() → 2019-12-23 14:39:53.662522-05
timeofday() → text 現在の日付と時刻(clock_timestampと似ているが、text型文字列として返す)。9.9.4を参照。 timeofday() → Mon Dec 23 14:39:53.662522 2019 EST
transaction_timestamp() → timestamp with time zone 現在の日付と時刻(現在のトランザクションの開始時)。9.9.4を参照。 transaction_timestamp() → 2019-12-23 14:39:53.662522-05
to_timestamp(double precision) → timestamp with time zone Unixエポック時間(1970-01-01 00:00:00+00からの経過秒数)をtimestamp with time zoneに変換 to_timestamp(1284352323) → 2010-09-13 04:32:03+00

これらの関数に加え、OVERLAPS SQL演算子がサポートされています。

```
(start1, end1) OVERLAPS (start2, end2)
(start1, length1) OVERLAPS (start2, length2)
```

この式は、2つの時間間隔(その両端で定義されます)が重なる時に真を返します。重ならない場合は偽を返します。両端は2つの日付、時刻、タイムスタンプとして、もしくは、日付/時刻/タイムスタンプとそれに続く時間間隔として指定できます。値の組み合わせで指定する場合、開始と終了のいずれを先に記述しても構いません。OVERLAPSは与えられた値のうち、早い方を開始として扱います。各時間間隔は、start <= time < endという半開区間として見なされます。ただし、startとendが同じ値の場合には単一の時間点となります。これは、例えば端点のみが共通である2つの時間間隔は、重ならないということを意味します。

```
SELECT (DATE '2001-02-16', DATE '2001-12-21') OVERLAPS
       (DATE '2001-10-30', DATE '2002-10-30');
Result: true
SELECT (DATE '2001-02-16', INTERVAL '100 days') OVERLAPS
       (DATE '2001-10-30', DATE '2002-10-30');
Result: false
SELECT (DATE '2001-10-29', DATE '2001-10-30') OVERLAPS
       (DATE '2001-10-30', DATE '2001-10-31');
Result: false
SELECT (DATE '2001-10-30', DATE '2001-10-30') OVERLAPS
       (DATE '2001-10-30', DATE '2001-10-31');
Result: true
```

timestamp with time zoneの値にintervalの値を加える時(またはintervalの値を引く時)、日にちの部分は、timestamp with time zoneの日付を指定された日数だけ先に進める、もしくは後に戻し、時刻は同じに保ちます。(セッションの時間帯がDSTを認識する設定の場合)夏時間の移行に跨っての変化に関しては、interval '1 day'がinterval '24 hours'に等しいとは限りません。例えば、セッションの時間帯がAmerica/Denverに設定されている時には以下のようになります。

```
SELECT timestamp with time zone '2005-04-02 12:00:00-07' + interval '1 day';
結果: 2005-04-03 12:00:00-06
SELECT timestamp with time zone '2005-04-02 12:00:00-07' + interval '24 hours';
結果: 2005-04-03 13:00:00-06
```

その理由はAmerica/Denver時間帯で2005-04-03 02:00に夏時間への変更があるからです。

異なる月では日数が異なりますのでageで返されるmonthsフィールドにはあいまいさがあります。PostgreSQLのやり方は月をまたがる2つの日付の計算において、日付の早いほうの月を使用します。例えば、age('2004-06-01', '2004-04-30')は4月を使用して1 mon 1 dayを得ます。5月は31日あり、4月は30日のため、もし5月を使用するなら結果は1 mon 2 daysとなるでしょう。

日付とタイムスタンプの引き算は複雑になることがあります。引き算をする概念的に単純な方法は、それぞれの値を秒数にEXTRACT(EPOCH FROM ...)で変換してから、結果を引き算する方法です。この結果は2つの値の間の秒数になります。これは各月の日数、時間帯の変更、夏時間の調整に対して調整されるでしょう。「-」演算子での日付やタイムスタンプの引き算は値の間の(24時間の)日数と時間/分/秒を、同様に調整して返します。age関数は年、月、日、時間/分/秒をフィールド毎に引き算し、負のフィールドの値を調整します。以下の問い合わせは上の各方法の違いを説明する例です。例の結果はtimezone = 'US/Eastern'で生成されました。2つの日付の間には夏時間の変更があります。

```
SELECT EXTRACT(EPOCH FROM timestamptz '2013-07-01 12:00:00') -
       EXTRACT(EPOCH FROM timestamptz '2013-03-01 12:00:00');

結果: 10537200
SELECT (EXTRACT(EPOCH FROM timestamptz '2013-07-01 12:00:00') -
        EXTRACT(EPOCH FROM timestamptz '2013-03-01 12:00:00'))
        / 60 / 60 / 24;

結果: 121.958333333333
SELECT timestamptz '2013-07-01 12:00:00' - timestamptz '2013-03-01 12:00:00';

結果: 121 days 23:00:00
SELECT age(timestamptz '2013-07-01 12:00:00', timestamptz '2013-03-01 12:00:00');

結果: 4 mons
```

9.9.1. EXTRACT, date_part

```
EXTRACT(field FROM source)
```

extract関数は、日付/時刻の値から年や時などの部分フィールドを抽出します。sourceはtimestamp型、time型、またはinterval型の評価式でなければなりません(date型の式はtimestamp型にキャストされますので、同様に使用可能です)。fieldはsourceの値からどのフィールドを抽出するかを選択する識別子もしくは文字列です。extract関数はdouble precision型の値を返します。以下に有効なフィールド名を示します。

century

世紀

```
SELECT EXTRACT(CENTURY FROM TIMESTAMP '2000-12-16 12:21:13');
Result: 20
SELECT EXTRACT(CENTURY FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 21
```

当時の人々にはそのような意識はありませんでしたが、最初の世紀は0001-01-01 00:00:00 ADから始まります。この定義は全てのグレゴリオ暦を使用する国で適用されています。0という値の世紀はありません。-1世紀の次は1世紀です。この定義に納得できなければ、苦情をバチカンローマ聖ペテロ大聖堂のローマ法王に伝えてください。

day

timestamp値については、(月内の)日付フィールド(1–31)。interval値については日数。

```
SELECT EXTRACT(DAY FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 16

SELECT EXTRACT(DAY FROM INTERVAL '40 days 1 minute');
Result: 40
```

decade

年フィールドを10で割ったもの

```
SELECT EXTRACT(DECADE FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 200
```

dow

日曜日(0)から土曜日(6)までの曜日

```
SELECT EXTRACT(DOW FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 5
```

extract関数の曜日番号はto_char(...,'D')関数のそれとは異なる点に注意してください。

doy

年内での通算日数(1–365/366)

```
SELECT EXTRACT(DOY FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 47
```

epoch

timestamp with time zone型の値において、1970-01-01 00:00:00 UTCからの秒数（負の数の場合もあり）。dateとtimestamp型の値において、ローカルタイムの1970-01-01 00:00:00からの秒数。interval型の値ではその時間間隔における全体の秒数。

```
SELECT EXTRACT(EPOCH FROM TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40.12-08');
Result: 982384720.12

SELECT EXTRACT(EPOCH FROM INTERVAL '5 days 3 hours');
Result: 442800
```

to_timestampで経過秒数をタイムスタンプ値に変換することができます。

```
SELECT to_timestamp(982384720.12);
Result: 2001-02-17 04:38:40.12+00
```

hour

時フィールド（0–23）

```
SELECT EXTRACT(HOUR FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 20
```

isodow

月曜日（1）から日曜日（7）までの曜日

```
SELECT EXTRACT(ISODOW FROM TIMESTAMP '2001-02-18 20:38:40');
Result: 7
```

日曜日を除きdowと同一です。これはISO 8601曜日番号付けに一致します。

isoyear

日付に当てはまるISO 8601週番号年（intervalには適用できない）。

```
SELECT EXTRACT(ISOYEAR FROM DATE '2006-01-01');
Result: 2005
SELECT EXTRACT(ISOYEAR FROM DATE '2006-01-02');
Result: 2006
```

すべてのISO 8601週番号年は1月4日を含む週の月曜日から開始されます。従って、1月上旬、または12月下旬でISO年がグレゴリオ年と異なる可能性があります。より詳細はweekフィールドを参照してください。

このフィールドは8.3より前のPostgreSQLリリースでは有効でありません。

microseconds

端数部分も含む秒フィールドに、1,000,000を乗じた値。秒の整数部を含むことに注意。

```
SELECT EXTRACT(MICROSECONDS FROM TIME '17:12:28.5');
Result: 28500000
```

millennium

ミレニアム（千年期）

```
SELECT EXTRACT(MILLENNIUM FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 3
```

1900年代の年は第2ミレニアムです。第3ミレニアムは2001年1月1日から始まりました。

milliseconds

端数部分も含む秒フィールドに、1000を乗た値。秒の整数部を含むことに注意してください。

```
SELECT EXTRACT(MILLISECONDS FROM TIME '17:12:28.5');
Result: 28500
```

minute

分フィールド (0–59)

```
SELECT EXTRACT(MINUTE FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 38
```

month

timestamp型の値に対しては年内の月番号（1–12）。interval型の値に対しては月数を12で割った余り（0–11）。

```
SELECT EXTRACT(MONTH FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 2

SELECT EXTRACT(MONTH FROM INTERVAL '2 years 3 months');
Result: 3

SELECT EXTRACT(MONTH FROM INTERVAL '2 years 13 months');
Result: 1
```

quarter

その日付が含まれる年の四半期(1–4)

```
SELECT EXTRACT(QUARTER FROM TIMESTAMP '2001-02-16 20:38:40');
```

```
Result: 1
```

second

端数を含んだ秒フィールド

```
SELECT EXTRACT(SECOND FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 40

SELECT EXTRACT(SECOND FROM TIME '17:12:28.5');
Result: 28.5
```

timezone

秒単位のUTCからの時間帯オフセット。正の値はUTCより東の時間帯に対応し、負の値はUTCより西の時間帯に対応。(技術的に言えば、PostgreSQLはうるう秒を制御しないためUTCを使用していない。)

timezone_hour

時間帯オフセットの時の成分。

timezone_minute

時間帯オフセットの分の成分。

week

ISO 8601週番号。定義ではISO週は月曜日から始まり、その年の1月4日を含む週をその年の第1週としています。つまり、年の最初の木曜日がある週がその年の第1週となります。

ISO週番号システムでは、1月の早い日にちは前年の第52週もしくは第53週となることがあり、12月の遅い日にちには次年の最初の週の一部となることがあります。例えば、2005-01-01は2004年の第53週であり、2006-01-01は2005年の第52週の一部です、一方2012-12-31は2013年の第1週の一部となります。整合性のある結果を得るため、isoyearフィールドとweekを併用することを推奨します。

```
SELECT EXTRACT(WEEK FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 7
```

year

年フィールド。AD零年が存在しないことは忘れないでください。このためADの年からBCの年を減ずる時には注意が必要です。

```
SELECT EXTRACT(YEAR FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 2001
```

注記

入力値が+/-無限大の場合、extractは単調増加するフィールド(epoch、julian、year、isoyear、decade、century、millennium)に対し、+/-無限大を返します。その他のフィールドに対してはNULL

> が返されます。PostgreSQLの9.6より前のバージョンでは、入力が無限大のすべての場合に対してゼロを返していました。

extract関数は主に演算処理を意図しています。日付/時刻の値を表示する目的での書式については9.8を参照してください。

date_part関数は伝統的なIngres上で設計されたもので、標準SQLのextract関数と等価です。

```
date_part('field', source)
```

ここでfieldパラメータが名前ではなく文字列値である必要があることに注意してください。date_partで有効なフィールド名はextractと同じです。

```
SELECT date_part('day', TIMESTAMP '2001-02-16 20:38:40');
Result: 16

SELECT date_part('hour', INTERVAL '4 hours 3 minutes');
Result: 4
```

9.9.2. date_trunc

date_trunc関数は概念的に数値に対するtrunc関数と類似しています。

```
date_trunc(field, source [, time_zone ])
```

sourceは、データ型timestamp、timestamp with time zoneもしくはintervalの評価式です。(date型とtime型の値はそれぞれ自動的にtimestampもしくはintervalにキャストされます。) fieldは、入力値の値をどの精度で切り捨てるかを選択します。同様に戻り値はtimestamp、timestamp with time zoneもしくはinterval型で、指定した精度より下のすべてのフィールドがゼロに設定(日と月については1に設定)されます。

fieldの有効値には次のものがあります。

microseconds
milliseconds
second
minute
hour
day
week
month
quarter
year
decade

century
millennium

入力値がtimestamp with time zone型の値なら、特定の時間帯を考慮して切り捨てが行われます。たとえば、日を切り捨てると値はその時間帯での真夜中になります。デフォルトでは切り捨ては現在のTimeZoneの設定に従いますが、別の時間帯を指定することができるようにオプションのtime_zone引数が提供されています。時間帯名は8.5.3に記述されている方法で指定できます。

timestamp without time zoneあるいはintervalの入力を処理している間は時間帯は指定できません。これらは額面通りの値で扱われます。

例(現地タイムゾーンはAmerica/New_Yorkと仮定します):

```
SELECT date_trunc('hour', TIMESTAMP '2001-02-16 20:38:40');
Result: 2001-02-16 20:00:00

SELECT date_trunc('year', TIMESTAMP '2001-02-16 20:38:40');
Result: 2001-01-01 00:00:00

SELECT date_trunc('day', TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40+00');
Result: 2001-02-16 00:00:00-05

SELECT date_trunc('day', TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40+00', 'Australia/Sydney');
Result: 2001-02-16 08:00:00-05

SELECT date_trunc('hour', INTERVAL '3 days 02:47:33');
Result: 3 days 02:00:00
```

9.9.3. AT TIME ZONE

AT TIME ZONE構文を使用することにより、time stamp *without* time zoneからtime stamp *with* time zoneへ、あるいはtime with time zoneの値を異なる時間帯に変換することができます。表 9.33にその種類を示します。

表9.33 AT TIME ZONEの種類

演算子 説明 例
timestamp without time zone AT TIME ZONE zone → timestamp with time zone 与えられた時間帯なしタイムスタンプを指定された時間帯にあるとして時間帯ありタイムスタンプに変換します。 timestamp '2001-02-16 20:38:40' at time zone 'America/Denver' → 2001-02-17 03:38:40+00
timestamp with time zone AT TIME ZONE zone → timestamp without time zone 与えられた時間帯付き時刻を、時刻がその時間帯にあるものとして時間帯なしタイムスタンプに変換します。 timestamp with time zone '2001-02-16 20:38:40-05' at time zone 'America/Denver' → 2001-02-16 18:38:40

演算子 説明 例
time with time zone AT TIME ZONE zone → time with time zone 与えられた時刻*with* time zoneを新しい時間帯に変換します。判断するためのデータがないので、現在の有効なUTCオフセットを目的の時間帯のために使用します。 time with time zone '05:34:17-05' at time zone 'UTC' → 10:34:17+00

これらの式では、設定する時間帯zoneは、('America/Los_Angeles'のような)テキスト値、または(INTERVAL '-08:00'のような)時間間隔で指定することができます。テキストの場合、8.5.3に示した方法で時間帯名称を指定することができます。時間間隔を使うのはUTCからの固定のオフセットを持つ時間帯でのみ有用なので、一般的に非常に有用であるとは言えません。

以下に例を示します(現在の時間帯(TimeZone)をAmerica/Los_Angelesと想定しています)。

```
SELECT TIMESTAMP '2001-02-16 20:38:40' AT TIME ZONE 'America/Denver';
Result: 2001-02-16 19:38:40-08

SELECT TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40-05' AT TIME ZONE 'America/Denver';
Result: 2001-02-16 18:38:40

SELECT TIMESTAMP '2001-02-16 20:38:40' AT TIME ZONE 'Asia/Tokyo' AT TIME ZONE 'America/Chicago';
Result: 2001-02-16 05:38:40
```

最初の例は、時間帯のない値に時間帯を追加し、現在のTimeZone設定を使ってその値を表示します。2番目の例は、time stamp with time zone値を指定した時間帯に変換し、その値をwithout a time zoneで返しています。これは、TimeZone設定とは異なる値の格納と表示を可能にします。3番目の例は、東京時間をシカゴ時間に変換します。

関数timezone(zone, timestamp)は、SQL準拠の構文timestamp AT TIME ZONE zoneと等価です。

9.9.4. 現在の日付/時刻

PostgreSQLは、現在の日付時刻に関した値を返す多くの関数を提供します。これらの標準SQL関数はすべて、現在のトランザクションの開始時刻に基づいた値を返します。

```
CURRENT_DATE
CURRENT_TIME
CURRENT_TIMESTAMP
CURRENT_TIME(precision)
CURRENT_TIMESTAMP(precision)
LOCALTIME
LOCALTIMESTAMP
LOCALTIME(precision)
LOCALTIMESTAMP(precision)
```

CURRENT_TIMEおよびCURRENT_TIMESTAMP関数では、時間帯を伴う値を扱います。一方、LOCALTIMEおよびLOCALTIMESTAMP関数では、時間帯を伴わない値を扱います。

CURRENT_TIME、CURRENT_TIMESTAMP、LOCALTIME、およびLOCALTIMESTAMP関数では、精度のパラメータをオプションで取ることができ、それに合わせて秒フィールドの端数桁を丸める結果をもたらします。精度のパラメータがない場合、結果は使用可能な最大精度で出力されます。

例:

```
SELECT CURRENT_TIME;
Result: 14:39:53.662522-05

SELECT CURRENT_DATE;
Result: 2019-12-23

SELECT CURRENT_TIMESTAMP;
Result: 2019-12-23 14:39:53.662522-05

SELECT CURRENT_TIMESTAMP(2);
Result: 2019-12-23 14:39:53.66-05

SELECT LOCALTIMESTAMP;
Result: 2019-12-23 14:39:53.662522
```

これらの関数は現在のトランザクションの開始時刻を返すため、その値はトランザクションが実行されている間は変化しません。これは仕様であると考えられており、その意図は、単一のトランザクションが一貫性のある「現在」時刻の概念を持ち、同一トランザクション内の複数の変更が同一のタイムスタンプを持つようにすることにあります。

> **注記**
>
> 他のデータベースシステムでは、これらの値をより頻繁に増加させることがあります。

PostgreSQLはまた、関数を呼び出した時の実際の現在時刻や現在の文の開始時刻を返す関数も提供します。非標準SQLの時間関数の全一覧を以下に示します。

```
transaction_timestamp()
statement_timestamp()
clock_timestamp()
timeofday()
now()
```

transaction_timestamp()はCURRENT_TIMESTAMPと等価ですが、明確に何を返すかを反映する名前になっています。statement_timestamp()は現在の文の実行開始時刻を返すものです(より具体的にいうと、直前のコマンドメッセージをクライアントから受け取った時刻です)。statement_timestamp()およびtransaction_timestamp()はトランザクションの最初のコマンドでは同じ値を返しますが、その後に

引き続くコマンドでは異なる可能性があります。clock_timestamp()は実際の現在時刻を返しますので、その値は単一のSQLコマンドであっても異なります。timeofday()はPostgreSQLの歴史的な関数です。clock_timestamp()同様、実際の現在時刻を返しますが、timestamp with time zone型の値ではなく、整形されたtext文字列を返します。now()はtransaction_timestamp()と同じもので、伝統的なPostgreSQL関数です。

すべての日付/時刻型はまた、特殊なリテラル値 nowを受け付け、これは現在の日付と時刻（ここでも、トランザクションの開始時刻として解釈されます）を表します。したがって、下記の3つの実行結果は全て同じものとなります。

```
SELECT CURRENT_TIMESTAMP;
SELECT now();
SELECT TIMESTAMP 'now';  -- but see tip below
```

ヒント

たとえばテーブルの列にDEFAULT句を指定するのに、後から評価される値を指定する際に3番目の形式は使わないでください。システムはnowという定数を解析すると、すぐにそれをtimestampに変換するので、デフォルト値が必要が時には、テーブルが作成された時刻が使われます。最初の2つの形式は関数呼び出しなので、デフォルト値が使用されるまで評価されません。ですから、これらの関数は列の挿入時間をデフォルトとする、望ましい振舞いをします。（8.5.1.4も見てください。）

9.9.5. 遅延実行

以下の関数は、サーバプロセスの実行を遅延させるために使用可能です。

```
pg_sleep ( double precision )
pg_sleep_for ( interval )
pg_sleep_until ( timestamp with time zone )
```

pg_sleepは、seconds秒経過するまで、現在のセッションのプロセスを休止させます。secondsはdouble precision型の値です。そのため、小数単位で休止秒数を指定することができます。pg_sleep_forはintervalでより長い休止時間を指定する便利な関数です。pg_sleep_untilは特定の起床時刻が望まれる場合に便利な関数です。以下に例を示します。

```
SELECT pg_sleep(1.5);
SELECT pg_sleep_for('5 minutes');
SELECT pg_sleep_until('tomorrow 03:00');
```

注記

休止時間の有効な分解能はプラットフォームに依存します。0.01秒が一般的な値です。休止による遅延は最短で指定した時間と同じになります。サーバの負荷などが要因となり、より長くなる可能性が

あります。特に、pg_sleep_untilは指定した時刻ちょうどに起床する保証はありませんが、それより早く起床することはありません。

警告

pg_sleepまたはその亜種を呼び出す時、セッションが必要以上のロックを保持していないことを確実にしてください。さもないと、他のセッションが休止中のプロセスを待機しなければならないかもしれません。そのためシステム全体の速度が低下することになるかもしれません。

9.10. 列挙型サポート関数

列挙型（8.7で解説）に対し、特に列挙型の値をハードコーディングせず簡潔なプログラミングを可能にするいくつかの関数があります。それらの関数は表 9.34で一覧されています。例は以下のようにして列挙型が作成されていることを想定しています。

```
CREATE TYPE rainbow AS ENUM ('red', 'orange', 'yellow', 'green', 'blue', 'purple');
```

表9.34 列挙型サポート関数

関数 説明 例
enum_first (anyenum) → anyenum 入力列挙型の最初の値を返します。 enum_first(null::rainbow) → red
enum_last (anyenum) → anyenum 入力列挙型の最後の値を返します。 enum_last(null::rainbow) → purple
enum_range (anyenum) → anyarray 入力列挙型の全ての値を順序付き配列として返します。 enum_range(null::rainbow) → {red,orange,yellow,green,blue,purple}
enum_range (anyenum, anyenum) → anyarray 与えられた2つの列挙型値の範囲を、順序配列として返します。値は同一の列挙型に拠らなければなりません。1番目のパラメータがNULLの場合、結果は列挙型の最初の値から始まります。2番目のパラメータがNULLの場合、結果は列挙型の最後の値で終わります。 enum_range('orange'::rainbow, 'green'::rainbow) → {orange,yellow,green} enum_range(NULL, 'green'::rainbow) → {red,orange,yellow,green} enum_range('orange'::rainbow, NULL) → {orange,yellow,green,blue,purple}

enum_rangeの2引数の形式を除き、これらの関数は、渡された特定の値を無視することに注意してください。関数は宣言されたデータ型のみ配慮します。その型のNULLまたは特定の値を渡すことができ、同一の結果が得られます。例で使われているような直書きした型名に対してではなく、テーブル列もしくは関数引数にこれらの関数を適用することがより一般的です。

9.11. 幾何関数と演算子

point、box、lseg、line、path、polygon、およびcircle幾何データ型には、PostgreSQLが元々サポートしている関数と演算子が豊富に揃っています（表 9.35、表 9.36、および表 9.37を参照してください）。

表9.35 幾何データ演算子

演算子 説明 例
geometric_type + point → geometric_type 最初の引数の各々の点に二番目のpointの座標を加え、平行移動します。point、box、path、circleで利用可能です。 box '(1,1),(0,0)' + point '(2,0)' → (3,1),(2,0)
path + path → path 2つの開経路を結合します。（どちらかの経路が閉じていればNULLを返します。） path '[(0,0),(1,1)]' + path '[(2,2),(3,3),(4,4)]' → [(0,0),(1,1),(2,2),(3,3),(4,4)]
geometric_type - point → geometric_type 最初の引数の各々の点に二番目のpointの座標を減算し、平行移動します。point、box、path、circleで利用可能です。 box '(1,1),(0,0)' - point '(2,0)' → (-1,1),(-2,0)
geometric_type * point → geometric_type 最初の引数の各々の点に2番目のpointの座標を乗じます。（点を実数部と虚数部で表現する複素数として扱い、標準複素乗法を行います。）2番目のpointをベクトルと解釈すると、これはオブジェクトの大きさと原点からの距離をベクトルの長さで拡大し、x軸に対する角度分原点周りで反時計方向に回転させたものになります。point、box [a] path、circleで利用可能です。 path '((0,0),(1,0),(1,1))' * point '(3.0,0)' → ((0,0),(3,0),(3,3)) path '((0,0),(1,0),(1,1))' * point(cosd(45), sind(45)) → ((0,0),(0.7071067811865475,0.7071067811865475),(0,1.414213562373095))
geometric_type / point → geometric_type 最初の引数の各々の点を2番目のpointの座標で除算します。（点を実数部と虚数部で表現する複素数として扱い、標準複素除法を行います。2番目のpointをベクトルと解釈すると、これはオブジェクトの大きさと原点からの距離をベクトルの長さで縮小し、x軸に対する角度分原点周りで時計方向に回転させたものになります。point、box[a]、path、circleで利用可能です。 path '((0,0),(1,0),(1,1))' / point '(2.0,0)' → ((0,0),(0.5,0),(0.5,0.5)) path '((0,0),(1,0),(1,1))' / point(cosd(45), sind(45)) → ((0,0),(0.7071067811865476,-0.7071067811865476),(1.4142135623730951,0))
@-@ geometric_type → double precision 全長を計算します。lseg、pathで利用可能です。 @-@ path '[(0,0),(1,0),(1,1)]' → 2
@@ geometric_type → point 中心点を計算します。box、lseg、path、polygon、circleで利用可能です。 @@ box '(2,2),(0,0)' → (1,1)
# geometric_type → integer 点の数を返します。path、polygonで利用可能です。 # path '((1,0),(0,1),(-1,0))' → 3
geometric_type # geometric_type → point 交点を計算します。交点がなければNULLを返します。lseg、lineで利用可能です。

演算子 説明 例
lseg '[(0,0),(1,1)]' # lseg '[(1,0),(0,1)]' → (0.5,0.5)
box # box → box 2つの矩形の共通部を計算します。共通部がなければNULLを返します。 box '(2,2),(-1,-1)' # box '(1,1),(-2,-2)' → (1,1),(-1,-1)
geometric_type ## geometric_type → point 最初のオブジェクトから2番目のオブジェクトへの2番目のオブジェクト上の最近点を計算します。以下の型の対で利用可能です。(point, box), (point, lseg), (point, line), (lseg, box), (lseg, lseg), (lseg, line), (line, box), (line, lseg). point '(0,0)' ## lseg '[(2,0),(0,2)]' → (1,1)
geometric_type <-> geometric_type → double precision オブジェクト間の距離を計算します。7つのすべての幾何型、pointと他のすべての幾何型との組み合わせ、そして次の型の組み合わせで利用できます。(box、lseg)、(box、line)、(lseg、line)、(polygon、circle)(そして可換の組み合わせ)。 circle '<(0,0),1>' <-> circle '<(5,0),1>' → 3
geometric_type @> geometric_type → boolean 最初のオブジェクトは2番目のオブジェクトを含んでいるか？次の型の組み合わせで利用できます。(box、point)、(box、box)、(path、point)、(polygon、point)、(polygon、polygon)、(circle、point)、(circle、circle)。 circle '<(0,0),2>' @> point '(1,1)' → t
geometric_type <@ geometric_type → boolean 最初のオブジェクトは2番目のオブジェクトに含まれているかあるいはその上にあるか？次の型の組み合わせで利用できます。(point、box)、(point、lseg)、(point、line)、(point、path)、(point、polygon)、(point、circle)、(box、box)、(lseg、box)、(lseg、line)、(polygon、polygon)、(circle、circle)。 point '(1,1)' <@ circle '<(0,0),2>' → t
geometric_type && geometric_type → boolean これらのオブジェクトは重なり合っているか？（共通の点があれば真となります。）box、polygon、circleで利用可能です。 box '(1,1),(0,0)' && box '(2,2),(0,0)' → t
geometric_type << geometric_type → boolean 最初のオブジェクトは完全に2番目のオブジェクトの左にあるか？point、box、polygon、circleで利用可能です。 circle '<(0,0),1>' << circle '<(5,0),1>' → t
geometric_type >> geometric_type → boolean 最初のオブジェクトは完全に2番目のオブジェクトの右にあるか？point、box、polygon、circleで利用可能です。 circle '<(5,0),1>' >> circle '<(0,0),1>' → t
geometric_type &< geometric_type → boolean 最初のオブジェクトは2番目のオブジェクトの右にはみ出していないか？box、polygon、circleで利用可能です。 box '(1,1),(0,0)' &< box '(2,2),(0,0)' → t
geometric_type &> geometric_type → boolean 最初のオブジェクトは2番目のオブジェクトの左にはみ出していないか？box、polygon、circleで利用可能です。 box '(3,3),(0,0)' &> box '(2,2),(0,0)' → t
geometric_type <<| geometric_type → boolean 最初のオブジェクトは完全に2番目のオブジェクトの下にあるか？box、polygon、circleで利用可能です。 box '(3,3),(0,0)' <<| box '(5,5),(3,4)' → t

演算子 説明 例
geometric_type |>> geometric_type → boolean 最初のオブジェクトは完全に2番目のオブジェクトの上にあるか？box、polygon、circleで利用可能です。 box '(5,5),(3,4)' |>> box '(3,3),(0,0)' → t
geometric_type &<| geometric_type → boolean 最初のオブジェクトは2番目のオブジェクトの上にはみ出していないか？box、polygon、circleで利用可能です。 box '(1,1),(0,0)' &<| box '(2,2),(0,0)' → t
geometric_type |&> geometric_type → boolean 最初のオブジェクトは2番目のオブジェクトの下にはみ出していないか？box、polygon、circleで利用可能です。 box '(3,3),(0,0)' |&> box '(2,2),(0,0)' → t
box <^ box → boolean 最初のオブジェクトは2番目のオブジェクトの下か？（辺が接しているのを許容します） box '((1,1),(0,0))' <^ box '((2,2),(1,1))' → t
point <^ point → boolean 最初のオブジェクトは完全に2番目のオブジェクトの下か？（この演算子名は不適切です。<<|であるべきです。） point '(1,0)' <^ point '(1,1)' → t
box >^ box → boolean 最初のオブジェクトは2番目のオブジェクトの上か？（辺が接しているのを許容します） box '((2,2),(1,1))' >^ box '((1,1),(0,0))' → t
point >^ point → boolean 最初のオブジェクトは完全に2番目のオブジェクトの上か？（この演算子名は不適切です。|>>であるべきです。） point '(1,1)' >^ point '(1,0)' → t
geometric_type ?# geometric_type → boolean これらのオブジェクトは交差しているか？次の型の組み合わせで利用できます。(box、box)、(lseg、box)、(lseg、lseg)、(lseg、line)、(line、box)、(line、line)、(path、path)。 lseg '[(-1,0),(1,0)]' ?# box '(2,2),(-2,-2)' → t
?- line → boolean ?- lseg → boolean 線は水平か？ ?- lseg '[(-1,0),(1,0)]' → t
point ?- point → boolean 点は水平に並んでいるか？（つまりy座標が同じであるということです。） point '(1,0)' ?- point '(0,0)' → t
?| line → boolean ?| lseg → boolean 線は垂直か？ ?| lseg '[(-1,0),(1,0)]' → f
point ?| point → boolean 点は垂直に並んでいるか？（つまりx座標が同じであるということです。） point '(0,1)' ?| point '(0,0)' → t

演算子 説明 例
line ?- | line → boolean lseg ?- | lseg → boolean (指定された)2つの線は垂直か? lseg '[(0,0),(0,1)]' ?- | lseg '[(0,0),(1,0)]' → t
line ?|| line → boolean lseg ?|| lseg → boolean 線は平行か? lseg '[(-1,0),(1,0)]' ?|| lseg '[(-1,2),(1,2)]' → t
geometric_type ~= geometric_type → boolean オブジェクトは同じか? point、box、polygon、circleで利用可能です。 polygon '((0,0),(1,1))' ~= polygon '((1,1),(0,0))' → t

[a]boxをこれらの演算子で「回転」してもその頂点を動かすだけです。原点に対して矩形の辺は平行のままです。ですから矩形の大きさは保存されません。真の回転ならば保存します。

> **注意**
>
> 「同じを示す」~=演算子はpoint、box、polygon、およびcircle型に対し通常の等価概念を示すことに注意してください。これらのいくつかの型は=演算子を持ちますが、=は面積の等しさのみを比較します。これらの型で利用可能であれば、その他のスカラー比較演算子(<=など)は同様に面積を比較します。

> **注記**
>
> PostgreSQLの8.2より前では、包含演算子@>および<@はそれぞれ~および@という名前でした。これらの名前はまだ利用できますが、削除予定であり最終的にはなくなるでしょう。

表9.36 幾何データ型関数

関数 説明 例
area (geometric_type) → double precision 面積を計算します。box、path、circleで利用可能です。入力pathは閉じていなければなりません。さもなければNULLが返ります。またpathが自分自身と交わっていれば、結果は無意味なものになります。 area(box '(2,2),(0,0)') → 4
center (geometric_type) → point 中心点を計算します。box、circleで利用可能です。 center(box '(1,2),(0,0)') → (0.5,1)
diagonal (box) → lseg 矩形の対角線を線分として取り出します。(lseg(box)と同じです。)

関数 説明 例
diagonal(box '(1,2),(0,0)') → [(1,2),(0,0)]
diameter (circle) → double precision 円の直径を計算します。 diameter(circle '<(0,0),2>') → 4
height (box) → double precision 矩形の高さを計算します。 height(box '(1,2),(0,0)') → 2
isclosed (path) → boolean 閉経路か? isclosed(path '((0,0),(1,1),(2,0))') → t
isopen (path) → boolean 開経路か? isopen(path '[(0,0),(1,1),(2,0)]') → t
length (geometric_type) → double precision 全長を計算します。lseg、pathで利用可能です。 length(path '((-1,0),(1,0))') → 4
npoints (geometric_type) → integer 点の数を返します path、polygonで利用可能です。 npoints(path '[(0,0),(1,1),(2,0)]') → 3
pclose (path) → path 経路を閉じた状態に変換します。 pclose(path '[(0,0),(1,1),(2,0)]') → ((0,0),(1,1),(2,0))
popen (path) → path 経路を開いた状態に変換します。 popen(path '((0,0),(1,1),(2,0))') → [(0,0),(1,1),(2,0)]
radius (circle) → double precision 円の半径を計算します。 radius(circle '<(0,0),2>') → 2
slope (point, point) → double precision 2つの点で描いた直線の傾きを計算します。 slope(point '(0,0)', point '(2,1)') → 0.5
width (box) → double precision 矩形の幅を計算します。 width(box '(1,2),(0,0)') → 1

表9.37 幾何型変換関数

関数 説明 例
box (circle) → box 円に内接する矩形を計算します。 box(circle '<(0,0),2>') → (1.414213562373095,1.414213562373095),(-1.414213562373095,-1.414213562373095)
box (point) → box 点を空の矩形に変換します。 box(point '(1,0)') → (1,0),(1,0)
box (point, point) → box 2つの対角する点を矩形に変換します。 box(point '(0,1)', point '(1,0)') → (1,1),(0,0)
box (polygon) → box 多角形の外接矩形を計算します。 box(polygon '((0,0),(1,1),(2,0))') → (2,1),(0,0)
bound_box (box, box) → box 2つの矩形の外接矩形を計算します。 bound_box(box '(1,1),(0,0)', box '(4,4),(3,3)') → (4,4),(0,0)
circle (box) → circle 矩形を含む最小の円を計算します。 circle(box '(1,1),(0,0)') → <(0.5,0.5),0.7071067811865476>
circle (point, double precision) → circle 中心と半径から円を作成します。 circle(point '(0,0)', 2.0) → <(0,0),2>
circle (polygon) → circle 多角形を円に変換します。円の中心は多角形の点の位置の平均で、半径は中心から多角形の点の平均距離です。 circle(polygon '((0,0),(1,3),(2,0))') → <(1,1),1.6094757082487299>
line (point, point) → line 2点を通過する直線に変換します。 line(point '(-1,0)', point '(1,0)') → {0,-1,0}
lseg (box) → lseg 矩形の対角線を線分として取り出します。 lseg(box '(1,0),(-1,0)') → [(1,0),(-1,0)]
lseg (point, point) → lseg 2つの点から線分を作ります。 lseg(point '(-1,0)', point '(1,0)') → [(-1,0),(1,0)]
path (polygon) → path 同じ点のリストで多角形を閉経路に変換します。 path(polygon '((0,0),(1,1),(2,0))') → ((0,0),(1,1),(2,0))
point (double precision, double precision) → point 座標から点を作ります。

関数 説明 例
point(23.4, -44.5) → (23.4,-44.5)
point (box) → point 矩形の中心点を計算します。 point(box '(1,0),(-1,0)') → (0,0)
point (circle) → point 円の中心点を計算します。 point(circle '<(0,0),2>') → (0,0)
point (lseg) → point 線分の中心を計算します。 point(lseg '[(-1,0),(1,0)]') → (0,0)
point (polygon) → point 多角形の中心を計算します。(多角形の点の位置の平均です。) point(polygon '((0,0),(1,1),(2,0))') → (1,0.3333333333333333)
polygon (box) → polygon 矩形を4点の多角形に変換します。 polygon(box '(1,1),(0,0)') → ((0,0),(0,1),(1,1),(1,0))
polygon (circle) → polygon 円を12点の多角形に変換します。 polygon(circle '<(0,0),2>') → ((-2,0),(-1.7320508075688774,0.9999999999999999),(-1.0000000000000002,1.7320508075688772),(-1.2246063538223773e-16,2),(0.9999999999999996,1.7320508075688774),(1.732050807568877,1.0000000000000007),(2,2.4492127076447545e-16),(1.7320508075688776,-0.9999999999999994),(1.0000000000000009,-1.7320508075688767),(3.673819061467132e-16,-2),(-0.9999999999999987,-1.732050807568878),(-1.7320508075688767,-1.0000000000000009))
polygon (integer, circle) → polygon 円をn点の多角形に変換します。 polygon(4, circle '<(3,0),1>') → ((2,0),(3,1),(4,1.2246063538223773e-16),(3,-1))
polygon (path) → polygon 同じ点のリストで閉経路を多角形に変換します。 polygon(path '((0,0),(1,1),(2,0))') → ((0,0),(1,1),(2,0))

あたかもpointは添字0、1を有する配列であるかのように、pointの2つの構成要素にアクセスすることができます。例えば、t.pがpoint列の場合、SELECT p[0] FROM tという式でX座標を抽出できます。また、UPDATE t SET p[1] = ...でY座標を変更できます。同様に、box型またはlseg型の値も、2つのpoint型の値の配列のように扱えます。

9.12. ネットワークアドレス関数と演算子

IPネットワークアドレス型であるcidrとinetは表 9.1に示す通常の比較演算子に加え、表 9.38と表 9.39で示す特定目的の演算子と関数をサポートしています。

すべてのcidr値は暗黙的にinetにキャストできます。ですから以下で示すinetで使える演算子と関数はcidrでも使えます。(inetとcidr用の別々の関数があるのは、この両者で振る舞いが異なっているべきである場合があるからです。) またinet値をcidrにキャストすることが許されています。これが行われると、ネットマスクの右側のすべてのビットは有効なcidr値を作るために暗黙的にゼロになります。

表9.38 IPアドレス演算子

演算子 説明 例
inet << inet → boolean サブネットが完全にサブネットに含まれているか？この演算子と次の4つの演算子はサブネットの包含をテストします。それらは2つのアドレスのネットワーク部分だけを考慮し(ネットマスクの右のビットは無視されます)、ネットワークが他のネットワークと同一か、あるいはサブネットであるかどうかを決定します。 inet '192.168.1.5' << inet '192.168.1/24' → t inet '192.168.0.5' << inet '192.168.1/24' → f inet '192.168.1/24' << inet '192.168.1/24' → f
inet <<= inet → boolean サブネットがサブネットに含まれているか、あるいは同じか？ inet '192.168.1/24' <<= inet '192.168.1/24' → t
inet >> inet → boolean サブネットが完全にサブネットを含んでいるか？ inet '192.168.1/24' >> inet '192.168.1.5' → t
inet >>= inet → boolean サブネットがサブネットを含んでいるか、あるいは同じか？ inet '192.168.1/24' >>= inet '192.168.1/24' → t
inet && inet → boolean サブネットが他を含んでいるか、あるいは同じか？ inet '192.168.1/24' && inet '192.168.1.80/28' → t inet '192.168.1/24' && inet '192.168.2.0/28' → f
~ inet → inet ビット否定を計算します。 ~ inet '192.168.1.6' → 63.87.254.249
inet & inet → inet ビット積を計算します。 inet '192.168.1.6' & inet '0.0.0.255' → 0.0.0.6
inet | inet → inet ビット和を計算します。 inet '192.168.1.6' | inet '0.0.0.255' → 192.168.1.255
inet + bigint → inet オフセットをアドレスに加算します。 inet '192.168.1.6' + 25 → 192.168.1.31
bigint + inet → inet オフセットをアドレスに加算します。

演算子 説明 例
200 + inet '::ffff:fff0:1' → ::ffff:255.240.0.201
inet - bigint → inet アドレスからオフセットを減算します。 inet '192.168.1.43' - 36 → 192.168.1.7
inet - inet → bigint 2つのアドレスの差を計算します。 inet '192.168.1.43' - inet '192.168.1.19' → 24 inet '::1' - inet '::ffff:1' → -4294901760

表9.39 IPアドレス関数

関数 説明 例
abbrev (inet) → text 表示用テキスト省略形を作成します。(結果はinet出力関数が生成するものと同じです。明示的にtextにキャストしたもの(歴史的な理由でネットマスク部分が抑止されていません)と比べると「省略」されているだけです。 abbrev(inet '10.1.0.0/32') → 10.1.0.0
abbrev (cidr) → text 表示用テキスト省略形を作成します。(ネットマスクの右側のすべてのゼロオクテットを削除することによって省略形にします。表 8.22に他の例があります。) abbrev(cidr '10.1.0.0/16') → 10.1/16
broadcast (inet) → inet アドレスのネットワーク部のネットワークブロードキャストアドレスを計算します。 broadcast(inet '192.168.1.5/24') → 192.168.1.255/24
family (inet) → integer アドレスファミリーを返します。IPv4なら4で、IPv6なら6です。 family(inet '::1') → 6
host (inet) → text IPアドレスをテキストとして返します。ネットマスクは無視されます。 host(inet '192.168.1.0/24') → 192.168.1.0
hostmask (inet) → inet アドレスのネットワークに対するホストマスクを計算します。 hostmask(inet '192.168.23.20/30') → 0.0.0.3
inet_merge (inet, inet) → cidr 与えられたネットワークを両方含む最小のネットワークを計算します。 inet_merge(inet '192.168.1.5/24', inet '192.168.2.5/24') → 192.168.0.0/22
inet_same_family (inet, inet) → boolean アドレスが同じIPファミリーに属しているかどうかを判定します。 inet_same_family(inet '192.168.1.5/24', inet '::1') → f

関数 説明 例
masklen (inet) → integer ネットマスクのビット長を返します。 masklen(inet '192.168.1.5/24') → 24
netmask (inet) → inet アドレスのネットワークに対するネットワークマスクを計算します。 netmask(inet '192.168.1.5/24') → 255.255.255.0
network (inet) → cidr ネットマスクの右側をすべてゼロにしてアドレスのネットワーク部を返します。(これは値をcidrにキャストするのと同じです。) network(inet '192.168.1.5/24') → 192.168.1.0/24
set_masklen (inet, integer) → inet ネットマスク長をinet値に設定します。アドレスの部分は変更しません。 set_masklen(inet '192.168.1.5/24', 16) → 192.168.1.5/16
set_masklen (cidr, integer) → cidr ネットマスク長をcidr値に設定します。新しいネットマスクの右側のアドレスビットは0に設定されます。 set_masklen(cidr '192.168.1.0/24', 16) → 192.168.0.0/16
text (inet) → text 省略形ではないIPアドレスとネットマスク長をテキストとして返します。(これはtextに明示的にキャストするのと同じ効果があります。) text(inet '192.168.1.5') → 192.168.1.5/32

ヒント

関数abbrev、host、およびtext、は主として、代替のIPアドレスの整形表示を提供する目的のものです。

MACアドレス型であるmacaddrとmacaddr8は、表 9.1で示す通常の比較演算子と表 9.40で示す特定目的のための関数をサポートします。加えて上記のIPアドレス用に示したのと同様に、ビットごとの論理演算子~、&、|(NOT、AND、OR)をサポートします。

表9.40 MACアドレス関数

関数 説明 例
trunc (macaddr) → macaddr アドレスの終わりの3バイトをゼロに設定します。残りの前の部分は(PostgreSQLには含まれないデータを使って)特定の製造業者に関連付けることもできます。 trunc(macaddr '12:34:56:78:90:ab') → 12:34:56:00:00:00
trunc (macaddr8) → macaddr8 アドレスの終わりの5バイトをゼロに設定します。残りの前の部分は(PostgreSQLには含まれないデータを使って)特定の製造業者に関連付けることもできます。

関数 説明 例
trunc(macaddr8 '12:34:56:78:90:ab:cd:ef') → 12:34:56:00:00:00:00:00
macaddr8_set7bit (macaddr8) → macaddr8 7番目のビットを1にし、修正EUI-64と呼ばれる形式にして、IPv6アドレスに含められるようにします。 macaddr8_set7bit(macaddr8 '00:34:56:ab:cd:ef') → 02:34:56:ff:fe:ab:cd:ef

9.13. テキスト検索関数と演算子

表 9.41、表 9.42および表 9.43は全文検索用に提供されている関数と演算子を要約しています。PostgreSQLのテキスト検索機能の詳細は第12章を参照してください。

表9.41 テキスト検索演算子

演算子 説明 例
tsvector @@ tsquery → boolean tsquery @@ tsvector → boolean tsvectorがtsqueryの条件に合うか？（引数は任意の順で与えることができます。） to_tsvector('fat cats ate rats') @@ to_tsquery('cat & rat') → t
text @@ tsquery → boolean テキスト文字列はto_tsvector()の暗黙的な呼び出し後にtsqueryの条件に合うか？ 'fat cats ate rats' @@ to_tsquery('cat & rat') → t
tsvector @@@ tsquery → boolean tsquery @@@ tsvector → boolean @@に対する廃止予定の同義語です。 to_tsvector('fat cats ate rats') @@@ to_tsquery('cat & rat') → t
tsvector || tsvector → tsvector 2つのtsvectorを連結します。両方の入力が語彙素の位置を含んでいるなら2番目の入力の位置はそれにしたがって調整されます。 'a:1 b:2'::tsvector || 'c:1 d:2 b:3'::tsvector → 'a':1 'b':2,5 'c':3 'd':4
tsquery && tsquery → tsquery 2つのtsqueryの論理積を取り、両方の入力問い合わせにマッチする文書にマッチする問い合わせを生成します。 'fat | rat'::tsquery && 'cat'::tsquery → ('fat' | 'rat') & 'cat'
tsquery || tsquery → tsquery 2つのtsqueryの論理和を取り、どちらかの入力問い合わせにマッチする文書にマッチする問い合わせを生成します。 'fat | rat'::tsquery || 'cat'::tsquery → 'fat' | 'rat' | 'cat'
!! tsquery → tsquery tsqueryの否定を取り、入力問い合わせにマッチしない文書にマッチする問い合わせを生成します。 !! 'cat'::tsquery → !'cat'
tsquery <-> tsquery → tsquery

演算子 説明 例
2つの入力問い合わせが連続する語彙素にマッチする場合にマッチする語句問い合わせを作成します。 to_tsquery('fat') <-> to_tsquery('rat') → 'fat' <-> 'rat'
tsquery @> tsquery → boolean 最初のtsqueryは2番目を含んでいるか？（これは結合演算子を無視して、単に一方の問い合わせ中のすべての語彙素が他方に現れるかどうかだけを考慮します。） 'cat'::tsquery @> 'cat & rat'::tsquery → f
tsquery <@ tsquery → boolean 最初のtsqueryは2番目に含まれているか？（これは結合演算子を無視して、単に一方の問い合わせ中のすべての語彙素が他方に現れるかどうかだけを考慮します。） 'cat'::tsquery <@ 'cat & rat'::tsquery → t 'cat'::tsquery <@ '!cat & rat'::tsquery → t

表に示された演算子に加え、表 9.1で示す通常の比較演算子型tsvectorおよびtsqueryに対して利用できます。これらはテキスト検索に対してそれほど有用ではありませんが、例えばこれらの型の列に一意インデックスを作成することを可能にします。

表9.42 テキスト検索関数

関数 説明 例
array_to_tsvector (text[]) → tsvector 語彙素の配列をtsvectorに変換します。与えられた文字列は特に処理は施されずにそのまま利用されます。 array_to_tsvector('{fat,cat,rat}'::text[]) → 'cat' 'fat' 'rat'
get_current_ts_config ( ) → regconfig （default_text_search_configで設定された）現在のテキスト検索設定のOIDを返します。 get_current_ts_config() → english
length (tsvector) → integer tsvectorにある語彙素の数を返します。 length('fat:2,4 cat:3 rat:5A'::tsvector) → 3
numnode (tsquery) → integer tsqueryにある語彙素の数と演算子の数の和を返します。 numnode('(fat & rat) | cat'::tsquery) → 5
plainto_tsquery ([config regconfig,] query text) → tsquery 指定されたデフォルト設定にしたがって単語を正規化してテキストをtsqueryに変換します。文字列中の句読点はすべて無視されます。（句読点は問い合わせ演算子を決定しません。）結果の問い合わせはテキスト中の非ストップワードをすべて含む文書にマッチします。 plainto_tsquery('english', 'The Fat Rats') → 'fat' & 'rat'
phraseto_tsquery ([config regconfig,] query text) → tsquery 指定されたデフォルト設定にしたがって単語を正規化してテキストをtsqueryに変換します。文字列中の句読点はすべて無視されます。（句読点は問い合わせ演算子を決定しません。）結果の問い合わせはテキスト中の非ストップワードをすべて含む句にマッチします。

関数 説明 例
phraseto_tsquery('english', 'The Fat Rats') → 'fat' <-> 'rat' phraseto_tsquery('english', 'The Cat and Rats') → 'cat' <2> 'rat'
websearch_to_tsquery ([config regconfig,] query text) → tsquery 指定されたデフォルト設定にしたがって単語を正規化してテキストをtsqueryに変換します。引用符で囲まれた一連の語は句の検査に変換されます。「or」はOR演算子を生成するものとして扱われ、ダッシュはNOT演算子として扱われます。それ以外の句読点は無視されます。これにより通常のweb検索ツールに近い振る舞いをします。 websearch_to_tsquery('english', '"fat rat" or cat dog') → 'fat' <-> 'rat' | 'cat' & 'dog'
querytree (tsquery) → text tsqueryのインデックス付可能な部分の表現を生成します。空あるいはTはインデックス付できる部分が無い問い合わせであることを意味します。 querytree('foo & ! bar'::tsquery) → 'foo'
setweight (vector tsvector, weight "char") → tsvector vectorの各要素に指定したweightを割り当てます。 setweight('fat:2,4 cat:3 rat:5B'::tsvector, 'A') → 'cat':3A 'fat':2A,4A 'rat':5A
setweight (vector tsvector, weight "char", lexemes text[]) → tsvector vectorの各要素にlexemesで列挙したweightを割り当てます。 setweight('fat:2,4 cat:3 rat:5,6B'::tsvector, 'A', '{cat,rat}') → 'cat':3A 'fat':2,4 'rat':5A,6A
strip (tsvector) → tsvector 位置と重みをtsvectorから削除します。 strip('fat:2,4 cat:3 rat:5A'::tsvector) → 'cat' 'fat' 'rat'
to_tsquery ([config regconfig,] query text) → tsquery 指定されたデフォルト設定にしたがって単語を正規化してテキストをtsqueryに変換します。単語は有効なtsquery演算子と組み合わされていなければなりません。 to_tsquery('english', 'The & Fat & Rats') → 'fat' & 'rat'
to_tsvector ([config regconfig,] document text) → tsvector 指定されたデフォルト設定にしたがって単語を正規化してテキストをtsvectorに変換します。位置情報が結果に含まれます。 to_tsvector('english', 'The Fat Rats') → 'fat':2 'rat':3
to_tsvector ([config regconfig,] document json) → tsvector to_tsvector ([config regconfig,] document jsonb) → tsvector 指定されたデフォルト設定にしたがって正規化してJSON文書中の文字列値をtsvectorに変換します。そして結果は文書中の順序にしたがって結合されます。位置情報は、あたかも文字列値の各々の対の間にストップワードが存在するかのように生成されます。（入力がjsonbの場合、JSONオブジェクトのフィールドの「ドキュメント順」は実装依存であることに注意してください。例中の差異を見てください。） to_tsvector('english', '{"aa": "The Fat Rats", "b": "dog"}'::json) → 'dog':5 'fat':2 'rat':3 to_tsvector('english', '{"aa": "The Fat Rats", "b": "dog"}'::jsonb) → 'dog':1 'fat':4 'rat':5
json_to_tsvector ([config regconfig,] document json, filter jsonb) → tsvector jsonb_to_tsvector ([config regconfig,] document jsonb, filter jsonb) → tsvector filterによって要求された項目をJSON文書から検索し、指定されたデフォルト設定にしたがって正規化してtsvectorに変換します。そして結果は文書中の順序にしたがって結合されます。位置情報は、あたかも文字列値の各々の対の間にストップワードが存在するかのように生成されます。（入力がjsonbの場合、JSONオブジェクトのフィールドの「ドキュメント順」は実装

関数 説明 例
依存であることに注意してください。例中の差異を見てください。) filterは0個以上の以下のキーワードを含むjsonbの配列でなければなりません: "string" (すべての文字列値を含めます)、"numeric" (すべての数値を含めます)、"boolean" (すべての論理値を含めます)、"key" (すべてのキーを含めます)、"all" (すべてを含めます)。特別な場合として、filterはこれらのキーワードのどれかである単純なJSON値とすることもできます。 json_to_tsvector('english', '{"a": "The Fat Rats", "b": 123}'::json, '["string", "numeric"]') → '123':5 'fat':2 'rat':3 json_to_tsvector('english', '{"cat": "The Fat Rats", "dog": 123}'::json, '"all"') → '123':9 'cat':1 'dog':7 'fat':4 'rat':5
ts_delete (vector tsvector, lexeme text) → tsvector vectorから与えられたlexemeを削除します。 ts_delete('fat:2,4 cat:3 rat:5A'::tsvector, 'fat') → 'cat':3 'rat':5A
ts_delete (vector tsvector, lexemes text[]) → tsvector vectorからlexemes中のすべての語彙素を削除します。 ts_delete('fat:2,4 cat:3 rat:5A'::tsvector, ARRAY['fat','rat']) → 'cat':3
ts_filter (vector tsvector, weights "char"[]) → tsvector vectorからweightsを持つ要素だけを検索します。 ts_filter('fat:2,4 cat:3b,7c rat:5A'::tsvector, '{a,b}') → 'cat':3B 'rat':5A
ts_headline ([config regconfig,] document text, query tsquery [, options text]) → text document中のqueryにマッチするものを省略形で表示します。documentはtsvectorではなくて生のテキストでなければなりません。問い合わせのマッチ処理を行う前に、指定した、あるいはデフォルトの設定にしたがって単語が正規化されます。12.3.4にこの関数の使い方が記述されています。可能なoptionsについても言及されています。 ts_headline('The fat cat ate the rat.', 'cat') → The fat cat ate the rat.
ts_headline ([config regconfig,] document json, query tsquery [, options text]) → text ts_headline ([config regconfig,] document jsonb, query tsquery [, options text]) → text JSON document中に出現する文字列値にqueryがマッチしたものを省略形で表示します。詳細は12.3.4を見てください。 ts_headline('{"cat":"raining cats and dogs"}'::jsonb, 'cat') → {"cat": "raining cats and dogs"}
ts_rank ([weights real[],] vector tsvector, query tsquery [, normalization integer]) → real vectorがqueryにどれほどマッチするかのスコアを計算します。詳細は12.3.3を見てください。 ts_rank(to_tsvector('raining cats and dogs'), 'cat') → 0.06079271
ts_rank_cd ([weights real[],] vector tsvector, query tsquery [, normalization integer]) → real 被覆密度アルゴリズムを用いてvectorがqueryにどれほどマッチするかのスコアを計算します。詳細は12.3.3を見てください。 ts_rank_cd(to_tsvector('raining cats and dogs'), 'cat') → 0.1
ts_rewrite (query tsquery, target tsquery, substitute tsquery) → tsquery query中に出現するtargetをsubstituteに置き換えます。詳細は12.4.2.1を見てください。 ts_rewrite('a & b'::tsquery, 'a'::tsquery, 'foo|bar'::tsquery) → 'b' & ('foo' | 'bar')
ts_rewrite (query tsquery, select text) → tsquery SELECTを実行して取得したターゲットと代替を使用してqueryの一部を置き換えます。詳細は12.4.2.1を見てください。 SELECT ts_rewrite('a & b'::tsquery, 'SELECT t,s FROM aliases') → 'b' & ('foo' | 'bar')
tsquery_phrase (query1 tsquery, query2 tsquery) → tsquery

関数 説明 例
連続する語彙素でquery1とquery2のマッチを検索する語句問い合わせを作成します。(<->演算子と同じです。) tsquery_phrase(to_tsquery('fat'), to_tsquery('cat')) → 'fat' <-> 'cat'
tsquery_phrase (query1 tsquery, query2 tsquery, distance integer) → tsquery 語彙素が正確にdistanceだけ離れているquery1とquery2へのマッチを検索する語句問い合わせを作成します。 tsquery_phrase(to_tsquery('fat'), to_tsquery('cat'), 10) → 'fat' <10> 'cat'
tsvector_to_array (tsvector) → text[] tsvectorを語彙素の配列に変換します。 tsvector_to_array('fat:2,4 cat:3 rat:5A'::tsvector) → {cat,fat,rat}
unnest (tsvector) → setof record (lexeme text, positions smallint[], weights text) 1行につき1語彙素でtsvectorを行の集合に変換します。 select * from unnest('cat:3 fat:2,4 rat:5A'::tsvector) → lexeme | positions | weights --------+-----------+--------- cat | {3} | {D} fat | {2,4} | {D,D} rat | {5} | {A}

注記

オプションのregconfig引数を受け付ける全てのテキスト検索関数は、その引数が省略された場合default_text_search_configで指定された設定を使用します。

表 9.43の関数は、日常のテキスト検索操作では通常使用されないので、別の表にしました。これらは主に新しいテキスト検索設定の開発およびデバッグに役立ちます。

表9.43 テキスト検索デバッグ関数

関数 説明 例
ts_debug ([config regconfig,] document text) → setof record (alias text, description text, token text, dictionaries regdictionary[], dictionary regdictionary, lexemes text[]) 指定した、あるいはデフォルトの設定にしたがってdocumentから正規化されたトークンを取り出し、各トークンがどのように処理されたかの情報を返します。詳細は12.8.1を見てください。 ts_debug('english', 'The Brightest supernovaes') → (asciiword,"Word, all ASCII",The,{english_stem},english_stem,{}) ...
ts_lexize (dict regdictionary, token text) → text[] 入力トークンが辞書にあれば代替の語彙素の配列、辞書にあるがストップワードである場合には空の配列、未知の単語ならNULLを返します。詳細は12.8.3を見てください。 ts_lexize('english_stem', 'stars') → {star}

関数 説明 例
ts_parse (parser_name text, document text) → setof record (tokid integer, token text) 名前で指定したパーサを使ってdocumentからトークンを取り出します。詳細は12.8.2を見てください。 ts_parse('default', 'foo - bar') → (1,foo) ...
ts_parse (parser_oid oid, document text) → setof record (tokid integer, token text) OIDで指定されたパーサを使ってdocumentからトークンを取り出します。詳細は12.8.2を見てください。 ts_parse(3722, 'foo - bar') → (1,foo) ...
ts_token_type (parser_name text) → setof record (tokid integer, alias text, description text) 名前で指定したパーサが認識できるトークンの型を記述するテーブルを返します。詳細は12.8.2を見てください。 ts_token_type('default') → (1,asciiword,"Word, all ASCII") ...
ts_token_type (parser_oid oid) → setof record (tokid integer, alias text, description text) OIDで指定したパーサが認識できるトークンの型を記述するテーブルを返します。詳細は12.8.2を見てください。 ts_token_type(3722) → (1,asciiword,"Word, all ASCII") ...
ts_stat (sqlquery text [, weights text]) → setof record (word text, ndoc integer, nentry integer) 単一のtsvector列を返さなければならないsqlqueryを実行し、データに含まれる別個の語彙素に関する統計情報を返します。詳細は12.4.4をご覧ください。 ts_stat('SELECT vector FROM apod') → (foo,10,15) ...

9.14. UUID関数

PostgreSQL includes one function to generate a UUID:

```
gen_random_uuid () → uuid
```

この関数はバージョン4(ランダム)UUIDを返します。これはUUIDのもっとも一般的な使い方で、大抵のアプリケーションに適しています。

uuid-osspモジュールはUUID生成のための他の標準アルゴリズムを実装した追加の関数を提供します。

PostgreSQLは表 9.1で示すUUIDのための通常の比較演算子を提供しています。

9.15. XML関数

この節で説明される関数および擬似関数式は、xml型の値に対して機能します。xml型についての情報は8.13を参照してください。xml型のやりとりを変換するxmlparseおよびxmlserialize擬似関数式はこの節ではなく、そこに記載されています。

これらの関数の大半はPostgreSQLがconfigure --with-libxmlでビルドされていることを必要としています。

9.15.1. XML内容の生成

SQLデータからXML内容を生成するために関数と擬似関数式の一式が提供されています。そのようなものとして、クライアントアプリケーションが問い合わせ結果を処理のためXML文書に書式化するのにこれらは特に適しています。

9.15.1.1. xmlcomment

```
xmlcomment ( text ) → xml
```

関数xmlcommentは指定のテキストを内容とするXMLコメントを含んだXML値を作成します。テキストは「--」を含むこと、または「-」で終結することはできません。さもないと結果として構築されるXMLコメントは有効になりません。引数がNULLならば結果もNULLになります。

例:

```
SELECT xmlcomment('hello');

  xmlcomment
--------------
 <!--hello-->
```

9.15.1.2. xmlconcat

```
xmlconcat ( xml [, ...] ) → xml
```

関数xmlconcatは、個々のXML値のリストを結合し、XMLの内容断片を含む単一の値を作成します。NULL値は削除され、NULL以外の引数が存在しないときのみ結果はNULLになります。

例:

```
SELECT xmlconcat('<abc/>', '<bar>foo</bar>');

      xmlconcat
----------------------
 <abc/><bar>foo</bar>
```

XML宣言が提示されている場合は次のように組み合わされます。全ての引数の値が同一のXML version宣言を持っていれば、そのversionが結果に使用されます。さもなければversionは使用されません。全ての引数の値でstandaloneの宣言値が「yes」であれば、その値が結果に使用されます。全ての引数の値にstandalone宣言値があり、その中で1つでも「no」がある場合、それが結果に使用されます。それ以外の場合は、結果はstandalone宣言を持ちません。standalone宣言を必要とするが、standalone宣言がないと

いう結果になった場合には、version 1.0のversion宣言が使用されます。これはXMLがXML宣言においてversion宣言を含むことを要求するためです。encoding宣言は無視され、全ての場合で削除されます。

例:

```
SELECT xmlconcat('<?xml version="1.1"?><foo/>', '<?xml version="1.1" standalone="no"?><bar/>');

             xmlconcat
-----------------------------------
 <?xml version="1.1"?><foo/><bar/>
```

9.15.1.3. xmlelement

```
xmlelement ( NAME name [, XMLATTRIBUTES ( attvalue [ AS attname ] [, ...] ) ] [, content
 [, ...]] ) → xml
```

xmlelement式は与えられた名前、属性、および内容を持つXML要素を生成します。構文中に示すnameとattname項目は単純な識別子で値ではありません。attvalueとcontent項目は式で、PostgreSQLの任意のデータ型を出力できます。XMLATTRIBUTES中の引数はXML要素の属性を生成します。content値は結合して内容を構成します。

例:

```
SELECT xmlelement(name foo);

 xmlelement
------------
 <foo/>

SELECT xmlelement(name foo, xmlattributes('xyz' as bar));

    xmlelement
------------------
 <foo bar="xyz"/>

SELECT xmlelement(name foo, xmlattributes(current_date as bar), 'cont', 'ent');

              xmlelement
-------------------------------------
 <foo bar="2007-01-26">content</foo>
```

有効なXML名ではない要素名と属性名は、シーケンス_xHHHH_により障害となる文字を置換することでエスケープされます。ここで、HHHHは16進数によるその文字のUnicode文字コード番号です。例をあげます。

```
SELECT xmlelement(name "foo$bar", xmlattributes('xyz' as "a&b"));
```

```
            xmlelement
--------------------------------
 <foo_x0024_bar a_x0026_b="xyz"/>
```

属性値が列参照の場合、明示的な属性名を指定する必要はありません。この場合、デフォルトで列名が属性名として使用されます。その他の場合には、属性は明示的な名前で与えられなければなりません。従って、以下の例は有効です。

```
CREATE TABLE test (a xml, b xml);
SELECT xmlelement(name test, xmlattributes(a, b)) FROM test;
```

しかし、以下の例は有効ではありません。

```
SELECT xmlelement(name test, xmlattributes('constant'), a, b) FROM test;
SELECT xmlelement(name test, xmlattributes(func(a, b))) FROM test;
```

もし要素内容が指定されればそのデータ型に従って書式化されます。もし内容そのものがxml型であれば、複合XML文書が構築されます。例をあげます。

```
SELECT xmlelement(name foo, xmlattributes('xyz' as bar),
                            xmlelement(name abc),
                            xmlcomment('test'),
                            xmlelement(name xyz));

                  xmlelement
----------------------------------------------
 <foo bar="xyz"><abc/><!--test--><xyz/></foo>
```

そのほかの型の内容は有効なXML文字データにフォーマットされます。これは特に文字<、>、および&がエンティティに変換されることを意味します。バイナリデータ(データ型はbytea)は、設定パラメータxmlbinaryの設定にしたがって、base64もしくは16進符号化方式で表現されます。個々のデータ型に対する特定の動作は、XMLスキーマ仕様でのSQLおよびPostgreSQLデータ型に調整するため発展すると期待されます。その時点で記述がより詳細になるでしょう。

9.15.1.4. xmlforest

```
xmlforest ( content [ AS name ] [, ...] ) → xml
```

xmlforest式は与えられた名前と内容を使用し、要素のXMLフォレスト(シーケンス)を生成します。xmlelementでは、各nameは単純な識別子でなければなりませんが、content式はどんな型のデータも持つことができます。

例:

```
SELECT xmlforest('abc' AS foo, 123 AS bar);
```

```
          xmlforest
-----------------------------
 <foo>abc</foo><bar>123</bar>


SELECT xmlforest(table_name, column_name)
FROM information_schema.columns
WHERE table_schema = 'pg_catalog';

                                xmlforest
----------------------------------------------------------------------
 <table_name>pg_authid</table_name><column_name>rolname</column_name>
 <table_name>pg_authid</table_name><column_name>rolsuper</column_name>
 ...
```

第2の例に見られるように、内容の値が列参照の場合、要素名は省略可能です。この時は、列名がデフォルトで使用されます。そうでない時は、名前が指定されなければなりません。

有効なXML名ではない要素名は上のxmlelementで説明した通りエスケープされます。同様にして、既にxml型であるものを除き、内容データは有効なXML内容になるようにエスケープされます。

XMLフォレストは2つ以上の要素からなる場合、有効なXML文書ではないことに注意してください。したがって、xmlelement内にxmlforest式をラップすることが有用なことがあります。

9.15.1.5. xmlpi

```
xmlpi ( NAME name [, content ] ) → xml
```

xmlpi式はXML処理命令を作成します。xmlelementでは、各nameは単純な識別子でなければなりませんが、content式はどんな型のデータも持つことができます。contentが存在するときは、それは?>という文字シーケンスを含んではいけません。

例:

```
SELECT xmlpi(name php, 'echo "hello world";');

            xmlpi
-----------------------------
 <?php echo "hello world";?>
```

9.15.1.6. xmlroot

```
xmlroot ( xml, VERSION {text|NO VALUE} [, STANDALONE {YES|NO|NO VALUE} ] ) → xml
```

xmlroot式はXML値のルートノードの属性を変更します。versionが指定されていると、ルートノードのversion宣言での値を変更し、standalone設定が指定されていると、ルートノードのstandalone宣言での値を変更します。

```
SELECT xmlroot(xmlparse(document '<?xml version="1.1"?><content>abc</content>'),
               version '1.0', standalone yes);

                xmlroot
----------------------------------------
 <?xml version="1.0" standalone="yes"?>
 <content>abc</content>
```

9.15.1.7. xmlagg

```
xmlagg ( xml ) → xml
```

ここで説明している他の関数とは異なり、xmlagg関数は集約関数です。これはxmlconcatが行うように、入力値を連結する集約関数ですが、単一行内の複数の式にまたがった連結ではなく、複数行にまたがった連結を行います。集約関数についての追加情報は9.21を参照してください。

例:

```
CREATE TABLE test (y int, x xml);
INSERT INTO test VALUES (1, '<foo>abc</foo>');
INSERT INTO test VALUES (2, '<bar/>');
SELECT xmlagg(x) FROM test;
        xmlagg
----------------------
 <foo>abc</foo><bar/>
```

連結の順序を決定するため、4.2.7に記述されているようにORDER BY句を集計呼び出しに追加することができます。以下は例です。

```
SELECT xmlagg(x ORDER BY y DESC) FROM test;
        xmlagg
----------------------
 <bar/><foo>abc</foo>
```

下記は以前のバージョンで推奨されていた、非標準的な方法例です。特定のケースでは依然として有用かもしれません。

```
SELECT xmlagg(x) FROM (SELECT * FROM test ORDER BY y DESC) AS tab;
        xmlagg
----------------------
 <bar/><foo>abc</foo>
```

9.15.2. XML述語

この節で記述されている式は、xml値の属性をチェックします。

9.15.2.1. IS DOCUMENT

```
xml IS DOCUMENT → boolean
```

式IS DOCUMENTは引数XML値が適切なXML文書であれば真を返し、そうでなければ（つまり、内容の断片）偽を返すか、もしくは引数がNULLであればNULLを返します。文書と内容の断片の差異については8.13を参照してください。

9.15.2.2. IS NOT DOCUMENT

```
xml IS NOT DOCUMENT → boolean
```

式IS NOT DOCUMENTは引数XML値が適切なXML文書であれば偽を返し、そうでなければ（つまり、内容の断片）真を返すか、もしくは引数がNULLであればNULLを返します。

9.15.2.3. XMLEXISTS

```
XMLEXISTS ( text PASSING [BY {REF|VALUE}] xml [BY {REF|VALUE}] ) → boolean
```

関数xmlexistsは渡されたXML値をコンテキスト項目としてXPath 1.0式（第一引数）を評価します。この関数は評価が空のノード集合を生成する場合には偽を返し、それ以外の値を返すならば真を返します。もしどれかの引数がNULLであった場合はNULLを返します。コンテキスト項目として渡される非NULLの値は、内容の断片や非XML値ではなく、XML文書でなければなりません。

例:

```
SELECT xmlexists('//town[text() = ''Toronto'']' PASSING BY VALUE '<towns><town>Toronto</
town><town>Ottawa</town></towns>');

 xmlexists
------------
 t
(1 row)
```

PostgreSQLはBY REF句とBY VALUE句を受け付けますが、D.3.2で議論されているように無視します。

SQL標準ではxmlexists関数はXML問い合わせ言語における式を評価しますが、D.3.1で議論されているように、PostgreSQLはXPath 1.0の式だけを受け付けます。

9.15.2.4. xml_is_well_formed

```
xml_is_well_formed ( text ) → boolean
xml_is_well_formed_document ( text ) → boolean
xml_is_well_formed_content ( text ) → boolean
```

これらの関数はtext文字列が整形式かどうかをチェックし、論理値で結果を返します。xml_is_well_formed_documentは文書が整形式かをチェックし、一方xml_is_well_formed_contentは内容が整形式かをチェックします。xml_is_well_formedは、xmloptionパラメータ値がDOCUMENTに設定されていれば前者を、CONTENTが設定されていれば後者のチェックを実施します。これは、xml_is_well_formedは単純なxml型へのキャストが成功するかの判断に有用であり、その他の2つの関数はXMLPARSEの対応による変換が成功するかの判断に有用であることを意味します。

例:

```
SET xmloption TO DOCUMENT;
SELECT xml_is_well_formed('<>');
 xml_is_well_formed
--------------------
 f
(1 row)

SELECT xml_is_well_formed('<abc/>');
 xml_is_well_formed
--------------------
 t
(1 row)

SET xmloption TO CONTENT;
SELECT xml_is_well_formed('abc');
 xml_is_well_formed
--------------------
 t
(1 row)

SELECT xml_is_well_formed_document('<pg:foo xmlns:pg="http://postgresql.org/stuff">bar</
pg:foo>');
 xml_is_well_formed_document
-----------------------------
 t
(1 row)

SELECT xml_is_well_formed_document('<pg:foo xmlns:pg="http://postgresql.org/stuff">bar</
my:foo>');
 xml_is_well_formed_document
-----------------------------
```

```
 f
(1 row)
```

最後の例は、名前空間が正しく一致しているかのチェックも含むことを示しています。

9.15.3. XMLの処理

データ型xmlの値を処理するため、PostgreSQLはXPath 1.0式を評価する関数xpathおよびxpath_existsと、テーブル関数XMLTABLEを提供しています。

9.15.3.1. xpath

```
xpath ( xpath text, xml xml [, nsarray text[] ] ) → xml[]
```

関数xpathは、XML値xmlに対し、XPath 1.0式xpath(テキストとして指定)を評価します。そして、XPath式で作成されたノード集合に対応するXML値の配列を返します。もし、XPath式がノード集合ではなくスカラー値を返す場合、単一要素の配列が返されます。

2番目の引数は整形済XML文書でなければなりません。特に、単一のルートノード要素を持たなければなりません。

オプショナルな関数の3番目の引数は名前空間マッピング配列です。この配列は、第2軸が2に等しい長さをもつ2次元text配列です（つまり、それは配列の配列で、それぞれは正確に2つの要素からなります）。それぞれの配列のエントリの最初の要素は名前空間の名前（別名）で、2番目は名前空間のURIです。この配列内で提供される別名がXML文書自身で使用されるものと同じであることは必要ではありません（言い換えると、XML文書内およびxpath関数の両方の文脈の中で、別名はローカルです）。

例：

```
SELECT xpath('/my:a/text()', '<my:a xmlns:my="http://example.com">test</my:a>',
             ARRAY[ARRAY['my', 'http://example.com']]);

 xpath
--------
 {test}
(1 row)
```

デフォルト(匿名)名前空間を取り扱うためには、以下のようなことを実施してください。

```
SELECT xpath('//mydefns:b/text()', '<a xmlns="http://example.com"><b>test</b></a>',
             ARRAY[ARRAY['mydefns', 'http://example.com']]);

 xpath
--------
```

```
 {test}
(1 row)
```

9.15.3.2. xpath_exists

```
xpath_exists ( xpath text, xml xml [, nsarray text[] ] ) → boolean
```

関数xpath_existsは、xpath関数の特別な形式です。この関数は、XPath 1.0を満足する個別のXML値を返す代わりに、問い合わせがそれを満足するかどうか（具体的には空のノード集合以外の値を返すかどうか）を論理値で返します。この関数は、名前空間にマッピングされた引数をもサポートする点を除き、標準のXMLEXISTS述語と同じです。

例:

```
SELECT xpath_exists('/my:a/text()', '<my:a xmlns:my="http://example.com">test</my:a>',
                    ARRAY[ARRAY['my', 'http://example.com']]);

 xpath_exists
--------------
 t
(1 row)
```

9.15.3.3. xmltable

```
XMLTABLE (
    [ XMLNAMESPACES ( namespace_uri AS namespace_name [, ...] ), ]
    row_expression PASSING [BY {REF|VALUE}] document_expression [BY {REF|VALUE}]
    COLUMNS name { type [PATH column_expression] [DEFAULT default_expression] [NOT NULL | NULL]
                 | FOR ORDINALITY }
            [, ...]
) → setof record
```

xmltable式は、与えられたXML値、行を抽出するXPathフィルタ、オプションの列定義の集合に基づいてテーブルを生成します。関数と構文的に似ていますが、これは問い合わせ中のFROM句におけるテーブルとしてのみ使用できます。

オプションのXMLNAMESPACES句はカンマで区切られた名前空間のリストを与えます。各々のnamespace_uriはtext式で、namespace_nameは単純な識別子です。これは文書とその別名で使用されるXML名前空間を指定します。デフォルトの名前空間指定は現在のところサポートされていません。

必須のrow_expression引数は評価されるXPath 1.0式(textで与えます)で、XMLノード集合を得るためにdocument_expressionをそのコンテキスト項目として渡します。このノードはxmltableが出力行に変換します。document_expressionがNULLであるか、row_expressionが空のノード集合あるいはノード集合以外の値を生成するなら行は出力されません。

document_expressionはrow_expressionのためのコンテキスト項目を提供します。それは整形式XMLの文書でなければならず、フラグメントやフォレストは受け付けられません。D.3.2で議論されているように、BY REF句とBY VALUE句は受け付けられますが、無視されます。

SQL標準ではxmltable関数はXML問い合わせ言語の式を評価しますが、D.3.1で議論されているようにPostgreSQLではXPath 1.0式だけを受け付けます。

必須のCOLUMNS句は、出力テーブルに現れる列を指定します。形式については上記の構文サマリーを参照してください。各列には名前が必須で、データ型についても同様です。(FOR ORDINALITYが指定された場合を除きます。その場合は暗黙的にintegerが想定されます。) パス、デフォルト値、NULLを許すかどうかの句は省略できます。

FOR ORDINALITYと印がつけられた列には、row_expressionの結果ノード集合から取得されたノードの順序に対応する1から始まる行番号が入ります。FOR ORDINALITYの印が付けられるのは最大でも1列です。

> **注記**
>
> XPath 1.0はノード集合内のノードの順序を指定しません。ですから、結果が特定の順序になっていることに依存するコードは実装依存となります。詳細はD.3.1.2をご覧ください。

列のcolumn_expressionはXPath 1.0式で、row_expressionの結果における現在のノードをそのコンテキスト項目としてrow_expressionの結果に対応する各行について評価されて、列の値を得ます。column_expression が与えられなかった場合は、暗黙的なパスとして列名が使用されます。

列のXPath式が非XML値(XPath 1.0における文字列、論理値、倍精度浮動小数点数に限られます)を返し、その列がxml以外のPostgreSQL型なら、あたかも値の文字列表現をPostgreSQL型にアサインしたように列に値がセットされます。(値が論理値の場合、出力列型が数値カテゴリに属するならその文字列表現は1または0になり、それ外ならtrueまたはfalseになります。)

列のXPath表現が空ではないXMLノードの集合を返し、列のPostgreSQL型がxmlである場合には、式が文書あるいはフォームの内容なら、列には正確に式の結果がアサインされます。[2]

xml出力列にアサインされた非XMLの結果は、結果の値が文字列値となる単一のテキストノードであるコンテントを生成します。それ以外の型の列にアサインされたXMLの結果は複数のノードを持たないかも知れませんし、エラーを生じするかも知れません。正確に一つのノードだけが存在するなら、列にはあたかもノードの文字列値(XPath 1.0 string関数の定義されているように) がPostgreSQL型にアサインされたように設定されます。

ある要素と、その子孫に含まれるすべてのテキストノードをドキュメントの順に結合したものがXML要素の文字列値です。テキストノードの子孫を持たない要素の文字列値は空文字列です。(NULLではありません。) すべてのxsi:nil属性は無視されます。非テキスト要素の間にある空白のみからなるtext()2つのノードは保存され、text()の先頭の空白は平坦化されないことに注意してください。XPath 1.0 string関数が、他のXMLノード型と非XML値の文字列値を定義するルールのために参照されるかも知れません。

[2] トップレベルにおいて複数の要素ノードを含むか、あるいは要素の外側の非空白テキストであるような結果は、コンテントフォームの例です。XPathの結果はそのどちらでもないフォームであることがあり得ます。たとえば、それを含む要素から選択された属性ノードを返す場合です。XPath 1.0のstring関数で定義されているように、そうした結果は、許可されないノードを文字列値で置き換えたコンテントフォームに設定されます。

ここで示した変換ルールは、D.3.1.3で議論されているように、正確にSQL標準に従っているわけではありません。

パス式がある行に対して空のノード集合（典型的にはマッチしなかった場合）を返した時は、default_expressionが指定されている場合を除き、列にはNULLが設定されます。そしてその式を評価した結果から生じる値が使用されます。

xmltableが呼び出されて直ちに評価されるのと異なり、default_expressionはその列に対してデフォルトが必要になるたびに評価されます。式が安定（stable）または不変（immutable）とみなされる場合、評価は繰り返し行われないかもしれません。これはdefault_expressionの中でnextvalのような揮発性関数を使用できることを意味します。

列にはNOT NULLの印をつけることができます。NOT NULLの列のcolumn_expressionが何にもマッチせず、DEFAULTがない、あるいはdefault_expressionの評価結果もNULLになるという場合はエラーが報告されます。

例:

```
CREATE TABLE xmldata AS SELECT
xml $$
<ROWS>
  <ROW id="1">
    <COUNTRY_ID>AU</COUNTRY_ID>
    <COUNTRY_NAME>Australia</COUNTRY_NAME>
  </ROW>
  <ROW id="5">
    <COUNTRY_ID>JP</COUNTRY_ID>
    <COUNTRY_NAME>Japan</COUNTRY_NAME>
    <PREMIER_NAME>Shinzo Abe</PREMIER_NAME>
    <SIZE unit="sq_mi">145935</SIZE>
  </ROW>
  <ROW id="6">
    <COUNTRY_ID>SG</COUNTRY_ID>
    <COUNTRY_NAME>Singapore</COUNTRY_NAME>
    <SIZE unit="sq_km">697</SIZE>
  </ROW>
</ROWS>
$$ AS data;

SELECT xmltable.*
  FROM xmldata,
       XMLTABLE('//ROWS/ROW'
                PASSING data
                COLUMNS id int PATH '@id',
                        ordinality FOR ORDINALITY,
                        "COUNTRY_NAME" text,
                        country_id text PATH 'COUNTRY_ID',
                        size_sq_km float PATH 'SIZE[@unit = "sq_km"]',
```

```
                        size_other text PATH
                             'concat(SIZE[@unit!="sq_km"], " ", SIZE[@unit!="sq_km"]/@unit)',
                        premier_name text PATH 'PREMIER_NAME' DEFAULT 'not specified');

 id | ordinality | COUNTRY_NAME | country_id | size_sq_km |  size_other  | premier_name
----+------------+--------------+------------+------------+--------------+---------------
  1 |          1 | Australia    | AU         |            |              | not specified
  5 |          2 | Japan        | JP         |            | 145935 sq_mi | Shinzo Abe
  6 |          3 | Singapore    | SG         |        697 |              | not specified
```

以下の例では、複数のtext()ノードの結合、列名のXPathフィルターとしての使用、空白文字、XMLコメント、処理命令の取扱いを示します。

```
CREATE TABLE xmlelements AS SELECT
xml $$
  <root>
   <element>  Hello<!-- xyxxz -->2a2<?aaaaa?> <!--x-->  bbb<x>xxx</x>CC  </element>
  </root>
$$ AS data;

SELECT xmltable.*
  FROM xmlelements, XMLTABLE('/root' PASSING data COLUMNS element text);
         element
-------------------------
   Hello2a2   bbbxxxCC
```

以下の例では、XMLNAMESPACES句を使ってXMLドキュメントやXPath式で使われる追加の名前空間のリストを指定する方法を示します。

```
WITH xmldata(data) AS (VALUES ('
<example xmlns="http://example.com/myns" xmlns:B="http://example.com/b">
 <item foo="1" B:bar="2"/>
 <item foo="3" B:bar="4"/>
 <item foo="4" B:bar="5"/>
</example>'::xml)
)
SELECT xmltable.*
  FROM XMLTABLE(XMLNAMESPACES('http://example.com/myns' AS x,
                              'http://example.com/b' AS "B"),
              '/x:example/x:item'
                 PASSING (SELECT data FROM xmldata)
                 COLUMNS foo int PATH '@foo',
                   bar int PATH '@B:bar');
 foo | bar
-----+-----
   1 |   2
```

```
   3 |   4
   4 |   5
(3 rows)
```

9.15.4. XMLにテーブルをマップ

以下の関数はリレーショナルテーブルの内容をXML値にマップします。これらはXMLエクスポート機能と考えることができます。

```
table_to_xml ( table regclass, nulls boolean,
               tableforest boolean, targetns text ) → xml
query_to_xml ( query text, nulls boolean,
               tableforest boolean, targetns text ) → xml
cursor_to_xml ( cursor refcursor, count integer, nulls boolean,
                tableforest boolean, targetns text ) → xml
```

table_to_xmlは、パラメータtableとして渡された名前付きのテーブルの内容をマップします。regclass型はオプションのスキーマ修飾と二重引用符を含む、通常の表記法を使用しテーブルを特定する文字列を受け付けます。query_to_xmlは、パラメータqueryとしてテキストが渡された問い合わせを実行し、結果セットをマップします。cursor_to_xmlは、パラメータcursorで指定されたカーソルから提示された行数を取得します。それぞれの関数により結果値がメモリーに構築されるため、この異形は巨大なテーブルをマップする必要がある場合推奨されます。

tableforestが偽であれば、結果のXML文書は以下のようになります。

```
<tablename>
  <row>
    <columnname1>data</columnname1>
    <columnname2>data</columnname2>
  </row>

  <row>
    ...
  </row>

  ...
</tablename>
```

tableforestが真であれば、結果は以下のようなXML文書の断片です。

```
<tablename>
  <columnname1>data</columnname1>
  <columnname2>data</columnname2>
</tablename>

<tablename>
  ...
```

```
</tablename>

...
```

テーブル名が利用できないとき、つまり、問い合わせ、またはカーソルをマップする時は、最初の書式では文字列tableが使用され、2番目の書式ではrowが使用されます。

これらどの書式を選択するのかはユーザ次第です。最初の書式は適切なXML文書で、多くのアプリケーションにおいて重要です。第2の書式は、後に結果値が1つの文書に再び組み立てられる場合、cursor_to_xml関数内でより有用になる傾向があります。上記で説明したXML内容を作成する関数、特にxmlelementは結果を好みにかえるために使用することができます。

データの値は上記関数xmlelementで説明したのと同じ方法でマップされます。

パラメータnullsは出力にNULL値が含まれる必要があるかを決定します。もし真であれば列内のNULL値は以下のように表現されます。

```
<columnname xsi:nil="true"/>
```

ここでxsiはXMLスキーマインスタンスに対するXML名前空間接頭辞です。適切な名前空間宣言が結果値に追加されます。もし偽の場合、NULL値を含む列は単に出力から削除されます。

パラメータtargetnsは結果の希望するXML名前空間を指定します。特定の名前空間が必要なければ、空文字列を渡す必要があります。

以下の関数は、対応する上記関数により行われたマッピングを記述するXMLスキーマ文書を返します。

```
table_to_xmlschema ( table regclass, nulls boolean,
                     tableforest boolean, targetns text ) → xml
query_to_xmlschema ( query text, nulls boolean,
                     tableforest boolean, targetns text ) → xml
cursor_to_xmlschema ( cursor refcursor, nulls boolean,
                      tableforest boolean, targetns text ) → xml
```

一致するXMLデータマッピングとXMLスキーマ文書を取得するため、同じパラメータが渡されることが不可欠です。

以下の関数は、XMLデータマッピングとそれに対応するXMLスキーマがお互いにリンクされた、1つの文書（またはフォレスト）を作成します。これらは自己完結した、自己記述的な結果を希望する場合に便利です。

```
table_to_xml_and_xmlschema ( table regclass, nulls boolean,
                             tableforest boolean, targetns text ) → xml
query_to_xml_and_xmlschema ( query text, nulls boolean,
                             tableforest boolean, targetns text ) → xml
```

さらに、以下の関数がスキーマ全体、または現在のデータベース全体の類似マッピングを作成するため利用できます。

```
schema_to_xml ( schema name, nulls boolean,
```

```
                tableforest boolean, targetns text ) → xml
schema_to_xmlschema ( schema name, nulls boolean,
                      tableforest boolean, targetns text ) → xml
schema_to_xml_and_xmlschema ( schema name, nulls boolean,
                              tableforest boolean, targetns text ) → xml

database_to_xml ( nulls boolean,
                  tableforest boolean, targetns text ) → xml
database_to_xmlschema ( nulls boolean,
                        tableforest boolean, targetns text ) → xml
database_to_xml_and_xmlschema ( nulls boolean,
                                tableforest boolean, targetns text ) → xml
```

これらの関数は現在のユーザが読めないテーブルは無視します。加えてデータベース中全体に渡る関数は現在のユーザがUSAGE(検索)権限を持たないスキーマを無視します。

これらはメモリー内に作成される必要がある、多くのデータを生成する潜在的可能性があることに注意してください。巨大なスキーマ、またはデータベースの内容マッピングを要求する際は、その代わりにテーブルを別々にマップすること、さらにはカーソル経由とすることさえ、検討することは無駄ではありません。

スキーマ内容マッピングの結果は以下のようになります。

```
<schemaname>

table1-mapping

table2-mapping

...

</schemaname>
```

ここで、テーブルマッピング書式は上で説明したとおりtableforestパラメータに依存します。

データベース内容マッピング書式は以下のようになります。

```
<dbname>

<schema1name>
  ...
</schema1name>

<schema2name>
  ...
</schema2name>

...
```

```
</dbname>
```

ここで、スキーママッピングは上記のとおりです。

これらの関数で作成された出力を使用する1つの例として、例 9.1は、テーブルデータの表形式への翻訳を含むtable_to_xml_and_xmlschemaからHTML文書への出力の変換をおこなうXSLTスタイルシートを示します。同じようにして、これらの関数の結果は他のXML基準書式に変換されます。

例9.1 SQL/XML出力をHTMLに変換するXSLTスタイルシート

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://www.w3.org/1999/xhtml"
>

  <xsl:output method="xml"
      doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
      doctype-public="-//W3C/DTD XHTML 1.0 Strict//EN"
      indent="yes"/>

  <xsl:template match="/*">
    <xsl:variable name="schema" select="//xsd:schema"/>
    <xsl:variable name="tabletypename"
                  select="$schema/xsd:element[@name=name(current())]/@type"/>
    <xsl:variable name="rowtypename"
                  select="$schema/xsd:complexType[@name=$tabletypename]/xsd:sequence/
xsd:element[@name='row']/@type"/>

    <html>
      <head>
        <title><xsl:value-of select="name(current())"/></title>
      </head>
      <body>
        <table>
          <tr>
            <xsl:for-each select="$schema/xsd:complexType[@name=$rowtypename]/xsd:sequence/
xsd:element/@name">
              <th><xsl:value-of select="."/></th>
            </xsl:for-each>
          </tr>

          <xsl:for-each select="row">
            <tr>
              <xsl:for-each select="*">
                <td><xsl:value-of select="."/></td>
```

```
              </xsl:for-each>
            </tr>
          </xsl:for-each>
        </table>
      </body>
    </html>
  </xsl:template>

</xsl:stylesheet>
```

9.16. JSON関数と演算子

この節では次のことを説明します。

- JSONデータを処理、生成する関数と演算子
- SQL/JSONパス言語

SQL/JSON標準を更に学ぶためには、[sqltr-19075-6]をご覧ください。PostgreSQLでサポートされているJSON型の詳細に関しては、8.14をご覧ください。

9.16.1. JSONデータの処理と生成

表 9.44にJSONデータ型(8.14を参照)で使用可能な演算子を示します。加えて表 9.1で示す通常の比較演算子がjsonbで利用できますが、jsonでは利用できません。比較演算子は8.14.4で概要が示されているように示すBツリー操作用の順序付け規則にしたがいます。

表9.44 jsonとjsonb演算子

演算子 説明 例
json -> integer → json jsonb -> integer → jsonb JSON配列のn番目の要素を取り出します。(配列要素はゼロから始まりますが、負の整数は最後から数えられます。) '[{"a":"foo"},{"b":"bar"},{"c":"baz"}]'::json -> 2 → {"c":"baz"} '[{"a":"foo"},{"b":"bar"},{"c":"baz"}]'::json -> -3 → {"a":"foo"}
json -> text → json jsonb -> text → jsonb 与えられたキーでJSONオブジェクトフィールドを取り出します。 '{"a": {"b":"foo"}}'::json -> 'a' → {"b":"foo"}
json ->> integer → text jsonb ->> integer → text JSON配列のn番目の要素をtextとして取り出します。 '[1,2,3]'::json ->> 2 → 3

演算子 説明 例
json ->> text → text jsonb ->> text → text 与えられたキーでJSONオブジェクトフィールドをtextとして取り出します。 '{"a":1,"b":2}'::json ->> 'b' → 2
json #> text[] → json jsonb #> text[] → jsonb 指定したパスにおけるJSONの副オブジェクトを取り出します。パス要素はフィールドキーあるいは配列のインデックスでも構いません。 '{"a": {"b": ["foo","bar"]}}'::json #> '{a,b,1}' → "bar"
json #>> text[] → text jsonb #>> text[] → text 指定したパスにおけるJSONの副オブジェクトをtextとして取り出します。 '{"a": {"b": ["foo","bar"]}}'::json #>> '{a,b,1}' → bar

> **注記**
>
> JSON入力が要求と一致する正しい構造をしていなければ、フィールド/要素/パス抽出演算子は失敗するのではなくNULLを返します。例えばそのような要素が存在しない場合です。

ほかにjsonbだけで利用可能な演算子もいくつか存在します。それらを表 9.45に示します。8.14.4には、インデックス付されたjsonbデータを効率的に検索するためにこれらの演算子をどのように利用できるかについて書いてあります。

表9.45 追加jsonb演算子

演算子 説明 例
jsonb @> jsonb → boolean 最初のJSON値は二番目を含んでいるか？（包含の詳細は8.14.3を参照してください。） '{"a":1, "b":2}'::jsonb @> '{"b":2}'::jsonb → t
jsonb <@ jsonb → boolean 最初のJSON値は二番目に含まれているか？ '{"b":2}'::jsonb <@ '{"a":1, "b":2}'::jsonb → t
jsonb ? text → boolean そのテキスト文字列はトップレベルのキーあるいは配列要素としてJSON値中に存在しているか？ '{"a":1, "b":2}'::jsonb ? 'b' → t '["a", "b", "c"]'::jsonb ? 'b' → t
jsonb ?| text[] → boolean テキスト配列中のどれかの文字列がトップレベルのキーあるいは配列要素として存在しているか？ '{"a":1, "b":2, "c":3}'::jsonb ?| array['b', 'd'] → t

演算子 説明 例
jsonb ?& text[] → boolean テキスト配列のすべての文字列がトップレベルのキーあるいは配列要素として存在しているか？ '["a", "b", "c"]'::jsonb ?& array['a', 'b'] → t
jsonb || jsonb → jsonb 2つのjsonb値を結合します。2つのオブジェクトを結合するとそれらのキーの和を持つオブジェクトを生成します。キーが重複している場合は2番目のオブジェクトの値が使用されます。再帰操作は行いません。トップレベルの配列あるいはオブジェクト構造だけがマージされます。 '["a", "b"]'::jsonb || '["a", "d"]'::jsonb → ["a", "b", "a", "d"] '{"a": "b"}'::jsonb || '{"c": "d"}'::jsonb → {"a": "b", "c": "d"}
jsonb - text → jsonb キー（及びその値）をJSONオブジェクトから削除します。あるいはマッチする文字列値をJSON配列から削除します。 '{"a": "b", "c": "d"}'::jsonb - 'a' → {"c": "d"} '["a", "b", "c", "b"]'::jsonb - 'b' → ["a", "c"]
jsonb - text[] → jsonb 左のオペランドからマッチするすべてのキーあるいは配列要素を削除します。 '{"a": "b", "c": "d"}'::jsonb - '{a,c}'::text[] → {}
jsonb - integer → jsonb 指定したインデックス（負の整数は最後から数えます）の配列要素を削除します。JSON値が配列でなければエラーが生じます。 '["a", "b"]'::jsonb - 1 → ["a"]
jsonb #- text[] → jsonb 指定パスのフィールドあるいは配列要素を削除します。パス要素はフィールドキーあるいは配列インデックスが指定できます。 '["a", {"b":1}]'::jsonb #- '{1,b}' → ["a", {}]
jsonb @? jsonpath → boolean JSONパスは指定したJSON値の要素を返すか？ '{"a":[1,2,3,4,5]}'::jsonb @? '$.a[*] ? (@ > 2)' → t
jsonb @@ jsonpath → boolean 指定したJSON値に対するJSONパス述語チェックの結果を返します。結果の最初の項目だけが考慮されます。結果が論理値でなければNULLが返ります。 '{"a":[1,2,3,4,5]}'::jsonb @@ '$.a[*] > 2' → t

注記

jsonpath演算子の@?および@@演算子は以下のエラーを抑止します。オブジェクトフィールドあるいは配列要素の欠如、期待しないJSON要素型、日付時刻及び数値エラー。以下に示すjsonpath関連の関数もこれらのエラーを抑止するようにすることもできます。この振る舞いは、異なる構造のJSON文書集合を検索する際に役に立つかも知れません。

表 9.46に、json値およびjsonb値を作成するために利用可能な関数を示します。

表9.46 JSON作成関数

関数 説明 例
to_json (anyelement) → json to_jsonb (anyelement) → jsonb SQL値をjsonあるいはjsonbに変換します。配列と複合型は再帰的に配列とオブジェクトに変換されます。(多次元配列はJSONにおける配列の配列になります。) それ以外は、そのSQLデータ型からjsonにキャストがあれば、キャスト関数が変換のために用いられます。[a] そうでなければスカラーJSON値が生成されます。数値、論理値、NULL以外のスカラーには、有効なJSON文字列値にするための必要なエスケープ処理が施されたテキスト表現が使われます。 to_json('Fred said "Hi."'::text) → "Fred said \"Hi.\"" to_jsonb(row(42, 'Fred said "Hi."'::text)) → {"f1": 42, "f2": "Fred said \"Hi.\""}
array_to_json (anyarray [, boolean]) → json SQL配列をJSON配列に変換します。追加の論理引数が真であるときに改行がトップレベルの配列要素の間に加えられる以外は、その振る舞いはto_jsonと同じです。 array_to_json('{{1,5},{99,100}}'::int[]) → [[1,5],[99,100]]
row_to_json (record [, boolean]) → json SQL複合値をJSONオブジェクトに変換します。追加の論理引数が真であるときに改行がトップレベルの配列要素の間に加えられる以外は、その振る舞いはto_jsonと同じです。 row_to_json(row(1,'foo')) → {"f1":1,"f2":"foo"}
json_build_array (VARIADIC "any") → json jsonb_build_array (VARIADIC "any") → jsonb 異なる型から構成される可能性のあるJSON配列をvariadic引数リストから作成します。各々の引数はto_jsonあるいはto_jsonbに従って変換されます。 json_build_array(1, 2, 'foo', 4, 5) → [1, 2, "foo", 4, 5]
json_build_object (VARIADIC "any") → json jsonb_build_object (VARIADIC "any") → jsonb variadic引数リストからJSONオブジェクトを作成します。慣例により引数リストは代替キーと値が交互に並んだものです。キー引数はテキストに強制的に変換されます。値引数はto_jsonあるいはto_jsonbに従って変換されます。 json_build_object('foo', 1, 2, row(3,'bar')) → {"foo" : 1, "2" : {"f1":3,"f2":"bar"}}
json_object (text[]) → json jsonb_object (text[]) → jsonb テキスト配列からJSONオブジェクトを作成します。配列は、偶数個の要素からなる1次元(キー／値の対が交互に並んでいるものと扱われます)あるいは内側の配列が2つの要素を持つ2次元(2つの要素がキー／値のペアとして扱われます)のいずれかでなければなりません。すべての値はJSON文字列に変換されます。 json_object('{a, 1, b, "def", c, 3.5}') → {"a" : "1", "b" : "def", "c" : "3.5"} json_object('{{a, 1}, {b, "def"}, {c, 3.5}}') → {"a" : "1", "b" : "def", "c" : "3.5"}
json_object (keys text[], values text[]) → json jsonb_object (keys text[], values text[]) → jsonb この形のjson_objectは2つの別々の配列からキーと値の対を取ります。他の点ではすべて、引数1つの形と同じです。 json_object('{a,b}', '{1,2}') → {"a": "1", "b": "2"}

[a] たとえばhstore拡張にはhstoreからjsonへのキャストがあり、JSON生成関数で変換されたhstore値は、原始的な文字列値としてではなく、JSONオブジェクトとして表示されます。

表 9.47にjsonとjsonb値を処理するのに使える関数を示します。

表9.47 JSON処理関数

関数 説明 例
json_array_elements (json) → setof json jsonb_array_elements (jsonb) → setof jsonb トップレベルのJSON配列をJSON値の集合に展開します。 select * from json_array_elements('[1,true, [2,false]]') → value ----------- 1 true [2,false]
json_array_elements_text (json) → setof text jsonb_array_elements_text (jsonb) → setof text トップレベルのJSON配列をtext値の集合に展開します。 select * from json_array_elements_text('["foo", "bar"]') → value ----------- foo bar
json_array_length (json) → integer jsonb_array_length (jsonb) → integer トップレベルのJSON配列の要素数を返します。 json_array_length('[1,2,3,{"f1":1,"f2":[5,6]},4]') → 5
json_each (json) → setof record (key text, value json) jsonb_each (jsonb) → setof record (key text, value jsonb) トップレベルのJSONオブジェクトをキー／値のペアの集合に展開します。 select * from json_each('{"a":"foo", "b":"bar"}') → key | value -----+------- a | "foo" b | "bar"
json_each_text (json) → setof record (key text, value text) jsonb_each_text (jsonb) → setof record (key text, value text) トップレベルのJSONオブジェクトをキー／値のペアの集合に展開します。返り値のvalueはtext型です。 select * from json_each_text('{"a":"foo", "b":"bar"}') → key | value

関数

説明

例

```
-----+-------
 a   | foo
 b   | bar
```

json_extract_path (from_json json, VARIADIC path_elems text[]) → json

jsonb_extract_path (from_json jsonb, VARIADIC path_elems text[]) → jsonb

指定したパスにおけるJSONの副オブジェクトを取り出します。(これは#>演算子と機能的に同じですが、パスをvariadicリストで書き出す方がより便利な場合があります。)

json_extract_path('{"f2":{"f3":1},"f4":{"f5":99,"f6":"foo"}}', 'f4', 'f6') → "foo"

json_extract_path_text (from_json json, VARIADIC path_elems text[]) → text

jsonb_extract_path_text (from_json jsonb, VARIADIC path_elems text[]) → text

指定したパスにおけるJSONの副オブジェクトをtextとして取り出します。(これは機能的には#>>演算子と同じです。)

json_extract_path_text('{"f2":{"f3":1},"f4":{"f5":99,"f6":"foo"}}', 'f4', 'f6') → foo

json_object_keys (json) → setof text

jsonb_object_keys (jsonb) → setof text

トップレベルのJSONオブジェクト中のキーの集合を返します。

select * from json_object_keys('{"f1":"abc","f2":{"f3":"a", "f4":"b"}}') →

```
json_object_keys
------------------
 f1
 f2
```

json_populate_record (base anyelement, from_json json) → anyelement

jsonb_populate_record (base anyelement, from_json jsonb) → anyelement

トップレベルのJSONオブジェクトをbase引数である複合型を持つ行に展開します。JSONオブジェクトは出力行型の列名と一致するフィールドが検査されます。(出力列名と関連のないフィールドは無視されます。) 典型的な使い方としては、baseの値が単にNULLで、これはオブジェクトフィールドと一致しない出力列にはNULLがセットされることを意味します。しかし、baseがNULLでないなら、それが持つ値が一致しない列に使われます。

JSON値を出力列のSQL型に変換する際に以下のルールが順に適用されます。

- すべての場合にJSONのNULL値はSQLのNULLに変換されます。
- 出力列がjson型あるいはjsonb型なら、JSON値は単にそのまま複製されます。
- 出力行が複合(行)型でJSON値がJSONオブジェクトなら、これらのルールを再帰的に適用することによって、オブジェクトのフィールドが出力行型の列に変換されます。
- 同様に、出力行が配列型でJSON値がJSON配列なら、これらのルールを再帰的に適用することによって、JSON配列の要素が出力配列の要素に変換されます。
- それ以外の場合で、JSON値が文字列なら、その文字列の内容が列のデータ型に対応する入力変換関数に送られます。
- さもなければ、通常のJSON値のテキスト表現が列のデータ型に対応する入力変換関数に送られます。

これらの関数の例ではJSON定数を使用していますが、典型的な使用法はそのjsonまたはjsonb列をFROM句の別のテーブルから外側に参照することです。FROM句でjson_populate_recordを書くのは良い練習になります。すべての取り出された列を重複した関数呼び出しなしに利用できるからです。

create type subrowtype as (d int, e text); create type myrowtype as (a int, b text[], c subrowtype);

関数

説明

例

select * from json_populate_record(null::myrowtype, '{"a": 1, "b": ["2", "a b"], "c": {"d": 4, "e": "a b c"}, "x": "foo"}') →

```
a |   b       |     c
---+-----------+-------------
 1 | {2,"a b"} | (4,"a b c")
```

json_populate_recordset (base anyelement, from_json json) → setof anyelement

jsonb_populate_recordset (base anyelement, from_json jsonb) → setof anyelement

トップレベルのJSONオブジェクトをbase引数である複合型を持つ行の集合に展開します。JSON配列の個々の要素は上のjson[b]_populate_recordで説明したように処理されます。

```
create type twoints as (a int, b int);
select * from json_populate_recordset(null::twoints, '[{"a":1,"b":2}, {"a":3,"b":4}]') →
```

```
a | b
---+---
 1 | 2
 3 | 4
```

json_to_record (json) → record

jsonb_to_record (jsonb) → record

トップレベルのJSONオブジェクトをAS句で定義した複合型を持つ行に展開します。(recordを返すすべての関数では、呼び出す問い合わせは明示的にAS句でレコードの構造を定義しなければなりません。) 上のjson[b]_populate_recordで説明した方法で、出力レコードはJSONオブジェクトのフィールドで満たされます。入力レコード値がないので、一致しない列は常にNULLで満たされます。

```
create type myrowtype as (a int, b text);
select * from json_to_record('{"a":1,"b":[1,2,3],"c":[1,2,3],"e":"bar","r": {"a": 123, "b": "a b c"}}')
as x(a int, b text, c int[], d text, r myrowtype) →
```

```
a |    b    |    c    | d |       r
---+---------+---------+---+---------------
 1 | [1,2,3] | {1,2,3} |   | (123,"a b c")
```

json_to_recordset (json) → setof record

jsonb_to_recordset (jsonb) → setof record

トップレベルのJSON配列をAS句で定義した複合型を持つ行に展開します。(recordを返すすべての関数では、呼び出す問い合わせは明示的にAS句でレコードの構造を定義しなければなりません。) 上のjson[b]_populate_recordで説明した方法で、JSON配列の要素は処理されます。

```
select * from json_to_recordset('[{"a":1,"b":"foo"}, {"a":"2","c":"bar"}]') as x(a int, b text) →
```

```
a |  b
---+-----
 1 | foo
 2 |
```

関数 説明 例
jsonb_set (target jsonb, path text[], new_value jsonb [, create_if_missing boolean]) → jsonb pathで指定された要素をnew_valueで置き換えてtargetを返します。create_if_missingが真なら(デフォルトです)、pathで指定された項目が無い時にnew_valueが追加されます。パス中のすべての初期のステップは存在しなければならず、さもなければtargetは変わらないままに返却されます。パスの位置についての演算子については、pathの中にある負の整数はJSON配列の終わりから数えます。パスの最後のステップが範囲外の配列のインデックスで、create_if_missingが真のときは、インデックスが負なら配列の最初に、正なら配列の最後に新しい値が追加されます。 jsonb_set('[{"f1":1,"f2":null},2,null,3]', '{0,f1}', '[2,3,4]', false) → [{"f1": [2, 3, 4], "f2": null}, 2, null, 3] jsonb_set('[{"f1":1,"f2":null},2]', '{0,f3}', '[2,3,4]') → [{"f1": 1, "f2": null, "f3": [2, 3, 4]}, 2]
jsonb_set_lax (target jsonb, path text[], new_value jsonb [, create_if_missing boolean [, null_value_treatment text]]) → jsonb new_valueがNULLでないなら、jsonb_setと同じ振る舞いをします。そうでなければnull_value_treatmentにしたがいます。null_value_treatmentは、'raise_exception'、'use_json_null'、'delete_key'、'return_target'のいずれかでなければなりません。デフォルトは'use_json_null'です。 jsonb_set_lax('[{"f1":1,"f2":null},2,null,3]', '{0,f1}', null) → [{"f1":null,"f2":null},2,null,3] jsonb_set_lax('[{"f1":99,"f2":null},2]', '{0,f3}', null, true, 'return_target') → [{"f1": 99, "f2": null}, 2]
jsonb_insert (target jsonb, path text[], new_value jsonb [, insert_after boolean]) → jsonb new_valueを挿入してtargetを返します。pathで指定した項目が配列要素で、insert_afterが偽(デフォルトです)ならばnew_valueはその項目の前に挿入され、insert_afterが真であれば後に挿入されます。pathで指定した項目がオブジェクトフィールドならば、オブジェクトがすでにそのキーを含んでいない場合にのみnew_valueが挿入されます。パス中のすべての初期のステップは存在しなければならず、さもなければtargetは変わらないままに返却されます。pathについての演算子について言うと、path内の負の整数はJSON配列の終わりから数えます。パスの最後のステップが範囲外の配列のインデックスで、インデックスが負なら配列の最初に、正なら配列の最後に新しい値が追加されます。 jsonb_insert('{"a": [0,1,2]}', '{a, 1}', '"new_value"') → {"a": [0, "new_value", 1, 2]} jsonb_insert('{"a": [0,1,2]}', '{a, 1}', '"new_value"', true) → {"a": [0, 1, "new_value", 2]}
json_strip_nulls (json) → json jsonb_strip_nulls (jsonb) → jsonb 与えられたJSON値からNULLを持つオブジェクトフィールドをすべて削除します。オブジェクトフィールドではないNULL値は変わりません。 json_strip_nulls('[{"f1":1, "f2":null}, 2, null, 3]') → [{"f1":1},2,null,3]
jsonb_path_exists (target jsonb, path jsonpath [, vars jsonb [, silent boolean]]) → boolean JSONパスが指定したJSON値に対して項目を返すかどうかをチェックします。varsが指定されるなら、それはJSONオブジェクトでなければならず、そのフィールドはjsonpath式に置き換えられる名前を持つ値を提供します。silent引数が指定されていてtrueなら、この関数は@?と@@演算子が生成するのと同じエラーを抑止します。 jsonb_path_exists('{"a":[1,2,3,4,5]}', '$.a[*] ? (@ >= $min && @ <= $max)', '{"min":2, "max":4}') → t
jsonb_path_match (target jsonb, path jsonpath [, vars jsonb [, silent boolean]]) → boolean 指定したJSON値のJSONパス述語チェックの結果を返します。結果の最初の項目だけが考慮されます。結果がBooleanでないなら、nullが返ります。オプションのvarsとsilent引数はjsonb_path_existsと同じように働きます。 jsonb_path_match('{"a":[1,2,3,4,5]}', 'exists($.a[*] ? (@ >= $min && @ <= $max))', '{"min":2, "max":4}') → t

関数

説明

例

jsonb_path_query (target jsonb, path jsonpath [, vars jsonb [, silent boolean]]) → setof jsonb

JSON値に対するJSONパスによって返されるすべてのJSON項目を返します。オプションのvarsとsilent引数はjsonb_path_existsと同じように働きます。

```
select * from jsonb_path_query('{"a":[1,2,3,4,5]}', '$.a[*] ? (@ >= $min && @ <= $max)', '{"min":2, "max":4}') →
```

```
jsonb_path_query
------------------
 2
 3
 4
```

jsonb_path_query_array (target jsonb, path jsonpath [, vars jsonb [, silent boolean]]) → jsonb

JSON値に対するJSONパスによって返されるすべてのJSON項目をJSON配列として返します。オプションのvarsとsilent引数はjsonb_path_existsと同じように働きます。

```
jsonb_path_query_array('{"a":[1,2,3,4,5]}', '$.a[*] ? (@ >= $min && @ <= $max)', '{"min":2, "max":4}') → [2, 3, 4]
```

jsonb_path_query_first (target jsonb, path jsonpath [, vars jsonb [, silent boolean]]) → jsonb

JSON値に対するJSONパスによって返される最初のJSON項目を返します。結果がなければNULLが返ります。オプションのvarsとsilent引数はjsonb_path_existsと同じように働きます。

```
jsonb_path_query_first('{"a":[1,2,3,4,5]}', '$.a[*] ? (@ >= $min && @ <= $max)', '{"min":2, "max":4}') → 2
```

jsonb_path_exists_tz (target jsonb, path jsonpath [, vars jsonb [, silent boolean]]) → boolean

jsonb_path_match_tz (target jsonb, path jsonpath [, vars jsonb [, silent boolean]]) → boolean

jsonb_path_query_tz (target jsonb, path jsonpath [, vars jsonb [, silent boolean]]) → setof jsonb

jsonb_path_query_array_tz (target jsonb, path jsonpath [, vars jsonb [, silent boolean]]) → jsonb

jsonb_path_query_first_tz (target jsonb, path jsonpath [, vars jsonb [, silent boolean]]) → jsonb

これらの関数は、時間帯を考慮する日時値の比較をサポートすることを除いて、上で述べた、_tz接尾を除いた片割れの関数のように動作します。以下の例では日付のみの値2015-08-02を時間帯付きタイムスタンプとして解釈することが必要で、結果はTimeZone設定に依存します。この依存性のために、これらの関数は安定(stable)、として印付けされており、インデックスにはこれらの関数は使えないことを意味します。これらの関数の片割れは不変(immutable)なので、インデックスで使えます。しかし、そうした比較を要求されるとエラーを吐きます。

```
jsonb_path_exists_tz('["2015-08-01 12:00:00 -05"]', '$[*] ? (@.datetime() < "2015-08-02".datetime())') → t
```

jsonb_pretty (jsonb) → text

与えられたJSON値を整形されたインデント付きテキストに変換します。

```
jsonb_pretty('[{"f1":1,"f2":null}, 2]') →
```

```
[
    {
        "f1": 1,
        "f2": null
```

関数 説明 例
}, 2]
json_typeof (json) → text jsonb_typeof (jsonb) → text トップレベルのJSON値の型をテキスト文字列として返します。可能な型は次のとおりです。object、array、string、number、boolean、null。(nullの結果をSQLのNULLと混同してはいけません。以下の例をご覧ください。) json_typeof('-123.4') → number json_typeof('null'::json) → null json_typeof(NULL::json) IS NULL → t

レコードの値をJSONに集約するjson_agg集約関数、値の対をJSONオブジェクトに集約するjson_object_agg集約関数、およびそれらのjsonb版のjsonb_aggとjsonb_object_aggについては9.21を参照して下さい。

9.16.2. SQL/JSONパス言語

SQL/JSONパス式は、XMLへのSQLアクセスで使用されるXPath同様、JSONデータから取り出す項目を指定します。PostgreSQLではパス式はjsonpathデータ型として実装されており、8.14.6で説明されているすべての要素を使うことができます。

JSON問い合わせ関数と演算子は与えられたパス式を*path engine*に渡して評価します。式が問い合わせ対象のJSONデータにマッチすれば、関連するSQL/JSON項目が返却されます。パス式はSQL/JSONパス言語で書かれ、算術式と関数を含むことができます。

パス式はjsonpathデータ型で認められた一連の要素からなります。パス式は通常左から右へと評価されますが、括弧を使って演算の順序を変更することができます。評価が成功すれば、一連のJSON項目が生成され、評価結果が指定した計算を完了したJSON問い合わせ関数に戻されます。

問い合わせ対象(*context item*)のJSONデータを参照するには、パス式内で$値を使います。複数のアクセサ演算子をその後に記述することもできます。それによってJSON構造をレベル順に訪れて文脈項目の副項目の内容を取り出します。後続の個々の演算子はその前の評価段階の結果を処理します。

たとえば、次のようなパースしたいGPSトラッカーからのJSONデータがあるとします。

```
{
  "track": {
    "segments": [
      {
        "location":   [ 47.763, 13.4034 ],
        "start time": "2018-10-14 10:05:14",
        "HR": 73
      },
      {
```

```
        "location":   [ 47.706, 13.2635 ],
        "start time": "2018-10-14 10:39:21",
        "HR": 135
      }
    ]
  }
}
```

存在するトラックセグメントを取り出すには、.keyアクセサ演算子を使用して、周辺のJSONオブジェクトを下っていく必要があります。

```
$.track.segments
```

配列の内容を取り出すには、典型的には[*]演算子を使います。たとえば次のパスはすべての存在するトラックセグメントの位置座標を返します。

```
$.track.segments[*].location
```

最初のセグメントの座標だけを返すには、[]アクセサ演算子の中で対応する添え字を指定することができます。JSON配列インデックスは0スタートであることに注意してください。

```
$.track.segments[0].location
```

各段階でのパス評価結果は9.16.2.2に列挙されている一つ以上のjsonpath演算子とメソッドで処理することができます。各々のメソッド名の前にピリオドを付けなければなりません。たとえば配列の大きさを得ることができます。

```
$.track.segments.size()
```

パス式内のjsonpath演算子とメソッドを使用する他の例については以下の9.16.2.2を参照してください。

パスを定義する際にはSQLのWHERE節のように働く一つ以上のフィルター式が利用できます。フィルター式はクェスチョンマークで始まり、カッコ内に条件を記述します。

```
? (condition)
```

フィルター式はそれを適用するパス評価段階の直後に指定しなければなりません。この段階の結果は、指定した条件を満たす項目だけが含まれるようにフィルターされます。SQL/JSONは3値論理を定義しており、条件はtrue、false、unknownのどれかです。unknownは値はSQLのNULLと同じ役割を果たし、is unknown述語で評価できます。その後の評価段階ではtrueを返すフィルター式に対応する項目だけが使われます。

フィルター式内で利用できる関数と演算子は表 9.49にリストされています。フィルター式内では、フィルターする必要のある値は@変数で示します。(つまり以前のパスステップの結果の一つです。) コンポーネント項目を取得するためにアクセサ演算子を@の後に記述することができます。

たとえば130より高いすべての心拍数を取り出したいとします。次の式を使ってそれを得ることができます。

```
$.track.segments[*].HR ? (@ > 130)
```

そうした値を持つセグメントの開始時刻を得たい場合は、開始時刻を返す前に無関係のセグメントを取り除く必要があります。そうすることにより前の段階にフィルター式が適用されるので、その条件で適用されるパスは異なります。

```
$.track.segments[*] ? (@.HR > 130)."start time"
```

必要なら複数のフィルター式を順に使用することができます。たとえば次の式は指定した座標と高い心拍数値を持つ位置を持つすべてのセグメントを選択します。

```
$.track.segments[*] ? (@.location[1] < 13.4) ? (@.HR > 130)."start time"
```

異なる入れ子レベルに対してフィルター式を適用することもできます。次の例では、まず位置ですべてのセグメントをフィルターし、もしあれば高い心拍数値を返します。

```
$.track.segments[*] ? (@.location[1] < 13.4).HR ? (@ > 130)
```

フィルター式をお互いに入れ子にすることもできます。

```
$.track ? (exists(@.segments[*] ? (@.HR > 130))).segments.size()
```

この式は高い心拍数値を含むトラックがあればそのすべてのサイズを返します。もしなければ空のシーケンスが返ります。

PostgreSQLのSQL/JSONパス言語の実装はSQL/JSON標準と次の点が異なります。

- SQL/JSON標準ではフィルター内でのみ述語が使えますが、パス式はBoolean述語でも構いません。これは@@演算子を実装するために必要です。たとえば、次のjsonpath式はPostgreSQLでは有効です。

  ```
  $.track.segments[*].HR < 70
  ```

- 9.16.2.3で述べるように、like_regexフィルターで使用される正規表現パターンの解釈には些細な違いがあります。

9.16.2.1. 厳密モードと非厳密モード

JSONデータを問い合わせる際、パス式は実際のJSONデータ構造に一致しないかも知れません。存在しないオブジェクトのメンバーあるいは配列要素にアクセスしようとすると、構造上のエラーとなります。SQL/JSONパス式には構造上のエラーを扱うための2つのモードがあります。

- 非厳密(lax)モード（デフォルト）— パスエンジンは指定したパスを問い合わせデータに暗黙的に適合させます。構造上のエラーは抑止され、空のSQL/JSONシーケンスへと変換されます。
- 厳密(strict)モード — 構造上のエラーがあるとエラーが発生します。

非厳密モードは、JSONデータが期待されるスキーマに沿わないときにJSON文書構造とパス式のマッチングを助けます。あるオペランドが操作の要件に合わないときにはそれをSQL/JSON配列にまとめたり、あるいは操作を行う前にそれをSQL/JSONシーケンスに展開することもできます。また非厳密モードにおいては、比較演算子は自動的にオペランドを展開し、SQL/JSON配列をそのまま比較することができます。大きさ1の配列はその単独要素と同じものとして扱われます。自動展開は以下の場合にのみ行われません。

- それぞれ配列の型、要素数を返すtype()、size()をパス式が含む。
- 問い合わせ対象のJSONデータが入れ子の配列を含む。この場合はもっとも外側の配列のみが展開され、内側の配列は変わりません。ですから、それぞれの評価段階において1レベルのみに暗黙的な展開が行われます。

たとえば、上述のGPSデータに問い合わせする際、非厳密モードでは配列のセグメントを含んでいることを抽象化できます。

```
lax $.track.segments.location
```

厳密モードでは、指定したパスはSQL/JSON項目を返す問い合わせ対象のJSON文書の構造に正確に一致していなければなりません。ですから、このパス式を使うとエラーになります。非厳密モードと同じ結果を得るためには、segments配列を明示的に展開する必要があります。

```
strict $.track.segments[*].location
```

9.16.2.2. SQL/JSONパス演算子とメソッド

表 9.48にjsonpathで利用可能な演算子とメソッドを示します。単項演算子とメソッドは以前のパスステップから生じた複数の値に適用できますが、二項演算子（加算など）は単一の値にしか適用できないことに注意してください。

表9.48 jsonpath演算子とメソッド

演算子/メソッド 説明 例
number + number → number 加算 jsonb_path_query('[2]', '$[0] + 3') → 5
+ number → number 単項のプラス（演算なし）。加算と違って、複数の値に渡って適用できます。 jsonb_path_query_array('{"x": [2,3,4]}', '+ $.x') → [2, 3, 4]
number - number → number 減算 jsonb_path_query('[2]', '7 - $[0]') → 5
- number → number 負符号。減算と違って、複数の値に渡って適用できます。 jsonb_path_query_array('{"x": [2,3,4]}', '- $.x') → [-2, -3, -4]
number * number → number 乗算 jsonb_path_query('[4]', '2 * $[0]') → 8
number / number → number 除算 jsonb_path_query('[8.5]', '$[0] / 2') → 4.2500000000000000

演算子/メソッド 説明 例
number % number → number 剰余（残り） jsonb_path_query('[32]', '$[0] % 10') → 2
value . type() → string JSON項目の型（json_typeofを参照） jsonb_path_query_array('[1, "2", {}]', '$[*].type()') → ["number", "string", "object"]
value . size() → number JSON項目の大きさ（配列の要素数。配列でなければ1） jsonb_path_query('{"m": [11, 15]}', '$.m.size()') → 2
value . double() → number JSON数値あるいは文字列から変換した概算の浮動小数点数 jsonb_path_query('{"len": "1.9"}', '$.len.double() * 2') → 3.8
number . ceiling() → number 引数より大きいか等しく、与えられた数に最も近い整数 jsonb_path_query('{"h": 1.3}', '$.h.ceiling()') → 2
number . floor() → number 引数より小さいか等しく、与えられた数に最も近い整数 jsonb_path_query('{"h": 1.7}', '$.h.floor()') → 1
number . abs() → number 与えられた数の絶対値 jsonb_path_query('{"z": -0.3}', '$.z.abs()') → 0.3
string . datetime() → datetime_type (see note) 文字列から変換した日時値 jsonb_path_query('["2015-8-1", "2015-08-12"]', '$[*] ? (@.datetime() < "2015-08-2".datetime())') → "2015-8-1"
string . datetime(template) → datetime_type (see note) 指定のto_timestampテンプレートを使って文字列から変換した日時値 jsonb_path_query_array('["12:30", "18:40"]', '$[*].datetime("HH24:MI")') → ["12:30:00", "18:40:00"]
object . keyvalue() → array 以下の3つのフィールドを含むオブジェクトの配列で表現したオブジェクトのキー/値ペア。"key"、"value"、"id"。"id"はキー/値ペアが属するオブジェクトのユニーク識別子です。 jsonb_path_query_array('{"x": "20", "y": 32}', '$.keyvalue()') → [{"id": 0, "key": "x", "value": "20"}, {"id": 0, "key": "y", "value": 32}]

注記

datetime()とdatetime(template)の結果型はdate、timetz、time、timestamptz、あるいはtimestampです。両方のメソッドは結果型を動的に決定します。

datetime()メソッドは入力文字列をdate、timetz、time、timestamptz、timestampのISO形式に対して順にマッチを試みます。最初にマッチした形式で停止し、関連するデータ型を出力します。

datetime(template)メソッドは与えられたテンプレート文字列にあるフィールドに従って結果型を決定します。

datetime()とdatetime(template)はto_timestampSQL関数と同じ解析ルール(参照9.8)を用いますが、3つの例外があります。一番目に、これらのメソッドは一致しないテンプレートパターンを許容しません。二番目に次の区切り文字のみを許容します。負符号、ピリオド、斜線(スラッシュ)、カンマ、アポストロフィー、セミコロン、コロン、空白、です。三番目にテンプレート文字列中の区切り文字は正確に入力文字列と一致しなければなりません。

異なる日時型の比較が必要なら、暗黙的なキャストが適用されます。date値はtimestampあるいはtimestamptzにキャストできます。timestampはtimestamptzに、timeはtimetzにキャストできます。しかし、これらの変換の最初のものは現在のTimeZone設定に依存します。ですから時間帯を認識するjsonpath関数中でのみ実行可能です。

表 9.49に利用可能なフィルター式要素を示します。

表9.49 jsonpathフィルター式要素

述語/値 説明 例
value == value → boolean 等値比較(これと他の比較演算子はすべてのJSONスカラー値で使えます) jsonb_path_query_array('[1, "a", 1, 3]', '$[*] ? (@ == 1)') → [1, 1] jsonb_path_query_array('[1, "a", 1, 3]', '$[*] ? (@ == "a")') → ["a"]
value != value → boolean value <> value → boolean 非等値比較 jsonb_path_query_array('[1, 2, 1, 3]', '$[*] ? (@ != 1)') → [2, 3] jsonb_path_query_array('["a", "b", "c"]', '$[*] ? (@ <> "b")') → ["a", "c"]
value < value → boolean 未満比較 jsonb_path_query_array('[1, 2, 3]', '$[*] ? (@ < 2)') → [1]
value <= value → boolean 以下比較 jsonb_path_query_array('["a", "b", "c"]', '$[*] ? (@ <= "b")') → ["a", "b"]
value > value → boolean より大きい比較 jsonb_path_query_array('[1, 2, 3]', '$[*] ? (@ > 2)') → [3]
value >= value → boolean 以上比較 jsonb_path_query_array('[1, 2, 3]', '$[*] ? (@ >= 2)') → [2, 3]
true → boolean JSON定数真

述語/値 説明 例
jsonb_path_query('[{"name": "John", "parent": false}, {"name": "Chris", "parent": true}]', '$[*] ? (@.parent == true)') → {"name": "Chris", "parent": true}
false → boolean JSON定数偽 jsonb_path_query('[{"name": "John", "parent": false}, {"name": "Chris", "parent": true}]', '$[*] ? (@.parent == false)') → {"name": "John", "parent": false}
null → value JSON定数null(SQLとは違ってnullとの比較は通常通り動作することに注意してください。) jsonb_path_query('[{"name": "Mary", "job": null}, {"name": "Michael", "job": "driver"}]', '$[*] ? (@.job == null) .name') → "Mary"
boolean && boolean → boolean 論理AND jsonb_path_query('[1, 3, 7]', '$[*] ? (@ > 1 && @ < 5)') → 3
boolean || boolean → boolean 論理OR jsonb_path_query('[1, 3, 7]', '$[*] ? (@ < 1 || @ > 5)') → 7
! boolean → boolean 論理NOT jsonb_path_query('[1, 3, 7]', '$[*] ? (!(@ < 5))') → 7
boolean is unknown → boolean 論理条件がunknownであるかどうかを検査します。 jsonb_path_query('[-1, 2, 7, "foo"]', '$[*] ? ((@ > 0) is unknown)') → "foo"
string like_regex string [flag string] → boolean 最初のオペランドが2番目のオペランドで与えられる正規表現にマッチするかどうか検査します。オプションでflag文字列で記述される変更を伴います。(9.16.2.3を参照してください。) jsonb_path_query_array('["abc", "abd", "aBdC", "abdacb", "babc"]', '$[*] ? (@ like_regex "^ab.*c")') → ["abc", "abdacb"] jsonb_path_query_array('["abc", "abd", "aBdC", "abdacb", "babc"]', '$[*] ? (@ like_regex "^ab.*c" flag "i")') → ["abc", "aBdC", "abdacb"]
string starts with string → boolean 2番目の文字列が1番目のオペランドの最初の部分文字列かどうかを検査します。 jsonb_path_query('["John Smith", "Mary Stone", "Bob Johnson"]', '$[*] ? (@ starts with "John")') → "John Smith"
exists (path_expression) → boolean パス式が少なくとも一つのSQL/JSON項目とマッチするかどうかを検査します。パス式がエラーとなる場合はunknownを返します。2番目の例は厳密モードでキーが存在しないエラーを回避するためにこれを使っています。 jsonb_path_query('{"x": [1, 2], "y": [2, 4]}', 'strict $.* ? (exists (@ ? (@[*] > 2)))') → [2, 4] jsonb_path_query_array('{"value": 41}', 'strict $? (exists (@.name)) .name') → []

9.16.2.3. SQL/JSON正規表現

SQL/JSONパス式ではlike_regexフィルターを使ってテキストを正規表現にマッチさせることができます。たとえば、次のSQL/JSONパス式問い合わせは、英語の母音で始まる配列内のすべての文字列に大文字小文字を無視してマッチするでしょう。

```
$[*] ? (@ like_regex "^[aeiou]" flag "i")
```

オプションのflag文字列は一つ以上の文字を含むことができます。iは大文字小文字を無視したマッチ、mは^と$で改行にマッチ、sは.が改行にマッチ、qはパターン全体を参照します。(振る舞いを単純な部分文字列マッチとします)

SQL/JSON標準は正規表現の定義を、XQuery標準を使用するLIKE_REGEX演算子から借りています。PostgreSQLは今の所LIKE_REGEX演算子をサポートしていません。ですから、like_regexフィルターは9.7.3で説明されているPOSIX正規表現で実装されています。このことにより、9.7.3.8で列挙されているSQL/JSON標準の振る舞いとの小さな違いが生じます。しかし、ここで述べているフラグ文字の非互換性はSQL/JSONには適用されないことに注意してください。SQL/JSONは、XQueryのフラグ文字をPOSIXエンジンが期待するのと一致するように解釈するからです。

like_regexのパターン引数は8.14.6で説明されているルールにしたがって書かれたJSONパス文字列リテラルであることに注意してください。これは、正規表現で使用するすべてのバックスラッシュを二重に書かなければならないことを意味します。たとえば、数字のみを含む文字列にマッチさせるには以下のようにします。

```
$ ? (@ like_regex "^\\d+$")
```

9.17. シーケンス操作関数

本節ではシーケンスオブジェクトに対し演算を行う関数について説明します。シーケンスオブジェクトは、シーケンスジェネレータ、あるいは単にシーケンスとも呼ばれます。シーケンスオブジェクトは特殊な一行だけのテーブルで、CREATE SEQUENCEで作成されます。シーケンスオブジェクトは一般的にテーブルの行に一意の識別子を生成するために使用されます。表 9.50に列挙されているシーケンス関数は、シーケンスオブジェクトから連続したシーケンス値を取得するための、簡易でマルチユーザに対応した関数です。

表9.50 シーケンス関数

関数 説明
nextval (regclass) → bigint シーケンスを次の値に進めてその値を返します。これは自動的に行われます。複数のセッションがnextvalを同時に実行しても、各々のシーケンスは異なったシーケンス値を安全に返します。シーケンスオブジェクトがデフォルト値を伴って作成されると、後続のnextval呼び出しは1から始まる次の値を返します。それ以外の動作は適切なパラメータをCREATE SEQUENCEコマンドで使うことによって得られます。 この関数はシーケンスオブジェクトのUSAGEあるいはUPDATE権限が必要です。
setval (regclass, bigint [, boolean]) → bigint シーケンスオブジェクトの現在の値をセットします。オプションでis_calledをセットします。2つのパラメータを持つ形式では、シーケンスのlast_valueフィールドを指定した値にセットし、is_calledフィールドをtrueに設定します。これは次のnextva

関数	説明

	lが値を返す前にシーケンスを増分することを意味します。currvalで報告される値も指定した値に設定されます。3つのパラメータを持つ形式では、is_calledはtrueあるいはfalseに設定されます。trueは2つのパラメータを持つ形式と同じ効果を持ちます。falseに設定されていると、次のnextvalはまさに指定した値を返し、後続のnextvalがシーケンスの増加を開始します。更に、この場合はcurrvalが報告する値は変化しません。たとえば次ようになります。
	`SELECT setval('myseq', 42);  次のnextvalは43を返す`
	`SELECT setval('myseq', 42, true);  同上`
	`SELECT setval('myseq', 42, false);  次のnextvalは42を返す`
	setvalが返した値はその2番目の引数と単に同じです。
	この関数はシーケンスのUPDATE権限が必要です。

currval (regclass) → bigint	現在のセッションでこのシーケンスに対して直近のnextvalによって得られた値を返します。(このセッションでnextvalが呼ばれていなければエラーが報告されます。) これはセッションローカルな値を返すので、他のセッションがnextvalを呼び出したかどうかに関わらず予測可能な値を返します。 この関数はシーケンスのUSAGEあるいはSELECT権限が必要です。
lastval () → bigint	現在のセッションでこのシーケンスに対して直近のnextvalによって得られた値を返します。この関数は、現在のセッションでnextvalが直近に適用されたシーケンス名を参照する引数を取ることを除き、currvalと同じです。このセッションでnextvalが呼ばれていないのにlastvalを呼び出すのはエラーです。 この関数はシーケンスのUSAGEあるいはSELECT権限が必要です。

注意

同一のシーケンスから数値を取得する同時実行トランザクション同士のブロックを防止するため、nextval演算は決してロールバックされません。つまり、値が一度取り出されたら、それは使用されたものと見なされ、同じ値が再び返されることはありません。これは、それを取り囲むトランザクションが後にアボートした場合でも、あるいは呼び出し側の問い合わせがその値を使用せずに終わった場合でも当てはまります。例えばON CONFLICT句のあるINSERTでは、挿入される予定のタプルについて、必要となるすべてのnextvalの呼び出しも含めて計算し、その後でON CONFLICTのルールを代わりに使用することになる競合について検知します。このような場合には、割り当てられた値のシーケンス内に未使用の「欠損」を残すことになります。従って、PostgreSQLのシーケンスオブジェクトは「欠番のない」シーケンスを得るために使うことはできません。

同様に、setvalが行ったシーケンス状態の変更はトランザクションがロールバックしても元には戻りません。

シーケンス関数により操作されるシーケンスはregclass引数で指定されますが、それはpg_classシステムカタログ内にある、そのシーケンスの単なるOIDです。しかしながら、手作業でOIDを検索する必要はなく、regclassデータ型の入力変換器が代わってその作業を行ってくれます。単一引用符で括られたシーケンス名を記述するだけで良いので、リテラル定数のように見えます。通常のSQLの名称での操作との互換のため、文字列はシーケンス名が二重引用符で括られていなければ、小文字に変換されます。よって、以下のようになります。

```
nextval('foo')      シーケンスfooの操作
nextval('FOO')      シーケンスfooの操作
nextval('"Foo"')    シーケンスFooの操作
```

必要であれば、以下のようにシーケンス名をスキーマで修飾することができます。

```
nextval('myschema.foo')     myschema.fooの操作
nextval('"myschema".foo')   上と同じ
nextval('foo')              fooを検索パスで探す
```

regclassに関してのより詳細な情報は8.19を参照してください。

注記

PostgreSQLの8.1より前においては、シーケンス関数の引数はregclass型ではなく、text型で、そして上記のテキスト文字列からOID値への変換はそれぞれの呼び出し実行時に起こりました。後方互換性のため、この仕組みはまだ存在しますが、内部的には関数が実行される前にtextからregclassへの暗黙的強制型変換として現在処理されています。

ありのままのリテラル文字列としてシーケンス関数の引数を記述すると、regclassデータ型の定数になります。これは単なるOIDなので、後で名前付けが再び行われたとか、スキーマの再割り振りとかに関わらず、最初に特定されたシーケンスを引き継ぎます。この「初期束縛」の動作は、列のデフォルトやビューからシーケンスを参照する場合は望ましいことが多いでしょう。しかし、実行時にシーケンス参照が解決されるような「動的束縛」が望まれる場合もあります。動的束縛の動作を得るには、その定数をregclassではなくtext定数として保存させます。

```
nextval('foo'::text)      fooは実行時に検索される
```

PostgreSQLのリリース8.1より前では動的束縛のみがサポートされる動作だったので、旧来のアプリケーションのセマンティクスを保ちたい場合このようにする必要があるかもしれません。

もちろん、シーケンス関数の引数は定数だけでなく、評価式とすることも可能です。テキスト式の場合は暗黙的型変換により、実行時検索が行われます。

9.18. 条件式

本節ではPostgreSQLで使用可能なSQL準拠の条件式について説明します。

ヒント

ここで説明する条件式より発展した機能を求める場合は、より表現の豊富なプログラム言語でストアドプロシージャを記述することで解決されます。

> **注記**
>
> COALESCE、GREATEST、LEASTは構文的には関数に似ていますが通常の関数ではなく、明示的なVARIADIC配列引数と一緒には使えません。

9.18.1. CASE

SQLのCASE式は他のプログラミング言語のif/else構文に類似した汎用条件式です。

```
CASE WHEN condition THEN result
     [WHEN ...]
     [ELSE result]
END
```

CASE句は式が有効な位置であればどこでも使用可能です。それぞれのconditionとはboolean型の結果を返す式です。もしconditionの結果が真であれば、CASE式の値は、conditionに続くresultとなります。そして、CASE式の残りは処理されません。もしconditionの結果が偽であれば後に続く全てのWHEN句が同じようにして調べられます。WHENのconditionの1つも真でない場合、CASE式の値はELSE句のresultになります。ELSE句が省略され、どのconditionも真でない場合、結果はNULLです。

以下に例を示します。

```
SELECT * FROM test;

 a
---
 1
 2
 3


SELECT a,
       CASE WHEN a=1 THEN 'one'
            WHEN a=2 THEN 'two'
            ELSE 'other'
       END
    FROM test;

 a | case
---+-------
 1 | one
 2 | two
 3 | other
```

全てのresult式のデータ型は単一の出力型に変換可能でなければなりません。詳細は10.5を参照してください。

以下のように、上記の一般的な形式と異なるCASE式の「単純な」形式が存在します。

```
CASE expression
    WHEN value THEN result
    [WHEN ...]
    [ELSE result]
END
```

最初のexpressionが計算され、そしてそれに等しいものが見つかるまでWHEN句のそれぞれのvalue式と比較されます。等しいものが見つからない場合、ELSE句のresult(もしくはNULL値)が返されます。これはC言語のswitch文に似ています。

上の例は簡略形CASE構文を使って次のように書くことができます。

```
SELECT a,
       CASE a WHEN 1 THEN 'one'
              WHEN 2 THEN 'two'
              ELSE 'other'
       END
    FROM test;

 a | case
---+-------
 1 | one
 2 | two
 3 | other
```

CASE式は、結果を決定するために不必要などんな副式をも評価しません。例えば、以下は0除算エラーを防ぐための方法です。

```
SELECT ... WHERE CASE WHEN x <> 0 THEN y/x > 1.5 ELSE false END;
```

> **注記**
>
> 4.2.14で説明したとおり、式の副式が異なる時点で評価される様々な状況があります。そのため「CASEは必要な副式のみを評価する」という原則は厳格なものではありません。例えば、定数1/0副式は、実行時には決して入らないCASE節の中にあったとしても、通常は計画時にゼロによる除算での失敗という結果に終わります。

9.18.2. COALESCE

```
COALESCE(value [, ...])
```

COALESCE関数は、NULLでない自身の最初の引数を返します。全ての引数がNULLの場合にのみNULLが返されます。データを表示目的で取り出す際、NULL値をデフォルト値で置き換えるためによく使用されています。以下に例を示します。

```
SELECT COALESCE(description, short_description, '(none)') ...
```

これはdescriptionがNULLでなければそれを返します。そうでない場合（NULLの場合）は、short_descriptionがNULLでなければそれを返します。それ以外の場合（short_descriptionもNULLの場合）は(none)が返ります。

引数はすべて共通の型に変換できる必要があり、それが結果の型になります。（詳細は10.5を参照してください。）

CASE式同様、COALESCEは結果を決定するために必要な引数のみを評価します。つまり、非NULL引数が見つかれば、その右側にある引数は評価されません。このSQL標準関数は、他のいくつかのデータベースで使用されているNVLおよびIFNULLと類似の機能を提供します。

9.18.3. NULLIF

```
NULLIF(value1, value2)
```

NULLIF関数は、value1がvalue2と等しい場合、NULL値を返します。その他の場合はvalue1を返します。これを使って、上記のCOALESCEの例の逆演算を実行できます

```
SELECT NULLIF(value, '(none)') ...
```

この例では、value1が(none)ならばNULLが返ります。さもなくばvalue1を返します

2つの引数は比較可能な型でなければなりません。具体的には、あたかもvalue1 = value2と書いたように比較されるので、適当な=演算子が使用できなければなりません。

結果は最初の引数と同じ型ですが、微妙な場合があります。実際に返却されるのは=演算子が暗示する最初の引数で、場合によっては2番目の引数にマッチするように昇格されています。たとえばNULLIF(1, 2.2)はnumericを出力します。なぜならinteger = numeric演算子はなく、numeric = numericがあるだけだからです。

9.18.4. GREATESTおよびLEAST

```
GREATEST(value [, ...])
```

```
LEAST(value [, ...])
```

GREATESTとLEAST関数は任意の数の式のリストから最大値もしくは最小値を選択します。評価される全ての式は共通の型に変換できる必要があり、それが結果の型になります（詳細は10.5を参照してください）。リストの中のNULL値は無視されます。全ての式がNULLと評価された場合に限って結果はNULLになります。

GREATESTおよびLEASTはSQL標準に載っていませんが、共通した拡張です。他のいくつかのデータベースでは、全てがNULLの場合に限定せず、いずれかの引数がNULLである場合にNULLを返すようにしているものもあります。

9.19. 配列関数と演算子

表 9.51に、配列型専用に利用可能な演算子を示します。これらに加えて表 9.1で示す通常の比較演算子が配列で利用できます。比較演算子は配列の内容をその要素のデータ型用のデフォルトのB-tree比較関数を要素単位で比較し、最初にどの要素に違いがあったかに基づいてソートします。多次元配列では配列の要素は行優先順にアクセスされます。（最後の添字が最初に変化します。）2つの配列の内容が同じで次元数が異なる場合は、どの次元で最初に違いがあったかによってソート順が決まります。（これはPostgreSQLの8.2より前のバージョンでは異なります。古いバージョンでは、次元数や添字の範囲が異なっても、2つの配列の内容が同じなら、同じ配列であるとしていました。）

表9.51 配列演算子

演算子 説明 例
anyarray @> anyarray → boolean 最初の配列が2番目を含んでいるか？すなわち、2番目の配列の各要素は最初の配列のいくつかの要素と同じであるか？（重複は特に考慮されないので、ARRAY[1]とARRAY[1,1]はそれぞれがお互いに相手を含んでいると見なされます。） ARRAY[1,4,3] @> ARRAY[3,1,3] → t
anyarray <@ anyarray → boolean 最初の配列は2番目に含まれているか？ ARRAY[2,2,7] <@ ARRAY[1,7,4,2,6] → t
anyarray && anyarray → boolean 配列は重なり合っているか？すなわち、共通の要素を持っているか？ ARRAY[1,4,3] && ARRAY[2,1] → t
anyarray || anyarray → anyarray 2つの配列を結合します。nullあるいは空の配列の結合は無処理です。そうでない場合は、配列は同じ次元数を持っていなければなりません。（最初の例にあるように）。さもなければ次元数でひとつ違わなければなりません（2番目の例にあるように）。 ARRAY[1,2,3] || ARRAY[4,5,6,7] → {1,2,3,4,5,6,7} ARRAY[1,2,3] || ARRAY[[4,5,6],[7,8,9]] → {{1,2,3},{4,5,6},{7,8,9}}
anyelement || anyarray → anyarray 配列（空か一次元の配列でなければなりません）の先頭に要素を結合します。 3 || ARRAY[4,5,6] → {3,4,5,6}
anyarray || anyelement → anyarray 配列（空か一次元の配列でなければなりません）の最後に要素を結合します。 ARRAY[4,5,6] || 7 → {4,5,6,7}

配列演算子の振舞いの詳細は8.15を参照してください。どの演算子がインデックス付きの操作をサポートしているかのより詳細については11.2を参照してください。

表 9.52に配列型で使用可能な関数を示します。これらの関数の情報と例については8.15を参照してください。

表9.52 配列関数

関数 説明 例
array_append (anyarray, anyelement) → anyarray 配列の最後に要素を追加します。(anyarray || anyelement演算子と同じです。) array_append(ARRAY[1,2], 3) → {1,2,3}
array_cat (anyarray, anyarray) → anyarray 2つの配列を結合します。(anyarray || anyarray演算子と同じです。) array_cat(ARRAY[1,2,3], ARRAY[4,5]) → {1,2,3,4,5}
array_dims (anyarray) → text 配列の次元をテキスト表現で返します。 array_dims(ARRAY[[1,2,3], [4,5,6]]) → [1:2][1:3]
array_fill (anyelement, integer[] [, integer[]]) → anyarray 与えられた値のコピーで満たされた2番目の引数で指定した次元の長さを持つ配列を返します。オプションの3番目の引数は各次元の下限値を与えます(デフォルトはすべて1です)。 array_fill(11, ARRAY[2,3]) → {{11,11,11},{11,11,11}} array_fill(7, ARRAY[3], ARRAY[2]) → [2:4]={7,7,7}
array_length (anyarray, integer) → integer 要求された配列の次元の長さを返します。 array_length(array[1,2,3], 1) → 3
array_lower (anyarray, integer) → integer 要求された配列の次元の下限を返します。 array_lower('[0:2]={1,2,3}'::integer[], 1) → 0
array_ndims (anyarray) → integer 配列の次元数を返します。 array_ndims(ARRAY[[1,2,3], [4,5,6]]) → 2
array_position (anyarray, anyelement [, integer]) → integer 2番目の引数が最初に配列に現れた添字を返します。存在しなければNULLを返します。3番目の引数が与えられるとその添字から検索が始まります。配列は一次元でなければなりません。比較はIS NOT DISTINCT FROMの意味論で行われるので、NULLを検索することができます。 array_position(ARRAY['sun', 'mon', 'tue', 'wed', 'thu', 'fri', 'sat'], 'mon') → 2
array_positions (anyarray, anyelement) → integer[] 2番目の引数が配列に現れるすべての添字を配列で返します。存在しなければNULLを返します。3番目の引数が与えられるとその添字から検索が始まります。配列は一次元でなければなりません。比較はIS NOT DISTINCT FROMの意味論で行われるので、NULLを検索することができます。配列がNULLのときのみNULLが返ります。値が配列中に見つからなければ空の配列が返ります。 array_positions(ARRAY['A','A','B','A'], 'A') → {1,2,4}

関数
説明
例

array_prepend (anyelement, anyarray) → anyarray
配列の先頭に要素を追加します。(anyelement || anyarray演算子と同じです。)
array_prepend(1, ARRAY[2,3]) → {1,2,3}

array_remove (anyarray, anyelement) → anyarray
与えられた値と等しい要素を配列から削除します。配列は一次元でなければなりません。比較はIS NOT DISTINCT FROMの意味論で行われるので、NULLを削除することができます。
array_remove(ARRAY[1,2,3,2], 2) → {1,3}

array_replace (anyarray, anyelement, anyelement) → anyarray
2番目の引数と等しい要素を3番目の引数で置き換えます。
array_replace(ARRAY[1,2,5,4], 5, 3) → {1,2,3,4}

array_to_string (array anyarray, delimiter text [, null_string text]) → text
配列要素をテキスト表現に変換しdelimiter文字列で区切って結合します。NULLでないnull_stringが与えられると、配列要素をその文字列で表現します。さもなければ無視されます。
array_to_string(ARRAY[1, 2, 3, NULL, 5], ',', '*') → 1,2,3,*,5

array_upper (anyarray, integer) → integer
要求された配列の次元の上限を返します。
array_upper(ARRAY[1,8,3,7], 1) → 4

cardinality (anyarray) → integer
配列中の要素数を返します。配列が空なら0が返ります。
cardinality(ARRAY[[1,2],[3,4]]) → 4

string_to_array (string text, delimiter text [, null_string text]) → text[]
stringをdelimiterが現れるところで分割し、残りのデータをtext配列にします。delimiterがNULLなら、string中の文字は配列中の別個の要素になります。delimiterが空文字列なら、stringは単一のフィールドとして扱われます。null_stringが提供されてNULLでなければ、その文字列とマッチするフィールドはNULLのエントリーに変換されます。
string_to_array('xx~~yy~~zz', '~~', 'yy') → {xx,NULL,zz}

unnest (anyarray) → setof anyelement
配列を行の集合に展開します。
unnest(ARRAY[1,2]) →

```
1
 2
```

unnest (anyarray, anyarray [, ...]) → setof anyelement, anyelement [, ...]
複数の配列(異なるデータ型の可能性があります)を行の集合に展開します。配列の長さが同じでなければ、短い配列にはNULLが詰められます。これはFROM句でのみ許されます。7.2.1.4を参照してください。
select * from unnest(ARRAY[1,2], ARRAY['foo','bar','baz']) as x(a,b) →

```
a |  b
---+-----
 1 | foo
 2 | bar
```

関数 説明 例	
	| baz

> **注記**
>
> string_to_arrayは、PostgreSQL9.1から、前のバージョンとは2つの異なる振る舞いするようになりました。1つ目は、入力した文字列長が0の場合、NULLを返すのではなく空の(要素数が0の)配列を返すようになりました。2つ目は区切り文字列がNULLの場合、以前はNULLを返していましたが9.1からは入力文字列を個別の文字に分割するようになりました。

配列を使用する集約関数array_aggについて、9.21も参照してください。

9.20. 範囲関数と演算子

範囲型の概要については8.17をご覧ください。

表 9.53に、範囲型専用に利用可能な演算子を示します。これらに加えて表 9.1で示す通常の比較演算子が配列で利用できます。この比較演算子は最初に範囲の下限で順序付けし、それが等しい場合にのみ上限を比較します。これは通常有用な全順序付けにはなりませんが、範囲に対して一意インデックスを構成することができる演算子が提供されます。

表9.53 範囲演算子

演算子 説明 例
anyrange @> anyrange → boolean 最初の範囲は2番目を含んでいるか? int4range(2,4) @> int4range(2,3) → t
anyrange @> anyelement → boolean 範囲はその要素を含んでいるか? '[2011-01-01,2011-03-01)'::tsrange @> '2011-01-10'::timestamp → t
anyrange <@ anyrange → boolean 最初の範囲は2番目に含まれるか? int4range(2,4) <@ int4range(1,7) → t
anyelement <@ anyrange → boolean その要素は範囲に含まれるか? 42 <@ int4range(1,7) → f
anyrange && anyrange → boolean 範囲は重なり合っているか?すなわち共通の要素があるか? int8range(3,7) && int8range(4,12) → t
anyrange << anyrange → boolean

演算子 説明 例
最初の範囲は厳密に2番目の左か？ int8range(1,10) << int8range(100,110) → t
anyrange >> anyrange → boolean 最初の範囲は厳密に2番目の右か？ int8range(50,60) >> int8range(20,30) → t
anyrange &< anyrange → boolean 最初の範囲は2番目の右を被覆していないか？ int8range(1,20) &< int8range(18,20) → t
anyrange &> anyrange → boolean 最初の範囲は2番目の左を被覆していないか？ int8range(7,20) &> int8range(5,10) → t
anyrange -|- anyrange → boolean 範囲は隣接しているか？ numrange(1.1,2.2) -|- numrange(2.2,3.3) → t
anyrange + anyrange → anyrange 範囲の和を計算します。範囲は和の結果が単一の範囲になるように、重なり合っているか、隣接していなければなりません。(ただしrange_merge()を参照してください。) numrange(5,15) + numrange(10,20) → [5,20)
anyrange * anyrange → anyrange 範囲の共通部分を計算します。 int8range(5,15) * int8range(10,20) → [10,15)
anyrange - anyrange → anyrange 範囲の差を計算します。差が単一の範囲にならないように、2番目の範囲は最初の範囲に含まれていてはいけません。 int8range(5,15) - int8range(10,20) → [5,10)

空の範囲が含まれる場合、「左」「右」「隣接」演算子は常に偽を返します。つまり、空の範囲は他の範囲の前でも後ろでもないと見なされます。

表 9.54に範囲型で利用可能な関数を示します。

表9.54 範囲関数

関数 説明 例
lower (anyrange) → anyelement 範囲の下限を取り出します。(範囲が空か下限が無限ならばNULLとなります。) lower(numrange(1.1,2.2)) → 1.1
upper (anyrange) → anyelement 範囲の上限を取り出します。(範囲が空か上限が無限ならばNULLとなります。) upper(numrange(1.1,2.2)) → 2.2

関数 説明 例
isempty (anyrange) → boolean 範囲は空か？ isempty(numrange(1.1,2.2)) → f
lower_inc (anyrange) → boolean 範囲の下限は境界を含むか？ lower_inc(numrange(1.1,2.2)) → t
upper_inc (anyrange) → boolean 範囲の上限は境界を含むか？ upper_inc(numrange(1.1,2.2)) → f
lower_inf (anyrange) → boolean 範囲の下限は無限か？ lower_inf('(,)'::daterange) → t
upper_inf (anyrange) → boolean 範囲の上限は無限か？ upper_inf('(,)'::daterange) → t
range_merge (anyrange, anyrange) → anyrange 与えられた両方の範囲を含む最小の範囲を計算します。 range_merge('[1,2)'::int4range, '[3,4)'::int4range) → [1,4)

lower_inc、upper_inc、lower_inf、upper_infはすべて空の範囲に対して偽を返します。

9.21. 集約関数

集約関数は入力値の集合から単一の結果を計算します。表 9.55に組み込みの汎用的な集約関数を、表 9.56に統計集約関数を示します。表 9.57には組み込みのグループ内順序集合集約関数を、一方表 9.58には組み込みのグループ内仮想集合用の順序集約関数を示します。表 9.59には、集約関数と密接に関係するグループ化演算を示します。集約関数の特殊な構文に関する考察は4.2.7で説明されています。また、初歩的な情報については2.7を参照して下さい。

部分モードをサポートする集約関数は並列集約など、様々な最適化に有用です。

表9.55 汎用集約関数

関数 説明	Partial Mode
array_agg (anynonarray) → anyarray NULLも含めてすべての入力値を収集して配列に格納します。	No
array_agg (anyarray) → anyarray すべての入力配列を結合して次元が1高い配列に格納します。（入力配列はすべて同じ次元数を持ち、空もしくはNULLであってはいけません。）	No
avg (smallint) → numeric	Yes

関数 説明	Partial Mode
avg (integer) → numeric avg (bigint) → numeric avg (numeric) → numeric avg (real) → double precision avg (double precision) → double precision avg (interval) → interval すべての非NULL入力値の平均（算術平均）を計算します。	
bit_and (smallint) → smallint bit_and (integer) → integer bit_and (bigint) → bigint bit_and (bit) → bit 全ての非NULLの入力値のビット積を計算します。	Yes
bit_or (smallint) → smallint bit_or (integer) → integer bit_or (bigint) → bigint bit_or (bit) → bit 全ての非NULLの入力値のビット和を計算します。	Yes
bool_and (boolean) → boolean 全ての入力が真ならば真、そうでなければ偽を返します。	Yes
bool_or (boolean) → boolean 入力のどれかが真ならば真、そうでなければ偽を返します。	Yes
count (*) → bigint 入力行数を返します。	Yes
count ("any") → bigint 非NULLの入力行数を返します。	Yes
every (boolean) → boolean これはSQL標準のbool_andと等価です。	Yes
json_agg (anyelement) → json jsonb_agg (anyelement) → jsonb NULLも含めてすべての入力値を収集し、JSON配列に格納します。入力はto_jsonあるいはto_jsonbでJSONに変換されます。	No
json_object_agg (key "any", value "any") → json jsonb_object_agg (key "any", value "any") → jsonb すべてのキー／値ペアをJSONオブジェクトに格納します。キー引数はテキストに変換されます。値はto_jsonあるいはto_jsonbにしたがって変換されます。値はNULLでも構いませんが、キーはNULLにはできません。	No
max (see text) → same as input type 非NULL入力値の最大を計算します。数値、文字列、日時、列挙型およびinet、interval、money、oid、pg_lsn、tid、およびこれらすべての配列でも同様に利用できます。	Yes
min (see text) → same as input type	Yes

関数 説明	Partial Mode
非NULL入力値の最小を計算します。数値、文字列、日時、列挙型およびinet、interval、money、oid、pg_lsn、tid、およびこれらすべての配列でも同様に利用できます。	
string_agg (value text, delimiter text) → text string_agg (value bytea, delimiter bytea) → bytea 非NULL入力を結合して文字列に格納します。最初の値以降、各値の前にdelimiterで指定した値が（NULLでなければ）追加されます。	No
sum (smallint) → bigint sum (integer) → bigint sum (bigint) → numeric sum (numeric) → numeric sum (real) → real sum (double precision) → double precision sum (interval) → interval sum (money) → money 非NULL入力値の合計を計算します。	Yes
xmlagg (xml) → xml 非NULLのXML入力値を結合します。(9.15.1.7参照。)	No

上記の関数は、count関数を除き、1行も選択されなかった場合NULL値を返すことに注意してください。特に、行の選択がないsum関数は、予想されるであろうゼロではなくNULLを返し、そしてarray_aggは、入力行が存在しない場合に、空配列ではなくNULLを返します。必要であれば、NULLをゼロまたは空配列と置換する目的でcoalesce関数を使うことができます。

集約関数array_agg、json_agg、jsonb_agg、json_object_agg、jsonb_object_agg、string_agg、およびxmlagg、そして類似のユーザ定義の集約関数は、入力値の順序に依存した意味のある別の結果値を生成します。この並び順はデフォルトでは指定されませんが、4.2.7に記述されているように、集計呼び出し中にORDER BY句を書くことで制御可能となります。別の方法として、並び替えられた副問い合わせから入力値を供給することでも上手くいきます。例をあげます。

```
SELECT xmlagg(x) FROM (SELECT x FROM test ORDER BY y DESC) AS tab;
```

外側の問い合わせのレベルで結合などの追加処理がある場合、この方法は失敗するかもしれないことに注意して下さい。なぜなら、集約の計算の前に副問い合わせの出力を並べ替える必要があるかも知れないからです。

注記

bool_and、bool_or論理集約関数は標準SQLの集約関数every、anyまたはsomeに対応します。PostgreSQLはeveryをサポートしますが、any、あるいはsomeはサポートしません。anyとsomeの標準の構文には曖昧さがあるからです。

```
SELECT b1 = ANY((SELECT b2 FROM t2 ...)) FROM t1 ...;
```

ここで、副問い合わせが論理値での1行を返す場合、ANYは副問い合わせを導入するもの、もしくは集約関数であるものいずれかとみなすことができます。従って、これらの集約関数に標準の名前を付けることはできません。

注記

他のSQLデータベース管理システムでの作業に親しんだユーザは、count集約関数がテーブル全体に適用される場合の性能に失望するかも知れません。

```
SELECT count(*) FROM sometable;
```

のような問い合わせはテーブルサイズに比例した労力が必要です。PostgreSQLはテーブル全体か、そのテーブルの全ての行を含んだインデックス全体のスキャンを必要とします。

統計解析処理によく使用される集約関数を表 9.56に示します。(これらは、より一般的に使用される集約関数との混乱を防ぐために別出ししました。) numeric_typeを受け付けると表示されている関数は、smallint、integer、bigint、numeric、real、double precisionのすべての型で利用可能です。説明の部分におけるNは、すべての入力式が非NULLの入力行の個数を表します。すべての場合にて、例えばNが0の時など計算が無意味である場合にはNULLが返されます。

表9.56 統計処理用の集約関数

関数 説明	部分モード
corr (Y double precision, X double precision) → double precision 相関係数を計算します。	可
covar_pop (Y double precision, X double precision) → double precision 母共分散を計算します。	可
covar_samp (Y double precision, X double precision) → double precision 標本の共分散を計算します。	可
regr_avgx (Y double precision, X double precision) → double precision 独立変数の平均値を計算します。sum(X)/N.	可
regr_avgy (Y double precision, X double precision) → double precision 従属変数の平均値を計算します。sum(Y)/N.	可
regr_count (Y double precision, X double precision) → bigint 両方の入力が非NULLとなる行数を計算します。	可
regr_intercept (Y double precision, X double precision) → double precision (X, Y)の組み合わせで決まる、最小二乗法による線形方程式のY切片を計算します。	可
regr_r2 (Y double precision, X double precision) → double precision 相関係数の二乗を計算します。	可
regr_slope (Y double precision, X double precision) → double precision	可

関数 説明	部分モード
(X, Y)の組み合わせで決まる、最小二乗法による線型方程式の勾配を計算します。	
regr_sxx (Y double precision, X double precision) → double precision 独立変数の「二乗和」、sum(X^2) - sum(X)^2/Nを計算します。	可
regr_sxy (Y double precision, X double precision) → double precision 独立変数と従属変数の「積の和」、sum(X*Y) - sum(X) * sum(Y)/Nを計算します。	可
regr_syy (Y double precision, X double precision) → double precision 従属変数の「積の和」、sum(Y^2) - sum(Y)^2/Nを計算します。	可
stddev (numeric_type) → 引数がdouble precisionあるいはrealに対してはdouble precision、それ以外はnumeric これはstddev_sampの歴史的な別名です。	可
stddev_pop (numeric_type) → 引数がdouble precisionあるいはrealに対してはdouble precision、それ以外はnumeric 入力値の母標準偏差を計算します。	可
stddev_samp (numeric_type) → 引数がdouble precisionあるいはrealに対してはdouble precision、それ以外はnumeric 入力値の標本標準偏差を計算します。	可
variance (numeric_type) → 引数がdouble precisionあるいはrealに対してはdouble precision、それ以外はnumeric これはvar_sampの歴史的な別名です。	可
var_pop (numeric_type) → 引数がdouble precisionあるいはrealに対してはdouble precision、それ以外はnumeric 入力値の母分散(母標準偏差の二乗)を計算します。	可
var_samp (numeric_type) → 引数がdouble precisionあるいはrealに対してはdouble precision、それ以外はnumeric 入力値の標本分散(標本標準偏差の二乗)を計算します。	可

表 9.57に順序集合集約構文を使う集約関数を示します。これらの関数は「逆分散」関数として参照されることがあります。これらの集約入力はORDER BYで導入され、集約ではない*direct argument*を取ることもでき、一度だけ計算されます。fractionパラメータを取る関数では、その値は0と1の間でなければなりません。そうでなければエラーが生じます。ただしNULLのfraction値は単にNULLの結果をもたらします。

表9.57 順序集合集約関数

関数 説明	部分モード
mode () WITHIN GROUP (ORDER BY anyelement) → anyelement 入力の最頻値、集約引数の値のうち最も頻出のもの(複数の同じ度数の結果があれば、任意に選んだ最初のもの)を計算します。集約引数はソート可能な型でなければなりません。	非
percentile_cont (fraction double precision) WITHIN GROUP (ORDER BY double precision) → double precision percentile_cont (fraction double precision) WITHIN GROUP (ORDER BY interval) → interval	非

関数 説明	部分モード
連続百分位数、引数の値の順序付け集合中で指定されたfractionに対応する値を計算します。これは必要なら隣り合う入力項目を補間します。	
percentile_cont (fractions double precision[]) WITHIN GROUP (ORDER BY double precision) → double precision[] percentile_cont (fractions double precision[]) WITHIN GROUP (ORDER BY interval) → interval[] 複数の連続百分位数を計算します。結果はfractionsパラメータと同じ次元数の配列です。各非NULL要素は(必要なら隣り合う入力項目を補間して)その百分位数に対応する値で置き換えられます。	非
percentile_disc (fraction double precision) WITHIN GROUP (ORDER BY anyelement) → anyelement 離散百分位数を計算します。集約引数の順序付け集合中で、その位置が指定したfractionと等しいか越えた最初の値です。集約引数はソート可能な型でなければなりません。	非
percentile_disc (fractions double precision[]) WITHIN GROUP (ORDER BY anyelement) → anyarray 複数の離散百分位数を計算します 結果はfractionsパラメータと同じ次元数の配列です。各非NULL要素はその百分位数に対応する値で置き換えられます。集約引数はソート可能な型でなければなりません。	非

表 9.58に列挙されている「仮想集合」集約は、それぞれ9.22で定義されている同じ名前のウィンドウ関数と関連します。どの場合も、集約結果は、argsから構築される「仮想的な」行に対して、関連するウィンドウ関数が返す値で、そのような行がsorted_argsから計算されるソートされた行のグループに追加される場合を想定します。これらの関数に対してargsで与えられる直接引数のリストは、sorted_argsで与えられる集約された引数の数と型に一致しなければなりません。ほとんどの組み込み集約とは異なり、この集約はSTRICTではありません、すなわち、NULLを含む入力行を落としません。NULL値はORDER BY節で指定されるルールに従って並べられます。

表9.58 仮想集合集約関数

関数 説明	部分モード
rank (args) WITHIN GROUP (ORDER BY sorted_args) → bigint 重複する行のギャップを含む仮想の行の順位を計算します。すなわち、ピアグループの先頭の行の番号です。	非
dense_rank (args) WITHIN GROUP (ORDER BY sorted_args) → bigint 重複する行のギャップなしの仮想の行の順位を計算します。この関数は実効的にピアグループを数えます。	非
percent_rank (args) WITHIN GROUP (ORDER BY sorted_args) → double precision 仮想行の相対的な順位を計算します。すなわち、(rank - 1) / (total rows - 1)です。ですから値の範囲は境界を含んで0から1までです。	非
cume_dist (args) WITHIN GROUP (ORDER BY sorted_args) → double precision 現在行の相対順位を計算します。すなわち、(仮想行より先行する、あるいはピアの行数) / (合計行数)です。ですから範囲は1/Nから1です。	非

表9.59 グループ化演算

関数 説明
GROUPING (group_by_expression(s)) → integer

関数	説明
	どのGROUP BY式が現在のグループ化セットに含まれないかを示す整数のビットマスクを返します。最も右側の引数が最下位ビットになるようにビットが割り当てられます。各ビットは対応する式が結果の行を生成するグループ化セットのグループ化条件に含まれていれば0、そうでなければ1です。

表 9.59で示すグループ化演算はグループ化セット(7.2.4参照)と一緒に使われ、結果の行を区別するものです。GROUPING関数の引数は実際には評価されませんが、関連する問い合わせのGROUP BY句にある式と正確に一致する必要があります。例えば以下のようになります。

```
=> SELECT * FROM items_sold;
 make  | model | sales
-------+-------+-------
 Foo   | GT    |  10
 Foo   | Tour  |  20
 Bar   | City  |  15
 Bar   | Sport |  5
(4 rows)

=> SELECT make, model, GROUPING(make,model), sum(sales) FROM items_sold GROUP BY
 ROLLUP(make,model);
 make  | model | grouping | sum
-------+-------+----------+-----
 Foo   | GT    |        0 | 10
 Foo   | Tour  |        0 | 20
 Bar   | City  |        0 | 15
 Bar   | Sport |        0 | 5
 Foo   |       |        1 | 30
 Bar   |       |        1 | 20
       |       |        3 | 50
(7 rows)
```

ここで、最初の4行のグループ化値0はこれらがグループ化列に対して正常にグループ化されたことを示します。値1はmodelが最後とその一つ前の行ではグループ化されなかったことを、値3はmakeもmodelも最後の行でグループ化されなかったことを意味します(ですから最後の行はすべての入力行に対する集約になっています)。

9.22. ウィンドウ関数

ウィンドウ関数は現在の問い合わせ行に関連した行集合に渡っての計算処理機能を提供します。この機能の手引きは3.5を、文法の詳細は4.2.8を参照してください。

組み込みウィンドウ関数は表 9.60に一覧されています。これらの関数は必ずウィンドウ関数構文で呼び出されなければなりません。つまり、OVER句が必要です。

これらの関数に加え、すべての組み込み、またはユーザ定義の汎用集約関数または統計集約関数もウィンドウ関数として使用できます(ただし順序集合や仮想集合集約はそうではありません)。組み込み集約関数一覧は9.21を参照してください。集約関数は、呼び出しの後にOVER句が続いた場合のみウィンドウ関数として動作します。それ以外の場合は、非ウィンドウの集約関数として動作し、集合全体に対して1行だけを返します。

表9.60 汎用ウィンドウ関数

関数 説明
row_number () → bigint 1から数える現在行のパーティション内での行番号を返します。
rank () → bigint ギャップを含んだ現在行の順位を返します。すなわちピアグループの先頭行のrow_numberと同じになります。
dense_rank () → bigint ギャップを含まない現在行の順位。この関数は実質的にピアのグループ数を数えます。
percent_rank () → double precision 現在行の相対順位、すなわち (rank - 1) / (パーティションの総行数 - 1)を返します。したがってこの値は境界を含み0から1となります。
cume_dist () → double precision 現在行の相対順位、すなわち (現在行より先行する行およびピアの行数) / (パーティションの総行数)を返します。したがってこの値は1/Nから1となります。
ntile (num_buckets integer) → integer できるだけ等価にパーティションを分割した、1から引数値までの整数を返します。
lag (value anyelement [, offset integer [, default anyelement]]) → anyelement パーティション内の現在行よりoffset行だけ前の行で評価されたvalueを返します。該当する行がない場合、その代わりとしてdefault(valueと同じ型でなければなりません)を返します。offsetとdefaultは共に現在行について評価されます。省略された場合、offsetはデフォルトで1となり、defaultはNULLになります。
lead (value anyelement [, offset integer [, default anyelement]]) → anyelement パーティション内の現在行よりoffset行だけ後の行で評価されたvalueを返します。該当する行がない場合、その代わりとしてdefault(valueと同じ型でなければなりません)を返します。offsetとdefaultは共に現在行について評価されます。省略された場合、offsetはデフォルトで1となり、defaultはNULLになります。
first_value (value anyelement) → anyelement ウィンドウフレームの最初の行である行で評価されたvalueを返します。
last_value (value anyelement) → anyelement ウィンドウフレームの最後の行である行で評価されたvalueを返します。
nth_value (value anyelement, n integer) → anyelement ウィンドウフレームの(1から数えて)n番目の行である行で評価されたvalueを返します。行が存在しない場合はNULLを返します。

表 9.60に列挙された関数はすべて、対応するウィンドウ定義のORDER BY句で指定されるソート順に依存します。ORDER BYの列だけを考慮した場合に重複する行はピアと呼ばれます。4つの順位付け関数(cume_distを含む)は、すべてのピア行に対して同じ答えになるように定義されています。

first_value、last_value、nth_value関数は「ウィンドウフレーム」内の行のみを考慮することに注意してください。デフォルトで、ウィンドウフレームにはパーティションの先頭から現在の行の最終ピアまでの行が含まれます。これはlast_value、または時々nth_valueでは有用ではない結果を得ることになりがちです。OVER句に適切なフレーム指定(RANGE、GROUP、もしくはROWS)を加えることで、フレームを再定義することができます。フレーム指定についての詳細は4.2.8を参照してください。

集約関数をウィンドウ関数として使用する場合、現在の行のウィンドウフレーム内の行に渡って集約処理を行います。ORDER BYおよび、デフォルトのウィンドウフレーム定義を使用した集約では、「中間和」のような動作を行います。これが望まれる場合もあれば、望まれない場合もあります。パーティション全体に渡る集約処理を行うためには、ORDER BYを省略するかROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWINGを使用してください。他のフレーム指定を使用することで様々な結果を得ることができます。

注記

SQL標準は、lead、lag、first_value、last_value、およびnth_valueに対しRESPECT NULLS、またはIGNORE NULLSオプションを定義します。これはPostgreSQLに実装されていません。動作は常に標準のデフォルトと同一です。つまり、RESPECT NULLSです。同様にして、標準のnth_valueに対するFROM FIRST、またはFROM LASTオプションは実装されていません。デフォルトのFROM FIRST動作のみに対応しています。(ORDER BY順序付けを逆に行うことで、FROM LASTの結果を得ることができます。)

9.23. 副問い合わせ式

本節ではPostgreSQLで使用できるSQL準拠の副問い合わせについて説明します。本節で記載した全ての式は結果として論理値(真/偽)を返します。

9.23.1. EXISTS

```
EXISTS (subquery)
```

EXISTSの引数は、任意のSELECT文、つまり副問い合わせです。副問い合わせはそれが何らかの行を返すか否かの決定のために評価されます。もし1つでも行を返すのであれば、EXISTSの結果は「true(真)」となり、副問い合わせが行を返さない場合、EXISTSの結果は「false(偽)」となります。

副問い合わせは、取り囲んでいる問い合わせから変数を参照することができ、その値は副問い合わせの評価時には定数として扱われます。

この副問い合わせは通常、最後まで実行されず、少なくとも1つの行が返されたかどうかを判定し得るに足りる時点まで実行されます。(シーケンス関数を呼び出すような)副作用のある副問い合わせを記述することは配慮不足です。副作用が生じるかどうかは予想できません。

結果は何らかの行が返されるのかのみに依存し、それらの行の内容には依存しないことから、副問い合わせの出力リストは通常重要ではありません。よく使われるコーディング規約は、全てのEXISTSテストをEXISTS(SELECT 1 WHERE ...)といった形式で記述することです。とは言っても、INTERSECTを使う副問い合わせのようにこの規則には例外があります。

以下の簡単な例はcol2上の内部結合に似ていますが、たとえtab2の行といくつか一致したとしてもtab1のそれぞれの行に対して最大限1つの出力行を生成します。

```
SELECT col1
FROM tab1
WHERE EXISTS (SELECT 1 FROM tab2 WHERE col2 = tab1.col2);
```

9.23.2. IN

```
expression IN (subquery)
```

右辺は括弧で括られた副問い合わせで、正確に1列を返すものでなければなりません。左辺式は評価され、副問い合わせの結果行と比較されます。副問い合わせの行のどれかと等しい場合、INの結果は「true(真)」です。(副問い合わせが行を返さない場合を含め)等しい行が見つからない場合、結果は「false(偽)」です。

左辺の式がNULLを生じる場合、または右側の値に等しいものがなくて少なくとも1つの右辺の行がNULLを持つ場合、IN構文の結果は偽ではなくNULLとなることに注意してください。これは、NULL値の論理的な組み合わせに対するSQLの標準規則に従うものです。

EXISTSと同様、副問い合わせが完全に評価されることを前提としてはなりません。

```
row_constructor IN (subquery)
```

INのこの形式の左辺は、4.2.13で説明する、行のコンストラクタです。右辺は括弧で括られた副問い合わせで、左辺の行にある式の数と正確に同じ数の列を返さなければなりません。左辺の式は副問い合わせの結果のそれぞれの行に対し、行に関して評価、比較が行われます。副問い合わせの行に等しいものが見つかった場合、IN の結果は「true(真)」となります。(副問い合わせが行を返さない場合を含め)等しい行が見つからない場合、結果は「false(偽)」です。

通常通り、行にあるNULL値はSQLの論理式の標準規則で結合されます。2つの行は対応する全ての構成要素が非NULLかつ等しい場合に等しいとみなされます。1つでも対応する構成要素が非NULLかつ等しくないものがあれば、2つの行は等しくないとみなされます。それ以外の場合、その行の比較結果は不明(NULL)です。行毎の結果すべてが不等もしくはNULLの場合、少なくとも1つのNULLがあると、INの結果はNULLとなります。

9.23.3. NOT IN

```
expression NOT IN (subquery)
```

右辺は括弧で括られた副問い合わせで、正確に1つの列を返さなければなりません。左辺の式は副問い合わせ結果の行それぞれに対して評価、比較されます。等しくない副問い合わせの行だけがある(副問い合わせが行を返さない場合を含む)と、NOT INの結果は「true(真)」です。等しい行が1つでもあれば、結果は「false(偽)」です。

左辺の式でNULLが生じる場合、または右辺の値に等しいものがなく、少なくとも1つの右辺の式がNULLを生み出す場合、NOT IN構文の結果は真ではなくNULLとなることに注意してください。これは、NULL値の論理的な組み合わせに対するSQLの標準規則に従うものです。

EXISTSと同様、副問い合わせが完全に評価されることを前提としてはなりません。

```
row_constructor NOT IN (subquery)
```

NOT INのこの形式の左辺は、4.2.13で説明する行コンストラクタです。右辺は括弧で括られた副問い合わせで、左辺の行にある式の数と正確に同じ数の列を返さなければなりません。左辺の式は副問い合わせの結果のそれぞれの行に対し、評価、比較が行われます。副問い合わせの行に不等のもののみが見つかった場合（副問い合わせが行を返さない場合を含む）、NOT INの結果は「true（真）」となります。等しい行が1つでも見つかった場合、結果は「false（偽）」です。

通常通り、行にあるNULL値はSQLの論理式の標準規則で結合されます。2つの行は対応する全ての構成要素が非NULLかつ等しい場合に等しいとみなされます。1つでも構成要素が非NULLかつ等しくない場合、2つの行は等しくないとみなされます。それ以外の場合、その行の比較結果は不明（NULL）です。行毎の結果すべてが不等もしくはNULLの場合、少なくとも1つのNULLがあると、NOT INの結果はNULLとなります。

9.23.4. ANY/SOME

```
expression operator ANY (subquery)
expression operator SOME (subquery)
```

右辺は括弧で括られた副問い合わせで、正確に1つの列を返さなければなりません。左辺の式は副問い合わせの結果行それぞれに対して、指定されたoperatorを使用して評価、比較されます。なお、operatorは結果として論理値を生成する必要があります。真の結果が1つでもあると、ANYの結果は「true（真）」です。真の結果がない（副問い合わせが行を返さない場合を含む）と、結果は「false（偽）」です。

SOMEはANYの同義語です。INは= ANYと等価です。

成功がなく、右辺の行が演算子の結果として1つでもNULLを生成した場合、ANY構文の結果は偽ではなくNULLになることに注意してください。これは、NULL値の論理的な組み合わせに対するSQLの標準規則に従うものです。

EXISTSと同様、副問い合わせが完全に評価されると前提してはなりません。

```
row_constructor operator ANY (subquery)
row_constructor operator SOME (subquery)
```

ANYのこの形式の左辺は、4.2.13で説明されている行コンストラクタです。右辺は括弧で括られた副問い合わせで、左辺の行にある式の数と正確に同じ数の列を返さなければなりません。左辺の式は副問い合わせの結果のそれぞれの行に対し、与えられたoperatorを使用して行に関する評価、比較が行われます。比較の結果、副問い合わせの行のどれかに対して真となる場合、ANYの結果は「true（真）」です。比較の結果、副問い合わせの全ての行に対して偽となる場合（副問い合わせが行を返さないという場合も含む）、結果は「false（偽）」です。いかなる副問合せ行との比較の結果も偽を返さず、かつ、少なくとも1つの比較がNULLを返す場合、結果はNULLになります。

行コンストラクタ比較の意味についての詳細は9.24.5を参照して下さい。

9.23.5. ALL

```
expression operator ALL (subquery)
```

右辺は括弧で括られた副問い合わせで、正確に1つの列を返さなければなりません。左辺の式は副問い合わせの結果行それぞれに対して、指定されたoperatorを使用して評価、比較されます。なお、operatorは結果として論理値を生成する必要があります。全ての行が真になる場合（副問い合わせが行を返さない場合を含む）、ALLの結果は「true（真）」です。1つでも偽の結果があると、結果は「false（偽）」です。比較がどの行でも偽を返さず、かつ、少なくとも1つの行でNULLを返した場合、結果はNULLとなります。

NOT INは<> ALLと等価です。

EXISTSと同様、副問い合わせが完全に評価されることを前提としてはなりません。

```
row_constructor operator ALL (subquery)
```

ALLのこの形式の左辺は、4.2.13で説明する行コンストラクタです。右辺は括弧で括られた副問い合わせで、左辺の行にある式の数と正確に同じ数の列を返さなければなりません。左辺の式は副問い合わせの結果のそれぞれの行に対し、与えられたoperatorを使用して行に関する評価、比較が行われます。比較した結果、すべての副問い合わせ行に対して真を返す場合（副問い合わせが行を返さないという場合も含む）、ALLの結果は「true（真）」となります。比較した結果、いずれかの副問い合わせ行で偽を返す場合、この結果は「false（偽）」となります。比較結果がすべての副問い合わせ行に対して偽を返さず、少なくとも1行でNULLを返す場合、結果はNULLとなります。

行コンストラクタに関する比較の意味については9.24.5を参照してください。

9.23.6. 単独行に関する比較

```
row_constructor operator (subquery)
```

左辺は、4.2.13で説明されている行コンストラクタです。右辺は括弧で括られた副問い合わせで、左辺の行とまったく同じ数の列を返さなければなりません。さらに、副問い合わせは複数行を返すことはできません。（行をまったく返さない場合、結果はNULLとみなされます。）左辺は副問い合わせの結果の単一行に対し行全体で評価、比較が行われます。

行コンストラクタに関する比較の意味についての詳細は9.24.5を参照してください。

9.24. 行と配列の比較

本節では、値のグループ間で複数の比較を行う、さまざまな特殊化したコンストラクトについて説明します。この形式は構文的には、前節の副問い合わせ形式と関係しています。しかし、副問い合わせを含みません。配列副式を含む形式はPostgreSQLの拡張ですが、それ以外はSQL準拠です。本節で記載した全ての式形式は結果として論理値（真/偽）を返します。

9.24.1. IN

```
expression IN (value [, ...])
```

右辺は括弧で括られたスカラ式のリストです。左辺の式の結果が右辺の式のいずれかと等しい場合、結果は「true（真）」になります。これは以下の省略形です。

```
expression = value1
OR
expression = value2
OR
...
```

左辺の式がNULLを生じる場合、または右側の値に等しいものがなくて少なくとも1つの右辺の行がNULLを持つ場合、IN構文の結果は偽ではなくNULLとなることに注意してください。これは、NULL値の論理的な組み合わせに対するSQLの標準規則に従うものです。

9.24.2. NOT IN

```
expression NOT IN (value [, ...])
```

右辺は括弧で括られたスカラ式のリストです。左辺の式の結果が右辺の式の全てと等しくない場合、結果は「真」です。これは以下の省略形です。

```
expression <> value1
AND
expression <> value2
AND
...
```

左辺の式でNULLが生じる場合、または右辺の値に左辺の式と等しいものがなく、かつ少なくとも1つの右辺の式がNULLを生じる場合、NOT IN構文の結果は、一部の人が予想する真ではなく、NULLとなることに注意してください。これは、NULL値の論理的な組み合わせに対するSQLの標準規則に従うものです。

> **ヒント**
>
> 全ての場合において、x NOT IN yはNOT (x IN y)と等価です。しかし、INを使用するよりもNOT INを使用する方が初心者がNULL値による間違いをしやすくなります。可能な限り条件を肯定的に表現することが最善です。

9.24.3. ANY/SOME (配列)

```
expression operator ANY (array expression)
expression operator SOME (array expression)
```

右辺は括弧で括られた式で、配列値を返さなければなりません。左辺の式は配列要素それぞれに対して、指定されたoperatorを使用して評価、比較されます。なお、operatorは結果として論理値を生成する必要があります。真の結果が1つでもあると、ANYの結果は「true(真)」です。真の結果がない(配列の要素数がゼロである場合を含む)と、結果は「false(偽)」です。

配列式がNULL配列を生成する場合、ANYの結果はNULLになります。左辺式がNULLとなる場合、ANYの結果は通常NULLになります(STRICTでない比較演算子では異なる結果になるかもしれません)。また、右辺の配列にNULL要素が含まれ、かつ、比較の結果、真が得られなかった場合、ANYの結果は偽ではなくNULLに

なります（ここでも、STRICTな演算子の場合です）。これは、NULLに対する、SQLの論理値組み合わせに関する標準規則に従うものです。

SOMEはANYの同義語です。

9.24.4. ALL (配列)

```
expression operator ALL (array expression)
```

右辺は括弧で括られた式で、配列値を返さなければなりません。左辺の式は配列の要素それぞれに対して、指定されたoperatorを使用して評価、比較されます。なお、operatorは結果として論理値を生成する必要があります。全ての比較が真になる場合（配列の要素数がゼロである場合を含む）、ALLの結果は「true（真）」です。1つでも偽の結果があると、結果は「false（偽）」です。

配列式がNULL配列を生成する場合、ALLの結果はNULLになります。左辺式がNULLとなる場合、ALLの結果は通常NULLになります（厳格でない比較演算子では異なる結果になるかもしれません）。また、右辺の配列にNULL要素が含まれ、かつ、比較の結果、偽が得られなかった場合、ALLの結果は真ではなくNULLになります（ここでも、厳格な演算子の場合です）。これは、NULLに対する、SQLの論理値組み合わせに関する標準規則に従うものです。

9.24.5. 行コンストラクタの比較

```
row_constructor operator row_constructor
```

両辺とも4.2.13で説明する行コンストラクタです。この2つの行値は同じフィールド数でなければなりません。両辺はそれぞれ評価され、行として比較されます。行コンストラクタの比較は、operatorが=、<>、<、<=、>、>=の場合に認められます。各行の要素はデフォルトのB-tree演算子クラスを持つ型でなければなりません。そうでなければ、比較を試みるとエラーが発生します。

> **注記**
>
> 比較が先行する列で解決された場合、要素の数や型に関係するエラーは起きないこともあります。

=と<>の場合、他と動作が多少異なります。2つの行は対応する全ての構成要素が非NULLかつ等しい場合に等しいとみなされます。1つでも構成要素が非NULLかつ等しくない場合、2つの行は等しくないとみなされます。それ以外の場合、その行の比較結果は不明（NULL）です。

<、<=、>、>=の場合、行の要素は左から右に比較されます。そして、不等またはNULLの組み合わせが見つかったところで停止します。要素の組み合わせのどちらかがNULLであった場合、行比較の結果は不明（NULL）です。さもなくば、要素の組み合わせの比較により結果が決まります。例えば、ROW(1,2,NULL) < ROW(1,3,0)は、3番目の要素の組み合わせまで進まないため、NULLではなく真を返します。

> **注記**
>
> PostgreSQLの8.2より前では、<、<=、> 、>=の場合SQL仕様に従っていませんでした。ROW(a,b) < ROW(c,d)などの比較は正しくはa < c OR (a = c AND b < d)ですが、a < c AND b < dとして実装されていました。

```
row_constructor IS DISTINCT FROM row_constructor
```

このコンストラクトは<>行比較と類似していますが、NULL入力に対してNULLを生成しない点が異なります。その代わりに、全てのNULL値は非NULL値と等しくない（DISTINCT FROM）ものとみなされ、また、2つのNULLは等しい（NOT DISTINCT）ものとみなされます。したがって、結果は真か偽のいずれかで、NULLにはなりません。

```
row_constructor IS NOT DISTINCT FROM row_constructor
```

このコンストラクトは=行比較と類似していますが、NULL入力に対してNULLを生成しません。代わりに、NULL値を、すべての非NULLの値に対して不等（DISTINCT FROM）とみなし、2つのNULLを等しいもの（NOT DISTINCT）とみなします。したがって、結果は常に真か偽となり、NULLになることはありません

9.24.6. 複合型の比較

```
record operator record
```

SQL仕様では、結果が2つのNULL値、またはNULLと非NULLの比較に依存するのであれば、行の観点からの比較はNULLを返すことを要求されています。PostgreSQLは、(9.24.5にあるように)2つの行コンストラクタの出力の比較を行う時、または副問い合わせの出力に対し(9.23にあるように)行コンストラクタの比較を行う時のみこれを実施します。2つの複合型の値が比較されるほかの状況では、2つのNULLフィールドの値は等しいと考えられ、NULLは非NULLより大きいとみなされます。複合型に対して、これは一貫した並び替えとインデックス付け動作担保のため必要です。

各辺が評価され、行単位で比較が行なわれます。複合型の比較はoperatorが=、<>、<、<=、>、>=またはそのいずれかと類似の意味を持つ場合に許されます。(正確には、演算子はB-tree演算子クラスのメンバである場合、またはB-tree演算子クラスの=メンバの否定子である場合に行比較演算子となり得ます。) 上記の演算子のデフォルトの動作は、行コンストラクタに対するIS [NOT] DISTINCT FROMと同じです（9.24.5参照）。

デフォルトのB-tree演算子クラスを持たない要素を含む行の一致をサポートするために、いくつかの演算子が複合型の比較のために定義されています。それは*=、*<>、*<、*<=、*>、*>=です。上記の演算子は2つの行の内部バイナリ表現を比較します。2つの行の等価演算子での比較が真であっても、2つの行はバイナリ表現が異なるかもしれません。上記の比較演算子での行の順序は決定論的ですが、それ以外は意味がありません。上記の演算子はマテリアライズドビューで内部的に使われ、レプリケーションやB-Treeの重複除去(63.4.2参照)のような他の特定の目的のためには有用かもしれませんが、問い合わせを書くのに一般的に有用であるようには意図していません。

9.25. 集合を返す関数

本節では、場合により複数行を返す関数について説明します。このクラスで最も広く用いられている関数は、表 9.61、および表 9.62にて詳細が触れられている、連続値生成関数です。他方、より特化された集合を返

す関数の記述がこのマニュアルの他の場所にあります。集合を返す関数を複数組み合わせる方法については7.2.1.4を参照してください。

表9.61 連続値生成関数

関数 説明
generate_series (start integer, stop integer [, step integer]) → setof integer generate_series (start bigint, stop bigint [, step bigint]) → setof bigint generate_series (start numeric, stop numeric [, step numeric]) → setof numeric startからstopまで、刻みstepで連続する値を生成します。stepのデフォルトは1です。
generate_series (start timestamp, stop timestamp, step interval) → setof timestamp generate_series (start timestamp with time zone, stop timestamp with time zone, step interval) → setof timestamp with time zone startからstopまで、刻みstepで連続する値を生成します。

stepが正の場合、startがstopよりも大きいと0行が返ります。反対に、stepが負の場合は、startがstopよりも小さいと0行が返ります。また、どれかの入力がNULLの場合も0行が返ります。stepが0の時はエラーになります。以下にいくつか例を示します。

```
SELECT * FROM generate_series(2,4);
 generate_series
-----------------
               2
               3
               4
(3 rows)

SELECT * FROM generate_series(5,1,-2);
 generate_series
-----------------
               5
               3
               1
(3 rows)

SELECT * FROM generate_series(4,3);
 generate_series
-----------------
(0 rows)

SELECT generate_series(1.1, 4, 1.3);
 generate_series
-----------------
             1.1
             2.4
             3.7
```

```
(3 rows)


-- この例は日付に整数を足し込む演算子に依存します。
SELECT current_date + s.a AS dates FROM generate_series(0,14,7) AS s(a);
   dates
------------
 2004-02-05
 2004-02-12
 2004-02-19
(3 rows)

SELECT * FROM generate_series('2008-03-01 00:00'::timestamp,
                              '2008-03-04 12:00', '10 hours');
   generate_series
---------------------
 2008-03-01 00:00:00
 2008-03-01 10:00:00
 2008-03-01 20:00:00
 2008-03-02 06:00:00
 2008-03-02 16:00:00
 2008-03-03 02:00:00
 2008-03-03 12:00:00
 2008-03-03 22:00:00
 2008-03-04 08:00:00
(9 rows)
```

表9.62 添え字生成関数

関数 説明
generate_subscripts (array anyarray, dim integer) → setof integer 指定した配列のdim次元で有効な添え字を構成する連番を生成します。
generate_subscripts (array anyarray, dim integer, reverse boolean) → setof integer 指定した配列dim次元で有効な添え字を構成する連番を生成します。reverseが真の場合、連番は逆順に返されます。

generate_subscriptsは、指定した配列の指定した次数で有効な添え字からなる集合を生成するために便利な関数です。要求された次数を持たない配列またはどれかの入力がNULLなら0行が返ります。いくつかの例を以下に示します。

```
-- basic usage:
SELECT generate_subscripts('{NULL,1,NULL,2}'::int[], 1) AS s;
 s
---
 1
 2
```

```
 3
 4
(4 rows)


-- 配列、
      添え字とその添え字が示す値を表示するには
-- 副問い合わせが必要です。
SELECT * FROM arrays;
         a
--------------------
 {-1,-2}
 {100,200,300}
(2 rows)

SELECT a AS array, s AS subscript, a[s] AS value
FROM (SELECT generate_subscripts(a, 1) AS s, a FROM arrays) foo;
     array     | subscript | value
---------------+-----------+-------
 {-1,-2}       |         1 |    -1
 {-1,-2}       |         2 |    -2
 {100,200,300} |         1 |   100
 {100,200,300} |         2 |   200
 {100,200,300} |         3 |   300
(5 rows)


-- 2次元配列の入れ子を解きます。
CREATE OR REPLACE FUNCTION unnest2(anyarray)
RETURNS SETOF anyelement AS $$
select $1[i][j]
   from generate_subscripts($1,1) g1(i),
        generate_subscripts($1,2) g2(j);
$$ LANGUAGE sql IMMUTABLE;
CREATE FUNCTION
SELECT * FROM unnest2(ARRAY[[1,2],[3,4]]);
 unnest2
---------
       1
       2
       3
       4
(4 rows)
```

FROM句の関数の後にWITH ORDINALITYが付いている場合、1から始まり関数の出力の行毎に1増えていくbigint列が関数の出力列に追加されます。これはunnest()のような集合を返す関数の場合に最も役に立ちます。

```
-- WITH ORDINALITYの付いた集合を返す関数
SELECT * FROM pg_ls_dir('.') WITH ORDINALITY AS t(ls,n);
       ls        | n
-----------------+----
 pg_serial       |  1
 pg_twophase     |  2
 postmaster.opts |  3
 pg_notify       |  4
 postgresql.conf |  5
 pg_tblspc       |  6
 logfile         |  7
 base            |  8
 postmaster.pid  |  9
 pg_ident.conf   | 10
 global          | 11
 pg_xact         | 12
 pg_snapshots    | 13
 pg_multixact    | 14
 PG_VERSION      | 15
 pg_wal          | 16
 pg_hba.conf     | 17
 pg_stat_tmp     | 18
 pg_subtrans     | 19
(19 rows)
```

9.26. システム情報関数と演算子

表 9.63に、セッションおよびシステムの情報を抽出する関数を示します。

本節で列挙されている関数のほかに、同様にシステム情報を提供する統計システムに関連した数多くの関数があります。27.2.2にさらに情報があります。

表9.63 セッション情報関数

関数 説明
current_catalog → name current_database () → name 現在のデータベースの名前を返します。(データベースはSQL標準では「カタログ」と呼ばれています。ですから標準での記述はcurrent_catalogとなります。)
current_query () → text

関数 説明
クライアントから送信された現在実行中問い合わせのテキスト（複数の文を含むことあります）を返します。
current_role → name current_userと同じです。
current_schema → name current_schema () → name サーチパスの先頭にあるスキーマの名前を返します。（サーチパスが空ならNULL値を返します。）これはターゲットスキーマを指定せずに作成されるすべてのテーブルあるいは名前付きのオブジェクトで使われるスキーマです。
current_schemas (include_implicit boolean) → name[] 現在有効な検索パス中にあるすべてのスキーマの名前を優先順に配列で返します。（現在のsearch_path設定にある項目で、存在する検索可能なスキーマに関連しないものは無視されます。）論理値引数が真ならpg_catalogのような暗黙的に検索されるシステムスキーマは結果に含まれます。
current_user → name 現在の実行コンテキストのユーザ名を返します。
inet_client_addr () → inet 現在のクライアントのIPアドレスを返します。UNIXドメインソケット経由の接続ならNULLが返ります。
inet_client_port () → integer 現在のクライアントのIPポート番号を返します。UNIXドメインソケット経由の接続ならNULLが返ります。
inet_server_addr () → inet サーバが受け付けている現在の接続のIPアドレスを返します。UNIXドメインソケット経由の接続ならNULLが返ります。
inet_server_port () → integer サーバが受け付けている現在の接続のIPポート番号を返します。UNIXドメインソケット経由の接続ならNULLが返ります。
pg_backend_pid () → integer 現在のセッションに結びついているサーバプロセスのプロセスIDを返します。
pg_blocking_pids (integer) → integer[] 指定したサーバプロセスIDによるロック取得をブロックしているプロセスIDを配列で返します。そのようなサーバプロセスが存在しないかあるいはブロックしていない場合は空の配列が返ります。 あるサーバプロセスが別のサーバプロセスをブロックするのは、ブロックされるプロセスのロック要求と競合するロックを保持している場合（ハードブロック）、あるいは、ブロックされるプロセスのロック要求と競合するロックを待っていて、かつロック待ちキュー内でより前方にいる場合（ソフトブロック）です。パラレルクエリを使っている場合、実際のロックを子ワーカプロセスが保持または待っている場合であっても、この結果には必ずクライアントから可視のプロセスID（つまり、pg_backend_pidの結果）が示されます。そのような仕様なので、この結果には重複するPIDが含まれることもあります。また、プリペアドのトランザクションが競合するロックを保持している場合、この関数の結果ではプロセスIDがゼロとして示されることにも注意して下さい。頻繁にこの関数を呼び出すとデータベースの性能に影響があるかも知れません。ロックマネージャの共有状態への短期的な排他ロックの取得が必要だからです。
pg_conf_load_time () → timestamp with time zone サーバ設定ファイルが最後に読み込まれた時の時刻を返します。現在のセッションがそのときに活動中だった場合、これはそのセッション自身が設定ファイルを再読み込みした時刻になります。（ですからその結果はセッションによって少し異なるかもしれません。）それ以外の場合は、postmasterプロセスが設定ファイルを再読み込みした時刻になります。
pg_current_logfile ([text]) → text

関数 説明
ログ収集機構が現在使用しているログファイルのパス名を返します。パスにはlog_directoryディレクトリとログファイルの名前が含まれます。ログ収集が無効ならば戻り値はNULLになります。複数のログファイルがそれぞれ異なる形式で存在する場合、引数なしのpg_current_logfileは、順序リスト(stderr、csvlog)の最初に出てくる形式のファイルのパスを返します。これらの形式のログファイルがないときはNULLが返されます。特定のファイル形式を要求するには、オプションパラメータの値としてcsvlogまたはstderrをオプション引数の値として渡してください。指定のログ形式がlog_destinationで設定されていない場合は、戻り値がNULLとなります。結果はcurrent_logfilesファイルの内容を反映します。
pg_my_temp_schema () → oid 現在のセッションの一時スキーマのOIDを返します。(一時テーブルをまだ1つも作成しておらず)存在しなければゼロを返します。
pg_is_other_temp_schema (oid) → boolean 指定したOIDが他のセッションの一時スキーマのOIDであれば、真を返します。(例えば、他のセッションの一時テーブルをカタログ表示から除外したい場合などで有用です。)
pg_jit_available () → boolean JITコンパイラ拡張が利用可能で(第31章参照)、jit設定パラメータがonなら真を返します。
pg_listening_channels () → setof text 今のセッションにおいて現在待ち受け中の非同期通知チャンネル名の集合を返します。
pg_notification_queue_usage () → double precision 非同期通知キューの最大サイズのうち、処理待ちの通知によって占められている現在の割合(0から1まで)を返します。詳細はLISTENとNOTIFYをご覧ください。
pg_postmaster_start_time () → timestamp with time zone サーバの起動時刻を返します。
pg_safe_snapshot_blocking_pids (integer) → integer[] 指定のサーバプロセスIDによる安全なスナップショットの取得をブロックしているセッションのサーバプロセスIDの配列を返します。そのようなサーバプロセスがないか、ブロックされていない場合は空の配列が返ります。 SERIALIZABLEトランザクションを実行しているセッションは、SERIALIZABLE READ ONLY DEFERRABLEトランザクションが述語ロックの取得をすべて回避しても安全であると決定するまで、後者がスナップショットを取得するのをブロックします。シリアライザブルトランザクションおよび遅延可能トランザクションについてのさらなる情報については13.2.3を参照してください。この関数を頻繁に呼び出すと、短時間に述語ロックマネージャの共有状態にアクセスする必要があるため、データベースのパフォーマンスに若干の影響が出るかもしれません。
pg_trigger_depth () → integer PostgreSQLのトリガの現在の入れ子の深さを返します。(直接的であれ間接的であれ、トリガ内部から呼ばれていなければ0を返します)。
session_user → name セッションのユーザ名を返します。
user → name current_userと等価です。
version () → text PostgreSQLサーバのバージョンを説明する文字列を返します。情報はserver_versionからも得られます。機械読み取り可能なバージョンはserver_version_numを使ってください。ソフトウェア開発者は文字列バージョンを解析するのではなく、server_version_num (8.2以降で利用可能)かPQserverVersionを使うべきです。

> **注記**
>
> current_catalog、current_role、current_schema、current_user、session_userおよびuserはSQLにおいて特殊な構文上の地位を持っており、最後に括弧を付けずに呼び出さなければなりません。PostgreSQLではcurrent_schemaの場合括弧を使用することができますが、他は使えません。

session_userは、通常、現在のデータベース接続を開始したユーザです。しかし、スーパーユーザはこの設定をSET SESSION AUTHORIZATIONを使用して変更することができます。current_userは、権限の検査に適用されるユーザ識別子です。通常はセッションユーザと同じですが、SET ROLEを使用して変更可能です。SECURITY DEFINER属性を持つ関数の実行中にも変わります。Unix用語で言うと、セッションユーザは「実ユーザ」で、現在のユーザは「実効ユーザ」です。current_roleとuserはcurrent_userの同義語です。(標準SQLではcurrent_roleとcurrent_userを区別していますが、PostgreSQLではユーザとロールを1種類のエンティティに統合しているため、両者に区別はありません。)

表 9.64に列挙した関数を使用して、ユーザはオブジェクトのアクセス権限をプログラムから問い合わせることができます。権限についての詳細は、5.7を参照してください。これらの関数では権限を検査されるユーザは名前かOID(pg_authid.oid)で指定できます。名前がpublicとして与えられるとPUBLIC仮想ロールの権限が検査されます。また、user引数を完全に省略できます。この場合はcurrent_userを指定したと見なされます。検査されるオブジェクトも名前かOIDで指定できます。名前で指定する時は関連するスキーマ名を含んでも構いません。対象となるアクセス権限はテキスト文字列で指定します。その文字列はオブジェクト型(たとえばSELECT)へと評価されなければならない適切なアクセスキーワードで指定します。その権限が許可オプションで保持されるかどうかをテストするためにオプションでWITH GRANT OPTIONを権限型に追加できます。また、複数の権限型をカンマで区切って列挙できます。この場合はどれかの権限が保持されていれば結果は真となります。(権限文字列は大文字小文字の区別がなく、追加の空白を権限文字列の間に入れることができますが、権限名の中に入れることはできません。)

```
SELECT has_table_privilege('myschema.mytable', 'select');
SELECT has_table_privilege('joe', 'mytable', 'INSERT, SELECT WITH GRANT OPTION');
```

表9.64 アクセス権限照会関数

関数 説明
has_any_column_privilege ([user name or oid,] table text or oid, privilege text) → boolean ユーザがテーブルのどれか1つの列に対して権限を所有しているか？テーブル全体あるいは列レベルの権限が少なくとも1つの列に与えられていれば成功します。可能な権限型はSELECT、INSERT、UPDATE、REFERENCESです。
has_column_privilege ([user name or oid,] table text or oid, column text or smallint, privilege text) → boolean ユーザがテーブルの指定された(1つの)列に対して権限を所有しているか？テーブル全体あるいはその列に対して列レベルの権限が与えられていれば成功します。名前かアトリビュート番号(pg_attribute.attnum)で列を指定できます。可能な権限型はSELECT、INSERT、UPDATE、REFERENCESです。
has_database_privilege ([user name or oid,] database text or oid, privilege text) → boolean ユーザはデータベースに対する権限があるか？可能な権限型はCREATE、CONNECT、TEMPORARY、TEMP (TEMPORARYと同じです)です。
has_foreign_data_wrapper_privilege ([user name or oid,] fdw text or oid, privilege text) → boolean 現在のユーザは外部データラッパに対する権限があるか？可能な権限型はUSAGEだけです。

関数 説明
has_function_privilege ([user name or oid,] function text or oid, privilege text) → boolean ユーザは関数に対する権限があるか？可能な権限型はEXECUTEだけです。 関数をOIDではなく名前で指定する場合、regprocedureデータ型（8.19を参照）と同じ入力が可能です。例を示します。 `SELECT has_function_privilege('joeuser', 'myfunc(int, text)', 'execute');`
has_language_privilege ([user name or oid,] language text or oid, privilege text) → boolean ユーザは言語に対する権限があるか？可能な権限型はUSAGEだけです。
has_schema_privilege ([user name or oid,] schema text or oid, privilege text) → boolean ユーザはスキーマに対する権限があるか？可能な権限型はCREATEとUSAGEです。
has_sequence_privilege ([user name or oid,] sequence text or oid, privilege text) → boolean ユーザはシーケンスに対する権限があるか？可能な権限型はUSAGE、SELECT、UPDATEです。
has_server_privilege ([user name or oid,] server text or oid, privilege text) → boolean ユーザは外部サーバに対する権限があるか？可能な権限型はUSAGEだけです。
has_table_privilege ([user name or oid,] table text or oid, privilege text) → boolean ユーザはテーブルに対する権限があるか？可能な権限型はSELECT、INSERT、UPDATE、DELETE、TRUNCATE、REFERENCES、TRIGGERです。
has_tablespace_privilege ([user name or oid,] tablespace text or oid, privilege text) → boolean ユーザはテーブル空間に対する権限があるか？可能な権限型はCREATEです。
has_type_privilege ([user name or oid,] type text or oid, privilege text) → boolean ユーザにデータ型に対する権限があるか？可能な権限型はUSAGEだけです。OIDではなく名前で型を指定する際は、可能な入力はregtypeデータ型に対するのと同じものです（8.19参照）。
pg_has_role ([user name or oid,] role text or oid, privilege text) → boolean ユーザにロールに対する権限があるか？可能な権限型はMEMBERとUSAGEです。MEMBERは直接あるいは間接的にそのロールのメンバであることを示します。（すなわち、SET ROLEを実行する権限です）一方、USAGEは、そのロールの権限がSET ROLEを実行することなく、直ちに利用可能であることを示します。この関数はuserをpublicに設定する特別なケースを許可しません。PUBLIC仮想ロールは実在するロールのメンバには決してなれないからです。
row_security_active (table text or oid) → boolean 現在のユーザと環境のコンテキストにおいて、指定のテーブルに対して行単位セキュリティは有効か？

アクセス権限のカタログ表現であるaclitem型で利用可能な演算子を表 9.65に示します。アクセス権限値を解釈する方法に関する情報は5.7をご覧ください。

表9.65 aclitem演算子

演算子 説明 例
aclitem = aclitem → boolean aclitemは等しいか？（aclitem型には通常の比較演算子の組がありません。等値だけです。同じようにaclitemの配列は等値比較だけが可能です。） 'calvin=r*w/hobbes'::aclitem = 'calvin=r*w*/hobbes'::aclitem → f

演算子 説明 例
aclitem[] @> aclitem → boolean 配列は指定した権限を含んでいるか？（これはaclitemを与えられる側と与える側にマッチする配列のエントリを含んでいて、少なくとも指定した権限の集合を持つ場合に真となります。） '{calvin=r*w/hobbes,hobbes=r*w*/postgres}'::aclitem[] @> 'calvin=r*/hobbes'::aclitem → t
aclitem[] ~ aclitem → boolean これは@>の廃止予定の別名です。 '{calvin=r*w/hobbes,hobbes=r*w*/postgres}'::aclitem[] ~ 'calvin=r*/hobbes'::aclitem → t

表 9.66にaclitem型を管理する追加の関数を示します。

表9.66 aclitem関数

関数 説明
acldefault (type "char", ownerId oid) → aclitem[] ownerIdのOIDを持つロールに所属するtype型のオブジェクトのデフォルト権限を持つaclitem配列を作成します。これはオブジェクトのACL権限がnullであるときに想定されるアクセス権限を示します。（デフォルトアクセス権限については5.7で述べています。）typeパラメータは以下のどれかでなければなりません。'c'でCOLUMN、'r'でTABLEおよびテーブルに見えるオブジェクト、's'でSEQUENCE、'd'でDATABASE、'f'でFUNCTIONあるいはPROCEDURE、'l'でLANGUAGE、'L'でLARGE OBJECT、'n'でSCHEMA、't'でTABLESPACE、'F'でFOREIGN DATA WRAPPER、'S'でFOREIGN SERVER、'T'でTYPEあるいはDOMAINを表します。
aclexplode (aclitem[]) → setof record (grantor oid, grantee oid, privilege_type text, is_grantable boolean) 行の集合としてaclitem配列を返します。アクセス権を与えられる側が仮想ロールPUBLICなら、granteeはゼロで表現されます。各々の与えられた権限はSELECT、INSERTなどで表現されます。各々の権限は別々の行に分解され、privilege_type列には一つのキーワードだけが現れることに注意してください。
makeaclitem (grantee oid, grantor oid, privileges text, is_grantable boolean) → aclitem 与えられた属性でaclitemを作成します。

表 9.67に、特定のオブジェクトが、現行スキーマの検索パスにおいて可視かどうかを判別する関数を示します。例えば、あるテーブルを含むスキーマが検索パス内に存在し、検索パス内の前方に同じ名前のテーブルがない場合、そのテーブルは可視であると言います。つまり、これは、テーブルが明示的なスキーマ修飾なしで名前によって参照可能であるということです。ですから全ての可視テーブルの名前を列挙するには以下のようにします。

```
SELECT relname FROM pg_class WHERE pg_table_is_visible(oid);
```

関数および演算子では、パスの前方に同じ名前かつ同じ引数のデータ型を持つオブジェクトが存在しなければ、検索パス内のオブジェクトは可視と言えます。演算子クラスと（演算子）族では、名前と関連するインデックスアクセスメソッドが考慮されます。

表9.67 スキーマ可視性照会関数

関数 説明
pg_collation_is_visible (collation oid) → boolean 照合順序が検索パスにおいて可視か？
pg_conversion_is_visible (conversion oid) → boolean 変換が検索パスにおいて可視か？
pg_function_is_visible (function oid) → boolean 関数が検索パスにおいて可視か？（これは手続きと集約にも使えます。）
pg_opclass_is_visible (opclass oid) → boolean 演算子クラスが検索パスにおいて可視か？
pg_operator_is_visible (operator oid) → boolean 演算子が検索パスにおいて可視か？
pg_opfamily_is_visible (opclass oid) → boolean 演算子族が可視か？
pg_statistics_obj_is_visible (stat oid) → boolean 統計情報オブジェクトが検索パスにおいて可視か？
pg_table_is_visible (table oid) → boolean テーブルが検索パスにおいて可視か？（これはビュー、マテリアライズドビュー、インデックス、シーケンス、外部テーブルを含むべての形式のリレーションで使用できます。）
pg_ts_config_is_visible (config oid) → boolean テキスト検索設定が検索パスにおいて可視か？
pg_ts_dict_is_visible (dict oid) → boolean テキスト検索辞書が検索パスにおいて可視か？
pg_ts_parser_is_visible (parser oid) → boolean テキスト検索パーサが検索パスにおいて可視か？
pg_ts_template_is_visible (template oid) → boolean テキスト検索テンプレートが検索パスにおいて可視か？
pg_type_is_visible (type oid) → boolean 型（またはドメイン）が検索パスにおいて可視か？

これらの関数は全て、検査するオブジェクトを識別するために、オブジェクトのOIDを必要とします。オブジェクトを名前でテストする場合、OID別名型（regclass、regtype、regprocedure、regoperator、regconfig、またはregdictionary）を使用すると便利です。例えば、以下のようにします。

```
SELECT pg_type_is_visible('myschema.widget'::regtype);
```

ただし、このようなやり方でスキーマ修飾されていない型名をテストしても、あまり意味がないことに注意してください。名前が認識されれば、それは必ず可視ということになります。

表 9.68に、システムカタログから情報を抽出する関数を列挙します。

表9.68 システムカタログ情報関数

関数 説明
format_type (type oid, typemod integer) → text 型OIDと型修飾子で決まるデータ型のSQL名を返します。型修飾子が不明な場合はNULLを型修飾子に渡してください。
pg_get_constraintdef (constraint oid [, pretty boolean]) → text 制約を作成したコマンドを再構築します。(これは逆コンパイルで構成したもので、元のコマンドのテキストではありません。)
pg_get_expr (expr pg_node_tree, relation oid [, pretty boolean]) → text 列のデフォルト値のような、システムカタログに格納された内部表現式を逆コンパイルします。式に変数が含まれている場合は2番目の引数として参照されているリレーションのOIDを指定してください。変数が含まれていない場合は、ゼロを渡しておけば十分です。
pg_get_functiondef (func oid) → text 関数あるいは手続きの作成コマンドを再構築します。(これは逆コンパイルによる再構築で、元のコマンドのテキストではありません。) 結果は完全なCREATE OR REPLACE FUNCTIONあるいはCREATE OR REPLACE PROCEDURE文です。
pg_get_function_arguments (func oid) → text CREATE FUNCTION中に現れる形で関数あるいは手続きの引数のリストを再構築します。(デフォルト値を含みます。)
pg_get_function_identity_arguments (func oid) → text ALTER FUNCTIONのようなコマンド中に現れる形で関数あるいは手続きの引数のリストを再構築します。この形式ではデフォルト値は省略します。
pg_get_function_result (func oid) → text CREATE FUNCTION中に現れる形で関数のRETURNS句を再構築します。
pg_get_indexdef (index oid [, column integer, pretty boolean]) → text インデックスを作成するコマンドを再構築します。(これは逆コンパイルによる再構築で、元のコマンドのテキストではありません。) columnが渡されていてゼロでないなら、その列の定義だけが再構築されます。
pg_get_keywords () → setof record (word text, catcode "char", catdesc text) サーバが認識するSQLキーワードを記述するレコードの集合を返します。word列にはキーワードが含まれます。catcode列にはカテゴリーコードが含まれます。Uは非予約キーワード、Cは列名になり得るキーワード、Tは型あるいは関数名になり得るキーワード、Rは完全な予約キーワードです。catdesc列にはカテゴリーを記述する、ローカライズ化されることもある文字列が含まれます。
pg_get_ruledef (rule oid [, pretty boolean]) → text ルールを作成するコマンドを再構築します。(これは逆コンパイルによる再構築で、元のコマンドのテキストではありません。)
pg_get_serial_sequence (table text, column text) → text 列に関連するシーケンスの名前を返します。列に関連するシーケンスが存在しなければ、NULLを返します。列がIDENTITY列の場合、関連するシーケンスはIDENTITY列に対して内部的に作成されたシーケンスとなります。SERIAL型(serial、smallserial、bigserial)の一つを使って作られた列については、そのSERIAL列の定義に対して作られたシーケンスとなります。後者の場合、この関連付けはALTER SEQUENCE OWNED BYで修正または削除することができます。(この関数はおそらくpg_get_owned_sequenceと呼ばれるべきだったのでしょうが、現在の名前はそれが主にserial列またはbigserial列と一緒に使われていたという事実によります。) 最初の入力パラメータはテーブル名で、スキーマを付けることもできます。2番目のパラメータは列名です。最初のパラメータは普通はスキーマとテーブルですので、二重引用符付の識別子としては解釈されません。つまり、デフォルトで小文字に変換されます。一方2番目のパラメータは単なる列名であり、二重引用符付として解釈され、その大文字小文字は保持されます。この関数は、シーケンス関数(9.17を参照)に渡すことができるよう適切な書式で値を返します。

関数 説明
典型的な使用法はIDENTITY列またはSERIAL列のシーケンスの現在値を読み取ることです。例を示します。 `SELECT currval(pg_get_serial_sequence('sometable', 'id'));`
pg_get_statisticsobjdef (statobj oid) → text 拡張統計情報オブジェクトを作成するコマンドを再構築します。(これは逆コンパイルによる再構築で、元のコマンドのテキストではありません。)
pg_get_triggerdef (trigger oid [, pretty boolean]) → text トリガを作成するコマンドを再構築します。(これは逆コンパイルによる再構築で、元のコマンドのテキストではありません。)
pg_get_userbyid (role oid) → name OIDで指定されるロール名を返します。
pg_get_viewdef (view oid [, pretty boolean]) → text ビューあるいはマテリアライズドビューの背後にあるSELECTコマンドを再構築します。(これは逆コンパイルによる再構築で、元のコマンドのテキストではありません。)
pg_get_viewdef (view oid, wrap_column integer) → text ビューあるいはマテリアライズドビューの背後にあるSELECTコマンドを再構築します。(これは逆コンパイルによる再構築で、元のコマンドのテキストではありません。) この関数形式では整形するオプションは常に有効で、列数よりも短く保つように長い行は折り返されます。
pg_get_viewdef (view text [, pretty boolean]) → text ビューあるいはマテリアライズドビューの背後にあるSELECTコマンドを再構築します。ビューのOIDではなく、テキスト形式の名前を使います。(これは廃止予定です。OIDのバージョンを使ってください。)
pg_index_column_has_property (index regclass, column integer, property text) → boolean インデックス列が名前付きのプロパティを持つかどうかを検査します。共通のインデックス列プロパティは表 9.69に列挙されています。(拡張アクセスメソッドはインデックスに対して追加のプロパティ名を持てることに注意してください。) プロパティ名が不明あるいは特定のオブジェクトに適用されない場合、OIDあるいは列番号が有効なオブジェクトを特定しない場合はNULLが返ります。
pg_index_has_property (index regclass, property text) → boolean インデックスが名前付きのプロパティを持つかどうかを検査します。共通のインデックスプロパティは表 9.70に列挙されています。(拡張アクセスメソッドはインデックスに対して追加のプロパティ名を持てることに注意してください。) プロパティ名が不明あるいは特定のオブジェクトに適用されない場合、OIDが有効なオブジェクトを特定しない場合はNULLが返ります。
pg_indexam_has_property (am oid, property text) → boolean インデックスアクセスメソッドが名前付きのプロパティを持つかどうかを検査します。アクセスメソッドプロパティは表 9.71に列挙されています。プロパティ名が不明あるいは特定のオブジェクトに適用されない場合、OIDが有効なオブジェクトを特定しない場合はNULLが返ります。
pg_options_to_table (options_array text[]) → setof record (option_name text, option_value text) pg_class.reloptionsあるいはpg_attribute.attoptionsの値で表現されるストレージオプションの集合を返します。
pg_tablespace_databases (tablespace oid) → setof oid 指定したテーブル空間に格納されるオブジェクトを持つデータベースのOIDの集合を返します。この関数が何らかの行を返すならば、そのテーブル空間は空ではなく、削除できません。特定のオブジェクトがそのテーブル空間にあるかどうかを確認するには、pg_tablespace_databasesで識別されるデータベースに接続してpg_classカタログを検索する必要があります。
pg_tablespace_location (tablespace oid) → text

関数 説明
テーブル空間が配置されているファイルシステムのパスを返します。

pg_typeof ("any") → regtype
渡された値のデータ型のOIDを返します。これはトラブル解決作業、または動的にSQL問い合わせを生成するのに便利です。この関数は、OIDの別名型であるregtypeを返すものとして宣言されます（8.19を参照）。つまり、比較目的のOIDと同一ですが、型名として表示されます。
以下に例をあげます。
`SELECT pg_typeof(33);`
` pg_typeof`
`-----------`
` integer`
 	
`SELECT typlen FROM pg_type WHERE oid = pg_typeof(33);`
` typlen`
`--------`
`      4`

COLLATION FOR ("any") → text
渡された値の照合順序の名前を返します。値は必要ならば引用符付きでスキーマ修飾されます。引数式から照合順序が生じなければ、NULLが返ります。引数が照合可能なデータ型でなければ、エラーが生じます。
例を挙げます。
`SELECT collation for (description) FROM pg_description LIMIT 1;`
` pg_collation_for`
`------------------`
` "default"`
 	
`SELECT collation for ('foo' COLLATE "de_DE");`
` pg_collation_for`
`------------------`
` "de_DE"`

to_regclass (text) → regclass テキスト形式のリレーション名をOIDに変換します。同様の結果はその文字列をregclass型にキャストすることによっても得られます。（8.19参照。）しかしこの関数は名前が見つからない場合にエラーを起こすのではなく、NULLを返します。またキャストと違って数値のOIDを入力として受け付けません。
to_regcollation (text) → regcollation テキスト形式の照合順名をOIDに変換します。同様の結果はその文字列をregcollation型にキャストすることによっても得られます。（8.19参照。）しかしこの関数は名前が見つからない場合にエラーを起こすのではなく、NULLを返します。またキャストと違って数値のOIDを入力として受け付けません。
to_regnamespace (text) → regnamespace

関数 説明
テキスト形式のスキーマ名をOIDに変換します。同様の結果はその文字列をregnamespace型にキャストすることによっても得られます。(8.19参照。) しかしこの関数は名前が見つからない場合にエラーを起こすのではなく、NULLを返します。またキャストと違って数値のOIDを入力として受け付けません。
to_regoper (text) → regoper テキスト形式の演算子名をOIDに変換します。同様の結果はその文字列をregoper型にキャストすることによっても得られます。(8.19参照。) しかしこの関数は名前が見つからない場合にエラーを起こすのではなく、NULLを返します。またキャストと違って数値のOIDを入力として受け付けません。
to_regoperator (text) → regoperator テキスト形式の演算子名(パラメータ型付き)をOIDに変換します。同様の結果はその文字列をregoperator型にキャストすることによっても得られます。(8.19参照。) しかしこの関数は名前が見つからない場合にエラーを起こすのではなく、NULLを返します。またキャストと違って数値のOIDを入力として受け付けません。
to_regproc (text) → regproc テキスト形式の関数名または手続き名をOIDに変換します。同様の結果はその文字列をregproc型にキャストすることによっても得られます。(8.19参照。) しかしこの関数は名前が見つからない場合にエラーを起こすのではなく、NULLを返します。またキャストと違って数値のOIDを入力として受け付けません。
to_regprocedure (text) → regprocedure テキスト形式の関数名または手続き名(引数型付き)をOIDに変換します。同様の結果はその文字列をregprocedure型にキャストすることによっても得られます。(8.19参照。) しかしこの関数は名前が見つからない場合にエラーを起こすのではなく、NULLを返します。またキャストと違って数値のOIDを入力として受け付けません。
to_regrole (text) → regrole テキスト形式のロール名をOIDに変換します。同様の結果はその文字列をregrole型にキャストすることによっても得られます。(8.19参照。) しかしこの関数は名前が見つからない場合にエラーを起こすのではなく、NULLを返します。またキャストと違って数値のOIDを入力として受け付けません。
to_regtype (text) → regtype テキスト形式の型名をOIDに変換します。同様の結果はその文字列をregtype型にキャストすることによっても得られます。(8.19参照。) しかしこの関数は名前が見つからない場合にエラーを起こすのではなく、NULLを返します。またキャストと違って数値のOIDを入力として受け付けません。

データベースオブジェクトを再構築(逆コンパイル)する関数の多くにオプションのprettyフラグがあり、trueなら結果が「整形」されるようになっています。整形によって不必要な括弧が抑止され、見やすさのために空白は追加されます。整形形式は見やすいですが、デフォルト形式は将来のバージョンのPostgreSQLでも同じように解釈される可能性が高いです。ですから、整形された出力をダンプ目的で使わないでください。pretty引数にfalseを渡すとパラメータを省略したとの同じ結果が得られます。

表9.69 インデックス列の属性

名前	説明
asc	前方スキャンで列は昇順にソートされるか
desc	前方スキャンで列は降順にソートされるか
nulls_first	前方スキャンで列はNULLを先頭にしてソートするか
nulls_last	前方スキャンで列はNULLを最後にしてソートするか

名前	説明
orderable	列は定義済みのソート順を所有しているか
distance_orderable	列は「距離」の演算子の順序に従ってスキャンできるか、例えばORDER BY col <-> 定数など
returnable	列の値をインデックスオンリースキャンで返すことができるか
search_array	列はcol = ANY(array)の検索をネイティブにサポートしているか
search_nulls	列はIS NULLおよびIS NOT NULLの検索をサポートしているか

表9.70 インデックスの属性

名前	説明
clusterable	インデックスをCLUSTERコマンドで使うことができるか
index_scan	インデックスは通常の(ビットマップでない)スキャンをサポートしているか
bitmap_scan	インデックスはビットマップスキャンをサポートしているか
backward_scan	スキャンの途中でスキャン方向を変更できるか(マテリアライゼーションを必要とせずにカーソルのFETCH BACKWARDをサポートするため)

表9.71 インデックスアクセスメソッドの属性

名前	説明
can_order	アクセスメソッドはCREATE INDEXにおいてASC、DESCおよび関連するキーワードをサポートしているか
can_unique	アクセスメソッドは一意インデックスをサポートしているか
can_multi_col	アクセスメソッドは複数列にまたがるインデックスをサポートしているか
can_exclude	アクセスメソッドは排他制約をサポートしているか
can_include	アクセスメソッドがCREATE INDEXのINCLUDE句をサポートしているか

表 9.72にデータベースオブジェクトの識別とアドレスに関連する関数を示します。

表9.72 オブジェクト情報とアドレスの関数

関数 説明
pg_describe_object (classid oid, objid oid, objsubid integer) → text カタログOID、オブジェクトOID、もしくはサブオブジェクトOID(たとえばテーブル中の列番号。オブジェクト全体を参照している場合は0)で指定されたデータベースオブジェクトのテキストによる説明を返します。この説明はサーバの設定に依存しますが、人が読んでわかる、そして翻訳も可能になることを目的としたのもです。これはpg_dependカタログに格納されたオブジェクトの識別判断の際に有用です。
pg_identify_object (classid oid, objid oid, objsubid integer) → record (type text, schema text, name text, identity text)

関数 説明
カタログOID、オブジェクトOID、そしてサブオブジェクトIDにより指定されるデータベースオブジェクトを一意に特定するために十分な情報を含む行を返します。この情報は機械による読み取りを目的としており、決して翻訳されません。typeはデータベースオブジェクトの型を識別するものです。schemaはオブジェクトが所属するスキーマの名前ですが、スキーマに所属しないオブジェクト型の場合はNULLになります。nameは（必要なら引用符で括った）オブジェクトの名前ですが、（適切ならスキーマ名と合わせて）オブジェクトの一意識別子として使用できる場合にのみ指定し、それ以外の場合はNULLにします。identityは完全なオブジェクトの識別で、オブジェクトの型に依存した正確なフォーマットを持っています。フォーマット内の各部分はスキーマ修飾されており、必要に応じて引用符で括られます。
pg_identify_object_as_address (classid oid, objid oid, objsubid integer) → record (type text, object_names text[], object_args text[]) カタログOID、オブジェクトOID、そしてサブオブジェクトIDにより指定されるデータベースオブジェクトを一意に特定するために充分な情報を含む行を返します。返される情報は現在のサーバに依存しません。つまり、他のサーバで全く同じ名前を付けられたオブジェクトを識別するために使うことができます。typeはデータベースオブジェクトの型を識別するものです。object_namesとobject_argsは文字列の配列で、それらが組み合わされてオブジェクトへの参照を構成します。これらの3つの値は、オブジェクトの内部アドレスを取得するためにpg_get_object_addressに渡すことができます。
pg_get_object_address (type text, object_names text[], object_args text[]) → record (classid oid, objid oid, objsubid integer) 型、オブジェクト名および引数の配列で指定されたデータベースオブジェクトを一意に特定するために十分な情報を含む行を返します。返される値は、pg_dependなどのシステムカタログで使用されるもので、pg_identify_objectやpg_describe_objectなど他のシステム関数に渡すことができます。classidはオブジェクトを含むシステムカタログのOIDです。objidはオブジェクト自体のOIDです。objsubidはオブジェクトのサブID、なければ0です。この関数はpg_identify_object_as_addressの逆関数です。

表 9.73に示される関数は、COMMENTコマンドによって以前に保存されたコメントを抽出します。指定されたパラメータに対するコメントが存在しない場合、NULL値が返されます。

表9.73 Comment Information Functions

関数 説明
col_description (table oid, column integer) → text テーブルのOIDと列番号で指定されたテーブル列のコメントを返します。（obj_descriptionはテーブル列には使えません。列は自身のOIDを持たないからです。）
obj_description (object oid, catalog name) → text OIDとそれを含むシステム型で指定されるデータベースオブジェクトのコメントを返します。たとえばobj_description(123456, 'pg_class')はOID 123456のテーブルのコメントを返します。
obj_description (object oid) → text OIDだけで指定されるデータベースオブジェクトのコメントを返します。これは廃止予定です。異なるシステムカタログに渡ってOIDが一意であるという保証はないからです。ですから、間違ったコメントが返されるかも知れません。
shobj_description (object oid, catalog name) → text OIDとそれを含むシステム型で指定されるデータベース共有オブジェクトのコメントを返します。これは共有オブジェクト（すなわちデータベース、ロール、テーブル空間）のコメントを取り出すために使うのを除くとobj_descriptionと同じです。システムカタログによってはクラスタ内ですべてのデータベースに対して広域的で、その中のオブジェクトの説明も広域的に格納されています。

表 9.74で示される関数はサーバトランザクション情報をエクスポートできる形式で提供します。これら関数の主な使用目的は2つのスナップショット間でどのトランザクションがコミットされたのかを特定するためです。

表9.74 トランザクションIDとスナップショット情報関数

関数 説明
pg_current_xact_id () → xid8 現在のトランザクションIDを返します。現在のトランザクションに(データベース更新をまったく行わなかったために)IDがなければ新規に割り当てます。
pg_current_xact_id_if_assigned () → xid8 現在のトランザクションIDを返します。もしまだ割り当てられていなければNULLを返します。(トランザクションが読み取り専用なら、無駄なXIDの消費を避けるためにこの関数を使うのが最良です。)
pg_xact_status (xid8) → text 最近のトランザクションのコミット状態について報告します。トランザクションが最近のもので、システムがそのトランザクションのコミット状態を保持している場合は、トランザクションの状態はin progress、committedあるいはabortedとして報告されます。トランザクションが古く、その参照がシステムに残っておらず、コミット状態の情報が破棄されている場合は、この関数はNULLを返します。COMMITの進行中にアプリケーションとデータベースが切断されたときに、アプリケーションはトランザクションがコミットされたか中断されたかを知るためにこれを使うことができます。プリペアドのトランザクションはin progressとして報告されること、そして指定のIDがプリペアドのトランザクションかどうかを確認する必要がある場合は、アプリケーションはpg_prepared_xactsを調べなければならないことに注意してください。
pg_current_snapshot () → pg_snapshot どのトランザクションIDがin-progressなのかを表示するデータ構造である現在の*snapshot*を返します。
pg_snapshot_xip (pg_snapshot) → setof xid8 スナップショットに含まれるin-progressのトランザクションIDの集合を返します。
pg_snapshot_xmax (pg_snapshot) → xid8 スナップショットのxmaxを返します。
pg_snapshot_xmin (pg_snapshot) → xid8 スナップショットのxminを返します。
pg_visible_in_snapshot (xid8, pg_snapshot) → boolean このスナップショットによると与えられたトランザクションIDが可視か(すなわちスナップショットが取得される前に完了していたか)？この関数は副トランザクションIDに対しては正しい答えを返さないことに注意してください。

内部トランザクションID型(xid)は32ビット幅なので40億トランザクション毎にラップします。とは言っても、表 9.74に示される関数はインストレーションの生涯にわたってラップしないxid8型の64ビット形式を使用しており、必要に応じてxidにキャストして変換できます。これらの関数で使用されるデータ型、pg_snapshotはある特定の時間におけるトランザクションIDの可視性に関する情報を格納します。構成要素は表 9.75に記載されています。pg_snapshotのテキスト表現はxmin:xmax:xip_listです。たとえば10:20:10,14,15はxmin=10, xmax=20, xip_list=10, 14, 15であることを意味します。

表9.75 スナップショット構成要素

名前	説明
xmin	現在実行中で最も小さいトランザクションID。xminより小さい全てのトランザクションはコミットされて可視となっているか、またはロールバックされて消滅しています。
xmax	完了した最も大きなトランザクションIDの一つ大きなID。xmaxと等しいかより大きい全てのトランザクションIDはスナップショットの時点で未完了であり、従って不可視です。
xip_list	スナップショット時の実行中のトランザクションです。xmin <= X < xmaxで、このリストにないトランザクションIDはスナップショット時点ですでに完了しており、コミット状態によって可視あるいはデッドのどちらかです。リストには副トランザクションのトランザクションIDは含まれません。

PostgreSQLリリースの13より前ではxid8型がなく、これらの関数の変種は、64ビットのXIDを表現するためにbigintを、スナップショットデータ型に対応する別のtxid_snapshot型を使っていました。これらの古い関数ではtxidが名前に含まれています。過去互換性のためにこれらはまだサポートされていますが、将来のリリースでは削除されるかも知れません。表 9.76を参照してください。

表9.76 廃止予定のトランザクションIDとスナップショット情報関数

関数 説明
txid_current () → bigint pg_current_xact_id()参照。
txid_current_if_assigned () → bigint pg_current_xact_id_if_assigned()参照。
txid_current_snapshot () → txid_snapshot pg_current_snapshot()参照。
txid_snapshot_xip (txid_snapshot) → setof bigint pg_snapshot_xip()参照。
txid_snapshot_xmax (txid_snapshot) → bigint pg_snapshot_xmax()参照。
txid_snapshot_xmin (txid_snapshot) → bigint pg_snapshot_xmin()参照。
txid_visible_in_snapshot (bigint, txid_snapshot) → boolean pg_visible_in_snapshot()参照。
txid_status (bigint) → text pg_xact_status()参照。

表 9.77に示す関数はいつ過去のトランザクションがコミットされたかの情報を提供します。track_commit_timestamp設定オプションが有効のときにだけ、かつそれが有効になった後にコミットされたトランザクションについてのみ意味のある情報を提供します。

表9.77 コミットされたトランザクションに関する情報関数

関数 説明
pg_xact_commit_timestamp (xid) → timestamp with time zone トランザクションのコミットタイムスタンプを返します。
pg_last_committed_xact () → record (xid xid, timestamp timestamp with time zone) 直近にコミットしたトランザクションのトランザクションIDとコミットタイムスタンプを返します。

表 9.78に示す関数は、カタログのバージョンなどといったinitdbの実行時に初期化される情報を表示します。それらはまた、先行書き込みログとチェックポイント処理についての情報も示します。この情報はクラスタ全体に渡るもので、どれか1つのデータベースに特有のものではありません。これらの関数はpg_controldataアプリケーションと同じ情報源から、ほぼ同じ情報を提供します。

表9.78 制御データ関数

関数 説明
pg_control_checkpoint () → record 表 9.79に示すように現在のチェックポイントの状態に関する情報を返します。
pg_control_system () → record 表 9.80に示すように現在の制御ファイルの状態に関する情報を返します。
pg_control_init () → record 表 9.81に示すようにクラスタの初期化状態に関する情報を返します。
pg_control_recovery () → record 表 9.82に示すようにリカバリ状態に関する情報を返します。

表9.79 pg_control_checkpointの出力列

列名	データ型
checkpoint_lsn	pg_lsn
redo_lsn	pg_lsn
redo_wal_file	text
timeline_id	integer
prev_timeline_id	integer
full_page_writes	boolean
next_xid	text
next_oid	oid
next_multixact_id	xid
next_multi_offset	xid
oldest_xid	xid
oldest_xid_dbid	oid

列名	データ型
oldest_active_xid	xid
oldest_multi_xid	xid
oldest_multi_dbid	oid
oldest_commit_ts_xid	xid
newest_commit_ts_xid	xid
checkpoint_time	timestamp with time zone

表9.80 pg_control_systemの出力列

列名	データ型
pg_control_version	integer
catalog_version_no	integer
system_identifier	bigint
pg_control_last_modified	timestamp with time zone

表9.81 pg_control_init Output Columns

列名	データ型
max_data_alignment	integer
database_block_size	integer
blocks_per_segment	integer
wal_block_size	integer
bytes_per_wal_segment	integer
max_identifier_length	integer
max_index_columns	integer
max_toast_chunk_size	integer
large_object_chunk_size	integer
float8_pass_by_value	boolean
data_page_checksum_version	integer

表9.82 pg_control_recoveryの出力列

列名	データ型
min_recovery_end_lsn	pg_lsn
min_recovery_end_timeline	integer
backup_start_lsn	pg_lsn
backup_end_lsn	pg_lsn
end_of_backup_record_required	boolean

9.27. システム管理関数

本節で説明する関数は、PostgreSQLインストレーションの制御と監視を行うために使用されます。

9.27.1. 構成設定関数

表 9.83は、実行時設定パラメータの問い合わせや変更に使用できる関数を示しています。

表9.83 構成設定関数

関数 説明 例
current_setting (setting_name text [, missing_ok boolean]) → text 現在のsetting_nameの設定値を返します。そのような設定がなければ、missing_okが渡され、それがtrueでない限りcurrent_settingはエラーを引き起こします。この関数はSQLコマンドのSHOWに関連します。 current_setting('datestyle') → ISO, MDY
set_config (setting_name text, new_value text, is_local boolean) → text setting_nameパラメータにnew_valueを設定し、その値を返します。is_localが渡され、それがtrueなら新しい値は現在のトランザクションにのみ適用されます。現在のセッションに新しい値を適用したければ、代わりにfalseとしてください。このコマンドはSQLコマンドのSETに関連します。 set_config('log_statement_stats', 'off', false) → off

9.27.2. サーバシグナル送信関数

表 9.84に示す関数は、制御用シグナルを他のサーバプロセスに送信します。これらの関数の使用は、デフォルトでスーパーユーザのみに制限されていますが、注記された例外を除き、GRANTを使用して他のユーザにアクセスを許可できます。

これらのそれぞれの関数は成功の場合true(真)を返し、そうでない場合はfalse(偽)を返します。

表9.84 サーバシグナル送信関数

関数 説明
pg_cancel_backend (pid integer) → boolean 指定したプロセスIDを持つバックエンドプロセスの現在のセッションの問い合わせを取り消します。呼び出し側のロールがキャンセルされるバックエンドのロールのメンバーであるか、pg_signal_backendの権限を与えられている場合に実行できます。ただし、スーパーユーザのバックエンドはスーパーユーザのみが取り消せます。
pg_reload_conf () → boolean PostgreSQLのすべてのサーバプロセスに構成ファイルの再読み込みをさせます。(これはSIGHUPシグナルをpostmasterプロセスに送ることによって始まり、postmasterは続いてSIGHUPを子に送ります。)
pg_rotate_logfile () → boolean ログファイルマネージャにシグナルを送って新しいファイルに直ちに切り替えさせます。これは組み込みのログ収集機構が実行中のみ動作します。でないとログマネージャサブプロセスが存在しないからです。
pg_terminate_backend (pid integer) → boolean

関数 説明
バックエンドが指定したプロセスIDを持つセッションを終了させます。呼び出し側のロールが終了されるバックエンドのロールのメンバーであるか、pg_signal_backendの権限を与えられている場合に実行できます。ただし、スーパーユーザのバックエンドはスーパーユーザのみが終了できます。

pg_cancel_backendとpg_terminate_backendは(それぞれ、SIGINTまたはSIGTERM)シグナルをプロセス識別子で特定されたバックエンドプロセスに送ります。使用中のバックエンドのプロセス識別子はpg_stat_activityビューのpid列から、もしくは、(Unixではps、WindowsではTask Managerにより)サーバ上のpostgresプロセスをリストすることで見つけられます。実行中のバックエンドのロールはpg_stat_activityのusename列から確認することができます。

9.27.3. バックアップ制御関数

表 9.85に示す関数はオンラインバックアップの作成を支援するものです。これらの関数は、リカバリ中には実行できません(非排他的pg_start_backup、非排他的pg_stop_backup、pg_is_in_backup、pg_backup_start_time、およびpg_wal_lsn_diffは除く)。

これらの関数の正しい使用方法については、25.3を参照してください。

表9.85 バックアップ制御関数

関数 説明
pg_create_restore_point (name text) → pg_lsn 先行書き込みログ中に後でリカバリターゲットとして使用できる名前付けされたマーカーレコードを作成し、関連する先行書き込みログの位置を返します。与えられた名前はリカバリをどこまで進めるかを指定するためにrecovery_target_nameとともに利用できます。同じ名前で複数のリストアポイントを作成するのは避けてください。リカバリターゲットが一致した最初のところでリカバリが停止するからです。 この関数はデフォルトではスーパーユーザのみ実施可能ですが、他のユーザにも関数を実行するEXECUTE権限を与えることができます。
pg_current_wal_flush_lsn () → pg_lsn 先行書き込みログの現在のフラッシュ位置を取得します。(下の注釈を参照してください。)
pg_current_wal_insert_lsn () → pg_lsn 現在の先行書き込みログの挿入位置を取得します。(下の注釈を参照してください。)
pg_current_wal_lsn () → pg_lsn 現在の先行書き込みログの書き込み位置を取得します。(下の注釈を参照してください。)
pg_start_backup (label text [, fast boolean [, exclusive boolean]]) → pg_lsn サーバがオンラインバックアップを開始するのを準備します。必須パラメータはユーザが任意に定義したバックアップラベルだけです。(通常、格納に使用するバックアップダンプファイルにちなんだ名前が付けられます。) オプションの2番目のパラメータがtrueとして与えられると、pg_start_backupを可能な限り素早く実行することが指定されます。これによりI/O操作の急上昇をもたらして同時に実行中のすべての問い合わせを遅くする即時チェックポイントが強制されます。オプションの3番目のパラメータは、排他あるいは非排他のどちらのバックアップを実行するかを指定します。(デフォルトは排他です。) 排他モードで使用される場合、この関数は、データベースクラスタのデータディレクトリにバックアップラベルファイル(backup_label)およびpg_tblspc/ディレクトリにリンクがあるならテーブル空間マップファイル(tablespace_map)を書き出し、チェックポイントを実行し、先行書き込みログのバックアップ開始位置をテキスト形式で返します。(ユーザはこの結果値を無視す

関数 説明
ることもできますが、便利なこともあるので提供されています。) 非排他モードで使用される場合は、排他モードとは違い、これらのファイルの内容がpg_stop_backup関数によって戻され、呼び出しユーザはそれをバックアップに書き込む必要があります。 この関数はデフォルトではスーパーユーザのみ実施可能ですが、他のユーザにも関数を実行するEXECUTE権限を与えることができます。
pg_stop_backup (exclusive boolean [, wait_for_archive boolean]) → setof record (lsn pg_lsn, labelfile text, spcmapfile text) 排他的あるいは非排他的オンラインバックアップを終了します。exclusiveパラメータは以前のpg_start_backupの呼び出しと一致していなければなりません。排他的バックアップでは、pg_stop_backupは、pg_start_backupで作成されたラベルファイルおよび、もしあればテーブル空間マップファイルを削除します。非排他的バックアップでは、これらのファイルの必要な内容が関数の結果の一部として返され、それをバックアップ内のファイルに書き込む必要があります(データディレクトリ内のファイルに書いてはいけません)。 オプションで2番目のboolean型パラメータがあります。falseの場合、pg_stop_backupはバックアップの完了後、WALがアーカイブされるのを待たずに、即座に戻ります。この動作はWALのアーカイブを独立して監視するバックアップソフトウェアに対してのみ有用です。それ以外の場合、バックアップを一貫性のあるものにするために必要なWALが欠けるためにバックアップが役立たなくなるかもしれません。省略時あるいはこのパラメータがtrueのとき、アーカイブが有効なら、pg_stop_backupはWALがアーカイブされるまで待機します。(スタンバイでは、これはつまりarchive_mode = alwaysのときのみ待機するということです。プライマリでの書き込み活動が少ないときは、セグメントの変更を即座に起こさせるためにプライマリでpg_switch_walを実行するのが有効かもしれません。) プライマリで実行された場合、この関数はまた、先行書き込みログの格納領域にバックアップ履歴ファイルを作成します。履歴ファイルにはpg_start_backupで付与されたラベル、バックアップの先行書き込みログの位置の開始位置、終了位置、バックアップ開始時刻、終了時刻が含まれます。終了位置を記録した後、現在の先行書き込みログの挿入位置は自動的に、次の先行書き込みログファイルに進みます。従って、終了先行書き込みログファイルをすぐにアーカイブし、バックアップを完了させることができます。 この関数の結果は単一レコードです。lsn列はバックアップの終了先行書き込みログの位置です(これもまた無視可能です)。2番目3番目の列は排他的バックアップの終了ではNULLです。非排他的バックアップの後では、2番目3番目の列にはラベルとテーブル空間マップファイルの必要な内容が含まれます。 この関数はデフォルトではスーパーユーザのみ実施可能ですが、他のユーザにも関数を実行するEXECUTE権限を与えることができます。
pg_stop_backup () → pg_lsn 排他的オンラインバックを終了します。pg_lsnを返す点だけを除けばこれはpg_stop_backup(true, true)の単純化されたバージョンです。 この関数はデフォルトではスーパーユーザのみ実施可能ですが、他のユーザにも関数を実行するEXECUTE権限を与えることができます。
pg_is_in_backup () → boolean 排他的オンラインバックアップが実行中なら真を返します。
pg_backup_start_time () → timestamp with time zone 実行中ならオンライン排他的バックアップの開始時刻を返します。実行中でなければNULLを返します。
pg_switch_wal () → pg_lsn サーバに対して新しい先行書き込みログファイルへ強制スイッチを行い、それによってその現在のファイルがアーカイブされるようにします。(継続的アーカイブを利用中だと仮定します。) 結果は今ちょうど終了した先行書き込みログファイル内の終

関数 説明
了先行書き込みログファイルの場所プラス1です。最後の先行書き込みログファイルのスイッチ以降先行書き込みログファイル活動がなければ、pg_switch_walは何もせず、現在使用中の先行書き込みログファイルの先頭位置を返します。 この関数はデフォルトではスーパーユーザのみ実施可能ですが、他のユーザにも関数を実行するEXECUTE権限を与えることができます。
pg_walfile_name (lsn pg_lsn) → text 先行書き込みログファイルの位置を、その位置を保持しているWALファイルの名前に変換します。
pg_walfile_name_offset (lsn pg_lsn) → record (file_name text, file_offset integer) 先行書き込みログファイルの位置を、そのファイルの名前とそのファイル内のバイトオフセットに変換します。
pg_wal_lsn_diff (lsn pg_lsn, lsn pg_lsn) → numeric 2つの先行書き込みログの位置のバイト単位の差分を計算します。これはpg_stat_replicationや表 9.85内の関数でレプリケーションの遅延を取得するために使用することができます。

pg_current_wal_lsnは、上記の関数で使用されるのと同じ書式で現在の先行書き込みログの書き込み位置を表示します。同様にpg_current_wal_insert_lsnは、現在の先行書き込みログの挿入位置を表示し、pg_current_wal_flush_lsnはトランザクションログの現在のフラッシュ位置を表示します。挿入位置は「論理的」な任意の時点の先行書き込みログの終了位置です。一方、書き込み位置は、サーバの内部バッファから書き出された実際の終了位置、またフラッシュ位置は永続的ストレージへの書き込みが保証される位置です。書き込み位置はサーバ外部から検証可能なものの終端です。通常は、部分的に完了した先行書き込みログファイルのアーカイブ処理を行いたい場合に必要とされるものです。挿入およびフラッシュ位置はサーバをデバッグする際に主に使用されます。これらはどちらも読み取りのみの操作であり、スーパーユーザ権限を必要としません。

pg_walfile_name_offsetを使用して、pg_lsn値から、対応する先行書き込みログファイル名とバイトオフセットを取り出すことができます。以下に例を示します。

```
postgres=# SELECT * FROM pg_walfile_name_offset(pg_stop_backup());
        file_name         | file_offset
--------------------------+-------------
 000000010000000000000002D |     4039624
(1 row)
```

同様に、pg_walfile_nameは、先行書き込みログファイル名のみを取り出します。指定した先行書き込みログの位置が正確に先行書き込みログファイルの境界であった場合、これらの両関数は前の先行書き込みログファイルの名前を返します。通常これは、先行書き込みログファイルのアーカイブ動作では好まれる動作です。前のファイルが現在のアーカイブで必要とする最後のファイルであるからです。

9.27.4. リカバリ制御関数

表 9.86に示される関数は、スタンバイサーバの現在のステータス情報を提供します。これらの関数はリカバリ中、および通常稼動時に実行することができるでしょう。

表9.86 リカバリ情報関数

関数 説明
pg_is_in_recovery () → boolean まだリカバリ実施中であれば真を返します。
pg_last_wal_receive_lsn () → pg_lsn ストリーミングレプリケーションにより受信されディスクに同期書き込みされた、先行書き込みログの最後の位置を返します。ストリーミングレプリケーションがまだ実行中の場合、この関数の戻り値は単調に増加します。リカバリが完了した場合は、受信されディスクに書き込まれた最後のWALレコードの位置の値のまま変化しません。ストリーミングレプリケーションが無効、もしくは開始されていない場合、この関数はNULLを返します。
pg_last_wal_replay_lsn () → pg_lsn リカバリ中に再生された最後の先行書き込みログの位置を返します。リカバリがまだ実行中の場合、この関数の戻り値は単調に増加します。リカバリが完了した場合は、リカバリ時に適用された最後のWALレコードの値のまま変化しません。サーバがリカバリ処理無しに正常に開始された場合、この関数はNULLを返します。
pg_last_xact_replay_timestamp () → timestamp with time zone リカバリ中に再生された最後のトランザクションのタイムスタンプを返します。このタイムスタンプは、プライマリにて該当するトランザクションがコミット、もしくはアボートされた際のWALレコードが生成された時刻です。リカバリ中に何のトランザクションも再生されていない場合、この関数はNULLを返します。リカバリがまだ実行中の場合、この関数の戻り値は単調に増加します。リカバリが完了している場合、この関数の戻り値はリカバリ中に再生した最後のトランザクションの時間のまま変化しません。サーバがリカバリ処理無しに正常に開始された場合、この関数はNULLを返します。

表 9.87に示す関数は、リカバリの進行を制御する関数です。これらの関数はリカバリ中のみ実行することが可能です。

表9.87 リカバリ制御関数

関数 説明
pg_is_wal_replay_paused () → boolean リカバリが中断中なら真を返します。
pg_promote (wait boolean DEFAULT true, wait_seconds integer DEFAULT 60) → boolean スタンバイサーバをプライマリ状態に昇格します。waitにtrue(デフォルト)を設定すると、この関数は昇格が完了するか、wait_seconds秒が経過するまで待ち、昇格に成功すればtrue、さもなければfalseを返します。waitにfalseを設定すると、この関数は昇格を起こすためにpostmasterにSIGUSR1を送信した後、直ちにtrueを返します。 デフォルトではこの関数の実行はスーパーユーザに限定されますが、他のユーザに関数を実行するEXECUTE権限を与えることができます。
pg_wal_replay_pause () → void リカバリを中断します。リカバリが中断している間、データベースへの変更は適用されません。ホットスタンバイが動作中はすべての新しい問い合わせはデータベースの一貫した同じスナップショットを参照することになり、リカバリが再開するまでそれ以上の問い合わせの衝突は起きません。 デフォルトではこの関数の実行はスーパーユーザに限定されますが、他のユーザに関数を実行するEXECUTE権限を与えることができます。
pg_wal_replay_resume () → void リカバリが中断中なら再開します。

関数 説明
デフォルトではこの関数の実行はスーパーユーザに限定されますが、他のユーザに関数を実行するEXECUTE権限を与えることができます。

pg_wal_replay_pauseとpg_wal_replay_resumeは昇格が進行中は実行できません。リカバリ中断中に昇格が引き起こされると中断状態は終了し、昇格が継続します。

ストリーミングレプリケーションが無効の場合、停止状態は特に問題なく永久に継続します。ストリーミングレプリケーションの進行中は、WALレコードの受信が継続され、停止時間、WALの生成速度、ディスクの残存容量によりますが、ディスク溢れが発生する可能性があります。

9.27.5. スナップショット同期関数

PostgreSQLはデータベースのセッションに対して、それらのスナップショットを同期させることが可能です。スナップショットは、そのスナップショットを使用しているトランザクションにどのデータが可視かを決定します。同期スナップショットは、2つ以上のセッションにおいて、全く同じデータベース内容を見たい場合に必要となります。単に2つのセッションが独立してそれぞれのトランザクションを開始するだけでは、第3のトランザクションのコミットが、2つのトランザクションのSTART TRANSACTIONの狭間で実行され、そのため一方のトランザクションではそのコミット結果が見え、他方では見えないという可能性が常にあります。

このような問題を解決するため、PostgreSQLではトランザクションが使用しているスナップショットをエクスポートできるようになっています。エクスポートしたトランザクションが開かれ続けている限り、他のトランザクションがそれをインポートすることができ、そしてこれにより最初のトランザクションと正確に同じとなるデータベースの可視性を保証されます。ただし、これらの(スナップショットを共有している)トランザクションによって発生したデータベースへの変更は、コミットされていないトランザクションによる変更と同様に、(スナップショットを共有している)他のトランザクションには見えないままです。つまり、既存データに対しては同期されますが、それら自身による変更については通常の振る舞いをします。

スナップショットは、表 9.88に示すpg_export_snapshot関数を用いてエクスポートされ、SET TRANSACTIONコマンドを用いてインポートされます。

表9.88 スナップショット同期関数

関数 説明
pg_export_snapshot () → text 現在のトランザクションのスナップショットを保存し、それを識別するtext文字列を返します。この文字列は(データベースの外側で)スナップショットを取り込みたいクライアントに渡さなければなりません。エクスポートしたトランザクションが終わるまでの間のみ、そのスナップショットをインポートすることができます。 必要ならばトランザクションは複数のスナップショットをエクスポートできます。これはREAD COMMITTEDのトランザクションにおいてのみ有用であることに注意してください。REPEATABLE READおよびそれ以上の分離レベルのトランザクションでは終了まで同じスナップショットを使うからです。一旦スナップショットをエクスポートしたトランザクションでは、PREPARE TRANSACTIONによる準備を使用することができなくなります。

9.27.6. レプリケーション管理関数

表 9.89に示す関数はレプリケーション機能を制御したり、情報を取得したりするためのものです。基盤となっている機能の情報に関しては26.2.5、26.2.6、第49章を参照してください。これらの関数のレプリケーションオリジンでの使用はスーパーユーザに限定されています。これらの関数のレプリケーションスロットでの使用はスーパーユーザとREPLICATION権限を持つユーザに限定されています。

これらの関数の多くには、レプリケーションプロトコルに等価なコマンドがあります。52.4を参照してください。

9.27.3、9.27.4、9.27.5に書かれている関数もレプリケーションに関係するものです。

表9.89 レプリケーション管理関数

関数 説明
pg_create_physical_replication_slot (slot_name name [, immediately_reserve boolean, temporary boolean]) → record (slot_name name, lsn pg_lsn) slot_nameという名前の新しい物理レプリケーションスロットを作成します。2番目のパラメータはオプションで、trueの場合、このレプリケーションスロットのLSNが即座に予約されることを指定します。それ以外の場合はLSNはストリーミングレプリケーションのクライアントから最初に接続された時に予約されます。物理スロットからのストリーミングの変更はストリーミングレプリケーションプロトコルでのみ可能です。52.4を参照してください。3番目のパラメータtemporaryはオプションで、trueに設定されるとそのスロットは永続的にディスクに保存されるものではなく、現在のセッションによってのみ用いられることを意図していることを指定します。一時的なスロットはエラーが発生したときも解放されます。この関数は、レプリケーションプロトコルコマンドCREATE_REPLICATION_SLOT ... PHYSICALに対応するものです。
pg_drop_replication_slot (slot_name name) → void slot_nameという名前の物理もしくは論理レプリケーションスロットを削除します。レプリケーションプロトコルコマンドDROP_REPLICATION_SLOTと同じです。論理スロットの場合、この関数はスロットが作成されたのと同じデータベースに接続している時に呼ばなければなりません。
pg_create_logical_replication_slot (slot_name name, plugin name [, temporary boolean]) → record (slot_name name, lsn pg_lsn) 出力プラグインpluginを使ってslot_nameという名前の新しい論理（デコーディング）レプリケーションスロットを作ります。3番目のオプションパラメータtemporaryをtrueに設定すると、このスロットを永続的にディスクに保存するべきではなく、現在のセッションでのみ使われることを意図します。また、一時スロットはエラーが起きると解放されます。この関数の呼び出しはレプリケーションプロトコルコマンドのCREATE_REPLICATION_SLOT ... LOGICALと同じ効果があります。
pg_copy_physical_replication_slot (src_slot_name name, dst_slot_name name [, temporary boolean]) → record (slot_name name, lsn pg_lsn) src_slot_nameという名前の既存の物理レプリケーションスロットをdst_slot_nameという名前の物理レプリケーションスロットにコピーします。コピーされた物理スロットはソーススロットと同じLSNからWALの保存を開始します。temporaryはオプションです。temporaryを省略すると、ソーススロットと同じ値を使用します。
pg_copy_logical_replication_slot (src_slot_name name, dst_slot_name name [, temporary boolean [, plugin name]]) → record (slot_name name, lsn pg_lsn) src_slot_nameという名前の既存の論理レプリケーションスロットをdst_slot_nameという名前の論理レプリケーションスロットにコピーします。オプションで出力プラグインと永続性を変更します。コピーされた論理スロットはソース論理スロットと同じLSNから開始します。temporaryとpluginはどちらもオプションです。省略するとソース論理スロットと同じ値が使用されます。
pg_logical_slot_get_changes (slot_name name, upto_lsn pg_lsn, upto_nchanges integer, VARIADIC options text[]) → setof record (lsn pg_lsn, xid xid, data text)

関数 説明
変更が最後に消費された時点から開始して、スロットslot_nameの変更を返します。upto_lsnとupto_nchangesがNULLならば論理デコードはWALの最後まで続きます。upto_lsnが非NULLであれば、デコードは指定されたLSNより前にコミットされたトランザクションのみを含みます。upto_nchangesが非NULLであれば、デコードにより生成された行の数が指定された値を越えたときに、デコードは止まります。しかしながら、新しいトランザクションの各コミットをデコードして生成された行を追加した後でしかこの制限は確認されませんので、実際に返される行の数は大きいかもしれないことに注意してください。
pg_logical_slot_peek_changes (slot_name name, upto_lsn pg_lsn, upto_nchanges integer, VARIADIC options text[]) → set of record (lsn pg_lsn, xid xid, data text) 変更が消費されないということを除いて、pg_logical_slot_get_changes()関数と同じように振る舞います。すなわち、将来の呼び出しでは再び同じものが返ります。
pg_logical_slot_get_binary_changes (slot_name name, upto_lsn pg_lsn, upto_nchanges integer, VARIADIC options text[]) → setof record (lsn pg_lsn, xid xid, data bytea) 変更はbyteaとして返されるということを除いてpg_logical_slot_get_changes()関数と同じように振る舞います。
pg_logical_slot_peek_binary_changes (slot_name name, upto_lsn pg_lsn, upto_nchanges integer, VARIADIC options text[]) → setof record (lsn pg_lsn, xid xid, data bytea) 変更はbyteaとして返されることを除いてpg_logical_slot_peek_changes()関数と同じように振る舞います。
pg_replication_slot_advance (slot_name name, upto_lsn pg_lsn) → record (slot_name name, end_lsn pg_lsn) slot_nameという名前のレプリケーションスロットの現在の確認された位置を進めます。スロットは後方には動きませんし、現在の挿入位置を超えて進むこともありません。スロットの名前と前に進んだ実際の位置を返します。前に進んだ場合は更新されたスロットに関する情報がこの後のチェックポイントで書き出されます。クラッシュが発生すると、そのスロットは以前の位置に戻るかもしれません。
pg_replication_origin_create (node_name text) → oid 指定した外部名でレプリケーション起点を作成し、割り当てられた内部IDを返します。
pg_replication_origin_drop (node_name text) → void 以前に作成されたレプリケーション起点を、それに関連するすべての再生の進捗も含めて削除します。
pg_replication_origin_oid (node_name text) → oid レプリケーション起点を名前で検索し、内部IDを返します。相当するレプリケーション起点が見つからない場合はエラーが発生します。
pg_replication_origin_session_setup (node_name text) → void 現在のセッションに、指定の起点から再生中であると印を付け、再生の進捗が追跡できるようにします。起点が選択されていない場合にのみ使うことができます。元に戻すにはpg_replication_origin_session_resetを使って下さい。
pg_replication_origin_session_reset () → void pg_replication_origin_session_setup()の効果を取り消します。
pg_replication_origin_session_is_setup () → boolean 現在のセッションでレプリケーション起点が選択されていれば真を返します。
pg_replication_origin_session_progress (flush boolean) → pg_lsn 現在のセッションで設定されたレプリケーション起点の再生位置を返します。パラメータflushにより、対応するローカルトランザクションがディスクにフラッシュされていることが保証されるかどうかを決定します。
pg_replication_origin_xact_setup (origin_lsn pg_lsn, origin_timestamp timestamp with time zone) → void 現在のトランザクションに、指定のLSNおよびタイムスタンプでコミットしたトランザクションを再生中であると印をつけます。事前にレプリケーション起点がpg_replication_origin_session_setupを使って選択されている場合にのみ呼び出せます。

関数 説明
pg_replication_origin_xact_reset () → void pg_replication_origin_xact_setup()の効果を取り消します。
pg_replication_origin_advance (node_name text, lsn pg_lsn) → void 指定したノードのレプリケーションの進捗を、指定の位置に設定します。これは主に設定変更の後で初期位置や新しい位置を設定するときなどに役立ちます。この関数を不注意に使うと、レプリケーションデータが一貫性を失うかもしれないことに注意して下さい。
pg_replication_origin_progress (node_name text, flush boolean) → pg_lsn 指定したレプリケーション起点の再生位置を返します。パラメータflushにより、対応するローカルトランザクションがディスクにフラッシュされていることが保証されるかどうかを決定します。
pg_logical_emit_message (transactional boolean, prefix text, content text) → pg_lsn pg_logical_emit_message (transactional boolean, prefix text, content bytea) → pg_lsn 論理デコードのメッセージを送出します これは汎用的なメッセージをWALを通して論理デコードのプラグインに渡すのに使うことができます。パラメータtransactionalは、メッセージが現在のトランザクションの一部なのか、あるいはすぐに書き込み、論理デコードがレコードを読んだらすぐにデコードされるべきものなのかを指定します。prefixパラメータは文字通りの接頭辞で、論理デコードのプラグインが、自分にとって関心のあるメッセージを容易に認識できるように使われます。contentパラメータはメッセージの内容で、テキストまたはバイナリ形式で与えられます。

9.27.7. データベースオブジェクト管理関数

表 9.90で示された関数はデータベースオブジェクトのディスク領域の使用状況を計算したり、使用結果の表示を補助します。これらの関数はすべてバイト単位の大きさを返します。関数に存在するオブジェクト以外のOIDが渡されるとNULLが返ります。

表9.90 データベースオブジェクトサイズ関数

関数 説明
pg_column_size ("any") → integer 個々のデータ値を格納するのに使用されるバイト数を表示します。テーブルの列の値に直接適用すると、圧縮が行われていればそれを反映します。
pg_database_size (name) → bigint pg_database_size (oid) → bigint 名前あるいはOIDで指定したデータベースによって使われている全ディスクスペースを計算します。この関数を使うには、指定したデータベースにCONNECT権限(デフォルトで付与されています)を持っているか、pg_read_all_statsロールのメンバーでなければなりません。
pg_indexes_size (regclass) → bigint 指定したテーブルに付与されたインデックスで使用されている全ディスクスペースを計算します。
pg_relation_size (relation regclass [, fork text]) → bigint 指定したリレーションの一つの「fork」で使用されているディスクスペースを計算します。(大抵の目的には、すべてのフォークのサイズを合計する高レベルのpg_total_relation_sizeあるいはpg_table_sizeを使う方が便利です。) 引数1つではリレーションの主データフォークのサイズを返します。2番目の引数で対象となるのがどのフォークであるかを指定できます。

関数 説明
• mainはリレーションの主データフォークのサイズを返します。 • fsmを指定すると、リレーションに関連した空き領域マップ(68.3を参照)のサイズを返します。 • vmを指定すると、リレーションに関連した可視性マップ(68.4を参照)のサイズを返します。 • initを指定すると、あれば、リレーションに関連した初期化フォークのサイズを返します。
pg_size_bytes (text) → bigint 可読性の高い形式(pg_size_prettyが返したもの)をバイトに変換します。
pg_size_pretty (bigint) → text pg_size_pretty (numeric) → text バイトサイズを、サイズ単位(バイト、kB、MB、GB、TBのうちの適切なもの)を使ったより人間が読みやすい形式に変換します。単位は10のべき乗ではなく、2のべき乗であることに注意してください。ですから1kBは1024バイトで、1MBは1024^2 = 1048576バイト、などとなります。
pg_table_size (regclass) → bigint 指定テーブルが使用している、インデックスを含まないディスクスペースを計算します。(ただしあればTOASTテーブル、空き領域マップ、可視性マップを含みます。)
pg_tablespace_size (name) → bigint pg_tablespace_size (oid) → bigint 名前あるいはOIDで指定されたテーブル空間で使用されているディスクスペースを計算します。現在のデータベースのデフォルトテーブル空間でない限り、この関数を使うには、指定したテーブル空間にCREATE権限を持っているか、pg_read_all_statsロールのメンバーでなければなりません。
pg_total_relation_size (regclass) → bigint 指定テーブルが使用している、インデックスとTOASTデータを含む全ディスクスペースを計算します。結果はpg_table_size + pg_indexes_sizeと等価です。

上記の関数において、テーブルやインデックスをregclass引数として受け取って処理するものがありますが、この引数は単にpg_classシステムカタログにあるテーブルやインデックスのOIDです。ただし、regclassデータ型が自動で入力変換を行うため、ユーザが手動で該当するOIDを調べる必要はありません。リテラル定数のようにシングルクオートで囲んだテーブル名を記述するだけです。通常のSQL名に対する処理互換のため、テーブル名をダブルクオートで囲わない限り、テーブル名として入力された文字列は小文字に変換されます。

表 9.91 に示される関数は、データベースオブジェクトに関連する特定のディスクファイルを確認する際の手助けとなります。

表9.91 データベースオブジェクト位置関数

関数 説明
pg_relation_filenode (relation regclass) → oid 指定されたリレーションに現在割り当てられている「ファイルノード」番号を返します。ファイルノードは、リレーションに使用しているファイル名の基本要素です。(詳しくは68.1を参照して下さい。) ほとんどのリレーションについては、結果はpg_class.relfilenodeと同じになります。ただし、いくつかのシステムカタログではrelfilenodeがゼロになるため、これらのシステムカタログの正しいファイルノードを取得するには、この関数を使用しなければなりません。この関数は、ビューの様にストレージに格納されないリレーションが指定された場合はNULLを返します。

関数 説明
pg_relation_filepath (relation regclass) → text リレーションのファイルパス名全体（データベースクラスタのデータディレクトリ、PGDATAからの相対）を返します。
pg_filenode_relation (tablespace oid, filenode oid) → regclass 与えられたテーブル空間OIDとファイルノードに格納されているリレーションのOIDを返します。これは本質的にpg_relation_filepathの逆マッピングです。データベースのデフォルトテーブル空間内のテーブルに対しては、テーブル空間は0と指定できます。当たられた値に対応する現在のデータベースにリレーションがなければNULLを返します。

表 9.92に照合順序の管理に使用される関数の一覧を示します。

表9.92 照合順序管理関数

関数 説明
pg_collation_actual_version (oid) → text オペレーティングシステムに現在インストールされている照合順序オブジェクトの実際のバージョンを返します。これがpg_collation.collversionと異なると、その照合順序に依存しているオブジェクトは再構築の必要があるかも知れません。ALTER COLLATIONも参照してください。
pg_import_system_collations (schema regnamespace) → integer オペレーティングシステム上にあるすべてのロケールに基づき、システムカタログpg_collationに照合順序を追加します。これはinitdbが使用しているもので、より詳細については23.2.2を参照してください。その後にオペレーティングシステムに追加のロケールをインストールした場合、この関数を再度実行して、その新しいロケールの照合順序を追加することができます。pg_collationの既存のエントリにマッチするロケールはスキップされます。（しかし、オペレーティングシステム上にもはや存在しなくなったロケールに基づく照合順序オブジェクトはこの関数では削除されません。）schemaパラメータは通常はpg_catalogですが、必ずしもそうでなければならないわけではなく、照合順序をどれか他のスキーマにインストールすることもできます。この関数は新しく作成された照合順序オブジェクトの数を返します。

表 9.93にパーティション化テーブルの構造に関する情報を提供する関数を示します。

表9.93 パーティション情報関数

関数 説明
pg_partition_tree (regclass) → setof record (relid regclass, parentrelid regclass, isleaf boolean, level integer) 1行1パーティションで与えられたパーティション化テーブルあるいはパーティション化インデックスのパーティションツリー内のテーブルあるいはインデックスの情報を表示します。提供される情報にはパーティションOID、その直接の親のOID、パーティションが葉かどうかを示す真偽値、階層内のレベルを表す整数が含まれます。レベル値は与えられたテーブルあるいはインデックスがパーティションツリーの根としての役割を持つことを表す0で始まり、1ならその直下のパーティション、2ならそのまた下のパーティションなどとなります。リレーションが存在しない、あるいはパーティションでない、もしくはパーティション化テーブルでない場合は行を返しません。
pg_partition_ancestors (regclass) → setof regclass パーティション自身を含む、与えられたパーティションの先祖リレーションを列挙します。リレーションが存在しない、あるいはパーティションでない、もしくはパーティション化テーブルでない場合は行を返しません。
pg_partition_root (regclass) → regclass

関数 説明
与えられたリレーションが所属するパーティションツリーの最上位の親を返します。リレーションが存在しない、あるいはパーティションでない、もしくはパーティション化テーブルでない場合はNULLを返します。

たとえばパーティション化テーブルmeasurementに含まれるデータの全体サイズを確認するには、次の問い合わせが利用できます。

```
SELECT pg_size_pretty(sum(pg_relation_size(relid))) AS total_size
  FROM pg_partition_tree('measurement');
```

9.27.8. インデックス保守関数

表 9.94にインデックスの保守タスクに使用可能な関数を示します。(これらの保守業務は通常自動バキュームが自動的に行うことに注意してください。これらの関数は特別な場合にのみ必要になります。) これらの関数はリカバリ中は実行できません。これらの関数の使用はスーパーユーザと対象のインデックスの所有者に限定されます。

表9.94 Index Maintenance Functions

Function 説明
brin_summarize_new_values (index regclass) → integer 指定されたBRINインデックスを検査してベーステーブル内のインデックスによって現在要約されていないページ範囲を探します。そのような範囲があれば、テーブルのページをスキャンして新しい要約インデックスタプルを作成します。インデックスに挿入された新しいページ範囲要約の数を返します。
brin_summarize_range (index regclass, blockNumber bigint) → integer 指定のブロックを含むページ範囲が、まだ要約されていなければ、要約します。これはbrin_summarize_new_valuesと似ていますが、指定されたテーブルブロック番号のみを対象としてページ範囲を処理するところだけが異なります。
brin_desummarize_range (index regclass, blockNumber bigint) → void 指定のブロックを含むページ範囲が要約されていれば、それを含むページ範囲を要約するBRINインデックスタプルを削除します。
gin_clean_pending_list (index regclass) → bigint 指定GINインデックスの「処理待ち」リストのエントリをメインのインデックスデータ構造に一括で移動します。処理待ちリストから削除されたページ数を返します。fastupdateオプションが無効で作成されたGINインデックスが引数なら、削除は起こらず結果がゼロになります。そのインデックスには処理待ちリストがないからです。処理待ちリストとfastupdateオプションの詳細については66.4.1と66.5をご覧ください。

9.27.9. 汎用ファイルアクセス関数

表 9.95で示されている関数はサーバをホスティングしているマシン上のファイルに対し、ネイティブのアクセスを提供します。ユーザがpg_read_server_filesロールを与えられていない限り、データベースクラスタディレクトリとlog_directoryに存在するファイルのみがアクセス可能です。クラスタディレクトリ内のファイ

ルに対して相対パスを、そしてログファイルに対してはlog_directory構成設定に一致するパスを使用してください。

ユーザに対してEXECUTE権限をpg_read_file()あるいは関連する関数に与えることは、サーバの上データベースサーバプロセスが読めるすべてのファイルを読めるようにすることになることに注意してください。これらの関数はデータベース内のすべての権限チェックをすり抜けます。このことは、たとえばそのようなアクセス権を持つユーザは、認証情報が格納されたpg_authidテーブルの中身を読むことができますし、同様にデータベース内のすべてのテーブルデータを読むことができるということを意味します。ですからこれらの関数にアクセス権限を与えるのは慎重に考慮したほうが良いでしょう。

これらの関数の一部はオプションでmissing_okパラメータをとり、ファイルまたはディレクトリが存在しない場合の動作を指定できます。trueの場合、関数はNULLを返すか、適切な場合には空の結果集合を返します。falseの場合はエラーが発生します。デフォルトはfalseです。

表9.95 汎用ファイルアクセス関数

関数 説明
pg_ls_dir (dirname text [, missing_ok boolean, include_dot_dirs boolean]) → setof text 指定されたディレクトリ内のすべてのファイル（およびディレクトリと他の特殊ファイル）の名前を返します。include_dot_dirsは「.」と「..」が結果集合に含まれるかどうかを指定します。デフォルト(false)ではそれらを除外しますが、それらを含めると、missing_okがtrueの場合は、空のディレクトリと存在しないディレクトリを区別するために役立つでしょう。 デフォルトではこの関数の実行はスーパーユーザに限定されますが、他のユーザに関数を実行するEXECUTE権限を与えることができます。
pg_ls_logdir () → setof record (name text, size bigint, modification timestamp with time zone) サーバのログディレクトリ内の各通常ファイルについて、名前、サイズ、最終更新時刻（mtime）を返します。ドットで始まるファイル名、ディレクトリ名その他の特殊なファイルは含まれません。 デフォルトではこの関数の実行はスーパーユーザとpg_monitorロールのメンバーに限定されますが、他のユーザに関数を実行するEXECUTE権限を与えることができます。
pg_ls_waldir () → setof record (name text, size bigint, modification timestamp with time zone) 先行書き込みログ（WAL）ディレクトリ内の各ファイルについて、名前、サイズ、最終更新時刻（mtime）を返します。ドットで始まるファイル名、ディレクトリ名その他の特殊なファイルは含まれません。 デフォルトではこの関数の実行はスーパーユーザとpg_monitorロールに限定されますが、他のユーザに関数を実行するEXECUTE権限を与えることができます。
pg_ls_archive_statusdir () → setof record (name text, size bigint, modification timestamp with time zone) WALアーカイブステータスディレクトリ(pg_wal/archive_status)内の各ファイルについて、名前、サイズ、最終更新時刻（mtime）を返します。ドットで始まるファイル名、ディレクトリ名その他の特殊なファイルは含まれません。 デフォルトではこの関数の実行はスーパーユーザとpg_monitorロールに限定されますが、他のユーザに関数を実行するEXECUTE権限を与えることができます。
pg_ls_tmpdir ([tablespace oid]) → setof record (name text, size bigint, modification timestamp with time zone) 指定されたtablespace内の一時ファイルディレクトリ内の各ファイルについて、名前、サイズ、最終更新時刻（mtime）を返します。tablespaceが与えられなければpg_defaultテーブル空間が検査されます。ドットで始まるファイル名、ディレクトリ名その他の特殊なファイルは含まれません。 デフォルトではこの関数の実行はスーパーユーザとpg_monitorロールのメンバーに限定されますが、他のユーザに関数を実行するEXECUTE権限を与えることができます。
pg_read_file (filename text [, offset bigint, length bigint [, missing_ok boolean]]) → text

関数 説明
与えられたoffsetから始まり、最大lengthバイト（先にファイルの終りに到達すればこれより少なくなります）テキストファイルの全部または一部分を返します。offsetが負の場合にはファイルの終りから数えた位置から読み出します。offsetとlengthが省略された場合、ファイル全体が返されます。ファイルから読み込まれたバイトは、そのサーバの符号化方式での文字列として解釈されます。読み込んだバイト列がその符号化方式において有効でない場合にはエラーが投げられます。 デフォルトではこの関数の実行はスーパーユーザに限定されますが、他のユーザに関数を実行するEXECUTE権限を与えることができます。
pg_read_binary_file (filename text [, offset bigint, length bigint [, missing_ok boolean]]) → bytea ファイルの一部あるいは全部を返します。結果がtextではなくてbytea値となり、任意のバイナリデータを読み出すことができることを除き、pg_read_fileと同じです。従って符号化方式の検査は行われません。 デフォルトではこの関数の実行はスーパーユーザに限定されますが、他のユーザに関数を実行するEXECUTE権限を与えることができます。 convert_from関数と組み合わせることで、この関数を、指定した符号化方式でファイルを読み込んでデータベース符号化方式に変換することができます。 `SELECT convert_from(pg_read_binary_file('file_in_utf8.txt'), 'UTF8');`
pg_stat_file (filename text [, missing_ok boolean]) → record (size bigint, access timestamp with time zone, modification timestamp with time zone, change timestamp with time zone, creation timestamp with time zone, isdir boolean) ファイル容量、最終アクセス時刻、最終更新時刻、最後にファイルステータスを変更した時刻（これはUnixプラットフォームのみ）、ファイル作成時刻（Windowsのみ）および、ディレクトリかどうかを示すフラグを返します。 デフォルトではこの関数の実行はスーパーユーザに限定されますが、他のユーザに関数を実行するEXECUTE権限を与えることができます。

9.27.10. 勧告的ロック用関数

表 9.96に示す関数は勧告的ロックを管理します。これらの関数の適切な使用方法についての詳細は、13.3.5を参照してください。

これらの関数はすべて単一の64ビットキー値か2つの32ビットキー値（これらのキー空間は重なり合わないことに注意してください）で識別されるアプリケーション定義のリソースをロックするために使うことを意図しています。他のセッションがすでに同じリソース識別子とコンフリクトするロックを保持していたら、関数はリソースが利用可能になるまで待つか、その関数にとって適切ならばfalseを結果として返します。ロックは共有はあるいは排他のどちらも可能です。共有ロックは同じリソースに対して他の共有ロックとコンフリクトしません。排他ロックとだけコンフリクトします。ロックはセッションレベル（ロックは解放されるまで保持するかセッションの終了まで保持します）あるいはトランザクションレベル（ロックは現在のトランザクションが終了するまで保持します。手動で解放する方法はありません）で取得できます。複数のセッションレベルロック要求は積み重ねられます。これにより、同じリソース識別子が3回ロックされると、セッション終了前にそのリソースを解放するアンロック要求が3回発行されなければならなくなります。

表9.96 勧告的ロック用関数

関数 説明
pg_advisory_lock (key bigint) → void

関数 説明
pg_advisory_lock (key1 integer, key2 integer) → void 必要なら待ってからセッションレベルの排他勧告的ロックを獲得します。
pg_advisory_lock_shared (key bigint) → void pg_advisory_lock_shared (key1 integer, key2 integer) → void 必要なら待ってからセッションレベルの共有勧告的ロックを獲得します。
pg_advisory_unlock (key bigint) → boolean pg_advisory_unlock (key1 integer, key2 integer) → boolean 事前に獲得したセッションレベルの排他的勧告ロックを解放します。ロックの解放に成功した場合、trueを返します。ロックを保持していない場合、falseを返し、さらに、SQL警告がサーバから報告されます。
pg_advisory_unlock_all () → void 現在のセッションで保持するセッションレベルの勧告的ロックをすべて解放します。(この関数は、クライアントとの接続が不用意に切れた場合でも、セッション終了時に暗黙的に呼び出されます。)
pg_advisory_unlock_shared (key bigint) → boolean pg_advisory_unlock_shared (key1 integer, key2 integer) → boolean 事前に獲得したセッションレベルの共有勧告的ロックを解放します。ロックの解放に成功した場合、trueを返します。ロックを保持していない場合、falseを返し、さらに、SQL警告がサーバから報告されます。
pg_advisory_xact_lock (key bigint) → void pg_advisory_xact_lock (key1 integer, key2 integer) → void 必要なら待ってからトランザクションレベルの排他勧告的ロックを獲得します。
pg_advisory_xact_lock_shared (key bigint) → void pg_advisory_xact_lock_shared (key1 integer, key2 integer) → void 必要なら待ってからトランザクションレベルの共有勧告的ロックを獲得します。
pg_try_advisory_lock (key bigint) → boolean pg_try_advisory_lock (key1 integer, key2 integer) → boolean 可能ならセッションレベルの排他勧告的ロックを獲得します。これは直ちにロックを取得してtrueを返すか、直ちにロックを取得できない場合は待たずにfalseを返します。
pg_try_advisory_lock_shared (key bigint) → boolean pg_try_advisory_lock_shared (key1 integer, key2 integer) → boolean 可能ならセッションレベルの共有勧告的ロックを獲得します。これは直ちにロックを取得してtrueを返すか、直ちにロックを取得できない場合は待たずにfalseを返します。
pg_try_advisory_xact_lock (key bigint) → boolean pg_try_advisory_xact_lock (key1 integer, key2 integer) → boolean 可能ならトランザクションレベルの排他勧告的ロックを獲得します。これは直ちにロックを取得してtrueを返すか、直ちにロックを取得できない場合は待たずにfalseを返します。
pg_try_advisory_xact_lock_shared (key bigint) → boolean pg_try_advisory_xact_lock_shared (key1 integer, key2 integer) → boolean 可能ならトランザクションレベルの共有勧告的ロックを獲得します。これは直ちにロックを取得してtrueを返すか、直ちにロックを取得できない場合は待たずにfalseを返します。

9.28. トリガ関数

多くの場合トリガにはユーザ記述のトリガ関数が必要になりますが、PostgreSQLはユーザ定義トリガで直接使用できる小数の組み込みの取り化関数を提供しています。これらは表 9.97にまとめられています。(追加の組み込みトリガ関数があり、外部キー制約と遅延インデックス制約を実装しています。ユーザがこれらを直接必要とすることはないので、ここには記述されていません。)

トリガ作成についてより詳細はCREATE TRIGGERを参照ください。

表9.97 組み込みトリガ関数

関数 説明 使用例
suppress_redundant_updates_trigger ( ) → trigger do-nothing更新操作を抑止します。詳細は以下を参照してください。 CREATE TRIGGER ... suppress_redundant_updates_trigger()
tsvector_update_trigger ( ) → trigger 関連付けされた平文文書列から自動的にtsvector列を更新します。使用するテキスト検索設定はトリガ引数で指定します。詳細は12.4.3をご覧ください。 CREATE TRIGGER ... tsvector_update_trigger(tsvcol, 'pg_catalog.swedish', title, body)
tsvector_update_trigger_column ( ) → trigger 関連付けされた平文文書列から自動的にtsvector列を更新します。使用するテキスト検索設定はテーブルのregconfig列が用いられます。詳細は12.4.3をご覧ください。 CREATE TRIGGER ... tsvector_update_trigger_column(tsvcol, tsconfigcol, title, body)

行レベルBEFORE UPDATEトリガとしてsuppress_redundant_updates_trigger関数が適用されると、実際には行の中でデータを変更しない更新が行われるのを防ぎます。これはデータが変更されるかどうかに関わらず、物理的に行の更新を行う通常の振る舞いを置き換えます。(この通常の動作は、検査を必要としないため更新をより迅速に行い、場合によっては便利です。)

理想的には、通常実際レコード内のデータを変更しない更新の実行を避けるべきです。冗長な更新により、特に変更対象の多くのインデックスが存在する場合、無視できない不要な時間にかかるコストが発生することがあります。また、最後にはバキュームしなければならなくなる不要行が場所を取ることになります。しかし、こうした状況をクライアント側で判定することは常に簡単ではありません。また、可能であったとしても、それを検知するための式の記述はエラーを招きがちです。他の方法として、suppress_redundant_updates_triggerを使用することがあります。これはデータを変更しない更新を飛ばします。しかしこの関数は注意して使用しなければなりません。このトリガはレコードごとに小さな、しかし僅かではない時間がかかります。このため、更新が影響するレコードのほとんどが実際に変更された場合、このトリガは平均すると更新の実行を低速にします。

suppress_redundant_updates_trigger関数は以下のようにテーブルに追加できます。

```
CREATE TRIGGER z_min_update
BEFORE UPDATE ON tablename
FOR EACH ROW EXECUTE FUNCTION suppress_redundant_updates_trigger();
```

ほとんどの場合、行を変更するかも知れない他のトリガを置き換えないために、それぞれの行に対しこのトリガを最後に起動させる必要があります。トリガは名前順に起動されることを判っているとして、テーブル上に存在する可能性のある他のトリガの名前の後に続くようトリガ名を選択できます。(それで例中に「z」接頭辞があります。)

9.29. イベントトリガ関数

PostgreSQLはイベントトリガについての情報を取得するために以下のヘルパ関数を提供しています。

イベントトリガについての詳細は第39章を参照して下さい。

9.29.1. コマンド側での変更を捕らえる

```
pg_event_trigger_ddl_commands () → setof record
```

pg_event_trigger_ddl_commandsがddl_command_endイベントトリガーに付与された関数から起動されると、各ユーザの操作によって実行されたDDLコマンドの一覧を返します。それ以外の環境から呼び出された場合はエラーが発生します。pg_event_trigger_ddl_commandsは、実行された基となるコマンドのそれぞれについて1行を返します。1つのSQL文として実行されるいくつかのコマンドに対して、複数の行が返されることもあります。この関数は以下の列を返します。

名前	型	説明
classid	oid	オブジェクトが属するカタログのOID
objid	oid	カタログ内のオブジェクトのOID
objsubid	integer	Sub-object ID (e.g., attribute number for a column)
command_tag	text	コマンドのタグ
object_type	text	オブジェクトの型
schema_name	text	オブジェクトが属するスキーマの名前(あれば)。なければNULL。引用符づけされない。
object_identity	text	オブジェクトの識別をテキスト表現したもので、スキーマ修飾される。識別内に存在する各識別子は、必要なら引用符で括られる。
in_extension	boolean	コマンドが拡張のスクリプトの一部なら真
command	pg_ddl_command	コマンドを内部形式で完全に表現したもの。これを直接出力することはできないが、コマンドについて他

名前	型	説明
		の情報を得るために、他の関数に渡すことができる。

9.29.2. DDLコマンドで削除されたオブジェクトの処理

```
pg_event_trigger_dropped_objects () → setof record
```

関数pg_event_trigger_dropped_objectsは、それが呼ばれたsql_dropイベントのコマンドにより削除された全てのオブジェクトのリストを返します。それ以外の状況で呼ばれた場合、エラーが生じます。この関数は以下の列を返します。

名前	型	説明
classid	oid	オブジェクトが所属するカタログのOID
objid	oid	カタログ内に所有するオブジェクトのOID
objsubid	integer	Sub-object ID (e.g., attribute number for a column)
original	boolean	これが削除のルートオブジェクトの一つなら真
normal	boolean	このオブジェクトへと至る依存関係グラフで、通常の依存があるなら真
is_temporary	boolean	オブジェクトが一時オブジェクトであったなら真
object_type	text	オブジェクトの型
schema_name	text	オブジェクトが所属しているスキーマの名前(あれば)。なければNULL。引用符づけされない。
object_name	text	スキーマと名前の組み合わせがオブジェクトに対する一意の識別子として使用可能な場合はオブジェクトの名前。そうでないときはNULL。引用符は適用されず、名前は決してスキーマで修飾されない。
object_identity	text	オブジェクト識別のテキスト表現で、スキーマ修飾される。識別内に存在する各識別子は必要であれば引用符で括られる。
address_names	text[]	object_typeおよびaddress_argsと一緒

名前	型	説明
		にpg_get_object_address()で使うことで、同じ種類で全く同じ名前のオブジェクトを含むリモートサーバ内のオブジェクトアドレスを再作成できる配列。
address_args	text[]	address_namesの補足。

関数pg_event_trigger_dropped_objectsは以下のようにイベントトリガとして使用可能です。

```
CREATE FUNCTION test_event_trigger_for_drops()
        RETURNS event_trigger LANGUAGE plpgsql AS $$
DECLARE
    obj record;
BEGIN
    FOR obj IN SELECT * FROM pg_event_trigger_dropped_objects()
    LOOP
        RAISE NOTICE '% dropped object: % %.% %',
                     tg_tag,
                     obj.object_type,
                     obj.schema_name,
                     obj.object_name,
                     obj.object_identity;
    END LOOP;
END;
$$;
CREATE EVENT TRIGGER test_event_trigger_for_drops
   ON sql_drop
   EXECUTE FUNCTION test_event_trigger_for_drops();
```

9.29.3. テーブル書き換えイベントの処理

表 9.98に示す関数は、table_rewriteイベントが呼び出されたばかりのテーブルについての情報を提供します。それ以外の状況で呼び出された場合はエラーが発生します。

表9.98 テーブル書き換え情報関数

関数 説明
pg_event_trigger_table_rewrite_oid () → oid 書き換えようとされているテーブルのOIDを返します。
pg_event_trigger_table_rewrite_reason () → integer 書き換えの理由を説明する理由コードを返します。コードの正確な意味はリリースに依存します。

これらの関数はイベントトリガ中で次のように使用できます。

```
CREATE FUNCTION test_event_trigger_table_rewrite_oid()
 RETURNS event_trigger
 LANGUAGE plpgsql AS
$$
BEGIN
  RAISE NOTICE 'rewriting table % for reason %',
                pg_event_trigger_table_rewrite_oid()::regclass,
                pg_event_trigger_table_rewrite_reason();
END;
$$;

CREATE EVENT TRIGGER test_table_rewrite_oid
                  ON table_rewrite
   EXECUTE FUNCTION test_event_trigger_table_rewrite_oid();
```

9.30. 統計情報関数

PostgreSQLはCREATE STATISTICSコマンドを使って定義した複雑な統計を調べる関数を提供しています。

9.30.1. MCVリストの検査

```
pg_mcv_list_items ( pg_mcv_list ) → setof record
```

pg_mcv_list_itemsは複数列MCVリストに格納されたすべての項目を列挙します。この関数は次の列を返します。

名前	型	説明
index	integer	MCVリスト内の項目のインデックス
values	text[]	MCV項目に格納された値
nulls	boolean[]	NULL値を識別するフラグ
frequency	double precision	このMCV項目の頻度
base_frequency	double precision	このMCV項目のベース頻度

pg_mcv_list_items関数は次のように使用することができます。

```
SELECT m.* FROM pg_statistic_ext join pg_statistic_ext_data on (oid = stxoid),
                pg_mcv_list_items(stxdmcv) m WHERE stxname = 'stts';
```

pg_mcv_list型の値はpg_statistic_ext_data.stxdmcv列からのみ得られます。

第10章 型変換

意図的かどうかにかかわらず、SQLの問い合わせでは1つの式の中に異なる型を混ぜ合わせた式を持つことができます。PostgreSQLは、異なる型が混在する式の評価に関して幅広い能力を持っています。

多くの場合、ユーザは型変換機構の詳細を理解する必要はありません。しかし、PostgreSQLによって暗黙的に行われる変換は問い合わせの結果に影響を及ぼします。必要に応じて、明示的な型変換を用いて結果を目的とするものに合わせることができます。

本章では、PostgreSQLの型変換機構とその規定について紹介します。特定のデータ型、使用できる関数と演算子についての情報については、第8章と第9章の関連する節を参照してください。

10.1. 概要

SQLは強く型付けされた言語です。つまり、各データ項目は、その動作と許される使用方法を決定するデータ型を所有しています。PostgreSQLには、他のSQLの実装よりもより一般的で柔軟性のある、拡張可能な型システムがあります。このために、PostgreSQLでのほとんどの型変換の動作は、特定の目的について勝手に作り上げられることなく一般的な規則で管理されています。これにより、ユーザ定義型についても型の混在する式を使用できます。

PostgreSQLのスキャナ/パーサは字句要素を、整数、非整数値、文字列、識別子、キーワードという5個の基礎カテゴリに分解します。ほとんどの非数値型定数は、まず文字列にクラス分けされます。SQL言語定義では、文字列で型の名前を指定することを許していて、パーサが正しい手順に沿って処理を始められるようにPostgreSQLも採用しています。例えば、以下のような問い合わせを考えてみましょう。

```
SELECT text 'Origin' AS "label", point '(0,0)' AS "value";

 label  | value
--------+-------
 Origin | (0,0)
(1 row)
```

この問い合わせは、textとpointという2つの型を指定したリテラル定数を持ちます。文字列リテラルに型が指定されていない場合、後述するように、後の段階で解決されるようにとりあえず場所を確保するための型であるunknownが割り当てられます。

PostgreSQLのパーサには、個別の型変換規則が必要な4つの基礎的なSQL構成要素があります。

関数呼び出し

> PostgreSQL型システムの大部分は、高度な関数群によって構築されています。関数は複数の引数を取ることができます。PostgreSQLでは関数のオーバーロードが可能ですので、関数名だけでは呼び出すべき関数を一意に識別できません。パーサは、提供される引数のデータ型に基づいて、正しい関数を選択しなければなりません。

演算子

PostgreSQLでは、(引数が2つの)二項演算子と同様に、(引数が1つの)前置、後置単項演算子を持つ式が使用できます。関数と同様、演算子もオーバーロード可能ですので、正しい演算子を選択する時に同じ問題が存在します。

値の格納

SQLのINSERTとUPDATE文は式の結果をテーブルの中に格納します。文内の式は対象となる列の型に一致する、または、変換できるものである必要があります。

UNION、CASE、および関連する構文

UNIONを構成するSELECT文からの選択結果は全て、ある1つの列集合として現れなければいけませんので、各SELECT句の結果型は統一された集合に一致し変換できる必要があります。同様に、CASE構文が全体として既知の出力型を持つようになるために、CASE構文の結果式は共通の型に変換される必要があります。ARRAY[]のような他のいくつかの構文やGREATEST関数、LEAST関数は、同様に副式に対して共通の型の決定を要求します。

システムカタログには、どのデータ型の間にどのような変換、すなわち*キャスト*があるのか、また、その変換をどのように実行するのかに関する情報を格納します。ユーザはCREATE CASTコマンドを使用してキャストを追加できます。(これは通常新しいデータ型を定義する時にまとめて行われます。組み込み型間のキャスト集合は注意深く作成されており、変更しないことが最善です。)

暗黙のキャストを持つデータ型間の処理において、適切なキャスト処理のより良い決定を行えるようパーサは追加の自律機構を備えています。データ型は、boolean、numeric、string、bitstring、datetime、timespan、geometric、network、及びユーザ定義を含むいくつかの基本的な*型カテゴリ*に分けられます。(一覧は表 51.63を参照してください。ですが、独自の型カテゴリを作成するのも可能なことに注意して下さい。) 各カテゴリには、候補となる型の選択があった場合に、優先される1つ以上の*優先される型*がある場合があります。優先される型と利用可能な暗黙のキャストを注意して選択すれば、曖昧な式(複数の解析結果候補を持つもの)が有効な方法で解決されることを保証することが可能です。

全ての型変換規則は次のようないくつかの基本的な考え方に基づいて設計されています。

- 暗黙的な変換は、意外な、あるいは予想できない結果を決して生成させてはなりません。
- 暗黙的な型変換を必要としない問い合わせの場合、パーサやエクゼキュータに余計なオーバーヘッドがあるべきではありません。つまり、問い合わせ文がきちんとまとめられ、型が既に一致するものになっていれば、パーサ内で余計な時間を費やさず、また、問い合わせに不要な暗黙的な型変換関数が使用されないように、問い合わせは処理されるべきです。
- さらに、もし問い合わせが関数のために暗黙的な変換を通常要求しており、そして、ユーザが正しい引数型を持つ関数を新しく定義した場合、パーサはこの新しい関数を使うべきであり、もはや古い関数を使うために暗黙的な変換を行わないようすべきです。

10.2. 演算子

演算式に参照される特定の演算子は、以下の手順を用いて決定されます。関連する演算子の優先順位によりどの下位式をどの演算子の入力と見なすかが決定されますので、この手順はこの優先順位により間接的な影響を受けることに注意して下さい。詳細は4.1.6を参照してください。

演算子における型の解決

1. pg_operatorシステムカタログから、調査の対象とする演算子を選択します。スキーマ修飾がされていない演算子名が使用される場合（通常の場合）、現行の検索パスで可視になっていて、同一の名前と引数の数を持つ演算子が調査対象であるとみなされます（5.9.3を参照してください）。修飾された演算子名が与えられている場合、指定されたスキーマの演算子のみが調査対象とみなされます。

   - (Optional) 検索パスで引数のデータ型が同じである複数の演算子を検出した場合、そのパスで最初に検出された演算子のみを調査対象とみなします。引数のデータ型が異なる演算子は、検索パス内の位置に関係なく、同じように調べられます。

2. 正確に入力引数型を受け付ける演算子があるかどうか検査します。該当する演算子があれば（調査される演算子の集合内で正確に一致するものは1つしかあり得ません）、それを使用します。正確に一致するものがない場合、信用できないユーザにオブジェクトの作成を許可しているスキーマで見つかる演算子を、(典型的なものではないですが)修飾された名前で [1] 呼び出す時にセキュリティの危険が発生します。そのような状況では、強制的に正確に一致するように引数をキャストしてください。

   a. (Optional) 二項演算子の1つの引数がunknown型であった場合、この検査のもう片方の引数と同一の型であると仮定します。2つのunknown入力、もしくはunknown入力を伴う単項演算子が呼び出された場合、この段階では対を見つけることはありません。

   b. (Optional) 二項演算子の1つの引数がunknown型であり、もう1つがドメイン型の場合、次に両側でドメインの基本型を厳密に受け付ける演算子があるかを確認します。

3. 最もよく合うものを検索します。

   a. 演算子の候補のうち、入力値のデータ型が一致せず、また、（暗黙的な変換を使用して）一致するように変換できないものを破棄します。unknownリテラルは、上記の目的で何にでも変換可能とみなされます。1つの候補しか残らない場合、それを使います。それ以外の場合は次の段階に進みます。

   b. 入力引数のいずれかがドメイン型であれば、以降の段階すべてでドメインの基本型であるかのように扱います。これにより、曖昧な演算子を解決するのを目的としてその基本型であるかのようにドメインが振る舞うことが確実になります。

   c. 全ての候補を検索し、入力型に最も正確に合うものを残します。正確に合うものが何もなければ全ての候補を残します。1つの候補しか残らない場合、それを使います。それ以外の場合は次の段階に進みます。

   d. 全ての候補を検索し、型変換が必要とされる所で（入力データ型カテゴリの）優先される型を受け付けるものを残します。優先される型を受け付けるものが何もなければ全ての候補を残します。1つの候補しか残らない場合、それを使います。それ以外の場合は次の段階に進みます。

[1] 信用できないユーザにオブジェクトの作成を許可するスキーマを含む検索パスは、安全なスキーマ使用パターンではありませんので、スキーマで修飾されていない名前では危険は起こりません。

e. 入力引数でunknownのものがあった場合、それらの残った候補に引数位置で受け入れられる型カテゴリを検査します。各位置において、候補がstringカテゴリを受け付ける場合は、そのカテゴリを選択します（unknown 型のリテラルは文字列のようなものですので、この文字列への重み付けは適切です）。そうでなければ、もし残った全ての候補が同じ型カテゴリを受け入れる場合はそのカテゴリを選択します。そうでもなければ、さらに手掛かりがなければ正しい選択が演繹されることができませんので、失敗となります。ここで、選択された型カテゴリを受け付けない演算子候補は破棄されます。さらに、それらカテゴリ内の優先される型を受け付ける候補が1つでもある場合、その引数の優先されない型を受け付ける候補は破棄されます。これらの検査をどれも通らなかったら全ての候補を残します。1つの候補しか残らない場合、それを使います。それ以外の場合は次の段階に進みます。

f. もしunknownと既知の型の引数の両方があり、そして全ての既知の型の引数が同じ型を持っていた場合、unknown引数も同じ型であると仮定し、どの候補がunknown引数の位置にある型を受け付けることができるかを検査します。正確に1つの候補がこの検査を通過した場合、それを使います。それ以外は失敗します。

以下に例を示します。

例10.1 平方根演算子の型解決

平方根演算子として、double precisionを引数とするものが標準カタログ内に1つのみ定義されています（|/を前に付けます）。スキャナは、以下の問い合わせ式の引数にまずinteger型を割り当てます。

```
SELECT |/ 40 AS "square root of 40";
 square root of 40
-------------------
 6.324555320336759
(1 row)
```

パーサはオペランドを型変換し、問い合わせは以下と等価になります。

```
SELECT |/ CAST(40 AS double precision) AS "square root of 40";
```

例10.2 文字列連結演算子の型解決

文字列類似構文は、文字列の作業の他、複雑な拡張型の作業にも使用されます。型の指定がない文字列は、類似演算子候補に一致します。

例えば、以下は指定がない引数が1つあります。

```
SELECT text 'abc' || 'def' AS "text and unknown";

 text and unknown
------------------
 abcdef
(1 row)
```

この場合、パーサは両引数でtextを取る演算子があるかどうかを検索します。この演算子は存在しますので、第二引数はtext型として解釈されるものと仮定されます。

以下は型の指定がない2つの値の連結です。

```
SELECT 'abc' || 'def' AS "unspecified";

 unspecified
-------------
 abcdef
(1 row)
```

この場合、問い合わせ内に型が指定されていませんので、どの型を使用すべきかについての初期の指針がありません。ですから、パーサは全ての演算子候補を検索し、文字列カテゴリとビット列カテゴリ入力を受け付ける候補を見つけます。使用できる場合は文字列カテゴリが優先されますので、文字列カテゴリが選択され、それから文字列に対して優先される型であるtextが、不明のリテラルを解決する型として使用されます。

例10.3 絶対値と否定演算子の型解決

PostgreSQLの演算子カタログには、前置演算子@用に複数の項目があります。これは全て各種数値データ型に対する絶対値計算を実装するものです。その1つは、数値カテゴリの優先される型であるfloat8型用の項目です。したがって、PostgreSQLは、unknownの入力があった場合にこれを使用します。

```
SELECT @ '-4.5' AS "abs";
 abs
-----
 4.5
(1 row)
```

ここでシステムは、選択した演算子を適用する前に、unknown型のリテラルをfloat8へ暗黙的に型変換します。以下のようにfloat8が使用され、他の型が使用されていないことを検証することができます。

```
SELECT @ '-4.5e500' AS "abs";

ERROR:  "-4.5e500" is out of range for type double precision
```

一方、前置演算子~(ビット否定)は、整数データ型のみで定義され、float8用は定義されていません。ですから、~における上と同様の場合では、以下のような結果になります。

```
SELECT ~ '20' AS "negation";

ERROR:  operator is not unique: ~ "unknown"
HINT:  Could not choose a best candidate operator. You might need to add
explicit type casts.
```

これは、システムが、複数の~演算子候補のうちどれが優先されるかを決定することができなかったため発生します。明示的なキャストを使用することで補助することができます。

```
SELECT ~ CAST('20' AS int8) AS "negation";

 negation
----------
      -21
(1 row)
```

例10.4 配列包含演算子の型解決

一方は既知でありもう一方は未知である入力を伴った演算子の解決のもう一つの例です。

```
SELECT array[1,2] <@ '{1,2,3}' as "is subset";

 is subset
-----------
 t
(1 row)
```

PostgreSQLの演算子カタログは、<@中置演算子のためのいくつかのエントリを持っていますが、数値型配列を左側に受け付けることができるのは配列包含(anyarray <@ anyarray)と範囲包含(anyelement <@ anyrange)の2つのみです。これらの多様な擬似データ型(8.21を参照)は優先されると見なされないため、このような方法ではパーサは曖昧さを解決することができません。しかし、ステップ 3.fでは、未知の型のリテラルを別の入力と同じ型であると仮定するために数値配列とみなします。今のところ2つのうち一つの演算子だけがマッチできるため、配列包含が選択されます。(範囲包含が選択された場合、演算子の右側にある文字列は正しい範囲型のリテラルではないため、エラーとなるでしょう。)

例10.5 ドメイン型の独自の演算子

利用者は時々ドメイン型にのみ適用される演算子を宣言しようとします。これは可能ですが、思ったほど便利ではありません。演算子の解決規則がドメイン基本型に適用される演算子を選ぶように設計されているからです。例として、以下を考えてください。

```
CREATE DOMAIN mytext AS text CHECK(...);
CREATE FUNCTION mytext_eq_text (mytext, text) RETURNS boolean AS ...;
CREATE OPERATOR = (procedure=mytext_eq_text, leftarg=mytext, rightarg=text);
CREATE TABLE mytable (val mytext);

SELECT * FROM mytable WHERE val = 'foo';
```

この問い合わせは独自の演算子を使いません．パーサはまずmytext = mytext演算子(ステップ 2.a)があるか確認しますが、ありません。次にドメイン基本型textを考慮してtext = text演算子(ステップ 2.b)があるか確認すると、あります。そのためunknown型はtextとして解決され、text = text演算子が使われます。独自の演算子を使う唯一の方法は、「正確な一致」規則に従ってmytext = text演算子がすぐに見つかるように、リテラルを明示的にキャストすることだけです。

```
SELECT * FROM mytable WHERE val = text 'foo';
```

もし、「最善の一致」規則に達した場合、ドメイン型の演算子を積極的に差別します。そうでなければ、そのような演算子は非常に多くの「曖昧な演算子」の失敗を引き起こします。キャストの規則はドメインをその基本型からもしくは基本型へキャスト可能と考え、ドメイン演算子は基本型の似たような名前の演算子とすべて同じ状況で利用できると考えられるからです。

10.3. 関数

関数呼び出しによって参照される特定の関数は、以下の手順に従って解決されます。

関数における型の解決

1. pg_procシステムカタログから、調査の対象とする関数を選択します。スキーマ修飾がされていない関数名が使用される場合、現行の検索パスで可視になっていて、同一の名前と引数の数を持つ関数が調査対象であるとみなされます（5.9.3を参照してください）。修飾された関数名が与えられている場合、指定されたスキーマの関数のみが調査対象とみなされます。

   a. (Optional) 検索パスで、引数のデータ型が同じである複数の関数を検出した場合、そのパスで最初に検出された関数のみを調査対象とみなします。引数のデータ型が異なる関数は、検索パス内の位置に関係なく、同じように調べられます。

   b. (Optional) もし関数がVARIADIC型の配列パラメータを伴って定義されており、そしてVARIADICキーワードを用いずに呼ばれた場合は、呼び出しに適合するよう、一つかそれ以上の要素の型に配列のパラメータを置き換えた形で扱われます。このような拡張後は、その関数は実際の引数の型を持つので、他の非可変長の引数を持つ関数と同一になるかもしれません。この場合、検索パスで先に見つかった関数が使われます。また、同じスキーマに2つの関数が見つかった場合は非可変長の関数が優先されます。

      信用できないユーザにオブジェクトの作成を許可しているスキーマで見つかる可変長引数の関数を、修飾された名前で[2]呼び出す時にセキュリティの危険が発生します。悪意のあるユーザは、支配権を奪い、あたかもあなたが実行したかのように任意のSQL関数を実行できます。VARIADICキーワードを持つ呼び出しを代わりに使ってください。そうすればこの危険は避けられます。VARIADIC "any"パラメータにデータを入れての呼び出しには、しばしば同等のVARIADICキーワード含む定式化がありません。この呼び出しを安全に行なうには、関数のスキーマは信用できるユーザだけがオブジェクトを作成できるようにしなければなりません。

   c. (Optional) パラメータにデフォルト値を持つ関数は、デフォルト指定可能なパラメータ位置のうち、0以上が省略されたどのような呼び出しに対しても適合すると見なされます。もし呼び出し時にこのような関数が2つ以上適合した場合、検索パスで先に見つかったものが使用されます。もし、デフォルト指定のない位置に同じパラメータ型を持つ関数(もしそれらが異なるデフォルト指定のあるパラメータのセットを持っていればあり得ます)が同じスキーマに2つ以上あった時は、システムはどの関数を使うべきか決定できず、呼び出しにより適合するものが見つからなければ「ambiguous function call」エラーが結果として返るでしょう。

      信用できないユーザにオブジェクトの作成を許可しているスキーマで見つかる関数を修飾された名前で[2]呼び出す時に、これは濫用の危険を起こします。悪意のあるユーザは、既存の関数の名前

[2] 信用できないユーザにオブジェクトの作成を許可するスキーマを含む検索パスは、安全なスキーマ使用パターンではありませんので、スキーマで修飾されていない名前では危険は起こりません。

で、その関数のパラメータを複製し、デフォルト値を持つ新しいパラメータを追加した関数を作成できます。これは元の関数への新しい呼び出しを妨げます。この危険を未然に防ぐには、関数を信用できるユーザだけがオブジェクトを作成できるスキーマに置いてください。

2. 正確に入力引数型を受け付ける関数があるかどうか検査します。該当する関数があれば（調査される関数の集合内で正確に一致するものは1つしかあり得ません）、それを使用します。正確に一致するものがない場合、信用できないユーザにオブジェクトの作成を許可しているスキーマで見つかる関数を、修飾された名前で[2]呼び出す時にセキュリティの危険が発生します。そのような状況では、強制的に正確に一致するように引数をキャストしてください。（unknownを含む場合は、この段階で一致するものは決して見つかりません。）

3. 正確に一致するものが存在しなかった場合、その関数呼び出しが特別な型変換要求であるかどうかを確認します。これは、関数呼び出しがただ1つの引数を取り、関数名が何らかのデータ型の（内部的な）名前と同一である場合に発生します。さらに、その関数の引数は、unknown型のリテラルか指定されたデータ型へのバイナリ変換可能な型か、型の入出力関数を適用することで指定された型に変換可能な型（つまり、変換が標準文字列型との間の変換である）であるかのいずれかでなければなりません。これらの条件に合う場合、関数呼び出しはCAST仕様の形式と同様に扱われます。[3]

4. 最適なものを検索します。

   a. 関数の候補のうち、入力値のデータ型が一致せず、また、（暗黙的な変換を使用して）一致するように変換できないものを破棄します。unknownリテラルは、上記の目的で何にでも変換可能とみなされます。1つの候補しか残らない場合、それを使います。それ以外の場合は次の段階に進みます。

   b. 入力引数のいずれかがドメイン型であれば、以降の段階すべてでドメインの基本型であるかのように扱います。これにより、曖昧な関数を解決するのを目的としてその基本型であるかのようにドメインが振る舞うことが確実になります。

   c. 全ての候補を検索し、入力型に最も正確に合うものを残します。正確に合うものが何もなければ全ての候補を残します。1つの候補しか残らない場合、それを使います。それ以外の場合は次の段階に進みます。

   d. 全ての候補を検索し、型変換が必要とされるところで（入力データ型カテゴリの）優先される型を受け付けるものを残します。優先される型を受け付けるものが何もなければ全ての候補を残します。1つの候補しか残らない場合、それを使います。それ以外の場合は次の段階に進みます。

   e. 入力引数でunknownのものがあった場合、それらの残った候補に引数位置で受け入れられる型カテゴリを検査します。各位置で候補がstringカテゴリを受け付ける場合は、そのカテゴリを選択します（unknown 型のリテラルは文字列のようなものですので、この文字列への重み付けは適切です）。そうでなければ、もし残った全ての候補が同じ型カテゴリを受け入れる場合はそのカテゴリを選択します。そうでもなければ、さらに手掛かりがなければ正しい選択が演繹されることができませんので、失敗となります。ここで、選択された型カテゴリを受け付けない演算子候補は破棄されます。さらに、このカテゴリ内の優先される型を受け付ける候補が1つでもある場合、その引数の優先されない型を受け付ける候補は破棄されます。これらの検査をどれも通らなかったら全ての候

[3] この処理の理由は、実際にはキャスト関数が存在しない状況において、関数形態のキャスト仕様をサポートすることです。キャスト関数が存在する場合、慣習的に出力型に因んで名付けられます。ですので、特殊な状況を持つ必要はありません。詳細な解説についてはCREATE CASTを参照してください。

補を残します。1つの候補しか残らない場合、それを使います。それ以外の場合は次の段階に進みます。

f. もしunknownと既知の型の引数の両方があり、そして全ての既知の型の引数が同じ型を持っていた場合、unknown引数も同じ型であると仮定し、どの候補がunknown引数の位置にある型を受け付けることができるかを検査します。正確に1つの候補がこの検査を通過した場合、それを使います。それ以外は失敗します。

この「最善一致」規則は演算子と関数の型解決で同一であることに注意してください。以下に例を示します。

例10.6 丸め関数引数の型解決

2つの引数を取るround関数は1つしかありません 第1引数としてnumeric型、第2引数としてinteger型を取ります。ですから、以下の問い合わせは自動的に、integer型の第1引数をnumericに変換します。

```
SELECT round(4, 4);

 round
--------
 4.0000
(1 row)
```

問い合わせはパーサによって実質以下のように変形されます。

```
SELECT round(CAST (4 AS numeric), 4);
```

小数点を持つ数値定数はまずnumericに割り当てられますので、以下の問い合わせでは型変換が不要です。そのためかなり効率的になる可能性があります。

```
SELECT round(4.0, 4);
```

例10.7 可変長引数の関数の解決

```
CREATE FUNCTION public.variadic_example(VARIADIC numeric[]) RETURNS int
  LANGUAGE sql AS 'SELECT 1';
CREATE FUNCTION
```

この関数は、必須ではないですがVARIADICキーワードを受け付けます。整数の引数と数値の引数の両方を許容します。

```
SELECT public.variadic_example(0),
       public.variadic_example(0.0),
       public.variadic_example(VARIADIC array[0.0]);
 variadic_example | variadic_example | variadic_example
------------------+------------------+------------------
                1 |                1 |                1
```

```
(1 row)
```

しかしながら、1番目と2番目の呼び出しは、もし利用可能なら、より特定の関数を優先します。

```
CREATE FUNCTION public.variadic_example(numeric) RETURNS int
  LANGUAGE sql AS 'SELECT 2';
CREATE FUNCTION

CREATE FUNCTION public.variadic_example(int) RETURNS int
  LANGUAGE sql AS 'SELECT 3';
CREATE FUNCTION

SELECT public.variadic_example(0),
       public.variadic_example(0.0),
       public.variadic_example(VARIADIC array[0.0]);
 variadic_example | variadic_example | variadic_example
------------------+------------------+------------------
                3 |                2 |                1
(1 row)
```

もしデフォルトの設定で最初の関数だけが存在しているなら、1番目と2番目の呼び出しは安全ではありません。ユーザは、2番目や3番目の関数を作成することで、それらを妨害できます。引数の型を厳密に一致させVARIADICキーワードを使うのなら、3番目の呼び出しは安全です。

例10.8 部分文字列関数の型解決

substr関数は複数存在します。その1つはtextとinteger型を取ります。型の指定がない文字列定数で呼び出した場合、システムは優先されるカテゴリstring（すなわちtext型）の引数を受け付ける候補関数を選択します。

```
SELECT substr('1234', 3);

 substr
--------
     34
(1 row)
```

文字列がvarchar型と宣言された場合、これはテーブルから取り出した場合が考えられますが、パーサはそれをtextになるように変換しようと試みます。

```
SELECT substr(varchar '1234', 3);

 substr
--------
     34
(1 row)
```

これは以下になるようにパーサによって変換されます。

```
SELECT substr(CAST (varchar '1234' AS text), 3);
```

> **注記**
>
> パーサはpg_castカタログからtextとvarcharがバイナリ互換、つまり、何らかの物理的な変換を行うことなく片方を受け付ける関数にもう片方を渡すことができることを知ります。したがって、この場合実際に挿入される型変換呼び出しはありません。

また、integer型の引数でこの関数が呼び出された場合、パーサはそれをtextに変換しようと試みます。

```
SELECT substr(1234, 3);
ERROR:  function substr(integer, integer) does not exist
HINT:  No function matches the given name and argument types. You might need
to add explicit type casts.
```

integerはtextへの暗黙的なキャストを持たないため、これは失敗します。成功させるには、以下のように明示的なキャストを行います。

```
SELECT substr(CAST (1234 AS text), 3);

 substr
--------
     34
(1 row)
```

10.4. 値の格納

以下の手順に従って、テーブルに挿入される値は対象とする列のデータ型に変換されます。

値の格納における型変換

1. 対象に正確に一致するかどうかを検査します。

2. なければ、式を対象の型に変換してみます。もし2つの型の間の代入キャストがpg_castカタログ(CREATE CASTを参照してください)に登録されている場合、これは可能です。あるいは、もし式がunknown型リテラルの場合、リテラル文字列の内容は対象の型の入力変換ルーチンに与えられます。

3. 対象の型に対してサイズ調整キャストがあるかどうかを検査します。サイズ調整キャストは、ある型からその同じ型へのキャストです。pg_castカタログに1つ見つかった場合は、格納先の列に収納する前に式に適用します。こうしたキャストを実装する関数は、常にinteger型のパラメータを1つ余計に取ります。このパラメータは、格納先の列のatttypmod値を受け付けます（atttypmodの解釈方法はデータ型によって異なりますが、典型的にはそれの宣言された長さです）。また、キャストが明示的か暗黙的かを

示す、第三のbooleanパラメータを取ることもできます。サイズ検査や切り詰めなど、長さに依存したセマンティクスの適用について、キャスト関数が責任を持ちます。

例10.9 character格納における型変換

character(20)として宣言された対象の列への以下の文では、対象の大きさが正確に調整されることを示します。

```
CREATE TABLE vv (v character(20));
INSERT INTO vv SELECT 'abc' || 'def';
SELECT v, octet_length(v) FROM vv;

          v           | octet_length
----------------------+--------------
 abcdef               |           20
(1 row)
```

ここで実際に起こったのは、デフォルトで||演算子がtextの連結として解決できるように、2つのunknownリテラルがtextに解決されたということです。そして演算子のtext型の結果は対象の列の型に合うようにbpchar(「空白が埋められる文字」, characterデータ型の内部名)に変換されます（しかし、textからbpcharへバイナリ変換可能ですので、この型変換のために実際の関数呼び出しは挿入されません）。最後に、bpchar(bpchar, integer, boolean)サイズ調整関数がシステムカタログの中から見つかり、演算子の結果と格納する列の長さを適用します。この型特有の関数は必要とされる長さを検査し、空白の埋め込みを行います。

10.5. UNION、CASEおよび関連する構文

UNION SQL構文は、似ていない可能性がある型を1つの検索結果になるように適合させなければなりません。解決アルゴリズムは1つのunion問い合わせの出力列ごとに適用されます。INTERSECT構文とEXCEPT構文は、UNIONと同じ方法で、似ていない可能性がある型の解決を行います。CASE、ARRAY、VALUESを含むいくつかの構文とGREATEST、LEAST関数は同一のアルゴリズムを使用して、その要素式を適合させ、結果のデータ型を選択します。

UNION、CASEおよび関連する構文の型解決

1. もし全ての入力値が同一型であり、unknownではない場合、その型として解決されます。
2. 入力のいずれかがドメイン型であれば、以降の段階すべてでドメインの基本型であるかのように扱います。[4]
3. もし全ての入力値がunknown型だった場合、text型（文字列カテゴリの優先される型）として解決されます。そうでない場合はunknown入力は残りの規則のために無視されます。
4. もしunknownではない入力値が全て同じ型カテゴリでなければ失敗します。

[4] 演算子や関数に対するドメイン入力の取り扱いとある程度似ていて、この振舞いにより、利用者が注意して入力をすべて厳密な型であると明示的にもしくは暗黙的に保証する限り、ドメイン型をUNIONや類似の構成体に保存できます。そうでなければ、ドメインの基本型が使われます。

5. 最初のunknownではない入力データ型を候補の型として選択します。それから、他のunknownではない入力データ型をそれぞれ左から右へ考慮します。[5] 候補の型が暗黙的にある別の型に変換できるが、その逆はできない場合、その別の型を新しい候補の型として選択します。それから残りの入力の考慮を続けます。この処理のある段階で、優先される型が選択されれば、追加の入力の考慮を止めます。

6. 入力値をすべて最終的な候補の型に変換します。指定された入力型から候補の型への暗黙の変換が存在しない場合は失敗します。

以下に例を示します。

例10.10 Unionにおける指定された型の型解決

```
SELECT text 'a' AS "text" UNION SELECT 'b';

 text
------
 a
 b
(2 rows)
```

ここで、unknown型のリテラル'b'はtextへと解決されます。

例10.11 簡単なUnionにおける型解決

```
SELECT 1.2 AS "numeric" UNION SELECT 1;

 numeric
---------
       1
     1.2
(2 rows)
```

numeric型のリテラル1.2とinteger型の値1は、暗黙的にnumericにキャスト可能です。したがって、この型が使用されます。

例10.12 転置されたUNIONにおける型解決

```
SELECT 1 AS "real" UNION SELECT CAST('2.2' AS REAL);

 real
------
    1
  2.2
(2 rows)
```

[5] 歴史的な理由により、CASEは(もしあれば)そのELSE句を「最初の」入力として扱い、THEN句はその後で考慮します。それ以外の場合では「左から右」は問い合わせテキスト内で式が現れる順を意味します。

ここで、real型を暗黙的にinteger型にキャストすることはできませんが、integer型を暗黙的にreal型にキャストすることはできるため、UNIONの結果データ型はreal型として解決されます。

例10.13 入れ子のUNIONにおける型解決

```
SELECT NULL UNION SELECT NULL UNION SELECT 1;

ERROR:  UNION types text and integer cannot be matched
```

この失敗は、PostgreSQLが複数のUNIONを二項演算の入れ子として扱うために起こります。すなわち、この入力は以下と同じです。

```
(SELECT NULL UNION SELECT NULL) UNION SELECT 1;
```

内側のUNIONは、上に挙げた規則に従って、型textになるものとして解決されます。すると、外側のUNIONは型textとintegerの入力を受け取ることになりますので、上のようなエラーになります。一番左のUNIONが望む結果型の入力を少なくとも1つ確実に受け取るようにすることで、この問題を修正できます。

INTERSECTとEXCEPT操作は同様に二項演算として解決されます。しかしながら、この節のその他の構文は入力をすべて解決の段階1つで考慮します。

10.6. SELECT出力列

これまでの節で挙げた規則は、SELECTコマンドの単純な出力列として現れる型の指定されていないリテラルを除いて、SQL問い合わせでunknownでないデータ型をすべての式に割り当てることになります。例えば、以下で

```
SELECT 'Hello World';
```

文字列リテラルをどの型とみなすべきかを示すものは何もありません。この状況ではPostgreSQLはリテラルの型をtextとして解決することになります。

SELECTがUNION(またはINTERSECT、またはEXCEPT)構文の片方である場合やINSERT ... SELECTの中に現れる場合は、これまでの節で挙げた規則が優先しますので、この規則は適用されません。型の指定されていないリテラルの型は、1番目の場合にはUNIONの他の側から、2番目の場合には対象とする列から取られるでしょう。

RETURNINGリストは、この目的のためにSELECT出力リストと同様に扱われます。

> **注記**
>
> PostgreSQL 10より前では、この規則は存在せず、SELECT出力リストの中の型の指定されていないリテラルは型unknownのままでした。これは様々な悪い結果をもたらしましたので、変更されました。

第11章 インデックス

インデックスは、データベースの性能を向上させるための一般的な方法です。データベースサーバでインデックスを使用すると、インデックスを使用しない場合に比べてかなり速く、特定の行を検出し抽出することができます。しかし、インデックスを使用すると、データベースシステム全体にオーバーヘッドを追加することにもなるため、注意して使用する必要があります。

11.1. 序文

次のようなテーブルを考えてみましょう。

```
CREATE TABLE test1 (
    id integer,
    content varchar
);
```

アプリケーションはこの形式の多くの問い合わせを発行します。

```
SELECT content FROM test1 WHERE id = constant;
```

事前に準備を行っていなければ、システムで一致する項目を全て検出するためには、test1テーブル全体を1行ごとにスキャンする必要があります。test1に数多くの行があり、その問い合わせで返されるのが数行（おそらく0行か1行）しかない場合、これは明らかに効率が悪い方法と言えます。システムがインデックスをid列上で維持するように指示されていれば、一致する行を検出するのにより効率の良い方法を使うことができます。例えば、検索ツリーを数層分検索するだけで済む可能性もあります。

ほとんどのノンフィクションの本で、同じような手法が使われています。読者が頻繁に調べる用語および概念は、その本の最後にアルファベット順に索引としてまとめられています。その本に興味を持った読者は、索引（インデックス）を調べ、比較的速く簡単に該当するページを開くことができるため、見たい場所を探すために本全部を読む必要はありません。読者がよく調べそうな項目を予想するのが著者の仕事であるように、どのインデックスが実用的であるかを予測するのはデータベースプログラマの仕事です。

上述のようにid列にインデックスを作成する場合は、以下のようなコマンドが使用できます。

```
CREATE INDEX test1_id_index ON test1 (id);
```

test1_id_indexというインデックス名には、何を選んでも構いませんが、そのインデックスを何のために作成したかを後で思い出せるような名前を選ぶべきです。

インデックスを削除するには、DROP INDEXコマンドを使用します。テーブルのインデックスは、いつでも追加および削除できます。

いったんインデックスを作成すれば、それ以上の処理は必要はありません。システムは、テーブルが変更される時インデックスを更新し、シーケンシャルスキャンよりもインデックススキャンを行うことがより効率的と判断した場合、問い合わせでインデックスを使用します。しかし、問い合わせプランナで情報に基づいた判断をするためには、定期的にANALYZEコマンドを実行し、統計情報を更新する必要があるかもしれません。イ

ンデックスが使われているかどうか、およびプランナがインデックスを使わないと判断した状況および理由を調べる方法については、第14章を参照してください。

インデックスは、UPDATEやDELETEコマンドの検索条件でも使用できます。さらに、インデックスは結合問い合わせでも使用されます。したがって、結合条件で記述されている列にインデックスを定義すれば、結合を伴った問い合わせにかかる時間もかなり短縮できます。

大規模テーブルに対するインデックス作成が長時間にわたる可能性があります。デフォルトでPostgreSQLはインデックス作成と並行してテーブルを読み取る(SELECT文)ことができますが、書き込み(INSERT、UPDATE、DELETE)はインデックス作成が終わるまでブロックされます。これは多くの運用環境では受け入れられません。インデックス作成中でも並行して書き込みできるようにすることができますが、いくつか注意しなければならないことがあります。Building Indexes Concurrentlyの情報を参照してください。

インデックスが作成された後、システムでは、テーブルとインデックスとの間で常に同期を取っておく必要があります。これにより、データ操作の処理にオーバーヘッドが加わります。したがって、めったに使用されないインデックスや、まったく使用されなくなったインデックスは、削除しておいた方が良いでしょう。

11.2. インデックスの種類

PostgreSQLでは、B-tree、Hash、GiST、SP-GiST、GIN、BRINといった複数の種類のインデックスを使用可能です。インデックスの各種類は、異なる種類の問い合わせに最も適した、異なるアルゴリズムを使用します。デフォルトでCREATE INDEXコマンドは、B-treeインデックスを作成し、それは最も一般的な状況に適合します。

B-treeインデックスは、ある順番でソート可能なデータに対する等価性や範囲を問い合わせることを扱うことができます。具体的には、PostgreSQLの問い合わせプランナは、インデックスの付いた列を次の演算子を使用して比較する場合に、B-treeインデックスの使用を検討します。

<
<=
=
>=
>

また、BETWEENやINなどのこれらの演算子の組み合わせと等価な式もB-treeインデックス検索で実装することができます。インデックスの付いた列に対するIS NULLやIS NOT NULLでもB-treeインデックスを使用することができます。

オプティマイザは、パターンマッチ演算子LIKE、~を含む問い合わせでも、そのパターンが定数であり、先頭文字列を指定しているのであればB-treeインデックスを使用することができます。例えば、col LIKE 'foo%'またはcol ~ '^foo'では使用されますが、col LIKE '%bar'では使用されません。しかし、データベースがCロケールを使用していない場合、パターンマッチ問い合わせのインデックス付けをサポートする特別な演算子クラスでインデックスを作成しなければなりません。後述の11.10を参照してください。なお、ILIKEと~*でもB-treeインデックスを使用することができますが、パターンが英字以外の文字、つまり、大文字小文字の違いの影響がない文字で始まる場合のみです。

B-treeインデックスをソートされた順序でデータを受けとるために使用することもできます。これは常に単純なスキャンとソート処理より高速になるものではありませんが、よく役に立つことがあります。

ハッシュインデックスは単純な等価性比較のみを扱うことができます。問い合わせプランナでは、インデックスの付いた列を=演算子を使用して比較する場合に、ハッシュインデックスの使用を検討します。ハッシュインデックスを作成するには、以下のようなコマンドを使用してください。

```
CREATE INDEX name ON table USING HASH (column);
```

GiSTインデックスは単一種類のインデックスではなく、多くの異なるインデックス戦略を実装することができる基盤です。したがって、具体的なGiSTインデックスで使用できる演算子はインデックス戦略(演算子クラス)によって異なります。例えば、PostgreSQLの標準配布物には、複数の二次元幾何データ型用のGiST演算子クラスが含まれており、以下の演算子を使用してインデックス付けされた問い合わせをサポートします。

<<
&<
&>
>>
<<|
&<|
|&>
|>>
@>
<@
~=
&&

(これらの演算子の意味については9.11を参照してください。) 標準配布物に含まれるGiST演算子クラスは表 64.1に記載されています。他の多くのGiST演算子クラスがcontrib群や別のプロジェクトとして利用可能です。詳細は第64章を参照してください。

GiSTインデックスは以下のような「最近傍」検索を最適化する機能も持ちます。

```
SELECT * FROM places ORDER BY location <-> point '(101,456)' LIMIT 10;
```

これは指定された対象地点に最も近い10箇所を見つけ出します。この場合も、これができるかどうかは使用される特定の演算子クラスに依存します。このように利用できる演算子は表 64.1の「順序付け演算子」列に表示されています。

SP-GiSTインデックスは、GiSTインデックスと同様に様々な種類の検索を支援する基盤を提供します。SP-GiSTインデックスは広域な異なる不均衡でディスクベースのデータ構造、つまり、四分木,kd木、基数木のような実装を認めます。例えば、PostgreSQL標準配布物には、以下の演算子を使用する問い合わせに対するインデックスをサポートする2次元の点用のSP-GiST用の演算子クラスが含まれています。

<<
>>
~=
<@
<^
>^

(演算子の意味は以下を参照してください9.11) 標準配布物に含まれるSP-GiSTクラスは表 65.1に記載されています。詳細は 第65章を参照してください。

GiSTと同様に、SP-GiSTは「最近傍」検索をサポートします。距離の順序付けをサポートするSP-GiST演算子クラスの場合、対応する演算子は表 65.1の「順序付け演算子」列に指定されます。

GINは「転置インデックス」であり、配列などのように複数の要素を持つデータ値に適しています。転置インデックスは各要素値に対して別々のエントリを持っており、特定の要素値の存在について検査する問い合わせを効率的に処理できます。

GiSTやSP-GiST同様、GINも多くの異なるユーザ定義のインデックス戦略を持つことができ、GINが使用できる具体的な演算子はインデックス戦略によって変化します。例えば、PostgreSQL標準配布物には、配列用のGIN演算子クラスが含まれており、これらは、以下の演算子を使用するインデックスによる問い合わせをサポートします。

<@
@>
=
&&

(これらの演算子の意味については9.19を参照してください。) 標準配布物に含まれるGIN演算子クラスは表 66.1に記載されています。他の多くのGIN演算子クラスはcontrib群または別のプロジェクトで利用可能です。詳細は第66章を参照してください。

BRINインデックス(ブロックレンジインデックス(Block Range INdex)を縮めたものです)はテーブルの連続的な物理ブロックの範囲に格納された値についての要約を格納します。GiST、SP-GiST、GINと同じように、BRINは多くの異なるインデックス戦略をサポートし、BRINインデックスが使用できる具体的な演算子はインデックス戦略によって変化します。線形のソート順を持つデータ型では、インデックス付けされたデータは各ブロックレンジの列の中の値の最小値と最大値に対応しています。これは以下の演算子を使用したインデックスによる問い合わせをサポートします。

<
<=
=
>=
>

標準配布物に含まれるBRIN演算子クラスは表 67.1.に記載されています。詳細は第67章を参照してください。

11.3. 複数列インデックス

インデックスは、テーブルの2つ以上の列に定義することができます。例えば、以下のようなテーブルがあるとします。

```
CREATE TABLE test2 (
  major int,
  minor int,
  name varchar
```

```
);
```

（例えば、/devディレクトリの内容をデータベースに保持していて）頻繁に下記のような問い合わせを発行するとします。

```
SELECT name FROM test2 WHERE major = constant AND minor = constant;
```

このような場合、majorおよびminorという2つの列に1つのインデックスを定義する方が適切かもしれません。

```
CREATE INDEX test2_mm_idx ON test2 (major, minor);
```

現在、B-tree、GiST、GINおよびBRINインデックス型でのみ、複数列インデックスをサポートしています。最高32列まで指定可能です。（この上限は、PostgreSQLを構築する際に変更可能です。pg_config_manual.hファイルを参照してください。）

複数列に対するB-treeインデックスをインデックス対象列の任意の部分集合を含む問い合わせ条件で使用することができます。しかし、先頭側の（左側）列に制約がある場合に、このインデックスはもっとも効率的になります。正確な規則は、先頭側の列への等価制約、および、等価制約を持たない先頭列への不等号制約がスキャン対象のインデックス範囲を制限するために使用されます。これらの列の右側の列に対する制約は、このインデックス内から検査されます。ですので、テーブルアクセスを適切に抑えますが、スキャンされるインデックスの範囲を減らしません。例えば、(a, b, c)に対するインデックスがあり、WHERE a = 5 AND b >= 42 AND c < 77という問い合わせ条件があったとすると、a = 5かつb = 42を持つ項目を先頭に、a = 5となる最後の項目までのインデックスをスキャンしなければなりません。c >= 77を持つインデックス項目は飛ばされますが、スキャンを行わなければなりません。このインデックスは原理上、aに対する制約を持たず、bあるいはcに制約に持つ問い合わせでも使用することができます。しかし、インデックス全体がスキャンされますので、ほとんどの場合、プランナはインデックスの使用よりもシーケンシャルテーブルスキャンを選択します。

複数列GiSTインデックスは、インデックス対象列の任意の部分集合を含む問い合わせ条件で使用することができます。他の列に対する条件は、インデックスで返される項目を制限します。しかし、先頭列に対する条件が、インデックスのスキャン量を決定するもっとも重要なものです。先頭列の個別値がわずかな場合、他の列が多くの個別値を持っていたとしても、相対的にGiSTインデックスは非効率的になります。

複数列GINインデックスは、インデックス対象列の任意の部分集合を含む問い合わせ条件で使用することができます。B-treeやGiSTと異なり、インデックス検索の効果はどのインデックス列が問い合わせ条件で使用されているかに関係なく同じです。

複数列BRINインデックスは、インデックス対象列の任意の部分集合を含む問い合わせ条件で使用することができます。GINと同様に、またB-treeやGiSTとは異なり、インデックス検索の効果はどのインデックス列が問い合わせ条件で使用されているかに関係なく同じです。一つのテーブルに対して複数列BRINインデックスを一つ持つ代わりに複数のBRINインデックスを持つ唯一の理由は、異なるpages_per_rangeストレージパラメータを持つためです。

当然ながら、インデックス種類に対して適切な演算子を各列に使用しなければなりません。他の演算子を含む句は考慮されません。

複数列インデックスは慎重に使用する必要があります。多くの場合、単一列のインデックスで十分であり、また、その方がディスク領域と時間を節約できます。テーブルの使用方法が極端に様式化されていない限り、

4つ以上の列を使用しているインデックスは、不適切である可能性が高いでしょう。異なるインデックス構成の利点に関するこの他の説明について11.5および11.9も参照してください。

11.4. インデックスとORDER BY

単に問い合わせによって返される行を見つけ出すだけではなく、インデックスは、その行を指定した順番で取り出すことができます。これにより、別途ソート処理を行うことなく、問い合わせのORDER BY指定に従うことが可能です。PostgreSQLが現在サポートするインデックスの種類の中で、B-treeのみがソート出力を行うことができます。他の種類のインデックスでは指定なし、または、実装固有の順序でマッチした行を返します。

プランナは、ORDER BY指定を満足させるために、指定に一致し利用可能なインデックスでスキャンするか、または、テーブルを物理的な順番でスキャンし明示的なソートを行うかを考慮します。テーブルの大部分のスキャンが必要な問い合わせでは、後に発生するシーケンシャルなアクセスパターンのために要求されるディスクI/Oが少ないため、インデックスを使用するよりも、明示的なソートの方が高速です。数行を取り出す必要がある場合のみ、インデックスの方が有用になります。ORDER BYとLIMIT nが組み合わされた場合が、重要かつ特別です。先頭のn行を識別するために、明示的なソートを全データに対して行う必要があります。しかし、もしORDER BYに合うインデックスが存在すれば、残りの部分をスキャンすることなく、先頭のn行の取り出しを直接行うことができます。

デフォルトでは、B-treeインデックスは項目を昇順で格納し、NULLを最後に格納します。(テーブルTIDはそれ以外が等しいエントリの中で勝ちを決める列として扱われます)。これは、x列に対するインデックスの前方方向のスキャンでORDER BY x(より冗長にいえばORDER BY x ASC NULLS LAST)を満たす出力を生成することを意味します。また、インデックスを後方方向にスキャンすることもでき、この場合、ORDER BY x DESC(より冗長にいえばORDER BY x DESC NULLS FIRST。NULLS FIRSTがORDER BY DESCのデフォルトだからです。)を満たす出力を生成します。

インデックスを作成する時に、以下のようにASC、DESC、NULLS FIRST、NULLS LASTオプションを組み合わせて指定することにより、B-treeインデックスの順序を調整することができます。

```
CREATE INDEX test2_info_nulls_low ON test2 (info NULLS FIRST);
CREATE INDEX test3_desc_index ON test3 (id DESC NULLS LAST);
```

昇順かつNULL先頭という順で格納されたインデックスは、スキャンされる方向に依存してORDER BY x ASC NULLS FIRSTまたはORDER BY x DESC NULLS LASTを満たすことができます。

4つの全方向を提供する理由が何か、後方方向へのスキャンの可能性があることを考慮した2方向で、すべての種類のORDER BYを網羅できるのではないかと疑問を持つかもしれません。単一列に対するインデックスでは、このオプションは実際冗長ですが、複数列に対するインデックスでは有用になります。(x, y)という2つの列に対するインデックスを仮定します。これを前方方向にスキャンすればORDER BY x, yを満たし、後方方向にスキャンすればORDER BY x DESC, y DESCを満たします。しかし、ORDER BY x ASC, y DESCをよく使用しなければならないアプリケーションが存在する可能性があります。簡素なインデックスからこの順序を取り出す方法がありません。しかし、インデックスが(x ASC, y DESC)または(x DESC, y ASC)として定義されていれば、取り出すことができます。

明確なことですが、デフォルト以外のソート順を持つインデックスはかなり特殊な機能です。しかし、特定の問い合わせにおいては恐ろしいほどの速度を向上させることがあります。こうしたインデックスを維持する価値があるかどうかは、特殊なソート順を要求する問い合わせを使用する頻度に依存します。

11.5. 複数のインデックスの組み合わせ

単一のインデックススキャンは、インデックスの列をその演算子クラスの演算子で使用する問い合わせ句と、それをAND結合したものでのみ使用されます。例えば、(a, b)というインデックスとWHERE a = 5 AND b = 6という問い合わせでは、インデックスが使用されます。しかし、WHERE a = 5 OR b = 6のような問い合わせではインデックスは直接使用されません。

幸いにも、PostgreSQLは、単一のインデックススキャンでは実装できない場合を扱うために、複数のインデックス(同じインデックスの複数回使用を含む)を組み合わせる機能を持ちます。システムは複数のインデックススキャンを跨がる、AND条件およびOR条件を形成できます。例えば、WHERE x = 42 OR x = 47 OR x = 53 OR x = 99という問い合わせは、問い合わせ句の1つを使用してx上のインデックスをスキャンする4つのスキャンに分割することができます。その後、これらのスキャンの結果はOR演算でまとめられ、結果を生成します。他の例としてxとyに別個のインデックスがある場合を考えます。WHERE x = 5 AND y = 6のような問い合わせに対して取り得る実装は、適切な問い合わせ句で各インデックスを使用し、インデックスの結果をANDでまとめ、結果行を識別することです。

複数のインデックスを組み合わせるために、システムは必要なインデックスそれぞれをスキャンし、インデックス条件に適合するものと報告されたテーブル行の位置を与えるためにメモリ上に*ビットマップ*を準備します。その後、このビットマップは問い合わせで必要とされたように、ANDまたはOR演算されます。最後に、実際のテーブル行がアクセスされ、返されます。テーブル行は物理的な順番でアクセスされます。ビットマップにこの順番で格納されているからです。これは、元のインデックスの順序が失われていることを意味します。そのため、もし問い合わせがORDER BY句を持つ場合、この他のソート手続きが必要となります。この理由、および、追加のインデックススキャンそれぞれのために余計な時間が加わることから、プランナは追加のインデックスが同様に使用できる場合であっても、単純なインデックススキャンを選択することがあります。

もっとも単純なアプリケーション以外のほとんどすべてのアプリケーションでは、インデックスの有用な組み合わせはいろいろあります。このため、データベース開発者は妥協点を探してどのようなインデックスを提供するかを決定しなければなりません。複数列インデックスが最善な場合がありますし、別々のインデックスを作成し、インデックスの組み合わせ機能に依存する方が優れている場合もあります。例えば、作業にx列のみを含む場合とy列のみを含む場合、両方の列を含む場合が混在する問い合わせが含まれる場合、xとyに対し、別個に2つのインデックスを作成し、両方の列を使用する問い合わせを処理する時にインデックスの組み合わせに依存することを選ぶことができます。また、(x, y)に対する複数列インデックスを作成することもできます。両方の列を含む問い合わせでは、通常このインデックスの方がインデックスの組み合わせよりも効率的です。しかし、11.3で説明した通り、yのみを含む問い合わせではほとんど意味がありません。従って、このインデックスのみとすることはできません。複数列インデックスとyに対する別のインデックスの組み合わせがかなりよく役に立ちます。xのみを含む問い合わせでは、複数列インデックスを使用することができます。しかし、これはより大きくなりますので、x のみインデックスよりも低速になります。最後の別方法は、3つのインデックスすべてを作成することです。しかしこれはおそらく、テーブルの検索頻度が更新頻度よりもかなり高く、3種類の問い合わせすべてが良く使用される場合のみ合理的です。問い合わせの中の1つの頻度が他よりも少なければ、おそらく良く使用される種類にもっとも合うように2つだけインデックスを作成した方がよいでしょう。

11.6. 一意インデックス

インデックスは、列値の一意性や、複数列を組み合わせた値の一意性を強制するためにも使用できます。

```
CREATE UNIQUE INDEX name ON table (column [, ...]);
```

現在、一意インデックスとして宣言できるのはB-treeインデックスのみです。

一意インデックスが宣言された場合、同じインデックス値を有する複数のテーブル行は許されなくなります。NULL 値は同じ値とはみなされません。複数列の一意インデックスは、インデックス列の全てが複数の行で同一の場合のみ拒絶されます。

PostgreSQLでは、テーブルに一意性制約または主キーが定義されると、自動的に一意インデックスを作成します。このインデックスが、主キーや一意性制約（適切ならば複数列のインデックスで）となる列に対して作成され、この制約を強制する機構となります。

> **注記**
>
> 手作業で一意列に対しインデックスを作成する必要はありません。これは、単に自動作成されるインデックスを二重にするだけです。

11.7. 式に対するインデックス

インデックス列は、基礎をなすテーブルにある列である必要はなく、そのテーブルの1つ以上の列から計算される関数やスカラ式とすることもできます。この機能は、ある演算結果に基づいた高速テーブルアクセスを行う時に有用です。

例えば、大文字小文字を区別せずに比較するための一般的な方法である、lower関数での使用例を以下に示します。

```
SELECT * FROM test1 WHERE lower(col1) = 'value';
```

lower(column)関数の結果にインデックスが定義されていれば、この問い合わせでインデックスを使用することができます。

```
CREATE INDEX test1_lower_col1_idx ON test1 (lower(col1));
```

このインデックスをUNIQUEと宣言したとすると、col1の値が同一となる行だけでなく、col1の大文字小文字だけが違う行の生成を防ぐことになります。したがって、式に対するインデックスを使用して、単なる一意性制約では定義できないような制約を強制することができます。

別の例として、以下のような問い合わせが頻繁に行われる場合を考えます。

```
SELECT * FROM people WHERE (first_name || ' ' || last_name) = 'John Smith';
```

この場合、以下のようなインデックスを作成する価値があるでしょう。

```
CREATE INDEX people_names ON people ((first_name || ' ' || last_name));
```

2番目の例に示すようにCREATE INDEXコマンドの構文は通常、インデックス式を括弧で括る必要があります。最初の例のように、式が単なる関数呼び出しの場合には括弧を省略することができます。

派生した式が、行が挿入、更新される度に実行されなければなりませんので、インデックス式は相対的に見て維持が高価です。しかし、インデックス式はインデックス内にすでに格納されているため、インデックスを使用する検索の間は再計算されません。上の両方の例では、システムは問い合わせを単なるWHERE indexedcolumn = 'constant'と理解しますので、この検索速度は他の単純なインデックス問い合わせと同じです。したがって、式に対するインデックスは取り出し速度が挿入、更新速度より重要な場合に有用です。

11.8. 部分インデックス

部分インデックスとは、テーブルの部分集合に構築されるインデックスです。部分集合は、(部分インデックスの述語と呼ばれる)条件式で定義されます。部分インデックスには、その述語を満たすテーブル行のみに対するエントリが含まれます。部分インデックスは特別な機能です。しかし、これらが有用となる状況が複数あります。

部分インデックスを利用する主な目的は、頻出値に対してインデックスを作成しないようにすることです。(テーブル全体の行のうち、数パーセント以上を占める)頻出値を検索する問い合わせでは、いかなる場合でもインデックスを使用しないため、インデックスにそれらの行を持ち続けることは全く意味がありません。これによりインデックスのサイズが小さくなりますので、インデックスを使用する問い合わせが速くなります。また、インデックスを更新する必要のないケースも生じるため、テーブルを更新する作業の多くも速くなります。例 11.1にこの概念に基づいた用例を示します。

例11.1 頻出値を除外するための部分インデックスの作成

ウェブサーバのアクセスログをデータベースに格納しているとします。多くのアクセスは、社内のIPアドレスの範囲内から発信されています。しかし、範囲外のアドレス(例えば、社員がダイアルアップ接続している場所)からの発信もあります。主に範囲外からのアクセスをIPアドレスで検索する場合、社内のサブネットに該当するIPアドレスの範囲にインデックスを作成する必要はないでしょう。

以下のようなテーブルがあると想定します。

```
CREATE TABLE access_log (
    url varchar,
    client_ip inet,
    ...
);
```

この例に適する部分インデックスを作成するには、以下のようなコマンドを使用します。

```
CREATE INDEX access_log_client_ip_ix ON access_log (client_ip)
WHERE NOT (client_ip > inet '192.168.100.0' AND
           client_ip < inet '192.168.100.255');
```

このインデックスを使用できる問い合わせの典型的な例は、以下のようなものです。

```
SELECT *
FROM access_log
WHERE url = '/index.html' AND client_ip = inet '212.78.10.32';
```

この問い合わせのIPアドレスは部分インデックスでカバーされています。以下の問い合わせは、インデックスから除外されているIPアドレスを使用しているので、部分インデックスを使用できません。

```
SELECT *
FROM access_log
WHERE url = '/index.html' AND client_ip = inet '192.168.100.23';
```

このような部分インデックスを使用するには、あらかじめ頻出値が何であるかを知っている必要があることに注意してください。値の分布が変わらない場合に、このような部分インデックスが最善です。データの分布が新しくなった場合はインデックスの再作成によって調整できますが、これはメンテナンス作業を増やしてしまいます。

部分インデックスを使用する有効な他の方法としては、一般的な問い合わせに必要のない値をインデックスから取り除くことです。例 11.2を参照してください。この方法の利点は上で示したものと同じです。ただ、この方法を使用すると、インデックススキャンが適している場合でも、「必要のない」値へのインデックスを介したアクセスが防止されてしまいます。以上のことから明白なように、このようなケースで部分インデックスを作成する際は、細心の注意を払い、十分な検証を行う必要があります。

例11.2 必要のない値を除外するための部分インデックスの作成

請求済み注文書および未請求注文書からなる、1つのテーブルがあるとします。そして、未請求注文書の方がテーブル全体に対する割合が小さく、かつその部分へのアクセス数が最も多かったとします。このような場合、未請求の行のみにインデックスを作成することにより、性能を向上させることができます。インデックスの作成には、以下のようなコマンドを使用します。

```
CREATE INDEX orders_unbilled_index ON orders (order_nr)
    WHERE billed is not true;
```

このインデックスを使用する問い合わせの例としては、次のものが考えられます。

```
SELECT * FROM orders WHERE billed is not true AND order_nr < 10000;
```

しかし、このインデックスは、order_nrをまったく使用しない問い合わせでも使用することができます。以下は、その例です。

```
SELECT * FROM orders WHERE billed is not true AND amount > 5000.00;
```

この問い合わせでは、システムがインデックス全体を検索する必要があるため、amount列に部分インデックスを作成した場合ほど効率は良くありません。しかし、未請求注文書データが比較的少ない場合は、この部分インデックスを未請求注文書を検出するためだけに使用した方が効率が良い可能性があります。

以下の問い合わせでは、このインデックスを使用できないことに注意してください。

```
SELECT * FROM orders WHERE order_nr = 3501;
```

注文番号3501は請求済みかもしれませんし、未請求かもしれないからです。

例 11.2でもわかるように、インデックスが付けられた列名と、述語で使用されている列名は、一致している必要はありません。PostgreSQLでは、インデックス付けされるテーブルの列だけが含まれているのなら、任

意の述語で部分インデックスを使用できます。しかし、この述語は、インデックスを使用させたい問い合わせの条件と一致する必要があることに留意してください。正確に言うと、部分インデックスを問い合わせで使用できるのは、インデックスの述語が問い合わせのWHERE条件に数学的に当てはまるとシステムが判断できる場合のみです。PostgreSQLには、異なった形式で記述された述語が数学的に同等のものであると判断できるような、洗練された定理証明機能はありません。(そのような汎用的な定理証明機能の作成は、非常に困難であるだけではなく、おそらく実際の利用にはあまりにも実行速度が遅過ぎるでしょう。) システムでは、例えば「x < 1」は「x < 2」を意味するというような、単純な比較演算子の意味は認識可能です。しかし、それ以外の場合は、述語条件は問い合わせのWHERE条件と完全に一致している必要があります。一致していない場合は、インデックスは使用可能と認識されません。一致するかどうかは、実行時ではなく、問い合わせ計画作成時に判定されます。したがって、パラメータ付きの問い合わせでは部分インデックスは動作しません。たとえば、「x < ?」と指定されたパラメータを持つ、プリペアド問い合わせでは、どのようなパラメータ値であっても「x < 2」を表しません。

部分インデックスの考えられる3つ目の用法では、問い合わせでインデックスをまったく使用しません。この考え方は、テーブルの部分集合に一意インデックスを作成するというものです。例 11.3を参照してください。これにより、インデックスの述語を満たさない行を制約することなく、その述語を満たす行での一意性を強制します。

例11.3 一意な部分インデックスの作成

テストの結果が格納されているテーブルがあるとします。与えられた件名(subject)および対象(target)の組み合わせに対して、「成功」のエントリが確実に1つしかないようにします。「失敗」のエントリは、複数あっても構いません。以下に、これを実行する一例を示します。

```
CREATE TABLE tests (
    subject text,
    target text,
    success boolean,
    ...
);

CREATE UNIQUE INDEX tests_success_constraint ON tests (subject, target)
    WHERE success;
```

これは、成功するテストが少なく、失敗するテストが多い場合に特に有効な手法です。また、IS NULL制限を使用して一意の部分インデックスを作成することにより、ひとつの列にNULL値をひとつのみ許可できます。

最後に、部分インデックスは、システムの問い合わせ計画の選択を変更するためにも使用できます。特殊なデータ分布を持つデータ集合では、システムが実際には使用すべきでないインデックスを使用してしまうことがあります。このような場合、特定の問い合わせでは使用することができないインデックスを設定することができます。通常、PostgreSQLはインデックスの使用について適切な選択を行います(例えば、頻出値の検索にはインデックスを使用しませんので、前述の例はインデックスのサイズを実際に小さくするだけのもので、インデックスの使用を制限するためには必要はありません)。まったく不適切な計画を選択するようであれば、バグとして報告してください。

部分インデックスを作成するには、少なくとも問い合わせプランナと同等の知識を持っていること、特に、インデックスが有益となる状況を理解している必要があることに留意してください。このような知識を得るためには、PostgreSQLでインデックスがどのように機能するかを理解し、経験を積むことが必要です。ほとんど

の場合、通常のインデックスと比べて、部分インデックスを使用する利点は微細です。例 11.4のように、かなり逆効果な場合があります。

例11.4 パーティショニングの代わりに部分インデックスを使用しない

例えば、重複しない部分インデックスの大きなセットを作りたいと思うかもしれません。

```
CREATE INDEX mytable_cat_1 ON mytable (data) WHERE category = 1;
CREATE INDEX mytable_cat_2 ON mytable (data) WHERE category = 2;
CREATE INDEX mytable_cat_3 ON mytable (data) WHERE category = 3;
...
CREATE INDEX mytable_cat_N ON mytable (data) WHERE category = N;
```

これは良くないアイディアです！ほとんどの場合、以下のように宣言された、部分的でない単一のインデックスを使用する方が良いでしょう。

```
CREATE INDEX mytable_cat_data ON mytable (category, data);
```

(11.3で説明されている理由から、最初にcategory列を指定します。) この大きなインデックスでの検索は、小さなインデックスでの検索よりも2,3ツリーレベルを下に移動する必要がありますが、部分インデックスの適切な1つを選択するためにプランナがおこなう作業よりも、ほぼ確実にコストが削減できます。この問題の核心は、システムが部分インデックス間の関係を理解していないことと、現在の問い合わせに適用出来るかどうかそれぞれ苦労してテストすることです。

テーブルが非常に大きくて、単一のインデックスが本当に悪いアイデアである場合は、代わりにパーティショニングを使用する必要があります(5.11を参照してください)。このメカニズムにより、テーブルとインデックスが重複していないことが、システムで認識されるため、パフォーマンスが大幅に向上します。

部分インデックスの詳細については、[ston89b]、[olson93]、および[seshadri95]を参照してください。

11.9. インデックスオンリースキャンとカバリングインデックス

PostgreSQLにおけるすべてのインデックスは*二次的な*インデックス、つまり各インデックスはテーブルの主要なデータ領域(PostgreSQLの用語ではテーブルの*ヒープ*と呼ばれます)とは別に格納されています。このことは、通常のインデックススキャンにおいて、各行の検索にはインデックスとヒープの両方からデータを取得する必要があることを意味します。さらに、指定のインデックス可能なWHERE条件に適合するインデックスのエントリは、通常、インデックス内の近い位置にあるのに対し、そこから参照されるテーブルの行はヒープ内のあらゆるところにあるかもしれません。このため、インデックススキャンにおけるヒープアクセスの部分では、ヒープに対する多くのランダムアクセスがありますが、これは遅い可能性があり、特に伝統的な回転型メディアでは遅くなります。(11.5で説明したように、ビットマップインデックススキャンはヒープアクセスをソートした順で行うことでこのコストを緩和しようとするものですが、それはある程度までしかできません。)

このパフォーマンス問題を解決するため、PostgreSQLは*インデックスオンリースキャン*をサポートします。これは、問い合わせに対してヒープアクセスをせずにインデックスのみで回答できるものです。基本的な考え

方は、関連するヒープのエントリを参照せずに、各インデックスエントリから直接に値を返すというものです。この方法が使用できるためには2つの基本的な制限があります。

1. インデックスの種類がインデックスオンリースキャンをサポートしている必要があります。B-treeインデックスはいつでもインデックスオンリースキャンをサポートしています。GiSTとSP-GiSTは一部の演算子クラスでインデックスオンリースキャンをサポートしていますが、サポートしない演算子クラスもあります。他のインデックスの種類はインデックスオンリースキャンをサポートしていません。根本的な必要条件は、インデックスが各インデックスのエントリに対応する元のデータ値を物理的に格納していなければならない、あるいはそれを再構築できる必要がある、ということです。その反例として、GINインデックスでは、各インデックスエントリが通常は元のデータ値の一部しか保持していないため、インデックスオンリースキャンをサポートすることができません。

2. 問い合わせはインデックスに格納されている列だけを参照しなければなりません。例えばテーブルの列xとyにインデックスがあり、そのテーブルにはさらに列zがある場合、次の問い合わせはインデックスオンリースキャンを使用できます。

```
SELECT x, y FROM tab WHERE x = 'key';
SELECT x FROM tab WHERE x = 'key' AND y < 42;
```

   しかし、以下の問い合わせはインデックスオンリースキャンを使用できません。

```
SELECT x, z FROM tab WHERE x = 'key';
SELECT x FROM tab WHERE x = 'key' AND z < 42;
```

   （以下で説明するように、式インデックスや部分インデックスは、この規則を複雑にします。）

この2つの基本的な要件が満たされるなら、問い合わせで要求されるすべてのデータ値はインデックスから利用できるので、インデックスオンリースキャンが物理的に可能になります。しかし、PostgreSQLのすべてのテーブルスキャンにおいて、さらなる必要条件があります。それは、第13章で説明するように、検索された各行が問い合わせのMVCCスナップショットに対して「可視」であることを確認しなければならない、ということです。可視性の情報はインデックスのエントリには格納されず、ヒープのエントリにのみあります。そのため、一見すると、すべての行検索はいずれにせよヒープアクセスが必要なように思われます。そして、テーブルの行が最近に更新された場合は、まさにその通りなのです。しかし、あまり更新されないデータについてはこの問題を回避する方法があります。PostgreSQLではテーブルのヒープの各ページについて、そのページに格納されているすべての行が、十分に古く、すべての現在および将来のトランザクションに対して可視であるかどうかを追跡しています。この情報はテーブルの*可視性マップ*のビットに格納されます。インデックスオンリースキャンでは、候補となるインデックスのエントリを見つけた後、対応するヒープページの可視性マップのビットを検査します。それがセットされていれば、行が可視であることがわかるので、それ以上の作業をすることなく、データを返すことができます。セットされていない場合は、それが可視かどうかを調べるためにヒープエントリにアクセスする必要があり、そのため標準的なインデックススキャンに対するパフォーマンス上の利点はありません。うまくいく場合であっても、この方法はヒープアクセスと引き換えに可視性マップにアクセスします。しかし、可視性マップはヒープに比べ、4桁の規模で小さいため、アクセスに必要な物理的I/Oははるかに少ないです。ほとんどの状況では、可視性マップは常にメモリ内にキャッシュされて残っています。

要するに、2つの基本的条件が満たされていればインデックスオンリースキャンが可能ですが、テーブルのヒープページのかなりの部分に対し、その全可視のビットがセットされている場合にのみ、性能が向上します。しかし大部分の行が変化しないテーブルは一般的であり、現実にはこの種のスキャンは非常に有効です。

インデックスオンリースキャンの機能を有効に利用するため、カバリングインデックスの作成を選択できます。これは、頻繁に実行する特定の種類の問い合わせに必要な列を含めるように特別に設計されたインデックスです。問い合わせは通常、検索対象の列よりも多くの列を取得する必要があるため、PostgreSQLはいくつかの列を単に「ペイロード」として検索キーの一部ではないインデックスを作成できます。これは追加の列リストをINCLUDE句に加えることで実行出来ます。例えば、次のような問い合わせをよく実行する場合を考えます。

```
SELECT y FROM tab WHERE x = 'key';
```

このような問い合わせを高速化する伝統的な手法は、xのみにインデックスを作成することです。しかし、次のようなインデックス定義では、

```
CREATE INDEX tab_x_y ON tab(x) INCLUDE (y);
```

yはヒープにアクセスしなくてもインデックスから取得できるため、この問い合わせをインデックスオンリースキャンとして処理できます。

y列はインデックスの検索キーの一部ではないため、インデックスが処理できるデータ型である必要はありません。単にインデックスに格納されているだけで、インデックス機構によって解釈されることはありません。また、インデックスが一意インデックスの場合は、

```
CREATE UNIQUE INDEX tab_x_y ON tab(x) INCLUDE (y);
```

x列のみに一意性条件が適用されます。xとyの組み合わせではありません。(INCLUDE句は、インデックスを設定するための代替構文を提供するUNIQUEやPRIMARY KEYの制約として書くことも出来ます。)

キー以外のペイロード列、特に幅の広い列をインデックスに追加することについては慎重になることが賢明です。インデックス型の最大サイズを超えるタプルをインサートしようとすると失敗します。いかなる場合でもキー以外の列が重複データだったり、インデックスサイズが膨張すると、検索が遅くなる可能性があります。それから、覚えておくべきもう一つの小さなポイントは、インデックスオンリースキャンがヒープへのアクセスが必要がないほどテーブルがゆっくり変化しない限り、インデックスにペイロード列を含めることにほとんど意味が無いことです。とにかくヒープタプルを訪れなければならなくなった場合は、そこから列値を取得するためにそれ以上のコストはかかりません。他の制限は、式を列に含めることは、現在のところサポートされていません。また、列を含めるサポートは現在のところB-treeとGiSTインデックスのみサポートされています。

INCLUDE機能がない以前のPostgreSQLでは、ペイロード列を通常のインデックス列としてカバリングインデックスを作成することがありました。

```
CREATE INDEX tab_x_y ON tab(x, y);
```

これは、yをWHERE句の一部で使用するつもりがなかったとしても書いています。余分な列が末尾の列である限り、これはうまく機能します。それらを先頭側の列にすることは、11.3で説明されている理由から賢明ではありません。しかし、この方法では、キー列に一意性を強制するインデックスがサポートされません。

末尾消去は、常に上位のB-treeレベルから非キーの列を削除します。ペイロード列として、それらはインデックススキャンを導くためには使われません。また、この消去プロセスは、キー列の残りのプレフィックスが、最下位のB-treeレベルのタプルを記述するのに十分である場合、1つ以上の後続キー列も削除します。実際

上、INCLUDE句を使用しないカバリングインデックスは、実質的に上位レベルにペイロードが含まれるカラムの格納を避けられます。ただし、ペイロード列を非キー列として明示的に定義すると確実に上位レベルのタプルが小さくなります。

原則として、インデックスオンリースキャンは式インデックスでも使うことができます。例えば、xがテーブルの列で、f(x)上にインデックスがある場合、次の問い合わせをインデックスオンリースキャンとして実行できるはずです。

```
SELECT f(x) FROM tab WHERE f(x) < 1;
```

そして、関数f()の計算が高価なら、この方法は非常に魅力的です。しかしPostgreSQLのプランナは現在のところ、このような場合についてあまり賢くありません。プランナは、問い合わせで必要となるすべての列がインデックスから利用可能な場合にのみ、その問い合わせが潜在的にインデックスオンリースキャンで実行可能と考えます。この例では、f(x)という文脈でしかxは必要になりませんが、プランナはそのことに気付かないため、インデックスオンリースキャンは不可能であると結論します。インデックスオンリースキャンは十分に価値があると思われるなら、含める列としてxを追加することで回避できます。例をあげます。

```
CREATE INDEX tab_f_x ON tab (f(x)) INCLUDE (x);
```

目的がf(x)の再計算を避けることの場合、さらなる注意として、プランナはインデックス可能なWHERE句にないf(x)の使用を必ずしもインデックス列とマッチしないという事があります。上記のような単純な問い合わせの場合は通常は正しく処理できるでしょうが、結合を含む問い合わせでは駄目でしょう。これらの欠点はPostgreSQLの将来のバージョンで解決されるかもしれません。

部分インデックスもインデックスオンリースキャンとの間に興味深い関係があります。例 11.3に示す部分インデックスを考えます。

```
CREATE UNIQUE INDEX tests_success_constraint ON tests (subject, target)
    WHERE success;
```

原則として、次のような問い合わせに対して、このインデックスを使ったインデックスオンリースキャンが可能です。

```
SELECT target FROM tests WHERE subject = 'some-subject' AND success;
```

しかし、WHERE句で参照されるsuccessがインデックスの結果列として利用できないという問題があります。それにも関わらず、インデックスオンリースキャンが可能です。なぜなら、このプランではWHERE句のその部分を実行時に再検査する必要がない、つまりインデックス内にあるすべてのエントリは必ずsuccess = trueなので、プラン内でこれを明示的に検査する必要がないからです。PostgreSQLのバージョン9.6およびそれ以降ではこのような場合を認識し、インデックスオンリースキャンを生成可能ですが、それより古いバージョンではできません。

11.10. 演算子クラスと演算子族

インデックス定義では、インデックスの各列に演算子クラスを指定することができます。

```
CREATE INDEX name ON table (column opclass [ ( opclass_options ) ] [sort options] [, ...]);
```

演算子クラスにより、その列のインデックスで使用される演算子が特定されます。例えば、int4型に対するB-treeインデックスには、int4_opsクラスを使用します。この演算子クラスには、int4型の値用の比較関数が含まれています。実際には、通常、列のデータ型のデフォルト演算子クラスで十分です。演算子クラスを持つ主な理由は、いくつかのデータ型では、複数の有意義なインデックスの振舞いがあり得るということです。例えば、複素数データ型を、絶対値でソートしたいかもしれませんし、実数部でソートしたいかもしれません。この処理は、そのデータ型の2つの演算子クラスを定義した上で、インデックスを作成する際に適切なクラスを選択することで、実行可能です。演算子クラスは基本的なソート順を決定します。(これはソートオプションCOLLATE、ASC/DESC、NULLS FIRST/NULLS LASTを付けることで変更できます。)

以下のように、デフォルトの演算子クラスとは別に、組み込み演算子クラスがいくつかあります。

- text_pattern_ops、varchar_pattern_ops、bpchar_pattern_ops演算子クラスは、それぞれ、text、varchar、char型上のB-treeインデックスをサポートします。デフォルトの演算子クラスとの違いは、ロケール特有の照合規則に従わずに、文字同士を厳密に比較する点です。これらの演算子クラスを、標準「C」ロケールを使用しないデータベースにおける、パターンマッチ式(LIKEやPOSIX正規表現)を含む問い合わせでの使用に適したものにします。例えば、以下のようにvarcharのインデックスを作成できます。

  ```
  CREATE INDEX test_index ON test_table (col varchar_pattern_ops);
  ```

  また、通常の<、<=、>、または>=比較を含む問い合わせでインデックスを使いたい場合も、デフォルトの演算子クラスでインデックスを作成しなければならないことに注意してください。こうした問い合わせではxxx_pattern_ops演算子クラスを使用することができません。(しかし、通常の等価比較はこれらの演算子クラスを使用することができます。) 同じ列に対して異なる演算子クラスを使用して複数のインデックスを作成することができます。Cロケールを使用する場合は、xxx_pattern_ops演算子クラスは必要ありません。Cロケールでのパターンマッチ問い合わせでは、デフォルト演算子クラスを使用したインデックスが使用できるためです。

以下の問い合わせは、定義済みの演算子クラスを全て返します。

```
SELECT am.amname AS index_method,
       opc.opcname AS opclass_name,
       opc.opcintype::regtype AS indexed_type,
       opc.opcdefault AS is_default
    FROM pg_am am, pg_opclass opc
    WHERE opc.opcmethod = am.oid
    ORDER BY index_method, opclass_name;
```

実際のところ演算子クラスは、演算子族と呼ばれる、より大きな構造の一部でしかありません。複数のデータ型が似たような動作を行う場合、データ型を跨る演算子を定義し、インデックスで使用可能とすることが有用な場合がよくあります。このためには、各型に対する演算子クラスが同一の演算子族にまとめられている必要があります。データ型を跨る演算子は演算子族の要素です。演算子族内の1つの演算子クラスに結びついているわけではありません。

以下は前述の問い合わせを拡張したバージョンで、各演算子クラスが属する演算子族を示します。

```
SELECT am.amname AS index_method,
```

```
       opc.opcname AS opclass_name,
       opf.opfname AS opfamily_name,
       opc.opcintype::regtype AS indexed_type,
       opc.opcdefault AS is_default
    FROM pg_am am, pg_opclass opc, pg_opfamily opf
    WHERE opc.opcmethod = am.oid AND
          opc.opcfamily = opf.oid
    ORDER BY index_method, opclass_name;
```

以下の問い合わせは、定義済みの演算子族と各演算子族に含まれる演算子をすべて表示します。

```
SELECT am.amname AS index_method,
       opf.opfname AS opfamily_name,
       amop.amopopr::regoperator AS opfamily_operator
    FROM pg_am am, pg_opfamily opf, pg_amop amop
    WHERE opf.opfmethod = am.oid AND
          amop.amopfamily = opf.oid
    ORDER BY index_method, opfamily_name, opfamily_operator;
```

ヒント

psqlには\dAc、\dAf、\dAoコマンドがあり、これらのクエリのもう少し洗練されたバージョンを提供します。

11.11. インデックスと照合順序

インデックスはインデックス列当たり1つの照合順序のみをサポートすることができます。複数の照合順序を考慮しなければならない場合、複数のインデックスが必要になるかもしれません。

以下の文を考えてみます。

```
CREATE TABLE test1c (
    id integer,
    content varchar COLLATE "x"
);

CREATE INDEX test1c_content_index ON test1c (content);
```

このインデックスは自動的に背後にある列の照合順序を使用することになり、

```
SELECT * FROM test1c WHERE content > constant;
```

という形式の問い合わせでは、この比較はデフォルトで列の照合順序を使用しますので、このインデックスを使用することになります。しかし、このインデックスは何らかの他の照合順序を含む問い合わせを高速化することはできません。このため

```
SELECT * FROM test1c WHERE content > constant COLLATE "y";
```

という形式の問い合わせも考慮しなければならない場合は、以下のように"y"照合順序をサポートする追加のインデックスを作成することになります。

```
CREATE INDEX test1c_content_y_index ON test1c (content COLLATE "y");
```

11.12. インデックス使用状況の検証

PostgreSQLでは、インデックスのメンテナンスやチューニングは必要ありませんが、どのインデックスが実際の問い合わせで使われているかを確認することは、やはり重要です。個々のコマンドでのインデックスの使用状況は、EXPLAINコマンドで検証できます。この目的のための用例を14.1に示します。また、27.2に示す通り、稼働中のサーバにおけるインデックス使用状況の全体的な統計情報を取り出すこともできます。

どのインデックスを作成すべきかを判断するための一般的な手順を定めることは困難です。これまでの節では、例として典型的なケースをいくつか記述してきました。十分な検証がしばしば必要です。本節の残りで、検証のためのヒントをいくつか説明しておきます。

- まず、必ずANALYZEコマンドを実行してください。このコマンドにより、テーブル内の値の分布に関する統計情報を収集します。この情報は、問い合わせにより返される行数を推測する際に必要となります。推測された行数は、可能な各問い合わせ計画に実際のコストを割り当てるために、プランナで必要となります。実際の統計情報が欠如している場合、何らかのデフォルト値が仮定されますが、このデフォルト値は、ほぼ間違いなく不正確です。したがって、ANALYZEコマンドを実行せずに、アプリケーションのインデックス使用状況を検証しても、あまり意味がありません。より詳細な情報は24.1.3と24.1.6を参照してください。

- 検証には、実際に使用するデータを使ってください。テストデータを使ってインデックスを作成した場合、テストデータに必要なインデックスはわかりますが、それ以上はわかりません。

  非常に小さなテストデータを使用することも、結果に特に致命的な影響を与えます。100,000行から1,000行を選択する場合は、インデックスが使用される可能性がありますが、100行から1行を選択する場合はインデックスはまず使用されません。なぜなら、100行はおそらく1つのディスクページに収まるため、1ページを逐次読み取るよりも高速な計画は存在しないからです。

  また、アプリケーションがまだ実動していない場合、テストデータを作成しなければならないことがよくありますが、その際にも注意が必要です。非常に類似した値や、完全にランダムな値、またはソートされた順序で値が挿入されている場合は、その統計情報は、実際のデータの分布とかけ離れたものになってしまいます。

- インデックスが使用されていない場合、テストのためにインデックスを強制的に使用するようにすると便利です。様々な計画の種類を無効にすることを設定できる実行時パラメータがあります（19.7.1を参照してください）。例えば、最も基本的な計画であるシーケンシャルスキャン（enable_seqscan）およびネステッドループ結合（enable_nestloop）を無効に設定すると、システムは別の計画を使用するように強制されます。そのような設定を行っても、システムがシーケンシャルスキャンやネステッドループ結合を選択する場合は、インデックスを使用しない理由としておそらくもっと根本的な理由があるということになります。例えば、問い合わせの条件がインデックスに適合しない、などが考えられます。（どのような問い合わせで、どのようなインデックスを使用できるかは、前節までで説明済みです。）

- 強制的にインデックスを使うように設定することで、インデックスを使用するようになった場合は、次の2つの可能性が考えられます。システムの判断が正しく、インデックスの使用が実際には適切ではないという可能性と、問い合わせ計画のコスト推定が実情を反映していない可能性です。したがって、インデックスを使った問い合わせの実行時間と、使わない場合の実行時間を計測する必要があります。この場合、`EXPLAIN ANALYZE`コマンドが便利です。

- コスト推定が間違っていると判明した場合、やはり2つの可能性が考えられます。総コストは、各計画ノードの行単位のコストに、計画ノードの推定選択度を掛けることで算出されます。計画ノードのコスト推定は、実行時パラメータによって設定することができます（19.7.2を参照してください）。推定選択度が不正確であるのは、統計情報が不十分であるのが原因です。統計情報収集用のパラメータを調節することによって、この状況を改善することができるかもしれません。（ALTER TABLEを参照してください）。

  コストを適切に調節できない場合は、明示的にインデックスの使用を強制する必要が考えられます。あるいは、PostgreSQL開発者に問題の調査を依頼することになるかもしれません。

第12章 全文検索

12.1. 導入

全文検索(または単にテキスト検索)は、問い合わせを満たす自然言語の文書を識別し、更には問い合わせとの関連性の順に並び替えることができます。もっとも一般的な検索は、与えられた検索語を含む文書を探し、問い合わせとの類似性の順に返す、というものです。問い合わせと類似性の記法は非常に柔軟で、特定の用途に適合できます。もっとも単純な検索では、問い合わせは単語の集合として、類似性は文書中の問い合わせ対象の単語の頻度として扱います。

テキスト検索演算子は、データベースシステムに長年存在していました。PostgreSQLは、テキストデータ型用に、~,~*, LIKE,ILIKEの各演算子を持っています。しかし、近代的な情報システムに必要な以下の本質的な特徴を欠いています。

- 英語にさえ、言語学的なサポートがありません。派生語、たとえばsatisfiesに対してsatisfyを容易に扱えないため、正規表現は十分ではありません。satisfyを探すときは、たぶんあなたはsatisfiesも探したいでしょうが、それらを含む文書は探せないかもしれません。ORを使えば複数の派生語を検索することができますが、退屈で間違いやすいです(ある種の単語は数千の派生語を持つことがあります)。
- 検索結果を順序付け(順位付け)することができません。その結果、数千の合致する文書が見つかったような場合に非効率的です。
- インデックスをサポートしないので毎回検索時にすべての文書を処理しなければならず、遅いです。

全文検索のインデックス付けでは、文書を前もって処理しておき、後で素早く検索するために、インデックスを保存しておくことができます。前処理には以下があります。

文書からトークンを解析します。トークンを色々なクラス、たとえば数、単語、複合単語、電子メールアドレスに分けて識別することが有効です。そうすれば、扱いを変えることができます。原則として、トークンのクラスは、特定の用途に依存します。しかし、ほとんどの目的には、あらかじめ定義されたクラスの集合を使うのが適当です。PostgreSQLは、パーサを使ってこの処理段階を実行します。標準搭載のパーサが提供されますが、特別な用途にはカスタム仕様のパーサを作ることもできます。

トークンを語彙素*(lexemes)*に変換します。語彙素はトークンと同じ文字列ですが、違う形態の同じ単語が同じになるように正規化されています。たとえば、正規化においてはほぼ常に大文字を小文字に変換し、(英語のsまたはesのような)接尾辞を取り除くことが多いです。これにより、可能性のあるすべての変種を地道に入力すること無く、同じ単語の変化形を検索できます。また、このステップでは、あまりにありふれていて、検索の役に立たないストップワードを取り除くことが多いです。(つまり、トークンは文書テキストの未加工の断片そのものであり、語彙素はインデックス付けや検索に有用と思われる単語です。) PostgreSQLは、辞書を使ってこのステップを実行します。いろいろな標準辞書が提供されています。特定の用途向けにカスタム辞書を作ることもできます。

検索に最適化された前処理済の文書を保存します。たとえば、個々の文書は、正規化された語彙素の整列済の配列として表現されます。語彙素とともに、適合性ランキング用に、位置情報を格納しておくことがしばしば望まれます。そうすることにより、問い合わせの語を「高密度」に含んでいる文書を、まばらに含む文書よりも高くランクづけすることができます。

辞書を使ってトークンの正規化を細かく制御できます。適当な辞書を用意すれば次のようなことができます。

- インデックスしたくないストップワードの定義
- Ispellを使って、同義語を単一の単語に関連づける
- 類語辞書(thesaurus)を使って、成句を単一の単語に関連づける
- Ispell辞書を使って、単語の変種を正規の単語に関連づける
- Snowball語幹規則を使って、単語の変種を正規の単語に関連づける

前処理した文書を格納するために、データ型tsvectorが提供されています。また、処理済問い合わせを表現するためにtsquery型も提供されています(8.11)。これらのデータ型のために、多数の関数と演算子が利用できますが(9.13)、もっとも重要なのは、12.1.2で紹介している@@演算子です。全文検索はインデックス(12.9)を使って高速化できます。

12.1.1. 文書とは何か?

文書は全文検索システムにおける検索の単位です。たとえば、雑誌記事やメールのメッセージです。テキスト検索エンジンは、文書をパースし、語彙素(キーワード)とそれが含まれる親文書の関連を格納できなければなりません。後で、この関連を使って問い合わせ語を含む文書を検索するのに使います。

PostgreSQLでの検索においては、ドキュメントはデータベースのテーブルの行内のテキストフィールドか、あるいはそのようなフィールドの組み合わせ(結合)でもよいです。そうしたフィールドはおそらく複数のテーブルに格納されていたり、動的に獲得されるものであったりします。言い換えると、文書はインデックス付けのために複数の異なる部分から構成されても良く、それらが全体としてはひとまとまりに格納されていなくても良いのです。例を示します。

```
SELECT title || ' ' ||  author || ' ' ||  abstract || ' ' || body AS document
FROM messages
WHERE mid = 12;

SELECT m.title || ' ' || m.author || ' ' || m.abstract || ' ' || d.body AS document
FROM messages m, docs d
WHERE m.mid = d.did AND m.mid = 12;
```

> **注記**
>
> 実際には、これらの例の問い合わせでは、coalesceを使って、一部NULLが含まれているためにドキュメント全体がNULLになってしまうのを防ぐべきです。

別な方法としては、ファイルシステム上に文書を単純なテキストファイルとして格納することです。この場合、データベースは、フルテキストインデックスを格納し、検索を実行するために使うことができます。ファイルシステムから文書を取り出すためには、何かのユニークな識別子を使います。しかし、データベースの外にあるファイルを取り出すには、スーパーユーザの許可か、特殊な関数のサポートが必要です。そういうわけでたいていの場合はPostgreSQLの中にすべてのデータを保持するのよりも不便です。また、すべてのデータをデータベースに保持することにより、文書のインデックス付けと表示の際に文書のメタデータにアクセスすることが容易になります。

テキスト検索という目的のため、各々の文書は前処理されてtsvector形式に変換しておかなければなりません。検索と順位付けはすべてtsvector表現の文書上で行われます。検索とランキングは文書のtsvector表

現上で実行されます — オリジナル文書は、ユーザに表示のため選択された場合にのみ取り出される必要があります。というわけで、ここではtsvectorを文書と見なすことがよくあります。といっても、tsvectorは完全な文書の縮小表現でしかありません。

12.1.2. 基本的なテキスト照合

PostgreSQLにおける全文検索は、tsvector(文書)が、tsquery(問い合わせ)に一致したら真を返す照合演算子@@に基づいています。どちらのデータ型を先に書いても構いません。

```
SELECT 'a fat cat sat on a mat and ate a fat rat'::tsvector @@ 'cat & rat'::tsquery;
 ?column?
----------
 t

SELECT 'fat & cow'::tsquery @@ 'a fat cat sat on a mat and ate a fat rat'::tsvector;
 ?column?
----------
 f
```

上記の例でわかるように、tsqueryは、tsvectorと違って、単なるテキストではありません。tsqueryは正規化済の語彙素である検索表現を含み、AND, OR, NOT, FOLLOWED BY演算子を使って複数の表現を組み合わせても構いません。(詳細は8.11.2を見てください。) 主にテキスト中の単語を正規化することにより、 ユーザが入力したテキストを適切なtsqueryに変換するのに便利な関数to_tsquery、plainto_tsquery、phraseto_tsqueryがあります。同様に、文書文字列をパースして正規化するためにto_tsvectorが利用できます。というわけで、実際にはテキスト検索照合はこんな感じになります。

```
SELECT to_tsvector('fat cats ate fat rats') @@ to_tsquery('fat & rat');
 ?column?
----------
 t
```

この照合は、もしつぎのように書くとうまくいかないことに注意してください。

```
SELECT 'fat cats ate fat rats'::tsvector @@ to_tsquery('fat & rat');
 ?column?
----------
 f
```

というのも、単語ratsに対して正規化が行われないからです。tsvectorの要素は、すでに正規化されている語彙素であることになっているので、ratsはratに一致しません。

また、@@演算子は、textを入力として受付けるので、簡単に使うときには、明示的にテキスト文字列をtsvectorまたはtsqueryに変換することを省略できます。応用として以下のものがあります。

```
tsvector @@ tsquery
tsquery  @@ tsvector
text @@ tsquery
```

```
text @@ text
```

最初の2つについてはすでに説明しました。text @@ tsqueryという形式は、to_tsvector(x) @@ yと同じです。text @@ textという形式は、to_tsvector(x) @@plainto_tsquery(y)と同じです。

tsquery内において、演算子 & (AND) は、マッチと見なされるには引数の両方がドキュメント内に現れる必要があるということを指定します。同様に、演算子 | (OR) では、引数の少なくとも一方が現れる必要があり、また演算子 ! (NOT) は、マッチと見なされるには引数が現れてはならないことを指定します。例えば、fat & ! ratという問い合わせは、fatは含むがratは含まないドキュメントとマッチします。

句の検索は、tsquery演算子 <-> (FOLLOWED BY)を使うことで可能です。この演算子は、その引数にマッチする語が隣接していて、かつ指定と同じ順序である場合にのみマッチします。例を示します。

```
SELECT to_tsvector('fatal error') @@ to_tsquery('fatal <-> error');
 ?column?
----------
 t

SELECT to_tsvector('error is not fatal') @@ to_tsquery('fatal <-> error');
 ?column?
----------
 f
```

FOLLOWED BY演算子にはもっと汎用的なバージョンがあり、それは<N>という構文で使います。ここでNは整数で、マッチする語彙素の位置の差を表します。<1>は<->と同じですが、<2>ではマッチする語の間にちょうど1つ、他の語彙素が現れることを許容する、という具合です。phraseto_tsquery関数は、この演算子を利用して、ストップワードを含む複数語の句にマッチ可能なtsqueryを構築するものです。例を示します。

```
SELECT phraseto_tsquery('cats ate rats');
       phraseto_tsquery
------------------------------
 'cat' <-> 'ate' <-> 'rat'

SELECT phraseto_tsquery('the cats ate the rats');
       phraseto_tsquery
------------------------------
 'cat' <-> 'ate' <2> 'rat'
```

ときに役立つことがある特別な場合として、<0>を2つのパターンが同じ語にマッチすることを要求するために使うことができます。

tsquery演算子を入れ子にして管理するために括弧を使うことができます。括弧がない場合、|の結合が最も弱く、次が&、その次が<->で、!が最も強く結合します。

FOLLOWED BYの中ではマッチの正確な位置が重要ですので、AND/OR/NOT演算子は、FOLLOWED BY演算子の引数の中で使われる場合にはそうでない場合と微妙に異なる意味になることに言及しておく価値はあります。例えば、通常!xはxをどこにも含まない文書とのみマッチします。しかし、!x <-> yは、xの直後にあるのでなければyとマッチします。文書の他のところでのxの出現は、マッチを邪魔しません。もう一つの例は、

x & yは通常xとyの両方が文書のどこかに現れることだけを要求しますが、(x & y) <-> zは、xとyが同じ場所、zの直前でマッチすることを要求します。そのため、この問い合わせはx <-> z & y <-> zとは異なった振る舞いをします。後者は2つの別の文字列、x zとy zを含む文書にマッチします。(xとyが同じ場所でマッチすることはあり得ませんので、上に書いたこの特別な問い合わせは、役に立ちません。しかし、接頭辞マッチパターンのようにより複雑な状況では、この形の問い合わせは役に立つかもしれません。)

12.1.3. 設定

今までのはすべて単純なテキスト検索の例でした。すでに述べたように、全文検索機能を使えば、もっと色々なことができます。インデックス付けの際に特定の単語をスキップ(ストップワード)、同義語(synonym)処理、賢いパース処理、すなわち、単に空白区切りに基づくパース処理以上のものです。この機能はテキスト検索設定で制御します。PostgreSQLには、多くの言語用の設定があらかじめ組み込まれていますが、ユーザ設定を容易に作ることもできます。(psqlの\dFコマンドで、利用できる設定を表示できます。)

インストールの際には、適当な設定が選ばれ、default_text_search_configがpostgresql.conf中にセットされます。クラスタ全体で同じ設定を使用する場合はpostgresql.confの設定値を利用できます。クラスタの設定とは異なるが、あるデータベースの中で同じ設定を使う場合には、ALTER DATABASE ... SETを利用します。さもなければ、セッション単位でdefault_text_search_configを設定できます。

設定に依存するテキスト検索関数は、オプションでregconfig引数を持っており、使用する設定を明示的に指定できます。default_text_search_configは、この引数が省略されたときだけ使用されます。

カスタムテキスト検索設定を作り易くするため、設定はより単純なデータベースオブジェクトから作られます。PostgreSQLのテキスト検索機能は、4つの設定関連のデータベースオブジェクトを提供しています。

- テキスト検索パーサは、文書をトークンに分解し、トークンを分類します(たとえば、単語とか数のように)。
- テキスト検索辞書はトークンを正規化された形式に変換し、ストップワードを排除します。
- テキスト検索テンプレートは、現在の辞書が利用する関数を提供します(辞書は、単にテンプレートと、その引数の集合を指定するだけです)。
- テキスト検索設定は、パーサと使用する辞書の集合を選択し、パーサが生成したトークンを正規化します。

テキスト検索パーサとテンプレートは、低レベルのC関数で作ります。したがって、新しく開発するためにはCのプログラミング能力と、データベースにインストールするためのスーパーユーザ権限が必要になります。(PostgreSQLの配布物のcontrib/には、追加パーサとテンプレートの例があります)。辞書と設定は、単に配下のパーサとテンプレートのパラメータを設定し、両者を結び付けるだけなので、新しい辞書と設定を作るために特別な権限は必要ありません。この章の後でカスタム辞書と設定を作る例が登場します。

12.2. テーブルとインデックス

前の節の例では、単純な文字列定数を使った全文検索照合を説明しました。この節では、テーブルのデータを検索する方法、そしてインデックスを使う方法を示します。

12.2.1. テーブルを検索する

インデックスがなくても全文検索をすることは可能です。bodyフィールド中のfriendという単語を含む行のtitleを印刷する単純な問い合わせは次のようになります。

```
SELECT title
FROM pgweb
WHERE to_tsvector('english', body) @@ to_tsquery('english', 'friend');
```

同時に、これは、friends、friendlyのように、関連する単語を見つけ出します。これらはすべて同じ正規化された語彙素に帰結するからです。

上の問い合わせはenglish設定を使って文字列をパースして正規化することを指定しています。別の方法としては、設定パラメータを省略することができます。

```
SELECT title
FROM pgweb
WHERE to_tsvector(body) @@ to_tsquery('friend');
```

この問い合わせはdefault_text_search_configで設定された設定を使用します。

もっと複雑な例として、createとtableをtitleまたはbodyに含む文書のうち新しい順に10個選ぶというものを示します。

```
SELECT title
FROM pgweb
WHERE to_tsvector(title || ' ' || body) @@ to_tsquery('create & table')
ORDER BY last_mod_date DESC
LIMIT 10;
```

細かいことですが、この例では、二つのうち一つのフィールドにNULLを含む行を探すために必要なcoalesce関数の呼び出しを省略しています。

これらの問い合わせはインデックスなしでも動きますが、たまに実行する一時的な問い合わせ用を除くと、たいていの用途には遅すぎます。実用上は、インデックスを作成することが必要なのが普通です。

12.2.2. インデックスの作成

テキスト検索を高速化するために、GINインデックス(12.9)を作ることができます。

```
CREATE INDEX pgweb_idx ON pgweb USING GIN (to_tsvector('english', body));
```

2引数バージョンのto_tsvectorを使っていることに注意してください。設定名を指定するテキスト検索関数だけが、式インデックス(11.7)で使えます。これは、インデックス内容が、default_text_search_configの影響を受けないためです。もし影響を受けるとすると、異なるテキスト検索設定で作られたtsvectorを持つエントリの間でインデックス内容が首尾一貫しなくなるからです。そして、どのエントリがどのようにして作られたのか、推測する方法はないでしょう。そのようなインデックスを正しくダンプ、リストアするのは不可能でしょう。

上記のインデックスでは、2引数バージョンのto_tsvectorが使われているので、同じ設定名の2引数バージョンのto_tsvectorを使う問い合わせ参照だけがそのインデックスを使います。すなわち、WHERE to_tsvector('english', body) @@ 'a & b'はインデックスが使えますが、WHERE to_tsvector(body)

@@ 'a & b'は使えません。これにより、インデックスエントリを作ったときの設定と、同じ設定のときだけインデックスが使われることが保証されます。

他の列によって設定名が指定されたより複雑な式インデックスを作ることができます。例えば、

```
CREATE INDEX pgweb_idx ON pgweb USING GIN (to_tsvector(config_name, body));
```

ここで、config_nameはpgwebテーブルの列です。これによって、各々のインデックスエントリで使用された設定を記録しつつ、同じインデックスの中で異なる設定を混在させることができます。これは、例えば文書の集まりが異なる言語の文書を含む場合に有用です。繰り返しになりますが、インデックスを使うよう考慮されている問い合わせは、合致するように書かれなければなりません。例えば、WHERE to_tsvector(config_name, body) @@ 'a & b'。

インデックスには、列を連結することさえできます。

```
CREATE INDEX pgweb_idx ON pgweb USING GIN (to_tsvector('english', title || ' ' || body));
```

別の方法として、to_tsvectorの出力を保持する別のtsvector列を作る方法があります。この列を元のデータに合わせて自動的に更新し続けるには、格納された生成列を使います。この例では、titleとbodyを連結、coalesceを使って、一つのフィールドがNULLであっても他のフィールドがインデックス付けされることを保証しています。

```
ALTER TABLE pgweb
    ADD COLUMN textsearchable_index_col tsvector
               GENERATED ALWAYS AS (to_tsvector('english', coalesce(title, '') || ' ' ||
 coalesce(body, ''))) STORED;
```

そして、GINインデックスを作って検索速度を上げます。

```
CREATE INDEX textsearch_idx ON pgweb USING GIN (textsearchable_index_col);
```

これで、高速全文検索を実行する準備ができました。

```
SELECT title
FROM pgweb
WHERE textsearchable_index_col @@ to_tsquery('create & table')
ORDER BY last_mod_date DESC
LIMIT 10;
```

別列方式が式インデックスに勝る点の一つは、インデックスを使うために問い合わせの中でテキスト検索設定を明示的に指定する必要がないことです。上の例で示したように、問い合わせはdefault_text_search_configに依存できます。もう一つの利点は、インデックスの合致を検証するためにto_tsvectorを再実行する必要がないのでより高速だという事です。(この点はGINインデックスを使うときよりも、GiSTインデックスを使う場合に重要です。12.9参照。)しかしながら、式インデックス方式はセットアップがより容易で、tsvector表現を明示的に保存する必要がないので、ディスクスペースの消費が少ないです。

12.3. テキスト検索の制御

全文検索を実装するためには、文書からtsvectorを、そしてユーザの問い合わせからtsqueryを作成する関数が存在しなければなりません。また、結果を意味のある順で返す必要があります。そこで、問い合わせとの関連性で文書を比較する関数も必要になってきます。結果を体裁良く表示できることも重要です。PostgreSQLはこれらすべての機能を提供しています。

12.3.1. 文書のパース

PostgreSQLは、文書をtsvectorデータ型に変換するto_tsvector関数を提供しています。

```
to_tsvector([ config regconfig, ] document text) returns tsvector
```

to_tsvectorは、テキスト文書をパースしてトークンにし、トークンを語彙素に変換、文書中の位置とともに語彙素をリストとして持つtsvectorを返します。文書は、指定したものか、あるいはデフォルトのテキスト検索設定にしたがって処理されます。単純な例を示します。

```
SELECT to_tsvector('english', 'a fat  cat sat on a mat - it ate a fat rats');
                  to_tsvector
-----------------------------------------------------
 'ate':9 'cat':3 'fat':2,11 'mat':7 'rat':12 'sat':4
```

上に示す例では、結果のtsvectorで、a、on、itという単語が含まれないこと、ratsという単語がratになっていること、句読点記号-が無視されていることがわかります。

to_tsvector関数は、文書をトークンに分解して、そのトークンに型を割り当てるパーサを内部的に呼び出しています。それぞれのトークンに対して辞書(12.6)のリストが検索されます。ここで、辞書のリストはトークンの型によって異なります。最初の辞書は、トークンを*認識*し、トークンを表現する一つ以上の正規化された*語彙素*を出力します。例えば、ある辞書はratsはratの複数形であることを認識しているので、ratsはratになります。ある単語は*ストップワード*(12.6.1)として認識されます。これは、あまりにも多く出現し検索の役に立たないため、無視されるものです。先の例では、a、on、およびitがそれです。もしリスト中の辞書のどれもがトークンを認識しなければ、そのトークンは無視されます。先の例では、句読点の-がそうです。なぜなら、実際にはそのトークン型(Space symbols)に対して辞書が割り当てられておらず、空白トークンは決してインデックス付けされないことを意味します。パーサ、辞書、そしてどのトークンがインデックス付けされるかという選択は、テキスト検索設定(12.7)によって決められます。同じデータベース中に多くの異なった設定を持つことができ、多くの言語用に定義済の設定が用意されています。先の例では、英語用として、デフォルトのenglish設定を使っています。

関数setweightを使ってtsvectorのエントリに与えられた*重み*のラベルを与えることができます。ここで重みは、A, B, C, Dのどれかの文字です。重みの典型的な使い方は、文書の各部分がどこから来たのかをマークすることです。たとえば、タイトルから来たのか、本文から来たのかなど。後でこの情報は検索結果のランキングに利用できます。

to_tsvector(NULL)はNULLを返すので、NULLになる可能性のある列に対してはcoalesceを使うことをお勧めします。構造化された文書からtsvectorを作るための推奨できる方法を示します。

```
UPDATE tt SET ti =
    setweight(to_tsvector(coalesce(title,'')), 'A')    ||
    setweight(to_tsvector(coalesce(keyword,'')), 'B')  ||
    setweight(to_tsvector(coalesce(abstract,'')), 'C') ||
    setweight(to_tsvector(coalesce(body,'')), 'D');
```

ここでは、完成したtsvectorの語彙素に対して、ラベル付けのためにsetweightを使っています。そして、tsvectorの連結演算子||を使って、ラベルづけされたtsvectorの値をマージします。(詳細は12.4.1を参照してください。)

12.3.2. 問い合わせのパース

PostgreSQLは、問い合わせをtsqueryに変換する関数to_tsquery、plainto_tsquery、phraseto_tsquery、websearch_to_tsqueryを提供しています。to_tsqueryは、plainto_tsqueryとphraseto_tsqueryのいずれよりも多くの機能を提供していますが、入力のチェックはより厳格です。websearch_to_tsqueryは、webサーチエンジンで使われているものに似た別の構文を使うto_tsqueryの簡易バージョンです。

```
to_tsquery([ config regconfig, ] querytext text) returns tsquery
```

to_tsqueryは、querytextからtsqueryとしての値を生成します。querytextは、tsquery演算子& (AND), | (OR)、! (NOT)、<-> (FOLLOWED BY)で区切られる単一のトークンから構成されなければなりません。これらの演算子は括弧でグループ化できます。言い換えると、to_tsqueryの入力は、8.11.2で述べられている一般規則にしたがっていなければなりません。違いは、基本的なtsqueryの入力はトークンの表面的な値を受け取るのに対し、to_tsqueryは指定した、あるいはデフォルトの設定を使ってトークンを語彙素へと正規化し、設定にしたがって、ストップワードであるようなトークンを破棄します。例を示します。

```
SELECT to_tsquery('english', 'The & Fat & Rats');
  to_tsquery
---------------
 'fat' & 'rat'
```

基本的なtsqueryの入力では、各々の語彙素に重みを付加することにより、同じ重みを持つtsvectorの語彙素のみに照合するようにすることができます。例を示します。

```
SELECT to_tsquery('english', 'Fat | Rats:AB');
    to_tsquery
------------------
 'fat' | 'rat':AB
```

また、明示的な前方一致検索のため、*を語彙素に与えることもできます。

```
SELECT to_tsquery('supern:*A & star:A*B');
        to_tsquery
```

```
--------------------------
 'supern':*A & 'star':*AB
```

このような語彙素は、与えられた文字列で始まるtsvector中のどんな単語にも照合するでしょう。

to_tsqueryは、単一引用符で囲まれた語句を受け付けることもできます。これは主に、設定の中にそういった語句を持つ同義語辞書を含んでいるときに有用です。以下の例では、ある同義語の中にsupernovae stars : snという規則が含まれています。

```
SELECT to_tsquery('''supernovae stars'' & !crab');
  to_tsquery
---------------
 'sn' & !'crab'
```

引用符がない場合は、to_tsqueryは、AND、ORあるいはFOLLOWED BY演算子で区切られていないトークンに対して構文エラーを引き起こします。

```
plainto_tsquery([ config regconfig, ] querytext text) returns tsquery
```

plainto_tsqueryは整形されていないテキストquerytextを、tsqueryの値に変換します。テキストはパースされ、to_tsvectorとしてできる限り正規化されます。そして、tsquery演算子& (AND) が存続した単語の間に挿入されます。

例:

```
SELECT plainto_tsquery('english', 'The Fat Rats');
 plainto_tsquery
-----------------
 'fat' & 'rat'
```

plainto_tsqueryは、入力中のtsquery演算子も、重み付けラベルも、前方一致ラベルも認識しないことに注意してください。

```
SELECT plainto_tsquery('english', 'The Fat & Rats:C');
   plainto_tsquery
---------------------
 'fat' & 'rat' & 'c'
```

ここでは、入力中のすべての句読点が破棄されています。

```
phraseto_tsquery([ config regconfig, ] querytext text) returns tsquery
```

phraseto_tsqueryはplainto_tsqueryとほぼ同じ動作をしますが、残った語の間に& (AND) 演算子ではなく、<-> (FOLLOWED BY) 演算子を挿入するところが違います。また、ストップワードを単に無視するのでなく、<->演算子の代わりに<N>演算子を挿入することで、意味のあるものとします。FOLLOWED BY演算子は、単

にすべての語彙素が存在することだけでなく、語彙素の順序についても確認するため、この関数は語彙素の正確な順序について検索するときに役立ちます。

例を示します。

```
SELECT phraseto_tsquery('english', 'The Fat Rats');
 phraseto_tsquery
------------------
 'fat' <-> 'rat'
```

plainto_tsqueryと同じく、phraseto_tsquery関数もその入力内のtsquery演算子、重み付けラベル、前方一致ラベルを認識しません。

```
SELECT phraseto_tsquery('english', 'The Fat & Rats:C');
      phraseto_tsquery
-----------------------------
 'fat' <-> 'rat' <-> 'c'
```

```
websearch_to_tsquery([ config regconfig, ] querytext text) returns tsquery
```

websearch_to_tsqueryは、問い合わせとして、単純で整形されていないテキストが代わりに使えるような構文を使ってquerytextからtsqueryを作り出します。plainto_tsqueryおよびphraseto_tsqueryと違って、ある種の演算子を理解します。更にこの関数は決して構文エラーを引き起こさないので、ユーザ入力をそのまま検索で使用することができます。以下の構文をサポートします。

- 引用符なしのテキスト:引用符の内側にないテキストは、あたかもplainto_tsqueryで処理されたように&演算子で区切られます。
- "引用符内のテキスト":引用符内のテキストは、あたかもphraseto_tsqueryで処理されたように<->で区切られた表現に変換されます。
- OR:単語「or」は|演算子に変換されます。
- -:ダッシュは!演算子に変換されます。

その他の句読点は無視されます。ですので、plainto_tsqueryやphraseto_tsqueryと同様、websearch_to_tsquery関数はtsquery演算子、重み付けラベルや前方一致ラベルを入力として認識しません。

例を示します。

```
SELECT websearch_to_tsquery('english', 'The fat rats');
 websearch_to_tsquery
----------------------
 'fat' & 'rat'
(1 row)

SELECT websearch_to_tsquery('english', '"supernovae stars" -crab');
       websearch_to_tsquery
---------------------------------
```

```
 'supernova' <-> 'star' & !'crab'
(1 row)

SELECT websearch_to_tsquery('english', '"sad cat" or "fat rat"');
       websearch_to_tsquery
-----------------------------------
 'sad' <-> 'cat' | 'fat' <-> 'rat'
(1 row)

SELECT websearch_to_tsquery('english', 'signal -"segmentation fault"');
         websearch_to_tsquery
---------------------------------------
 'signal' & !( 'segment' <-> 'fault' )
(1 row)

SELECT websearch_to_tsquery('english', '""" )( dummy \\ query <->');
 websearch_to_tsquery
----------------------
 'dummi' & 'queri'
(1 row)
```

12.3.3. 検索結果のランキング

ランキングはある問い合わせに対して、どの程度文書が関連しているかを計測しようとするものです。合致している文書が多数あるとき、もっとも関連している文書が最初に表示されるようにするためです。PostgreSQLは、2つの定義済ランキング関数を提供しています。それらは、辞書情報、近接度情報、構造的情報を加味します。すなわち、問い合わせの用語がどの位の頻度で文書に出現するか、文書中でどの程度それらの用語が近接しているか、どの用語が含まれる文書部位がどの程度重要なのかを考慮します。しかし、関連度という概念は曖昧で、用途に強く依存します。異なる用途は、ランキングのために追加の情報を必要とするかも知れません。たとえば、文書の更新時刻など。組み込みのランキング関数は例に過ぎません。利用者の目的に応じて、自分用のランキング関数を作ったり、その結果を追加の情報と組み合わせることができます。

今のところ、二種類のランキング関数が利用可能です。

`ts_rank([ weights float4[], ] vector tsvector, query tsquery [, normalization integer ]) returns float4`

それらの語彙素にマッチした頻度に基づくベクトルのランク。

`ts_rank_cd([ weights float4[], ] vector tsvector, query tsquery [, normalization integer ]) returns float4`

この関数は、1999年の"Information Processing and Management"ジャーナルに掲載されたClarke, Cormack, Tudhopeの"Relevance Ranking for One to Three Term Queries"で述べられている方法で、与えられた文書ベクトルと問い合わせの被覆密度*(cover density)*ランクを計算します。被覆密度は互いにマッチする語彙素の近接度を考慮に入れる点を除いてts_rankのランク付けと似ています。

この関数は、計算を実行するために語彙素の位置情報を必要とします。ですから、tsvector内の「剥き出しの」語彙素は無視します。入力に剥き出しでない語彙素がなければ、結果は0です。(strip関数とtsvector内の位置情報についてのより詳しい情報は12.4.1を参照してください。)

これらの関数では、単語がどの程度ラベル付けに依存するかを、単語ごとに指定する機能がweightsオプションパラメータによって提供されています。重み配列で、それぞれのカテゴリの単語がどの程度重み付けするかを指定します。その順は以下のようになっています。

```
{D-weight, C-weight, B-weight, A-weight}
```

weightsを与えない場合は、次のデフォルト値が使われます。

```
{0.1, 0.2, 0.4, 1.0}
```

重みの典型的な使い方は、文書のタイトルやアブストラクトのような特定の場所にある単語をマーク付けするような使い方です。そうすることにより、文書の本体に比べてそこにある単語がより重要なのか、そうでないのか、扱いを変えることができます。

文書が長ければ、それだけ問い合わせ用語を含む確率が高くなるため、文書のサイズを考慮にいれることは理にかなっています。たとえば、5つの検索語を含む100語の文書は、たぶん5つの検索語を含む1000語の文書よりも関連性が高いでしょう。ランキング関数には、どちらも整数型の正規化オプションがあります。これは、文書の長さがランクに影響を与えるのかどうか、与えるとすればどの程度か、ということを指定します。この整数オプションは、いくつかの挙動を制御するので、ビットマスクになっています。複数の挙動を|で指定できます(例:2|4)。

- 0(デフォルト):文書の長さを無視します
- 1:ランクを(1 + log(文書の長さ))で割ります
- 2:ランクを文書の長さで割ります
- 4:ランクをエクステントの間の調和平均距離で割ります(これはts_rank_cdのみで実装されています)
- 8: ランクを文書中の一意の単語の数で割ります
- 16: ランクをlog(文書中の一意の単語の数)+1 で割ります
- 32: ランクをランク自身+1 で割ります

2以上のフラグビットが指定された場合には、変換は上記に列挙された順に行われます。

これは重要なことですが、ランキング関数はグローバル情報を一切使わないので、時には必要になる1%から100%までの均一な正規化はできません。正規化オプション32(rank/(rank+1))を適用することにより、すべてのランクを0から1に分布させることができます。しかし、もちろんこれは表面的な変更に過ぎません。検索結果のならび順に影響を与えるものではありません。

マッチする順位の高い10位までを選ぶ例を示します。

```
SELECT title, ts_rank_cd(textsearch, query) AS rank
FROM apod, to_tsquery('neutrino|(dark & matter)') query
WHERE query @@ textsearch
ORDER BY rank DESC
LIMIT 10;
                     title                     |   rank
-----------------------------------------------+----------
```

```
 Neutrinos in the Sun                          |      3.1
 The Sudbury Neutrino Detector                 |      2.4
 A MACHO View of Galactic Dark Matter          |  2.01317
 Hot Gas and Dark Matter                       |  1.91171
 The Virgo Cluster: Hot Plasma and Dark Matter |  1.90953
 Rafting for Solar Neutrinos                   |      1.9
 NGC 4650A: Strange Galaxy and Dark Matter     |  1.85774
 Hot Gas and Dark Matter                       |   1.6123
 Ice Fishing for Cosmic Neutrinos              |      1.6
 Weak Lensing Distorts the Universe            | 0.818218
```

同じ例を正規化ランキングを使ったものを示します。

```
SELECT title, ts_rank_cd(textsearch, query, 32 /* rank/(rank+1) */ ) AS rank
FROM apod, to_tsquery('neutrino|(dark & matter)') query
WHERE  query @@ textsearch
ORDER BY rank DESC
LIMIT 10;
                     title                     |        rank
-----------------------------------------------+-------------------
 Neutrinos in the Sun                          | 0.756097569485493
 The Sudbury Neutrino Detector                 | 0.705882361190954
 A MACHO View of Galactic Dark Matter          | 0.668123210574724
 Hot Gas and Dark Matter                       |  0.65655958650282
 The Virgo Cluster: Hot Plasma and Dark Matter | 0.656301290640973
 Rafting for Solar Neutrinos                   | 0.655172410958162
 NGC 4650A: Strange Galaxy and Dark Matter     | 0.650072921219637
 Hot Gas and Dark Matter                       | 0.617195790024749
 Ice Fishing for Cosmic Neutrinos              | 0.615384618911517
 Weak Lensing Distorts the Universe            | 0.450010798361481
```

ランキングは、I/Oに結び付けられていて遅い可能性のある、一致する各文書のtsvectorへのアクセスが必要なので、高価な処理であるかもしれません。不幸なことに、実際の問い合わせでは往々にして大量の検索結果が生じるため、これはほとんど不可避であると言えます。

12.3.4. 結果の強調

検索結果を表示する際には、文書の該当部分を表示し、どの程度問い合わせと関連しているかを示すのが望ましいです。PostgreSQLはこの機能を実装したts_headline関数を提供しています。

```
ts_headline([ config regconfig, ] document text, query tsquery [, options text ]) returns text
```

ts_headlineは、問い合わせと一緒に文書を受け取り、問い合わせが注目した文書中の語句を抜粋して返します。文書をパースするのに使われる設定をconfigで指定することができます。configが省略された場合は、default_text_search_config設定が使われます。

options文字列を指定する場合は、一つ以上のoption=valueのペアをカンマで区切ったものでなければなりません。

- MaxWords, MinWords (整数): この数字を使って見出しの最大の長さと最小の長さを指定します。デフォルトは35と15です。
- ShortWord (整数): この長さか、それ以下の長さの単語は、検索語でない限り、見出しの最初と最後から削除されます。デフォルト値の3は、常用される英語の冠詞を取り除きます。
- HighlightAll (論理値): trueなら文書全体が見出しとして使われ、前の3つのパラメータは無視されます。デフォルトはfalseです。
- MaxFragments (整数): 表示するテキスト断片の最大数です。デフォルト値の0は断片化を起こさない見出しの生成の選択となります。0より大きい場合は断片化を基本とした見出しの生成の選択となります(下記参照)。
- StartSel, StopSel (文字列): 文書中に現れる問い合わせ単語を区切るこの文字列は、他の抜粋される単語と区別されます。デフォルト値は「」と「」であり、HTML出力には適切でしょう。
- FragmentDelimiter (文字列): 複数の断片が表示される時、その断片はこの文字列で区切られます。デフォルトは「 ... 」です。

これらのオプション名は大文字小文字の区別なく認識されます。空白やカンマを含む場合には、文字列の値を二重引用符で括ってください。

断片化を起こさない見出しの生成では、ts_headlineは与えられたqueryとの一致を見つけて、見出しの許される長さ以内でより多くの問い合わせの単語のある一致を優先して一つ選びます。断片化を基本とした見出しの生成では、ts_headlineは問い合わせの一致を見つけて、各一致を最大でMaxWords個の単語からなる「断片」に分割します。このとき、より多くの問い合わせの単語を含む断片を優先します。そして、可能であれば周囲の単語を含むよう断片を「広げます」。それゆえ、問い合わせの一致が文書の長い部分に渡る場合や複数の一致を表示するのが望ましい場合には、断片化を基本としたモードがより有用です。どちらのモードでも、もし問い合わせの一致が特定されなかった場合は、文書中の最初のMinWords個の単語から成る一つの断片が表示されます。

例を示します。

```
SELECT ts_headline('english',
  'The most common type of search
is to find all documents containing given query terms
and return them in order of their similarity to the
query.',
  to_tsquery('english', 'query & similarity'));
                        ts_headline
------------------------------------------------------------
 containing given <b>query</b> terms                       +
 and return them in order of their <b>similarity</b> to the+
 <b>query</b>.

SELECT ts_headline('english',
  'Search terms may occur
many times in a document,
requiring ranking of the search matches to decide which
occurrences to display in the result.',
```

```
  to_tsquery('english', 'search & term'),
  'MaxFragments=10, MaxWords=7, MinWords=3, StartSel=<<, StopSel=>>');
                        ts_headline
------------------------------------------------------------
 <<Search>> <<terms>> may occur                             +
 many times ... ranking of the <<search>> matches to decide
```

ts_headlineは、tsvectorの要約ではなく、元の文書を使います。ですので遅い可能性があり、注意深く使用する必要があります。

12.4. 追加機能

この節では、全文検索に関連する便利な追加の関数と演算子を説明します。

12.4.1. 文書の操作

12.3.1に、もとのテキスト形式の文書がどのようにしてtsvectorに変換されるのか書いてあります。また、PostgreSQLではtsvector形式に変換済の文書を操作する関数と演算子が提供されています。

`tsvector || tsvector`

tsvectorの結合演算子で、2つのベクトルの語彙素と位置情報を合成し、tsvectorを返します。位置と重み付けラベルは、結合では維持されます。右辺のベクトルの位置は左辺のベクトルの一番大きな位置情報のオフセットになります。その結果、この関数の結果は、元の2つの文書文字列を結合したものにto_tsvectorを適用したものとほぼ同じになります。(まったく同じと言うわけではありません。左辺の引数の最後から取り除かれたストップワードは結果に影響を与えないのに対し、テキストの結合が行われた場合は、右辺の引数にある語彙素位置に影響を与えるからです。)

to_tsvectorを適用する前のテキストを結合するよりも、ベクトルを結合することの利点の一つは、文書の異なる部分をパースするために、異なる設定を使うことができることです。なお、setweight関数は与えられたベクトルのすべての語彙素を同じ方法でマーク付けするため、もしも文書に異なる部分に別の重み付けを行いたいなら、結合する前に文書をパースしてsetweightを適用することが必要です。

`setweight(vector tsvector, weight "char") returns tsvector`

setweightは、A, B, C, Dのいずれかの与えられたweightを入力のベクトル中の位置にラベル付けし、そのコピーを返します。(Dは新しいベクトルのデフォルトで、出力する際には表示されません。) これらのラベルはベクトルが結合される際に保存されるので、ランキング関数によって文書中の異なる部分の語を別々に重み付けできます。

なお、重み付けラベルは語彙素ではなく位置に与えられることに注意してください。入力のベクトルから位置が削除されていると、setweightは何もしません。

`length(vector tsvector) returns integer`

ベクトル中に格納されている語彙素の数を返します。

`strip(vector tsvector) returns tsvector`

入力のベクトルと同じ語彙素のリストを持つが、位置と重みの情報が全くないベクトルを返します。その結果は、通常は情報を削除されていないベクトルよりもずっと小さくなりますが、有用性も低くなります。また、tsquery演算子<-> (FOLLOWED BY)は情報を削除した入力とマッチすることはありません。なぜなら語彙素が発生する間の距離を決定できないからです。

tsvectorに関連した関数の完全なリストが表 9.42にあります。

12.4.2. 問い合わせを操作する

12.3.2は、元のテキストがいかにしてtsquery値に変換されるかを解説しています。またPostgreSQLは、tsquery形式に変換済の問い合わせを操作するために使用できる関数と演算子を提供しています。

`tsquery && tsquery`

2つの問い合わせをANDで結合したものを返します。

`tsquery || tsquery`

2つの問い合わせをORで結合したものを返します。

`!! tsquery`

与えられた問い合わせの否定を返します。

`tsquery <-> tsquery`

1番目の問い合わせにマッチし、その直後に2番目の問い合わせにマッチするものを検索する問い合わせを、tsquery演算子<-> (FOLLOWED BY) を使って返します。例を示します。

```
SELECT to_tsquery('fat') <-> to_tsquery('cat | rat');
          ?column?
----------------------------
 'fat' <-> ( 'cat' | 'rat' )
```

`tsquery_phrase(query1 tsquery, query2 tsquery [, distance integer ]) returns tsquery`

1番目の問い合わせにマッチし、その後にちょうどdistance個の語彙素の距離で2番目の問い合わせにマッチするものを検索する問い合わせを、tsquery演算子<N>を使って返します。例を示します。

```
SELECT tsquery_phrase(to_tsquery('fat'), to_tsquery('cat'), 10);
  tsquery_phrase
------------------
 'fat' <10> 'cat'
```

`numnode(query tsquery) returns integer`

tsquery中のノード(語彙素と演算子)の数を返します。この関数は、問い合わせが意味のあるものか(返却値 > 0)、ストップワードだけを含んでいるか(返却値 0)を判断するのに役に立ちます。例を示します。

```
SELECT numnode(plainto_tsquery('the any'));
NOTICE:  query contains only stopword(s) or doesn't contain lexeme(s), ignored
 numnode
---------
       0

SELECT numnode('foo & bar'::tsquery);
 numnode
---------
       3
```

querytree(query tsquery) returns text

インデックス検索の際に使用できるtsqueryの部分を返します。この関数は、たとえばストップワードのみ、あるいは否定語だけのように、インデックス検索できない問い合わせを検出するのに役立ちます。例を示します。

```
SELECT querytree(to_tsquery('defined'));
 querytree
-----------
 'defin'

SELECT querytree(to_tsquery('!defined'));
 querytree
-----------
 T
```

12.4.2.1. 問い合わせの書き換え

ts_rewriteファミリー関数は、与えられたtsqueryから目的の副問い合わせ部分を探し、それを代わりの副問い合わせに置き換えます。本質的には、この操作は、部分文字列置き換えのtsquery版です。置き換え候補と置き換え内容の組は、問い合わせ書き換えルールであると考えることができます。そのような書き換えルールの集合は、強力な検索ツールとなり得ます。たとえば、同義語(たとえばnew york, big apple, nyc, gotham)を使って問い合わせをより広範囲にしたり、逆によりホットな話題にユーザを導くために問い合わせを狭い範囲に絞ったりすることができます。この機能と、同義語辞書(12.6.4)の間には、機能的な重複があります。しかし、再インデックス付けすることなしに、その場で書き換えルールを変更できるのに対し、同義語辞書の更新が有効になるためには、再インデックス付けを行わなければなりません。

ts_rewrite (query tsquery, target tsquery, substitute tsquery) returns tsquery

この形式の ts_rewrite は、単純に単一の書き換えルールを適用します。query中に表れるtargetは、substituteですべて置き換えられます。例を示します。

```
SELECT ts_rewrite('a & b'::tsquery, 'a'::tsquery, 'c'::tsquery);
 ts_rewrite
```

```
------------
 'b' & 'c'
```

ts_rewrite (query tsquery, select text) returns tsquery

この形式のts_rewriteは、開始問い合わせと、テキスト文字列で与えられるSQLのSELECTコマンドを受け取ります。SELECTは、tsquery型の2つの列を出力しなければなりません。現在の問い合わせは、SELECTのそれぞれの結果行中の最初の列の結果(ターゲット)が、2番目の列の結果(置き換え値)に、置き換えられます。例を示します。

```
CREATE TABLE aliases (t tsquery PRIMARY KEY, s tsquery);
INSERT INTO aliases VALUES('a', 'c');

SELECT ts_rewrite('a & b'::tsquery, 'SELECT t,s FROM aliases');
 ts_rewrite
------------
 'b' & 'c'
```

なお、複数の書き換えルールを適用する際は、適用する順番が重要です。ですから、実際には並び替えのキーを適用するORDER BYを問い合わせに入れておくのがよいでしょう。

天文学上の実際的な例を考えてみます。テーブル駆動の書き換えルールを使って、supernovaeを展開します。

```
CREATE TABLE aliases (t tsquery primary key, s tsquery);
INSERT INTO aliases VALUES(to_tsquery('supernovae'), to_tsquery('supernovae|sn'));

SELECT ts_rewrite(to_tsquery('supernovae & crab'), 'SELECT * FROM aliases');
           ts_rewrite
--------------------------------
 'crab' & ( 'supernova' | 'sn' )
```

テーブルを更新するだけで、書き換えルールを変更することができます。

```
UPDATE aliases
SET s = to_tsquery('supernovae|sn & !nebulae')
WHERE t = to_tsquery('supernovae');

SELECT ts_rewrite(to_tsquery('supernovae & crab'), 'SELECT * FROM aliases');
                 ts_rewrite
--------------------------------------------
 'crab' & ( 'supernova' | 'sn' & !'nebula' )
```

書き換えルールが多くなると、書き換えが遅くなる可能性があります。なぜなら、書き換えの対象になるものを求めて、すべてのルールをチェックするからです。明らかに使われないルールを取り除くために、tsqueryの包含演算子を使うことができます。以下の例では、元の問い合わせにマッチするルールだけを選ぶことができます。

```
SELECT ts_rewrite('a & b'::tsquery,
                  'SELECT t,s FROM aliases WHERE ''a & b''::tsquery @> t');
 ts_rewrite
------------
 'b' & 'c'
```

12.4.3. 自動更新のためのトリガ

> **注記**
>
> この節で説明する方法は、12.2.2で説明するように、格納された生成列の使用に置き換えられました。

tsvector形式の文書を格納するために別の列を使う場合、文書の内容を格納した列が変更されたときにtsvectorを格納した列を更新するトリガを作っておく必要があります。この目的のために、2つの組み込み関数を利用できます。自分で関数を書くこともできます。

```
tsvector_update_trigger(tsvector_column_name, config_name, text_column_name [, ... ])
tsvector_update_trigger_column(tsvector_column_name, config_column_name, text_column_name
 [, ... ])
```

これらのトリガ関数は、1つ以上のテキスト列から、CREATE TRIGGERコマンドで指定されたパラメータの制御により、tsvector列を自動的に計算します。使い方の例を示します。

```
CREATE TABLE messages (
    title       text,
    body        text,
    tsv         tsvector
);

CREATE TRIGGER tsvectorupdate BEFORE INSERT OR UPDATE
ON messages FOR EACH ROW EXECUTE FUNCTION
tsvector_update_trigger(tsv, 'pg_catalog.english', title, body);

INSERT INTO messages VALUES('title here', 'the body text is here');

SELECT * FROM messages;
   title    |         body          |            tsv
------------+-----------------------+----------------------------
 title here | the body text is here | 'bodi':4 'text':5 'titl':1

SELECT title, body FROM messages WHERE tsv @@ to_tsquery('title & body');
   title    |         body
------------+-----------------------
```

```
 title here | the body text is here
```

このトリガを作っておくことにより、titleまたはbodyへの変更は、アプリケーションで考慮しなくても自動的にtsvに反映されます。

トリガの最初の引数は更新対象のtsvectorの列名でなければなりません。2番目の引数は、変換を実行する際に使用されるテキスト検索の設定です。tsvector_update_triggerでは、設定の名前は単に2番目のトリガ引数で与えられます。上で示すように、スキーマ修飾されていなければなりません。search_pathの変更がトリガの振る舞いに影響を与えないためです。tsvector_update_trigger_columnでは、2番目のトリガ引数は別のテーブル列の列名です。この列の型はregconfigでなければなりません。この方法により、設定を行単位で変えることができます。残りの引数はテキスト型(text, varchar, charのいずれか)の列の名前です。与えられた順に、文書中に取り込まれます。NULL値はスキップされます(ただし、それ以外の列はインデックス付けされます)。

これらの組み込みトリガの制限事項として、すべての列を同じようにしか扱えないというものがあります。それぞれの列を違うように扱うには — たとえば本文とタイトルの重みを変えるとか —、カスタムトリガを書く必要があります。トリガ言語としてPL/pgSQLを使った例を示します。

```
CREATE FUNCTION messages_trigger() RETURNS trigger AS $$
begin
  new.tsv :=
     setweight(to_tsvector('pg_catalog.english', coalesce(new.title,'')), 'A') ||
     setweight(to_tsvector('pg_catalog.english', coalesce(new.body,'')), 'D');
  return new;
end
$$ LANGUAGE plpgsql;

CREATE TRIGGER tsvectorupdate BEFORE INSERT OR UPDATE
    ON messages FOR EACH ROW EXECUTE FUNCTION messages_trigger();
```

tsvector値をトリガ内で作るときには、設定名を明示的に与えることが重要であることを銘記しておいてください。そうすれば、default_text_search_configが変更されても列の内容は影響を受けません。これを怠ると、ダンプしてリロードすると検索結果が変わってしまうような問題が起きる可能性があります。

12.4.4. 文書の統計情報の収集

ts_stat関数は、設定をチェックしたり、ストップワードの候補を探すのに役立ちます。

```
ts_stat(sqlquery text, [ weights text, ]
        OUT word text, OUT ndoc integer,
        OUT nentry integer) returns setof record
```

sqlqueryは単一のtsvector列を返すSQL問い合わせのテキスト値です。ts_statは問い合わせを実行し、tsvectorデータに含まれる語彙素(単語)各々の統計情報を返します。返却される列は以下のものです。

- word text — 語彙素の値

- ndoc integer — 単語が含まれる文書(tsvector)の数
- nentry integer — 含まれる単語の数

weightsが与えられていたら、その重みを持つものだけがカウントされます。

たとえば、文書中もっとも頻繁に現れる単語の上位10位を探すには以下のようにします。

```
SELECT * FROM ts_stat('SELECT vector FROM apod')
ORDER BY nentry DESC, ndoc DESC, word
LIMIT 10;
```

同じ例で、重みがAかBの単語だけをカウントするには、以下のようにします。

```
SELECT * FROM ts_stat('SELECT vector FROM apod', 'ab')
ORDER BY nentry DESC, ndoc DESC, word
LIMIT 10;
```

12.5. パーサ

テキスト検索パーサは、もとの文書テキストを分割してトークンに変換し、それぞれのトークンの型を識別子する役割を持っています。ここで、可能な型の集合は、パーサ自身が定義します。パーサは文書をまったく変更しないことに注意してください — それは、単に可能な単語の境界を識別するだけです。このような制限があるため、カスタム辞書を作るのに比べ、用途限定のカスタムパーサを作る必要性は少ないです。今のところ、PostgreSQLはたった一つの組み込みパーサを提供しています。これは広い範囲の用途に対して有用であると考えられています。

組み込みのパーサはpg_catalog.defaultというものです。表 12.1に示す23のトークンを理解します。

表12.1 デフォルトパーサのトークン型

別名	説明	例
asciiword	単語、すべてのASCII文字	elephant
word	単語、すべての文字	ma ñ ana
numword	単語、文字、数字	beta1
asciihword	ハイフンでつながれた単語、すべてのASCII	up-to-date
hword	ハイフンでつながれた単語、すべての文字	l ó gico-matem á tica
numhword	ハイフンでつながれた単語、すべての文字、数字	postgresql-beta1
hword_asciipart	ハイフンでつながれた単語の一部、すべての ASCII	postgresql-beta1のpostgresql
hword_part	ハイフンでつながれた単語の一部、すべての文字	l ó gico-matem á ticaのl ó gicoまたはmatem á tica

別名	説明	例
hword_numpart	ハイフンでつながれた単語の文字+数字の部分	postgresql-beta1のbeta1
email	電子メールアドレス	foo@example.com
protocol	プロトコルヘッダー	http://
url	URL	example.com/stuff/index.html
host	ホスト名	example.com
url_path	URL中のパス名	URL中の/stuff/index.html
file	ファイルまたはパス名	URL中でない/usr/local/foo.txt
sfloat	科学技術表記	-1.234e56
float	10進表記	-1.234
int	符号付き整数	-1234
uint	符号なし整数	1234
version	バージョン番号	8.3.0
tag	XMLタグ	<a href="dictionaries.html">
entity	XMLエンティティ	&
blank	空白記号	(他のものに解釈できない空白または句読点)

注記

パーサにとっての「文字」は、データベースのロケールの設定、特にlc_ctypeによって決まります。基本的なASCIIのみを含む単語は、別のトークン型として報告されます。ときには、それらを他と区別することが有用だからです。ヨーロッパのたいていの言語では、wordとasciiwordは、同じように扱われます。

emailはRFC5322で定義されたすべての有効なメールアドレス文字をサポートしません。メールアドレスのユーザ名としてサポートされる英数字以外の文字はピリオド、ダッシュ、アンダースコアのみです。

パーサがテキストの同じ部分から重複したトークンを生成することはあり得ます。たとえば、ハイフン付の単語は、単語全体と、各部分の両方を報告します。例を示します。

```
SELECT alias, description, token FROM ts_debug('foo-bar-beta1');
      alias      |               description                |     token
-----------------+------------------------------------------+---------------
 numhword        | Hyphenated word, letters and digits      | foo-bar-beta1
 hword_asciipart | Hyphenated word part, all ASCII          | foo
 blank           | Space symbols                            | -
 hword_asciipart | Hyphenated word part, all ASCII          | bar
 blank           | Space symbols                            | -
 hword_numpart   | Hyphenated word part, letters and digits | beta1
```

この挙動は好ましいのものです。単語全体と、各々の部分の両方に対して検索ができるからです。初歩的な別の例を示します。

```
SELECT alias, description, token FROM ts_debug('http://example.com/stuff/index.html');
  alias   |  description  |            token
----------+---------------+------------------------------
 protocol | Protocol head | http://
 url      | URL           | example.com/stuff/index.html
 host     | Host          | example.com
 url_path | URL path      | /stuff/index.html
```

12.6. 辞書

辞書は、検索の対象とならない単語(ストップワード)を削除するために使われます。また、同じ単語から派生した異なる形態の単語が照合するようにするために、単語を正規化するためにも使われます。検索の品質を向上するという面以外にも、正規化とストップワードの削除は、tsvector表現の文書のサイズを小さくし、結果として性能を向上させます。正規化は常に言語学的な意味を持つとは限らず、通常は用途の意味論に依存します。

正規化の例を示します。

- 言語学的 — Ispell辞書は入力された単語を正規化された形式に変換しようとします。語幹辞書は単語の終了部を削除します。
- 以下のようなURLが同一のURLに一致するように正規化することができます。
  - http://www.pgsql.ru/db/mw/index.html
  - http://www.pgsql.ru/db/mw/
  - http://www.pgsql.ru/db/../db/mw/index.html
- 色の名前は、16進値に変換できます。例: red, green, blue, magenta -> FF0000, 00FF00, 0000FF, FF00FF
- 数をインデックス付けする際には、可能な範囲を縮小するために、端数を削除することができます。たとえば、もし正規化後に小数点未満2桁を保持するならば、*3.14*159265359、*3.14*15926、*3.14*は同じことになります。

辞書は、トークンを入力し、以下を返すプログラムです。

- 入力が辞書に登録されていれば語彙素の配列(一つのトークンが一つ以上の語彙素を生成する可能性があることに注意してください)
- 元々のトークンを新規のトークンに置き換え、それに続く辞書にその新規トークン渡す場合は、TSL_FILTERフラグセットを伴う単一の語彙素(このような置き換え機能をもつ辞書はフィルタリング辞書と呼ばれます)
- 辞書が入力を認識しないが、ストップワードであることは認識する場合は空の配列
- 辞書が入力トークンを認識しない場合はNULL

PostgreSQLは、多くの言語に定義済の辞書を提供しています。また、カスタムパラメータを使った新しい辞書を作るために使えるテンプレートもいくつかあります。定義済の辞書のテンプレートについては、以下で述べています。今あるテンプレートが適当でないのなら、新しいものを作ることもできます。例は、PostgreSQLの配布物のcontrib/をご覧下さい。

テキスト検索設定は、パーサと、パーサの出力トークンを処理する辞書の集合を結び付けます。パーサが返却する各々のトークン型に対して、設定で辞書のリストを指定します。パーサがあるトークン型を見つけると、ある辞書が単語を認識するまでリスト中の辞書が順番に調べられます。ストップワードであるか、あるいはどの辞書もトークンを認識しない場合はそれは捨てられ、インデックス付けや検索の対象となりません。通常、非NULLを返す最初の辞書の出力が結果を決めることになり、他の残りの辞書は参照されません。しかし、フィルタリング辞書は与えられたワードを変更し、それを続く辞書へ渡すことができます。

辞書をリストする一般的な方法は、まずもっとも範囲の狭い、特定用途向の辞書を配置し、次にもっと一般的な辞書を置き、最後にSnowball語幹処理やsimple辞書のような、すべてを認識する非常に一般的な辞書を置くことです。たとえば、天文学向の検索では(astro_en設定)では、asciiword (ASCII単語)型を天文学用語の同義語辞書、一般的な英語辞書、そしてSnowball英語語幹辞書に結び付けることができます。

```
ALTER TEXT SEARCH CONFIGURATION astro_en
    ADD MAPPING FOR asciiword WITH astrosyn, english_ispell, english_stem;
```

フィルタリング辞書は、リスト中の好きな場所へ配置できます。(役に立たなくなるリストの最後を除きます。)フィルタリング辞書は、後続の辞書の処理を単純化するために、一部の文字の正規化を行うのに有用です。例えば、フィルタリング辞書はunaccentモジュールで実施される様な、アクセント記号が付与された文字からアクセント記号を取り除くのに使用することができます。

12.6.1. ストップワード

ストップワードは、ほとんどすべての文書に現れるような非常に一般的で、ほかのものと同じようには扱う価値のない単語です。ですから、全文検索の際には無視して構いません。たとえば、すべての英語のテキストはaやtheのような単語を含んでおり、インデックスの中にそれらを入れても役に立ちません。しかし、ストップワードはtsvector中の位置に影響を与えるので、結局ランキングにも影響があります。

```
SELECT to_tsvector('english', 'in the list of stop words');
        to_tsvector
----------------------------
 'list':3 'stop':5 'word':6
```

位置1, 2, 4は、ストップワードのために失われています。ストップワードの有無により、文書のために計算されたランクは非常に影響を受けます。

```
SELECT ts_rank_cd (to_tsvector('english', 'in the list of stop words'), to_tsquery('list &
 stop'));
 ts_rank_cd
------------
       0.05

SELECT ts_rank_cd (to_tsvector('english', 'list stop words'), to_tsquery('list & stop'));
 ts_rank_cd
------------
        0.1
```

ストップワードをどのように扱うかは、特定の辞書に任されています。例えば、ispell辞書はまず単語を正規化し、そして、ストップワードのリストを検索します。一方、Snowball語幹抽出はまずストップワードのリストを検査します。動作が異なる理由は、ノイズが紛れ込む可能性を減らすことです。

12.6.2. simple辞書

simple辞書テンプレートは、入力トークンを小文字に変換し、ストップワードのファイルに対してチェックすることによって動作します。もしファイルの中にあれば、空の配列が返却され、そのトークンは捨てられます。そうでないときは、小文字形式の単語が正規化された語彙素として返却されます。別の方法としては、ストップワードではないものは、認識できないものとすることもできます。そうすることにより、それらをリスト中の次の辞書に渡すことができます。

simpleテンプレートを使った辞書定義の例を示します。

```
CREATE TEXT SEARCH DICTIONARY public.simple_dict (
    TEMPLATE = pg_catalog.simple,
    STOPWORDS = english
);
```

ここで、englishは、ストップワードファイルのベースネームです。ファイルのフルネームは、$SHAREDIR/tsearch_data/english.stopです。$SHAREDIRは、PostgreSQLインストール先の共有データディレクトリです。これは、よく/usr/local/share/postgresqlに置いてあります。(良くわからない場合はpg_config --sharedirを使ってください)。ファイル形式は、単に1行ごとに単語を書くだけです。空行と、後方の空白は無視されます。大文字は小文字に変換されます。このファイルの内容に関する処理はこれだけです。

これで辞書のテストができます。

```
SELECT ts_lexize('public.simple_dict', 'YeS');
 ts_lexize
-----------
 {yes}

SELECT ts_lexize('public.simple_dict', 'The');
 ts_lexize
-----------
 {}
```

また、ストップワードファイルの中に見つからないときに、小文字に変換した単語を返す代わりに、NULLを返すことを選ぶこともできます。この挙動は、辞書のAcceptパラメータをfalseに設定することで選択されます。さらに例を続けます。

```
ALTER TEXT SEARCH DICTIONARY public.simple_dict ( Accept = false );

SELECT ts_lexize('public.simple_dict', 'YeS');
 ts_lexize
-----------
```

```
SELECT ts_lexize('public.simple_dict', 'The');
 ts_lexize
-----------
 {}
```

デフォルト設定のAccept = trueでは、simple辞書は、辞書リストの最後に置かなければ意味がありません。なぜなら、後続の辞書にトークンを渡すことがないからです。逆にAccept = falseは、後続の辞書が少なくとも一つはあるときに意味があります。

> **注意**
>
> ほとんどの辞書の形式は、ストップワードファイルのように設定ファイルに依存します。これらのファイルは必ずUTF-8エンコーディングにしてください。サーバのエンコーディングがUTF-8でない場合は、サーバに読み込まれる際に実際のデータベースエンコーディングに変換されます。

> **注意**
>
> 通常、辞書はデータベースセッションの中で最初に使われる際に、一度だけ読み込まれます。辞書を変更し、現在使われているセッションの中で新しい内容が読み込まれるようにしたい場合は、その辞書に対してALTER TEXT SEARCH DICTIONARYを発行してください。これは実際にはどんなパラメータ値をも変更しない「ダミー」の更新でよいです。

12.6.3. 同義語辞書

この辞書テンプレートは、単語を同義語に置き換える辞書を作るために使われます。語句はサポートされていません(そのためには類語テンプレート(12.6.4)を使ってください)。同義語辞書は、言語学的な問題、たとえば、英語語幹辞書が「Paris」という単語を「pari」に縮小してしまうのを防ぎます。Paris parisという行を同義語辞書に登録し、english_stem辞書の前に置くようにするだけでよいのです。下記はその例です。

```
SELECT * FROM ts_debug('english', 'Paris');
   alias   |   description   | token |  dictionaries  |  dictionary  | lexemes
-----------+-----------------+-------+----------------+--------------+---------
 asciiword | Word, all ASCII | Paris | {english_stem} | english_stem | {pari}

CREATE TEXT SEARCH DICTIONARY my_synonym (
    TEMPLATE = synonym,
    SYNONYMS = my_synonyms
);

ALTER TEXT SEARCH CONFIGURATION english
    ALTER MAPPING FOR asciiword
```

```
    WITH my_synonym, english_stem;

SELECT * FROM ts_debug('english', 'Paris');
   alias   |   description   | token |       dictionaries        | dictionary | lexemes
-----------+-----------------+-------+---------------------------+------------+---------
 asciiword | Word, all ASCII | Paris | {my_synonym,english_stem} | my_synonym | {paris}
```

synonymテンプレートに必要なパラメータはSYNONYMSだけで、その設定ファイルのベースネームです — 上の例ではmy_synonymsです。ファイルのフルネームは、$SHAREDIR/tsearch_data/my_synonyms.synとなります(ここで$SHAREDIRは、PostgreSQLをインストールした際の、共有データディレクトリです)。ファイルの形式は、置き換え対象の1単語につき1行で、単語には空白で区切られた同義語が後に続きます。空行、後方の空白は無視されます。

synonymテンプレートはまた、CaseSensitiveというオプションパラメータを持っており、デフォルトはfalseです。CaseSensitiveがfalseの時は、同義語辞書内の単語は入力トークンと同様に小文字に変換されます。trueの時は、単語とトークンは小文字に変換されずそのまま比較されます。

アスタリスク(*)は設定ファイル中の同義語の最後に付与することができます。これは同義語を接頭語とすることを意味します。アスタリスクは、エントリがto_tsvector()で使用される場合には無視されますが、to_tsquery()で使用される場合、結果は前方一致を伴った問い合わせになるでしょう。(詳しくは12.3.2を見てください。) 例えば、$SHAREDIR/tsearch_data/synonym_sample.synに以下の様なエントリをもっていたとします。

```
postgres        pgsql
postgresql      pgsql
postgre pgsql
gogle   googl
indices index*
```

この場合、次のような結果を得ることになります。

```
mydb=# CREATE TEXT SEARCH DICTIONARY syn (template=synonym, synonyms='synonym_sample');
mydb=# SELECT ts_lexize('syn', 'indices');
 ts_lexize
-----------
 {index}
(1 row)

mydb=# CREATE TEXT SEARCH CONFIGURATION tst (copy=simple);
mydb=# ALTER TEXT SEARCH CONFIGURATION tst ALTER MAPPING FOR asciiword WITH syn;
mydb=# SELECT to_tsvector('tst', 'indices');
 to_tsvector
-------------
 'index':1
(1 row)

mydb=# SELECT to_tsquery('tst', 'indices');
```

```
 to_tsquery
------------
 'index':*
(1 row)

mydb=# SELECT 'indexes are very useful'::tsvector;
            tsvector
--------------------------------
 'are' 'indexes' 'useful' 'very'
(1 row)

mydb=# SELECT 'indexes are very useful'::tsvector @@ to_tsquery('tst', 'indices');
 ?column?
----------
 t
(1 row)
```

12.6.4. 類語辞書

類語辞書(TZと略されることがあります)は、単語と語句の関係情報を集めたものです。つまり、広義用語(BT)、狭義用語(NT)、優先用語、非優先用語、関連用語などです。

基本的には、類語辞書は、非優先用語を優先用語に置き換え、オプションで元の用語もインデックス付けのため保存します。PostgreSQLの現在の類語辞書の実装は、同義語辞書を拡張し、語句のサポートを追加したものです。類語辞書は、以下のようなフォーマットの設定ファイルを必要とします。

```
# this is a comment
sample word(s) : indexed word(s)
more sample word(s) : more indexed word(s)
...
```

ここで、コロン(:)は、語句とその置き換え対象の区切りです。

類語辞書は、副辞書(辞書設定で指定します)を、一致する語句をチェックする前に入力テキストを正規化するために使います。副辞書はただ一つだけ選べます。副辞書が単語を認識できない場合はエラーが報告されます。その場合は、その単語の利用を止めるか、副辞書にそのことを教えなければなりません。アスタリスク(*)をインデックス付けされた単語の先頭に置くことにより、副辞書の適用をスキップできます。しかしながら、すべてのサンプルの単語は、副辞書に認識されなければなりません。

複数の類語が照合するときは、類語辞書はもっとも長いものを選びます。そして、語句は、最後の定義を使って分解されます。

特定のストップワードを副辞書に認識するように指定することはできません。その代わり、ストップワードが出現する位置を?でマークします。たとえば、aとtheが副辞書によればストップワードだったとします。

```
? one ? two : swsw
```

は、a one the twoとthe one a twoに照合します。そして、両方ともswswに置き換えられます。

類語辞書は語句を認識することができるので、状態を記憶してパーサと連携を保たなければなりません。類語辞書は、この機能を使って次の単語を引き続き処理するのか、単語の蓄積を止めるのかを決定します。類語辞書の設定は注意深く行わなければなりません。たとえば、類語辞書がasciiwordトークンだけを扱うようになっている場合、one 7のような類語辞書の定義は、トークン型uintが類語辞書にアサインされていないので動きません。

> **注意**
>
> 類語辞書はインデックス付けの際に利用されるので、類語辞書を設定変更すると、再インデックス付けが必要になります。他のほとんどの辞書では、ストップワードを追加あるいは削除するような小さな変更は、インデックス付けを必要としません。

12.6.4.1. 類語設定

新しい類語辞書を定義するには、thesaurusテンプレートを使います。例を示します。

```
CREATE TEXT SEARCH DICTIONARY thesaurus_simple (
    TEMPLATE = thesaurus,
    DictFile = mythesaurus,
    Dictionary = pg_catalog.english_stem
);
```

ここで、

- thesaurus_simpleは新しい辞書の名前です。
- mythesaurusは、類語設定ファイルのベースネームです。(フルパスは、$SHAREDIR/tsearch_data/mythesaurus.thsとなります。ここで、$SHAREDIRはインストール時の共有データディレクトリです。)
- 類語正規化で使用するpg_catalog.english_stemは副辞書です(ここでは、Snowball英語語幹辞書)。副辞書にはそれ用の設定(たとえばストップワード)があることに注意してください。ここではそれは表示していません。

これで、類語辞書thesaurus_simpleを、設定中の希望のトークンにバインドすることができるようになります。例を示します。

```
ALTER TEXT SEARCH CONFIGURATION russian
    ALTER MAPPING FOR asciiword, asciihword, hword_asciipart
    WITH thesaurus_simple;
```

12.6.4.2. 類語の例

天文学の単語の組合わせを含む単純な天文学用のthesaurus_astro類語を考えます。

```
supernovae stars : sn
```

```
crab nebulae : crab
```

以下で辞書を作り、トークン型を天文学類語辞書と英語の語幹辞書に結び付けます。

```
CREATE TEXT SEARCH DICTIONARY thesaurus_astro (
    TEMPLATE = thesaurus,
    DictFile = thesaurus_astro,
    Dictionary = english_stem
);

ALTER TEXT SEARCH CONFIGURATION russian
    ALTER MAPPING FOR asciiword, asciihword, hword_asciipart
    WITH thesaurus_astro, english_stem;
```

さあ、これでどのように動くか試せます。ts_lexizeは類語をテストする目的にはあまり有用ではありません。なぜなら、それは入力を単一のトークンとして扱うからです。その代わりに、plainto_tsqueryとto_tsvectorを使って入力文字列を複数のトークンに分解します。

```
SELECT plainto_tsquery('supernova star');
 plainto_tsquery
-----------------
 'sn'

SELECT to_tsvector('supernova star');
 to_tsvector
-------------
 'sn':1
```

原則として、引数を引用符で囲めばto_tsqueryが使えます。

```
SELECT to_tsquery('''supernova star''');
 to_tsquery
------------
 'sn'
```

english_stem語幹辞書を同義語辞書の定義時に指定したので、supernova starがthesaurus_astro中のsupernovae starsに照合していることに注意してください。語幹処理がeとsを削除しています。

置き換え後の語句とオリジナルの語句の両方をインデックス付けするには、定義の右項にオリジナルを追加するだけで良いです。

```
supernovae stars : sn supernovae stars

SELECT plainto_tsquery('supernova star');
       plainto_tsquery
-----------------------------
 'sn' & 'supernova' & 'star'
```

12.6.5. Ispell辞書

Ispell辞書テンプレートは、形態論辞書を提供します。これによって、言語学的に多様な単語の形態を同じ語彙素に変換することができます。たとえば、英語Ispell辞書は、検索語bankの語形変化と活用変化、たとえばbanking, banked, banks, banks', bank'sに照合します。

PostgreSQLの標準配布には、Ispellの設定ファイルは含まれていません。多くの言語用の辞書がIspell[1]で入手できます。また、より新しい辞書のフォーマットもサポートされています — MySpell[2](OO < 2.0.1)とHunspell[3](OO >= 2.0.2)。多数の辞書のリストが OpenOffice Wiki[4]で入手できます。

Ispell辞書を作るには、以下の手順を実行します。

- 辞書の設定ファイルをダウンロードします。OpenOfficeの拡張ファイルは拡張子.oxtがあります。.affファイルと.dicファイルを抽出し、拡張子を.affixと.dictに変更する必要があります。一部の辞書ファイルでは、以下のコマンドで文字をUTF-8の符号化に変換する必要もあります(例えば、ノルウェー語の辞書では次のようになります)。

```
iconv -f ISO_8859-1 -t UTF-8 -o nn_no.affix nn_NO.aff
iconv -f ISO_8859-1 -t UTF-8 -o nn_no.dict nn_NO.dic
```

- ファイルを$SHAREDIR/tsearch_dataディレクトリにコピーします。
- 以下のコマンドでファイルをPostgreSQLにロードします。

```
CREATE TEXT SEARCH DICTIONARY english_hunspell (
    TEMPLATE = ispell,
    DictFile = en_us,
    AffFile = en_us,
    Stopwords = english);
```

ここで、DictFile, AffFile, およびStopWordsは、辞書のベースネーム、接辞ファイル、ストップワードファイルを指定します。ストップワードファイルは、上で説明したsimple辞書と同じ形式です。ほかのファイルの形式はここでは説明されませんが、上にあげたウェブサイトに説明があります。

Ispell辞書は通常限られた数の単語を認識します。ですので、なんでも認識できるSnowball辞書のような、より適用範囲の広い辞書による後処理が必要です。

Ispellの.affixファイルは次のような構造になっています。

```
prefixes
flag *A:
    .           >   RE      # As in enter > reenter
suffixes
flag T:
```

[1] https://www.cs.hmc.edu/~geoff/ispell.html

[2] https://en.wikipedia.org/wiki/MySpell

[3] https://sourceforge.net/projects/hunspell/

[4] https://wiki.openoffice.org/wiki/Dictionaries

```
    E           >   ST      # As in late > latest
    [^AEIOU]Y   >   -Y,IEST # As in dirty > dirtiest
    [AEIOU]Y    >   EST     # As in gray > grayest
    [^EY]       >   EST     # As in small > smallest
```

そして、.dictファイルは次のような構造になっています。

```
lapse/ADGRS
lard/DGRS
large/PRTY
lark/MRS
```

.dictファイルのフォーマットは次の通りです。

```
basic_form/affix_class_name
```

.affixファイルで、すべてのaffix(接辞)フラグは次のフォーマットで記述されています。

```
condition > [-stripping_letters,] adding_affix
```

ここで、condition(条件)は正規表現の形式と同じような形式になります。[...]および[^...]のグループ化を使うことができます。例えば[AEIOU]Yは、単語の最後の文字が"y"で、その前の文字が"a"、"e"、"i"、"o"、"u"のいずれかであることを意味します。[^EY]は最後の文字が"e"でも"y"でもないことを意味します。

Ispell辞書を使って複合語を分割することができます。これは優れた機能です。接辞ファイルは、複合語形式の候補になる辞書中の単語に印を付けるcompoundwords controlled文を使う特別なフラグを指定しなければならないことに注意してください。

```
compoundwords  controlled z
```

ノルウェー語の例をいくつか示します。

```
SELECT ts_lexize('norwegian_ispell', 'overbuljongterningpakkmesterassistent');
   {over,buljong,terning,pakk,mester,assistent}
SELECT ts_lexize('norwegian_ispell', 'sjokoladefabrikk');
   {sjokoladefabrikk,sjokolade,fabrikk}
```

MySpellのフォーマットはHunspellの部分集合です。Hunspellの.affixファイルは以下のような構造になっています。

```
PFX A Y 1
PFX A   0     re         .
SFX T N 4
SFX T   0     st         e
SFX T   y     iest       [^aeiou]y
SFX T   0     est        [aeiou]y
```

```
SFX T   0     est        [^ey]
```

接辞(affix)クラスの1行目はヘッダです。接辞ルールのフィールドはヘッダの後に列挙されます。

- パラメータ名（PFXまたはSFX）
- フラグ（接辞クラスの名前）
- 単語の先頭（接頭辞）から、あるいは終わり（接尾辞）から文字を削除する
- 接辞を追加する
- 正規表現の形式と類似の形式の条件

.dictファイルはIspellの.dictファイルと同じように見えます。

```
larder/M
lardy/RT
large/RSPMYT
largehearted
```

注記

MySpellは複合語をサポートしていません。Hunspellは複合語の高度なサポートを提供しています。いまのところ、PostgreSQLはHunspellの基本的な複合語操作しかサポートしていません。

12.6.6. Snowball辞書

Snowball辞書テンプレートは、有名な「英語用のポーターの語幹アルゴリズム」を発明したMartin Porterのプロジェクトに基づいています。Snowballは今では多くの言語用の語幹アルゴリズムを提供しています(詳細はSnowballのサイト[5]を参照してください)。各々のアルゴリズムにより、その言語において単語の共通部分を取りだし、基本部もしくは語幹の綴りに縮退させることができます。Snowball辞書には、どの語幹処理を使うかを識別する言語パラメータが必須で、加えて、オプションで無視すべき単語のリストを保持するストップワードファイルを指定することもできます。(PostgreSQLの標準的なストップワードファイルもまたSnowball projectから提供されています。) たとえば、以下と同じ組み込みの定義があります。

```
CREATE TEXT SEARCH DICTIONARY english_stem (
    TEMPLATE = snowball,
    Language = english,
    StopWords = english
);
```

ストップワードファイルの形式はすでに説明されているものと同じです。

Snowball辞書は、単純化できるかどうかに関係なく、すべての単語を認識するので、辞書リストの最後に置く必要があります。他の辞書の前に置くのは意味がありません。Snowball辞書は決してトークンを次の辞書に渡さないからです。

[5] https://snowballstem.org/

12.7. 設定例

テキスト検索設定は、文書をtsvectorに変換する必要なすべてのオプションを指定します。すなわち、テキストをトークンに分解するパーサ、そしてトークンを語彙素に変換する辞書です。to_tsvectorまたはto_tsqueryを呼び出すたびに、処理を進めるためにテキスト検索設定が必要になります。設定パラメータのdefault_text_search_configは、デフォルトの設定を指定します。これは、明示的な設定が省略されたときにテキスト検索関数が使用します。postgresql.confに設定するか、個々のセッションでSETコマンドを使って設定できます。

既定のテキスト検索設定がいくつか利用できます。また、カスタム設定を作るのも容易です。テキスト検索オブジェクトを管理する機能を実現するために、SQLコマンドが一通り用意されています。テキスト検索オブジェクトに関する情報を表示するpsqlコマンドもいくつか用意されています(12.10)。

例として、組み込みのenglish設定のコピーを用いて、新しいpg設定を作ります。

```
CREATE TEXT SEARCH CONFIGURATION public.pg ( COPY = pg_catalog.english );
```

PostgreSQL固有の同義語リストを使い、それを$SHAREDIR/tsearch_data/pg_dict.synに格納します。ファイルの内容は以下のようになります。

```
postgres    pg
pgsql       pg
postgresql  pg
```

同義語辞書を次のように定義します。

```
CREATE TEXT SEARCH DICTIONARY pg_dict (
    TEMPLATE = synonym,
    SYNONYMS = pg_dict
);
```

次に、Ispell辞書のenglish_ispellを登録します。これにはそれ自身の設定があります。

```
CREATE TEXT SEARCH DICTIONARY english_ispell (
    TEMPLATE = ispell,
    DictFile = english,
    AffFile = english,
    StopWords = english
);
```

ここで、pg設定に単語用のマッピングを設定します。

```
ALTER TEXT SEARCH CONFIGURATION pg
    ALTER MAPPING FOR asciiword, asciihword, hword_asciipart,
                      word, hword, hword_part
    WITH pg_dict, english_ispell, english_stem;
```

組み込み設定が扱っているいくつかのトークンに関しては、インデックス付けと検索に扱わないことにします。

```
ALTER TEXT SEARCH CONFIGURATION pg
    DROP MAPPING FOR email, url, url_path, sfloat, float;
```

これでここまで作った設定を試すことができます。

```
SELECT * FROM ts_debug('public.pg', '
PostgreSQL, the highly scalable, SQL compliant, open source object-relational
database management system, is now undergoing beta testing of the next
version of our software.
');
```

次に、セッションの中で新しい設定を使うようにします。この設定は、publicスキーマの中に作られています。

```
=> \dF
   List of text search configurations
 Schema  | Name | Description
---------+------+-------------
 public  | pg   |

SET default_text_search_config = 'public.pg';
SET

SHOW default_text_search_config;
 default_text_search_config
----------------------------
 public.pg
```

12.8. テキスト検索のテストとデバッグ

カスタムテキスト検索設定の挙動は複雑になりがちで、結果として混乱を招くことになります。この節では、テキスト検索オブジェクトのテストの際に役に立つ関数を説明します。完全な設定でテストすることも、パーサと辞書を別々にテストすることも可能です。

12.8.1. 設定のテスト

ts_debug関数により、テキスト検索設定の容易なテストができます。

```
ts_debug([ config regconfig, ] document text,
         OUT alias text,
         OUT description text,
         OUT token text,
```

```
        OUT dictionaries regdictionary[],
        OUT dictionary regdictionary,
        OUT lexemes text[])
        returns setof record
```

ts_debugは、パーサが生成し、設定された辞書が処理したdocumentのすべてのトークンの情報を表示します。その際、configで指定した設定が使われます。引数が省略されるとdefault_text_search_configが使われます。

ts_debugは、パーサが認識したテキスト中のトークンを1行につき一つ返します。返却される列は以下です。

- alias text —トークン型の短縮名
- description text —トークン型の説明
- token text —トークンテキスト
- dictionaries regdictionary[] — 設定によってこのトークン型用に選択された辞書
- dictionary regdictionary —トークンを認識した辞書。もし認識した辞書がなければ NULL
- lexemes text[] —トークンを認識した辞書が生成した語彙素。もしどの辞書も認識しなければNULL。空の配列({})が返った場合は、ストップワードとして認識されたことを示す

簡単な例を示します。

```
SELECT * FROM ts_debug('english', 'a fat  cat sat on a mat - it ate a fat rats');
   alias   |   description   | token |  dictionaries  |  dictionary  | lexemes
-----------+-----------------+-------+----------------+--------------+---------
 asciiword | Word, all ASCII | a     | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | fat   | {english_stem} | english_stem | {fat}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | cat   | {english_stem} | english_stem | {cat}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | sat   | {english_stem} | english_stem | {sat}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | on    | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | a     | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | mat   | {english_stem} | english_stem | {mat}
 blank     | Space symbols   |       | {}             |              |
 blank     | Space symbols   | -     | {}             |              |
 asciiword | Word, all ASCII | it    | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | ate   | {english_stem} | english_stem | {ate}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | a     | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | fat   | {english_stem} | english_stem | {fat}
 blank     | Space symbols   |       | {}             |              |
```

```
 asciiword | Word, all ASCII | rats  | {english_stem} | english_stem | {rat}
```

もう少し高度なデモをお見せするために、まず英語用のpublic.english設定と、Ispell辞書を作ります。

```
CREATE TEXT SEARCH CONFIGURATION public.english ( COPY = pg_catalog.english );

CREATE TEXT SEARCH DICTIONARY english_ispell (
    TEMPLATE = ispell,
    DictFile = english,
    AffFile = english,
    StopWords = english
);

ALTER TEXT SEARCH CONFIGURATION public.english
   ALTER MAPPING FOR asciiword WITH english_ispell, english_stem;
```

```
SELECT * FROM ts_debug('public.english', 'The Brightest supernovaes');
   alias   |   description   |    token    |         dictionaries          |   dictionary   |
 lexemes
-----------+-----------------+-------------+-------------------------------+----------------
+-------------
 asciiword | Word, all ASCII | The         | {english_ispell,english_stem} | english_ispell | {}
 blank     | Space symbols   |             | {}                            |                |
 asciiword | Word, all ASCII | Brightest   | {english_ispell,english_stem} | english_ispell |
 {bright}
 blank     | Space symbols   |             | {}                            |                |
 asciiword | Word, all ASCII | supernovaes | {english_ispell,english_stem} | english_stem   |
 {supernova}
```

この例では、単語Brightestは、ASCII word (別名はasciiword)として認識されています。このトークン型のための辞書リストはenglish_ispell とenglish_stemです。この単語はenglish_ispellに認識され、名詞brightへと縮退されています。単語supernovaesはenglish_ispell辞書には認識されず、次の辞書に渡され、幸い認識されました(実際には、english_stemはSnowball辞書で、何でも認識します。それで、この辞書は辞書リストの最後に置かれているわけです)。

単語Theは、english_ispell辞書によってストップワード(12.6.1)として認識されており、インデックス付けされません。空白も捨てられます。この設定では空白に関する辞書が提供されていないからです。

明示的に見たい列を指定することにより、出力の幅を減らすことができます。

```
SELECT alias, token, dictionary, lexemes
FROM ts_debug('public.english', 'The Brightest supernovaes');
   alias   |    token    |   dictionary   |   lexemes
-----------+-------------+----------------+-------------
 asciiword | The         | english_ispell | {}
 blank     |             |                |
 asciiword | Brightest   | english_ispell | {bright}
```

```
 blank     |             |                |
 asciiword | supernovaes | english_stem   | {supernova}
```

12.8.2. パーサのテスト

次にあげた関数により、テキスト検索パーサを直接テストすることができます。

```
ts_parse(parser_name text, document text,
         OUT tokid integer, OUT token text) returns setof record
ts_parse(parser_oid oid, document text,
         OUT tokid integer, OUT token text) returns setof record
```

ts_parseは与えられたdocumentをパースし、パーサが生成したトークンを1行に1個もつ一連のレコードを返します。それぞれのレコードには、割り当てられたトークン型を示すtokidと、テキストのトークンであるtokenが含まれます。例を示します。

```
SELECT * FROM ts_parse('default', '123 - a number');
 tokid | token
-------+--------
    22 | 123
    12 |
    12 | -
     1 | a
    12 |
     1 | number
```

```
ts_token_type(parser_name text, OUT tokid integer,
              OUT alias text, OUT description text) returns setof record
ts_token_type(parser_oid oid, OUT tokid integer,
              OUT alias text, OUT description text) returns setof record
```

ts_token_typeは、指定したパーサが認識できるトークン型を記述したテーブルを返します。各々のトークン型に対し、パーサがトークン型をラベル付けするのに使用する整数tokid、設定コマンド中のトークンの名前であるalias、簡単な説明であるdescriptionが含まれます。例を示します。

```
SELECT * FROM ts_token_type('default');
 tokid |      alias      |               description
-------+-----------------+------------------------------------------
     1 | asciiword       | Word, all ASCII
     2 | word            | Word, all letters
     3 | numword         | Word, letters and digits
     4 | email           | Email address
     5 | url             | URL
     6 | host            | Host
```

```
     7 | sfloat          | Scientific notation
     8 | version         | Version number
     9 | hword_numpart   | Hyphenated word part, letters and digits
    10 | hword_part      | Hyphenated word part, all letters
    11 | hword_asciipart | Hyphenated word part, all ASCII
    12 | blank           | Space symbols
    13 | tag             | XML tag
    14 | protocol        | Protocol head
    15 | numhword        | Hyphenated word, letters and digits
    16 | asciihword      | Hyphenated word, all ASCII
    17 | hword           | Hyphenated word, all letters
    18 | url_path        | URL path
    19 | file            | File or path name
    20 | float           | Decimal notation
    21 | int             | Signed integer
    22 | uint            | Unsigned integer
    23 | entity          | XML entity
```

12.8.3. 辞書のテスト

ts_lexize関数は辞書のテストを支援します。

```
ts_lexize(dict regdictionary, token text) returns text[]
```

ts_lexizeは、入力tokenが辞書に認識されれば語彙素の配列を返します。辞書に認識され、それがストップワードである場合には空の配列を返します。認識されなければNULLを返します。

例:

```
SELECT ts_lexize('english_stem', 'stars');
 ts_lexize
-----------
 {star}

SELECT ts_lexize('english_stem', 'a');
 ts_lexize
-----------
 {}
```

注記

ts_lexize関数には、テキストではなく単一のトークンを与えます。これを間違えると次のようになります。

```
SELECT ts_lexize('thesaurus_astro', 'supernovae stars') is null;
```

```
 ?column?
----------
 t
```

類語辞書thesaurus_astroは語句supernovae starsを認識しますが、ts_lexizeはしません。なぜなら、入力をテキストではなく、単一のトークンとして扱うからです。類語辞書をテストするには、plainto_tsqueryまたはto_tsvectorを使ってください。例を示します。

```
SELECT plainto_tsquery('supernovae stars');
 plainto_tsquery
-----------------
 'sn'
```

12.9. GINおよびGiSTインデックス種類

全文検索を高速化するために、2種類のインデックスが使えます。全文検索のためにインデックスが必須だと言うわけではないことを言っておかなければなりませんが、日常的に検索される列には、インデックスを使った方が良いでしょう。

```
CREATE INDEX name ON table USING GIN (column);
```

GIN (Generalized Inverted Index)インデックスを作ります。columnはtsvector型でなければなりません。

```
CREATE INDEX name ON table USING GIST (column [ { DEFAULT | tsvector_ops } (siglen =
number) ] );
```

GiST (Generalized Search Tree)インデックスを作ります。columnは tsvector または tsquery 型です。オプションの整数パラメータsiglenは署名の長さをバイト単位で決定します(詳細は以下を参照してください)。

GINインデックスの方がより好ましいテキスト検索インデックス形式です。転置インデックスなので、マッチした位置の圧縮されたリストと合わせて各単語(語彙素)へのインデックスエントリを含みます。複数単語での検索は最初のマッチを見つけることができ、その後、追加の単語がない行を削除するのにインデックスを使えます。GINインデックスはtsvector値の単語(語彙素)のみを格納しており、重み付けラベルは格納していません。そのため、重みを含む問い合わせを使う場合には、テーブル行の再検査が必要です。

GiSTインデックスは、非可逆です。つまり、インデックスは間違った結果を返すかも知れないので、間違った結果を排除するために、テーブルの行をチェックすることが必要です。(PostgreSQLはこの処理が必要とされた時に自動的に行います。) GiSTインデックスが非可逆なのは、インデックス中の各文書が固定長の署名によって表現されているからです。署名のバイト単位の長さはオプションの整数のパラメータsiglenの値で決まります。(siglenが指定されない場合)デフォルトの署名の長さは124バイトで、最大の署名の長さは2024バイトです。署名は、各々の単語をハッシュして単一なビットにして、これらのビットをnビットの文書署名にORし、nビットの列中のビットにすることで実現されています。2つの単語が同じビット位置を生成すると、間違った一致が起こります。問い合わせ対象のすべての単語が照合すると(それが正しいか間違っているかは別として)、その照合が正しいものかどうかテーブルの行を取得して調べなければなりません。長い署名で

は、インデックスはより大きくなってしまいますが、(インデックスのより小さな部分とより少ないヒープページを走査することで)検索がより正確になります。

GiSTインデックスはカバリングにできます、すなわちINCLUDE句を使えます。列には、GiST演算子クラスを持たないデータ型をINCLUDEで含めることができます。含まれる属性は圧縮されずに格納されます。

非可逆性は、間違った照合によるテーブルからの不必要なデータ取得のため、性能を劣化させます。テーブルへのランダムアクセスは遅いので、GiSTインデックスの有用性は制限されています。誤った照合がどの位あるかという可能性はいくつか要因によりますが、とりわけユニークな単語の数に依存します。ですから、辞書を使ってユニークな単語の数を減らすことをお勧めします。

GINインデックスの構築時間はmaintenance_work_memを増やすことによってしばしば改善することができることに注意してください。一方GiSTインデックスの構築時間にはあまりそのパラメータは効きません。

大きなデータをパーティショニングし、GIN、GiSTインデックスを適切に使うことによってオンラインの更新を伴いながら、非常に高速な検索を実現することができます。パーティショニングは、以下のどちらかの方法でデータベースレベルで実現できます。(1)テーブルの継承を使う。(2)文書を複数のサーバに分散させ、外部の検索結果を集約する。たとえば外部データアクセスを使います。2の方法は、ランキング関数がローカルな情報しか使わないため可能です。

12.10. psqlサポート

psqlでテキスト検索設定オブジェクトに関する情報は、コマンドの集まりを使って取得できます。

```
\dF{d,p,t}[+] [PATTERN]
```

オプションの+により、より詳細な情報を生成します。

オプションパラメータのPATTERNはテキスト検索オブジェクトの名前にすることができます。オプションとしてスキーマ修飾することができます。PATTERNが省略されると、すべての可視的なオブジェクトが表示されます。PATTERNは正規表現を与えることができ、さらにスキーマとオブジェクト名に対して別々のパターンを与えることができます。次の例はこれを説明するものです。

```
=> \dF *fulltext*
       List of text search configurations
 Schema |  Name        | Description
--------+--------------+-------------
 public | fulltext_cfg |
```

```
=> \dF *.fulltext*
       List of text search configurations
 Schema   |  Name        | Description
----------+--------------+-------------
 fulltext | fulltext_cfg |
 public   | fulltext_cfg |
```

以下のコマンドが利用できます。

\dF[+] [PATTERN]

テキスト検索設定を表示します(+追加で詳細表示)。

```
=> \dF russian
            List of text search configurations
   Schema   |  Name   |            Description
------------+---------+-----------------------------------
 pg_catalog | russian | configuration for russian language

=> \dF+ russian
Text search configuration "pg_catalog.russian"
Parser: "pg_catalog.default"
      Token      | Dictionaries
-----------------+--------------
 asciihword      | english_stem
 asciiword       | english_stem
 email           | simple
 file            | simple
 float           | simple
 host            | simple
 hword           | russian_stem
 hword_asciipart | english_stem
 hword_numpart   | simple
 hword_part      | russian_stem
 int             | simple
 numhword        | simple
 numword         | simple
 sfloat          | simple
 uint            | simple
 url             | simple
 url_path        | simple
 version         | simple
 word            | russian_stem
```

\dFd[+] [PATTERN]

テキスト検索辞書を表示します(+追加で詳細表示)。

```
=> \dFd
                              List of text search dictionaries
   Schema   |      Name       |                        Description
------------+-----------------+-----------------------------------------------------------
 pg_catalog | arabic_stem     | snowball stemmer for arabic language
 pg_catalog | danish_stem     | snowball stemmer for danish language
 pg_catalog | dutch_stem      | snowball stemmer for dutch language
 pg_catalog | english_stem    | snowball stemmer for english language
```

```
 pg_catalog | finnish_stem    | snowball stemmer for finnish language
 pg_catalog | french_stem     | snowball stemmer for french language
 pg_catalog | german_stem     | snowball stemmer for german language
 pg_catalog | greek_stem      | snowball stemmer for greek language
 pg_catalog | hungarian_stem  | snowball stemmer for hungarian language
 pg_catalog | indonesian_stem | snowball stemmer for indonesian language
 pg_catalog | irish_stem      | snowball stemmer for irish language
 pg_catalog | italian_stem    | snowball stemmer for italian language
 pg_catalog | lithuanian_stem | snowball stemmer for lithuanian language
 pg_catalog | nepali_stem     | snowball stemmer for nepali language
 pg_catalog | norwegian_stem  | snowball stemmer for norwegian language
 pg_catalog | portuguese_stem | snowball stemmer for portuguese language
 pg_catalog | romanian_stem   | snowball stemmer for romanian language
 pg_catalog | russian_stem    | snowball stemmer for russian language
 pg_catalog | simple          | simple dictionary: just lower case and check for stopword
 pg_catalog | spanish_stem    | snowball stemmer for spanish language
 pg_catalog | swedish_stem    | snowball stemmer for swedish language
 pg_catalog | tamil_stem      | snowball stemmer for tamil language
 pg_catalog | turkish_stem    | snowball stemmer for turkish language
```

\dFp[+] [PATTERN]

テキスト検索パーサを表示します(+追加で詳細表示)。

```
=> \dFp
        List of text search parsers
   Schema   |  Name   |     Description
------------+---------+---------------------
 pg_catalog | default | default word parser
=> \dFp+
    Text search parser "pg_catalog.default"
     Method      |    Function    | Description
-----------------+----------------+-------------
 Start parse     | prsd_start     |
 Get next token  | prsd_nexttoken |
 End parse       | prsd_end       |
 Get headline    | prsd_headline  |
 Get token types | prsd_lextype   |

        Token types for parser "pg_catalog.default"
   Token name    |               Description
-----------------+-----------------------------------------
 asciihword      | Hyphenated word, all ASCII
 asciiword       | Word, all ASCII
 blank           | Space symbols
 email           | Email address
 entity          | XML entity
```

```
 file            | File or path name
 float           | Decimal notation
 host            | Host
 hword           | Hyphenated word, all letters
 hword_asciipart | Hyphenated word part, all ASCII
 hword_numpart   | Hyphenated word part, letters and digits
 hword_part      | Hyphenated word part, all letters
 int             | Signed integer
 numhword        | Hyphenated word, letters and digits
 numword         | Word, letters and digits
 protocol        | Protocol head
 sfloat          | Scientific notation
 tag             | XML tag
 uint            | Unsigned integer
 url             | URL
 url_path        | URL path
 version         | Version number
 word            | Word, all letters
(23 rows)
```

\dFt[+] [PATTERN]

テキスト検索テンプレートを表示します(+追加で詳細表示)。

```
=> \dFt
                            List of text search templates
   Schema   |   Name    |                        Description
------------+-----------+------------------------------------------------------------
 pg_catalog | ispell    | ispell dictionary
 pg_catalog | simple    | simple dictionary: just lower case and check for stopword
 pg_catalog | snowball  | snowball stemmer
 pg_catalog | synonym   | synonym dictionary: replace word by its synonym
 pg_catalog | thesaurus | thesaurus dictionary: phrase by phrase substitution
```

12.11. 制限事項

PostgreSQLのテキスト検索機能の制限事項は以下です。

- 各々の語彙素の長さは2キロバイト未満でなければなりません
- tsvectorの長さ (語彙素 + 位置)は1Mバイト未満でなければなりません
- 語彙素の数は2^{64}未満でなければなりません
- tsvectorの位置量は、0より大きくかつ16,383以下でなければなりません
- tsquery演算子<N>におけるマッチの距離は16,384より大きくすることはできません
- 語彙素の位置情報は256以下でなければなりません
- tsquery中のノードの数(語彙素 + 演算子)は32,768未満でなければなりません

比較対象として述べておくと、PostgreSQL8.1 のドキュメントは10,441のユニークな単語を含み、全部の単語数は335,420で、最頻出の単語「postgresql」は655の文書中に6,127回出現しました。

別の例です — PostgreSQLメーリングリストのアーカイブは910,989のユニークな単語を含み、461,020のメッセージ中に57,491,343の語彙素がありました。

第13章 同時実行制御

本章では同時に2つ以上のセッションが同じデータにアクセスしようとした場合、PostgreSQLデータベースシステムがどう振舞うかについて説明します。このような状況でデータの整合性を確実に保つ一方、全てのセッションに対して効果的なアクセスを許可するようにすることが目的です。データベースアプリケーションを開発する方は、本章で扱われている内容を熟知していなければなりません。

13.1. 序文

PostgreSQLは、データへの同時アクセスを管理するために高度な開発者向けツール群を提供します。内部的に、データ一貫性は多版方式（多版型同時実行制御MVCC）を使用して管理されています。つまり、処理の基礎となっているデータの現在の状態にかかわらず、各SQL文は遡ったある時点におけるスナップショット（データベースバージョン）を参照する、というものです。これは、同時に並行しているトランザクションが同じ行を更新することによって引き起こす整合性を欠いたデータの参照を文にさせないようにし、それぞれのデータベースセッションに対して*トランザクションの分離*を提供します。MVCCは、マルチユーザ環境で理想的な性能を得るために、伝統的なデータベースシステムで行われるようなロック手法を避けることで、ロックの競合を最小化します。

ロックではなく同時実行制御のMVCCモデルを使用する主な利点は、MVCCでは問い合わせ（読み込み）ロックの獲得と、書き込みロックの獲得が競合しないことです。したがって、読み込みは書き込みを絶対にブロックしませんし、書き込みも読み込みをブロックすることがありません。革新的な*シリアライザブルスナップショット分離* (SSI)レベルの使用を通した最も厳密なトランザクションの分離レベルを提供する場合にもPostgreSQLはこれの保証を維持します。

全般的に完全なトランザクションの分離を必要とせず、明示的に競合する点を管理することを望むアプリケーションのために、PostgreSQLではテーブルレベルおよび行レベルのロック機能も使用可能です。とはいえ、MVCCを適切に使用すると通常ロックよりも性能が向上します。さらに、アプリケーションが定義した勧告的ロックは単一トランザクションに拘束されないロックの獲得機構を提供します。

13.2. トランザクションの分離

SQLの標準規格では、トランザクションの分離について4つのレベルを定義しています。標準規格で定義されているもののうち最も厳密なものはシリアライザブルです。1セットのシリアライザブルなトランザクションを同時実行した場合には、ある順番でひとつずつそれらを実行した場合と同じ結果となることが保証されるものです。本文で詳しく述べます。他の3レベルは、同時実行しているトランザクション間の相互作用に起因する、各レベルでは発生してはならない現象面に基づき定義されます。標準規格のシリアライザブルの定義では、このレベルではこれらの現象が起こりえないと述べています。(これは驚くことではありません。トランザクションの効果がひとつずつ実行された場合と一貫性を持たなければならないとしたら、相互作用によって発生した現象はどうやっても見つけ出すことはできないでしょう。)

各種レベルにおける禁止される現象を以下に示します。

ダーティリード

> 同時に実行されている他のトランザクションが書き込んで未だコミットしていないデータを読み込んでしまう。

反復不能読み取り

トランザクションが、以前読み込んだデータを再度読み込み、そのデータが(最初の読み込みの後にコミットした)別のトランザクションによって更新されたことを見出す。

ファントムリード

トランザクションが、複数行のある集合を返す検索条件で問い合わせを再実行した時、別のトランザクションがコミットしてしまったために、同じ検索条件で問い合わせを実行しても異なる結果を得てしまう。

直列化異常

複数のトランザクションを正常にコミットした結果が、それらのトランザクションを1つずつあらゆる可能な順序で実行する場合とは一貫性がない。

標準SQLおよびPostgreSQLで実装されているトランザクション分離レベルを表 13.1に示します。

表13.1 トランザクション分離レベル

分離レベル	ダーティリード	反復不能読み取り	ファントムリード	直列化異常
リードアンコミッティド	許容されるが、PostgreSQLでは発生しない	可能性あり	可能性あり	可能性あり
リードコミッティド	安全	可能性あり	可能性あり	可能性あり
リピータブルリード	安全	安全	許容されるが、PostgreSQLでは発生しない	可能性あり
シリアライザブル	安全	安全	安全	安全

PostgreSQLでは、4つの標準トランザクション分離レベルを全て要求することができます。しかし、内部的には3つの分離レベルしか実装されていません。つまり、PostgreSQLのリードアンコミッティドモードは、リードコミッティドのように動作します。これは、PostgreSQLの多版型同時実行制御という仕組みに標準の分離レベルを関連付ける実際的な方法がこれしかないからです。

このテーブルはまた、PostgreSQLのリピータブルリードの実装ではファントムリードが起こらないことを示しています。より厳密な動作をすることは標準SQLでも許されています。つまり、この4つの分離レベルでは、発生してはならない事象のみが定義され、発生しなければならない事象は定義されていません。利用可能な分離レベルでの動作については後で詳細に説明します。

トランザクションのトランザクション分離レベルを設定するにはSET TRANSACTIONコマンドを使用してください。

重要

いくつかのPostgreSQLデータ型と関数はトランザクションの振る舞いに関して特別の規則があります。特に、シーケンスに対しての変更は(従い、serialを使用して宣言された列のカウンタ)は直後に全ての他のトランザクションで可視となり、変更を行ったトランザクションが中止されるとロールバックはできません。9.17および8.1.4を参照してください。

13.2.1. リードコミッティド分離レベル

PostgreSQLではリードコミッティドがデフォルトの分離レベルです。トランザクションがこの分離レベルを使用すると、SELECT問い合わせ（FOR UPDATE/SHARE句を伴わない）はその問い合わせが実行される直前までにコミットされたデータのみを参照し、まだコミットされていないデータや、その問い合わせの実行中に別の同時実行トランザクションがコミットした更新は参照しません。結果として、SELECT問い合わせはその問い合わせが実行を開始した時点のデータベースのスナップショットを参照することになります。しかしSELECT文は、自分自身のトランザクション内で実行され更新された結果はたとえまだコミットされていなくても参照します。単一のトランザクション内であっても、SELECT文を2回連続して発行した場合、最初のSELECT文が開始した後で2番目のSELECT文が開始する前に他のトランザクションが更新をコミットすると、最初とその次に発行したSELECT問い合わせは異なるデータを参照してしまうことにも注意してください。

UPDATE、DELETE、SELECT FOR UPDATE、およびSELECT FOR SHAREコマンドは対象行を検索する際にSELECTコマンドと同じように振舞います。これらのコマンドは、問い合わせが開始された時点で既にコミットされた対象行のみを検出します。しかし、その対象行は、検出されるまでに、同時実行中の他のトランザクションによって、既に更新（もしくは削除あるいはロック）されてしまっているかもしれません。このような場合更新されるべき処理は、最初の更新トランザクションが（それがまだ進行中の場合）コミットもしくはロールバックするのを待ちます。最初の更新処理がロールバックされるとその結果は無視されて、2番目の更新処理で元々検出した行の更新を続行することができます。最初の更新処理がコミットされると、2番目の更新処理では、最初の更新処理により行が削除された場合はその行を無視します。行が削除されなかった時の更新処理は、最初のコミットで更新された行に適用されます。コマンドの検索条件（WHERE句）は、更新された行がまだその検索条件に一致するかどうかの確認のため再評価されます。検索条件と一致している場合、2番目の更新処理は、更新された行を使用して処理を開始します。SELECT FOR UPDATEおよびSELECT FOR SHAREの場合、ロックされクライアントに返されるのは、更新されるバージョンの行であることを意味します。

ON CONFLICT DO UPDATE句のあるINSERTは同じように動作します。リードコミッティドモードでは、挿入を提案された各行は挿入または更新されます。無関係なエラーが発生しなければ、それら2つの結果のうち1つが保証されます。まだその結果がINSERTに対して可視になっていない他のトランザクションに起因する競合では、慣習的な意味でそのコマンドに対して可視のバージョンの行が存在しないにも関わらず、UPDATE句がその行に対して動作します。

ON CONFLICT DO NOTHING句のあるINSERTでは、INSERTのスナップショットに対してその結果が可視になっていない他のトランザクションの結果のために、行の挿入が処理されないかもしれません。ここでも、問題になるのはリードコミッティドモードのときだけです。

このような仕組みにより、更新コマンドが、一貫しないスナップショットを参照する可能性があります。つまり、自分が更新を試みているのと同じ行に対して同時に更新するコマンドの結果は参照できますが、それらのコマンドがデータベース中の他の行に対して更新した結果は参照しません。このような動作をするために複雑な検索条件を含む問い合わせにリードコミッティドモードを使用することは適切ではありません。しかし、より単純な検索条件の場合、このモードの使用が適しています。例えば、銀行の残高を更新する以下のようなトランザクションを考えてみます。

```
BEGIN;
UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 12345;
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 7534;
COMMIT;
```

2つのこのようなトランザクションが同時に口座番号12345の残高を変更しようとした場合、口座の行の更新されたバージョンに対して2番目のトランザクションが開始されることは明らかに望まれるところです。各コマ

ンドは事前に決定していた行に対してのみ処理を行うため、行の更新されたバージョンを見せることによって、何の問題となる不整合も引き起こしません。

より複雑な使用法により、リードコミッティドモードでは好ましくない結果を生成する場合があります。例えば、別のコマンドによってDELETEの制約条件からデータが同時に追加・削除される場合を考えます。例えば、websiteは2行のテーブルで、website.hitsの値には9と10があるとします。

```
BEGIN;
UPDATE website SET hits = hits + 1;

-- 別のセッションから DELETE FROM website WHERE hits = 10; を実行します
COMMIT;
```

UPDATEの前後の両方でwebsite.hits = 10の行があるにも関わらず、DELETEは何もしません。なぜこうなるのかと言うと、更新前の行値9は読み飛ばされ、またUPDATEが完了してDELETEがロックを獲得した時点では、新しい行値は10ではなく11となり、判定条件にもはやマッチしなくなっているからです。

リードコミッティドモードは、それぞれのコマンドをその時点までにコミットされた全てのトランザクションを含む新規スナップショットを伴って開始するので、同一のトランザクション内でそれに続くコマンドは、いかなる場合でもコミットされた同時実行トランザクションの結果を参照します。上記問題の要点は単一のコマンドがデータベースの厳密に一貫性のある見え方を見るか否かです。

リードコミッティドモードで提供されている部分的なトランザクション分離は、多くのアプリケーションでは適切です。またこのモードは高速で、使い方も簡単ですが、全ての場合に対して充分ではありません。複雑な問い合わせや更新を行うアプリケーションは、リードコミッティドモードが提供する以上のより厳正なデータベースの厳密に一貫性のある見え方を必要とします。

13.2.2. リピータブルリード分離レベル

リピータブルリード分離レベルは、トランザクションが開始される前までにコミットされたデータのみを参照します。コミットされていないデータや、そのトランザクションの実行中に別のトランザクションでコミットされた変更を参照しません。(しかし、その問い合わせと同じトランザクション内で行われた過去の更新は、まだコミットされていませんが、参照します。) これはSQLの標準規格で求められるものよりもより強く保証するもので、直列化異常を除いて、表 13.1で述べている現象をすべて防ぎます。上で述べたように、これは標準規格によって明示的に許容されているもので、標準ではそれぞれの分離レベルが提供しなくてはならない最小の保護のみが示されています。

リピータブルリードのトランザクション内の問い合わせは、トランザクション内の現在の文の開始時点ではなく、トランザクションの最初のトランザクション制御以外の文の開始時点のスナップショットを見る、という点でこのレベルはリードコミッティドと異なります。従って、単一トランザクション内の連続するSELECT文は、同じデータを参照します。つまり、自身のトランザクションが開始した後にコミットされた他のトランザクションによる変更を参照しません。

このレベルを使ったアプリケーションでは、直列化の失敗によるトランザクションの再実行に備えておく必要があります。

UPDATE、DELETE、SELECT FOR UPDATE、およびSELECT FOR SHAREコマンドでは、SELECTと同じように対象行を検索します。これらのコマンドでは、トランザクションが開始された時点で既にコミットされている対象行のみを検出します。しかし、その対象行は、検出されるまでに、同時実行中の他のトランザクションによって、既

に更新(もしくは削除あるいはロック)されている可能性があります。このような場合、リピータブルリードトランザクションは、最初の更新トランザクションが(それらがまだ進行中の場合)コミットもしくはロールバックするのを待ちます。最初の更新処理がロールバックされると、その結果は無視され、リピータブルリードトランザクションでは元々検出した行の更新を続行することができます。しかし、最初の更新処理がコミット(かつ、単にロックされるだけでなく、実際に行が更新または削除)されると、リピータブルリードトランザクションでは、以下のようなメッセージを出力してロールバックを行います。

```
ERROR:  could not serialize access due to concurrent update
```

これは、リピータブルリードトランザクションでは、トランザクションが開始された後に別のトランザクションによって更新されたデータは変更またはロックすることができないためです。

アプリケーションがこのエラーメッセージを受け取った場合、現在のトランザクションを中止して、トランザクション全体を始めからやり直されなければなりません。2回目では、トランザクションはコミットされた変更を含めてデータベースの最初の状態とみなすので、新しいバージョンの行を新しいトランザクションにおける更新の始点としても、論理的矛盾は起こりません。

再実行する必要があるかもしれないのは、更新トランザクションのみです。読み込み専用トランザクションでは直列化の衝突は決して起こりません。

リピータブルリードモードでは、全てのトランザクションがデータベースの一貫した不変のビューの状態を参照することが保証されます。しかし、このビューは常にいくつかの同じレベルの同時実行トランザクションの直列(一度に一つずつの)実行と一貫性を持つとは限りません。例えば、このレベルの読み取りのみのトランザクションは、バッチが完了したことを示すために更新された制御レコードを参照することができますが、制御レコードのより以前のバージョンを読み取るため、論理的にそのバッチの一部となる詳細なレコードの1つを参照することはできません。この分離レベルで実行するトランザクションによりビジネスルールを強制しようとすることは、競合するトランザクションをブロックするために注意深く明示的なロックを持たないと、正確に動作しないことが多くあります。

リピータブルリード分離レベルは、学術的なデータベースの文献や他のデータベース製品のいくつかではスナップショット分離として知られる技術を用いて実装されています。同時実行性の面で劣る伝統的なロック技術を使うシステムと比較すると振舞いや性能の違いが観察されるかもしれません。他のシステムでは、リピータブルリードとスナップショット分離を異なる振舞いをする別の分離レベルとして提供しているかもしれません。2つの技術を区別する許容される現象は、標準SQLが制定されるまではデータベース研究者により定式化されておらず、この文書の範囲を超えます。詳細な取り扱いについては[berenson95]を参照してください。

> **注記**
>
> PostgreSQL version 9.1より前まででは、シリアライザブル分離レベルの要求はここで説明した通りの動作をそのまま提供していました。以前のシリアライザブルの動作を維持するためには、リピータブルリードを要求しなければならなくなりました。

13.2.3. シリアライザブル分離レベル

シリアライザブル分離レベルは、最も厳しいトランザクションの分離性を提供します。このレベルではトランザクションが同時にではなく、次から次へと、あたかも順に実行されているように逐次的なトランザクションの実行を全てのコミットされたトランザクションに対しエミュレートします。しかし、このレベルを使ったアプリ

ケーションでは、リピータブルリードレベルと同様に、直列化の失敗によるトランザクションの再実行に備えておく必要があります。実際、この分離レベルは、(ある時点で)逐次実行可能なすべてのトランザクションにおいて、シリアライザブルトランザクションの同時実行の組が一貫性のないような振る舞いをしていないか監視することを除き、リピータブルリードと全く同じ動きをします。この監視では、リピータブルリードが示すものを越えてブロックすることはありませんが、監視によりいくらかのオーバーヘッドがあり、直列化異常を引き起こすような状態の検知は、直列化の失敗を引き起こすでしょう。

例えば、以下の初期データを持つmytabというテーブルを考えてみます。

```
class | value
-------+-------
     1 |    10
     1 |    20
     2 |   100
     2 |   200
```

ここでシリアライザブルトランザクションAが以下を計算し、

```
SELECT SUM(value) FROM mytab WHERE class = 1;
```

そして、valueにその結果(30)を、class = 2の行として新たに挿入したとします。同時にシリアライザブルトランザクションBが以下を計算し、

```
SELECT SUM(value) FROM mytab WHERE class = 2;
```

その結果300を得、そして、この結果をclass = 1の新たな行として挿入したとします。その後、両方のトランザクションがコミットを試みます。もし一方の処理がリピータブルリード分離レベルで実行していれば、両方のコミットが許されるでしょう。しかし、この結果と一貫する実行順序が存在しないため、シリアライザブルトランザクションを使用した場合は、ひとつのトランザクションがコミットを許され、他方は次のメッセージとともにロールバックされることになります。

```
ERROR:  could not serialize access due to read/write dependencies among transactions
```

この理由は、もしAがBよりも前に実行されていた場合、Bの総和は300ではなく330と計算され、また同様に逆の順序で実行されたとすればAで計算される総和が異なる結果になるからです。

異常を防止するためにシリアライザブルトランザクションを使用するのであれば、恒久的なユーザテーブルから読み取られたいかなるデータも、それを読んだトランザクションがコミットされるまで有効とは認められない点は重要です。このことは読み取り専用トランザクションにも当てはまりますが、遅延可能な読み取り専用トランザクション内で読み込まれたデータは例外で、読み込まれてすぐに有効とみなされます。なぜなら、遅延可能なトランザクションはすべてのデータを読み込む前にこのような問題がないことを保証されているスナップショットを取得できるまで待機するからです。それ以外の全ての場合において、後に中止されたトランザクション内で読み込まれた結果をアプリケーションは信用してはならず、アプリケーションはトランザクションが成功するまで再試行すべきです。

真の直列性を保証するためにPostgreSQLでは、述語ロックを使います。述語ロックでは、トランザクションが最初に実行されたとしたら、それによる書き込みが同時実行トランザクションによる読み取り結果にいつ影響を及ぼしたかの決定を可能にするロックを保持します。PostgreSQLでは、これらのロックはブロッキング

を引き起こさないため、デッドロックの要因とならないものです。それらは、同時実行中のシリアライザブルトランザクションが、直列化異常につながる組み合わせであることを識別しフラグを立てることに使用されます。それとは対照的に、データの一貫性を保証したいリードコミッティドあるいはリピータブルリードトランザクションでは、テーブル全体のロック（そのテーブルを使用しようとしている他のユーザをブロックするかもしれません）を必要とするかもしれませんし、あるいは、他のトランザクションをブロックするだけでなくディスク・アクセスを引き起こすSELECT FOR UPDATEあるいはSELECT FOR SHAREを使用するかもしれません。

PostgreSQLの述語ロックは、他のほとんどのデータベースシステムと同様、トランザクションによって実際にアクセスされたデータを元にしています。これらは、pg_locksシステムビューにmodeがSIReadLockのデータとして現れます。問い合わせの実行期間中に獲得される個別のロックは、問い合わせが使用した計画に依存するでしょう。また、ロックを追跡するために使用されるメモリの消耗を防ぐために、トランザクションの過程において、多数のよりきめの細かいロック（例えばタプル・ロック）が結合されて、より少数のよりきめの粗いロック（例えばページ・ロック）になるかもしれません。直列化異常につながるような競合が継続して生じないことを検知すると、READ ONLYトランザクションは、それが完了する前にSIReadロックを解放できるかもしれません。実際、READ ONLYトランザクションは、よく開始時点でその事実を確証し、どんな述語ロックもとらないこともあります。SERIALIZABLE READ ONLY DEFERRABLEトランザクションを明示的に要求した場合には、この事実を確証できるまでブロックします。（これは、シリアライザブルトランザクションはブロックするけれども、リピータブルリードトランザクションはブロックしない唯一のケースです。）他方で、SIReadロックは、しばしば読み取りと書き込みが重なっているトランザクションが完了するまで、トランザクションのコミットが終わっても保持される必要があります。

シリアライザブルトランザクションの一貫した使用は開発を単純化することができます。正常にコミットされた同時実行のシリアライザブルトランザクションのどんな集合も、あたかもそれらが一度に一つずつ実行されたのと同じ結果になることが保証されるので、単独で実行されたときに単一トランザクションが正しく動作するよう書かれていると実証できるなら、他のトランザクションが何をしているかの情報が全く無くとも、複数シリアライザブルトランザクションが混在する中で正しく動作するかコミットに成功しないかであると確証を持つことができます。この技術を使用する環境では、直列化の失敗(常にSQLSTATE値が'40001'で返る)を扱うための、汎用的な手段を持っていることが重要です。なぜなら、どのトランザクションが読み取り/書き込みの依存性に影響し、直列化異常を防ぐためにロールバックさせる必要があるかということを、正確に予測することは非常に困難だからです。読み取り/書き込みの依存性を監視したり、直列化異常で終了したトランザクションを再起動することはコストがかかります。しかしながら、このコストと、明示的なロックとSELECT FOR UPDATEまたはSELECT FOR SHAREを使用したブロッキングとで比較検討すると、シリアライザブルトランザクションはいくつかの環境において最良な実行を選択することになります。

PostgreSQLのシリアライザブルトランザクション分離レベルが同じ結果を生む実行順序があることを証明できるときだけ、同時のトランザクションのコミットを許すとはいえ、本当のシリアル実行では起こらないエラーが常に防げるわけではありません。特に、たとえそのキーが生成されていないことを挿入しようとする前に明示的に調査した後でも重複しているシリアライザブルトランザクションとの競合が原因で一意性制約違反を見ることになる可能性があります。これは潜在的に競合しているキーを挿入する全てのシリアライザブルトランザクションで確実に挿入できるかどうか最初に明示的に調査することで防ぐことができます。例えば、ユーザに新しいキーを聞いてからまずselectでそれがすでに存在しているか確かめるアプリケーション、もしくは存在している中で一番大きなキーを選択しそれに1を足すことで新しいキーを生成するアプリケーションを想像してみてください。もしいくつかのシリアライザブルトランザクションがこのプロトコルに沿わずに直接新しいキーを挿入すれば、たとえそれがシリアル実行の同時トランザクションでは起こりえないケースでも一意性制約違反が報告されることになります。

同時実行制御のためにシリアライザブルトランザクションを使用する場合、最適な性能のためには、以下の問題を考慮すべきです。

- 可能であればトランザクションをREAD ONLYとして宣言してください。
- もし必要ならばコネクションプールを使用して、活動中の接続数を制御してください。これは常に重要な性能上の考慮点ですが、シリアライザブルトランザクションを使用した多忙なシステムにおいては、特に重要になる可能性があります。
- 完全性のために必要とされる以上のものを1つのトランザクションに入れないようにしてください。
- 必要以上に長く「トランザクション内で待機状態」で接続したまま放置しておかないようにしてください。長引くセッションを自動的に切断するために、設定パラメータidle_in_transaction_session_timeoutを使うことができます。
- シリアライザブルトランザクションにより自動的に提供される保護により、不必要な、明示的なロック、SELECT FOR UPDATEおよびSELECT FOR SHAREを取り除いてください。
- 述語ロックのテーブルがメモリ不足になると、複数のページレベルの述語ロックを単一のリレーションレベルの述語ロックへと結合するようシステムが強いられ、直列化失敗の発生割合が増加する恐れがあります。これは、max_pred_locks_per_transaction、max_pred_locks_per_relation、max_pred_locks_per_pageのいずれか、あるいは、すべてを増やすことにより回避することができます。
- シーケンシャルスキャンは常にリレーションレベルでの述語ロックを必要とします。これによって、直列化失敗の頻度が増える可能性があります。random_page_costを縮小および(または)cpu_tuple_costを増加することによりインデックススキャンの使用を促進することは有用かもしれません。トランザクションのロールバックや再実行の減少を、問い合わせ実行時間の全体的な変化と比較検討するようにしてください。

シリアライザブル分離レベルは、学術的なデータベースの文献ではシリアライザブルスナップショット分離として知られる技術を使って実装されています。シリアライザブルスナップショット分離は、スナップショット分離の上に直列化異常の確認を追加することで構築されています。伝統的なロック技術を使う他のシステムと比較すると振舞いや性能の違いが観察されるかもしれません。詳細な情報は[ports12]を参照してください。

13.3. 明示的ロック

PostgreSQLは、テーブル内のデータに対する同時アクセスを制御するために様々な種類のロックモードを備えています。これらのモードは、MVCCでは必要な動作を得られない場合、アプリケーション制御のロックに使用することができます。また、ほとんどのPostgreSQLコマンドでは、参照されるテーブルがそのコマンドの実行中に別の方法で削除もしくは変更されていないことを確実にするために、適切なモードのロックを自動的に獲得します。(例えば、TRUNCATEコマンドは、同じテーブルに対する他の操作と同時に安全に実行することはできないので、それを確実に実行するため、そのテーブルの排他ロックを獲得します。)

現在のデータベースサーバに残っているロックの一覧を確認するには、pg_locksシステムビューを使用してください。ロック管理サブシステムの状況監視についての詳細は第27章を参照してください。

13.3.1. テーブルレベルロック

以下のリストに、使用可能なロックモードとそれらがPostgreSQLで自動的に使用される文脈を示します。また、LOCKコマンドを使用して、こうしたロックを明示的に獲得することもできます。これらのロックモードは、たとえその名前に「row(行)」という言葉が付いていても、全てテーブルレベルのロックであることに注意して

ください。ロックモードの名前は歴史的なものです。これらの名前は、各ロックモードの代表的な使用方法をある程度表しています。しかし、意味的には全て同じです。ロックモード間における唯一の実質的な差異は、どのモードがどのモードと競合するかというロックモードの組み合わせです(表 13.2を参照してください)。2つのトランザクションで、競合するモードのロックを同時に同一テーブル上に保持することはできません（しかし、トランザクションは自分自身とは決して競合しません。例えば、ACCESS EXCLUSIVEロックを獲得し、その後同じテーブルにACCESS SHAREロックを獲得できる可能性があります）。競合しないロックモードは、多くのトランザクションで同時に保持することが可能です。特に、ロックモードには、自己競合するもの（例えば、ACCESS EXCLUSIVEは同時に複数のトランザクションで保持することは不可能）と、自己競合しないもの（例えば、ACCESS SHAREは複数のトランザクションで保持可能）があることに注意してください。

テーブルレベルロックモード

ACCESS SHARE

ACCESS EXCLUSIVEロックモードとのみ競合します。

SELECTコマンドにより、参照されるテーブルに対してこのモードのロックが獲得されます。通常、テーブルの読み取りのみで変更を行わない問い合わせであれば全て、このロックモードを獲得します。

ROW SHARE

EXCLUSIVEおよびACCESS EXCLUSIVEロックモードと競合します。

SELECT FOR UPDATEおよびSELECT FOR SHAREコマンドは、（参照はされているが、FOR UPDATE/FOR SHAREとして選択はされていない他のテーブルに対するACCESS SHAREロックに加えて）対象となるテーブル上にこのモードのロックを獲得します。

ROW EXCLUSIVE

SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE、およびACCESS EXCLUSIVEロックモードと競合します。

UPDATE、DELETE、およびINSERTコマンドは、（参照される他の全てのテーブルに対するACCESS SHAREロックに加えて）対象となるテーブル上にこのモードのロックを獲得します。通常、このロックモードは、テーブルのデータを変更する問い合わせにより獲得されます。

SHARE UPDATE EXCLUSIVE

SHARE UPDATE EXCLUSIVE、SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE、およびACCESS EXCLUSIVEロックモードと競合します。このモードにより、同時実行されるスキーマの変更およびVACUUMコマンドの実行から、テーブルを保護します。

（FULLなしの）VACUUM、ANALYZE、CREATE INDEX CONCURRENTLY、REINDEX CONCURRENTLY、CREATE STATISTICS、および、ALTER INDEXやALTER TABLEの特定の亜種(詳細はALTER INDEXやALTER TABLEを参照してください)によって獲得されます。

SHARE

ROW EXCLUSIVE、SHARE UPDATE EXCLUSIVE、SHARE ROW EXCLUSIVE、EXCLUSIVE、およびACCESS EXCLUSIVEロックモードと競合します。このモードは、同時実行されるデータ変更からテーブルを保護します。

（CONCURRENTLYなしの）CREATE INDEXによって獲得されます。

SHARE ROW EXCLUSIVE

ROW EXCLUSIVE、SHARE UPDATE EXCLUSIVE、SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE、およびACCESS EXCLUSIVEロックモードと競合します。このモードは、1つのセッションだけが一度にそれを保持することができるよう、自己排他的に同時のデータ変更からテーブルを保護します。

CREATE TRIGGER、および、ALTER TABLE（ALTER TABLE参照）のいくつか形式により獲得されます。

EXCLUSIVE

ROW SHARE、ROW EXCLUSIVE、SHARE UPDATE EXCLUSIVE、SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE、およびACCESS EXCLUSIVEロックモードと競合します。このモードは、同時実行されるACCESS SHAREのみを許可します。つまり、このロックモードを保持するトランザクションと並行して実行できる処理は、テーブルの読み取りだけです。

REFRESH MATERIALIZED VIEW CONCURRENTLYにより獲得されます。

ACCESS EXCLUSIVE

全てのモードのロック（ACCESS SHARE、ROW SHARE、ROW EXCLUSIVE、SHARE UPDATE EXCLUSIVE、SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE、および ACCESS EXCLUSIVE）と競合します。このモードにより、その保持者以外にテーブルにアクセスするトランザクションがないことが保証されます。

DROP TABLE、TRUNCATE、REINDEX、CLUSTER、VACUUM FULL、（CONCURRENTLYなしの）REFRESH MATERIALIZED VIEWコマンドによって獲得されます。ALTER INDEXとALTER TABLEの多くの形式もこのレベルでロックを獲得します。これはまた、明示的にモードを指定しないLOCK TABLE文のデフォルトのロックモードです。

> **ヒント**
>
> ACCESS EXCLUSIVEロックのみが、SELECT（FOR UPDATE/SHAREなし）文をブロックします。

通常ロックは獲得した後、トランザクションの終わりまで保持されます。しかし、ロックがセーブポイントの確立後に獲得された場合、セーブポイントがロールバックされると、ロックは即座に解放されます。これは、ROLLBACKがセーブポイント以降に行われたすべてのコマンドの効果を取消すという原則と整合性が取れています。PL/pgSQL例外ブロック内で獲得されたロックに対しても同様です。そのブロックからエラーで抜けた後、獲得されたロックは解放されます。

表13.2 ロックモードの競合

要求するロックモード	既存のロックモード							
	ACCESS SHARE	ROW SHARE	ROW EXCL.	SHARE UPDATE EXCL.	SHARE	SHARE ROW EXCL.	EXCL.	ACCESS EXCL.
ACCESS SHARE								X
ROW SHARE							X	X
ROW EXCL.					X	X	X	X
SHARE UPDATE EXCL.				X	X	X	X	X

要求するロックモード	既存のロックモード							
	ACCESS SHARE	ROW SHARE	ROW EXCL.	SHARE UPDATE EXCL.	SHARE	SHARE ROW EXCL.	EXCL.	ACCESS EXCL.
SHARE			X	X		X	X	X
SHARE ROW EXCL.			X	X	X	X	X	X
EXCL.		X	X	X	X	X	X	X
ACCESS EXCL.	X	X	X	X	X	X	X	X

13.3.2. 行レベルロック

テーブルレベルロックに加えて、行レベルロックがあります。PostgreSQLが自動的に使う文脈付きで以下に行レベルロックの一覧があります。行レベルロックの競合の完全な表については表 13.3を参照してください。トランザクションは異なる副トランザクション内であっても、同じ行に対して競合するロックを保持できることに注意してください。しかし、それ以外では、二つのトランザクションは同じ行に対して競合するロックを決して保持できません。行レベルロックは、データの問い合わせには影響を与えません。行レベルロックは、同じ行に対する書き込みとロックだけをブロックします。テーブルレベルロックと同じように、行レベルロックはトランザクションの終わり、または、セーブポイントへのロールバックで解放されます。

行レベルロックモード

FOR UPDATE

FOR UPDATEによりSELECT文により取り出された行が更新用であるかのようにロックされます。これにより、それらは現在のトランザクションが終わるまで、他のトランザクションがロック、変更、削除できなくなります。すなわち、これらの行に対してUPDATE、DELETE、SELECT FOR UPDATE、SELECT FOR NO KEY UPDATE、SELECT FOR SHARE、SELECT FOR KEY SHAREをしようとする他のトランザクションは現在のトランザクションが終わるまでブロックされます。逆に言えば、SELECT FOR UPDATEは同じ行に対して上記のコマンドを実行している同時実行トランザクションを待ち、それから更新された行をロックして返します(行が削除されていれば、行は返しません)。しかし、REPEATABLE READもしくはSERIALIZABLEトランザクション内では、ロックする行がトランザクションの開始した後に変更された場合にはエラーが返ります。これ以上の議論は13.4を参照してください。

FOR UPDATEロックモードは行に対するDELETEでも、ある列の値を変更するUPDATEでも獲得されます。現時点では、UPDATEの場合に考慮される列の集合は、外部キーとして使うことのできる一意のインデックス(つまり部分インデックスや式インデックスは考慮されません)があるものですが、これは将来変わるかもしれません。

FOR NO KEY UPDATE

獲得するロックが弱い以外はFOR UPDATEと同じように振る舞います。このロックは同じ行のロックを獲得しようとするSELECT FOR KEY SHAREコマンドをブロックしません。このロックモードはFOR UPDATEロックを獲得しないUPDATEによっても獲得されます。

FOR SHARE

取り出された各行に対して排他ロックではなく共有ロックを獲得する以外は、FOR NO KEY UPDATEと同じように振る舞います。共有ロックは、他のトランザクションがこれらの行に対してUPDATE、DELETE、

SELECT FOR UPDATE、SELECT FOR NO KEY UPDATEを実行するのをブロックしますが、SELECT FOR SHAREやSELECT FOR KEY SHAREを実行するのを阻害しません。

FOR KEY SHARE

獲得するロックが弱い以外はFOR SHAREと同じように振る舞います。SELECT FOR UPDATEはブロックされますが、SELECT FOR NO KEY UPDATEはブロックされません。キー共有ロックは、他のトランザクションがDELETEやキー値を変更するUPDATEを実行するのをブロックしますが、それ以外のUPDATEや、SELECT FOR NO KEY UPDATE、SELECT FOR SHARE、SELECT FOR KEY SHAREを阻害しません。

PostgreSQLでは、メモリ上に変更された行の情報を記憶しないため、同時にロックできる行数の上限はありません。しかし、行をロックする際に、ディスクに書き込む作業が発生するかもしれません。例えばSELECT FOR UPDATEは、選択された行をロックしたものと印を付けるために変更を行いますので、ディスクにその結果を書き込むことになります。

表13.3 行レベルロックの競合

要求するロックモード	現在のロックモード			
	FOR KEY SHARE	FOR SHARE	FOR NO KEY UPDATE	FOR UPDATE
FOR KEY SHARE				X
FOR SHARE			X	X
FOR NO KEY UPDATE		X	X	X
FOR UPDATE	X	X	X	X

13.3.3. ページレベルロック

テーブルと行ロックに加え、ページレベルの共有/排他ロックがあり、これらは共有バッファプールにあるテーブルページへの読み書きのアクセスを管理するために使用されます。これらのロックは、行が取得された後や更新された後に即座に解除されます。アプリケーション開発者は通常ページレベルロックを考慮する必要はありませんが、ロックについて全てを説明したかったためここで取り上げました。

13.3.4. デッドロック

明示的なロックの使用は、デッドロックの原因となる可能性があります。デッドロックとは、2つ(もしくはそれ以上)のトランザクションにおいて、それぞれが、他方のトランザクションが必要とするロックを所持してしまうことです。例えば、トランザクション1がテーブルAに排他ロックを獲得していて、次にテーブルBに排他ロックを獲得しようとする際に、トランザクション2が既にテーブルBに排他ロックを獲得済みであって、今からテーブルAに排他ロックを獲得しようと試みる場合、どちらのトランザクションも処理を進められません。PostgreSQLでは、自動的にデッドロック状況を検知し、関係するトランザクションの一方をアボートすることにより、この状況を解決し、もう一方のトランザクションの処理を完了させます (どちらのトランザクションをアボートするかを正確に予期するのは難しく、これに依存すべきではありません)。

デッドロックは行レベルロックの結果として発生する可能性があります (したがって、明示的なロック処理を使用していなくても発生する可能性があります)。2つの同時実行トランザクションがあるテーブルを変更する状況を考えてみます。1つ目のトランザクションは以下を実行します。

```
UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 11111;
```

これは、指定した口座番号の行に対し行レベルロックを獲得します。次に2番目のトランザクションが以下を実行します。

```
UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 22222;
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 11111;
```

1つ目のUPDATE文は指定された行に対する行レベルロックの獲得に成功し、この行の更新に成功します。しかし、2つ目のUPDATE文は、更新対象の行がロックされていることを検知し、ロックを獲得したトランザクションが完了するまで待機します。トランザクション2は、ここで、続きを実行する前にトランザクション1が完了するのを待機しています。さて、トランザクション1がここで以下を実行します。

```
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 22222;
```

トランザクション1は指定した行の行レベルロックを獲得しようとしますが、これは不可能です。トランザクション2がそのロックを既に獲得しているからです。そのため、トランザクション2が完了するのを待機することになります。こうして、トランザクション1はトランザクション2でブロックされ、トランザクション2はトランザクション1でブロックされる、つまり、デッドロック状態です。PostgreSQLはデッドロック状態を検知し、片方のトランザクションを中断させます。

デッドロックを防ぐ最も良い方法は、データベースを使用する全てのアプリケーションが、整合性のある順序で複数のオブジェクトに対するロックを獲得することです。前に示したデッドロックの例で、もし両方のトランザクションで同じ順序で行を更新していたらデッドロックは起こりません。また、トランザクション内のオブジェクトに対して獲得した最初のロックが、そのオブジェクトが必要とする最も制限的なモードであることを確実に保証すべきです。このことが事前に検証できない場合、デッドロックによりアボートするトランザクションを再試行すれば、デッドロックをデータベースを稼働させながらでも処理することができます。

デッドロック状況が検出されなければ、テーブルレベルロックもしくは行レベルロックを要求するトランザクションは、競合するロックが解放されるまで、無期限に待機します。したがって、アプリケーションで長時間（例えば、ユーザの入力待ち）トランザクションを開いたまま保持しておくのは、推奨されません。

13.3.5. 勧告的ロック

PostgreSQLは、アプリケーション独自の意味を持つロックを生成する手法を提供します。これは、その使用に関してシステムによる制限がないこと、つまり、正しい使用に関してはアプリケーションが責任を持つことから勧告的ロックと呼ばれます。勧告的ロックは、MVCC方式に合わせづらいロック戦略で有用に使用することができます。例えば、勧告的ロックのよくある利用として、いわゆる「フラットファイル」データ管理システムで典型的な、悲観的なロック戦略を模擬することです。この用途のためにテーブル内にフラグを格納することもできますが、勧告的ロックの方が高速で、テーブルの膨張を防ぐことができます。また、セッション終了時にサーバによる自動整理を行うこともできるようになります。

PostgreSQLには、セッションレベルとトランザクションレベルという2つの勧告的ロックの獲得方法があります。セッションレベルで獲得すると、勧告的ロックは明示的に解放されるか、セッションが終了するまで保持されます。標準のロック要求と異なり、セッションレベル勧告的ロックはトランザクションという意味には従いません。ロックがトランザクション期間中に獲得され、そのトランザクションを後でロールバックしたとして

も、ロールバック後も保持されます。そして、呼び出し元のトランザクションが後で失敗したとしてもロック解除は有効です。所有するプロセスの中で、同一のセッションレベルのロックを複数回獲得することもできます。この場合、個々のロック要求に対して、ロックを実際に解放する前に対応するロック解除要求がなければなりません。一方トランザクションレベルのロックはより通常のロックに似たように動作します。それらは、処理の終わりに自動的に解放されますので、明示的なロック解放操作はありません。短期間の勧告的ロックを利用する場合は、セッションレベルの動作よりもこの動作の方が便利なことが多くあります。同じ勧告的ロック識別子に対するセッションレベルのロックとトランザクションレベルのロック要求は、想像通り互いをブロックします。セッションがすでに指定された勧告的ロックを保持している場合、他のセッションがそのロックを待機していたとしても、追加の要求は常に成功します。これは保持されているロックと新しい要求がセッションレベルかトランザクションレベルかどうかに関わらず、この文は当てはまります。

PostgreSQLにおけるすべてのロックと同様に、現時点ですべてのセッションで保持されている勧告的ロックの全一覧はpg_locksシステムビューにあります。

勧告的ロックと通常のロックは共有メモリプールに割り当てられ、その容量はmax_locks_per_transactionとmax_connections設定変数により決定されます。このメモリを浪費しないように注意が必要です。さもないと、サーバはロック獲得をまったく許可することができなくなります。これは、サーバで許可できる勧告的ロック数に上限があることを意味します。サーバの設定によりますが、通常、1万から10万程度になります。

特に明示的な順序付けとLIMIT句を持つ問い合わせでは、この勧告ロックモードを使用する幾つかの場合において、SQL式が評価される順序を考慮し獲得されたロックを制御することに気を配らなければなりません。以下に例を示します。

```
SELECT pg_advisory_lock(id) FROM foo WHERE id = 12345; -- 問題なし
SELECT pg_advisory_lock(id) FROM foo WHERE id > 12345 LIMIT 100; -- 危険！
SELECT pg_advisory_lock(q.id) FROM
(
  SELECT id FROM foo WHERE id > 12345 LIMIT 100
) q; -- 問題なし
```

上の例では、ロック獲得関数が実行される前にLIMIT が適用されることを保障できないため、2番目の形式は危険です。これにより、アプリケーションが想定していないなんらかのロックが生成される可能性があります。そのため、（セッションが終了するまで）解放に失敗することになります。アプリケーションから見ると、こうしたロックはただの飾りですが、pg_locksからは参照され続けます。

勧告的ロックを扱うための関数については、9.27.10で説明します。

13.4. アプリケーションレベルでのデータの一貫性チェック

データの参照範囲は各ステートメントで変化するので、リードコミッティドトランザクションを使用して、データ保全性に関するビジネスルールを強化するのは非常に難しいことです。また、書き込み競合が生じる場合、単一のステートメントでさえステートメントのスナップショットに限定されないかもしれません。

リピータブルリードトランザクションは実行全体にわたってデータの安定した参照範囲を持ちますが、MVCCスナップショットをデータ完全性チェックに使用することによる、読み取り/書き込み競合として知られるものを含む、微妙な問題があります。1つのトランザクションがデータを書き、同時に実行するトランザクションが、同じデータ(書き込みの前に、あるいはその書き込みの後にも)を読むことを試みる場合、それは別のトランザクションの働きを見ることができません。その後、読み手は、どれが最初にスタートしたか、あるいは、どれが最初にコミットしたかにかかわらず最初に実行したように見えます。そのままいけば問題はありませんが、読み手がさらにデータを書けば、同時に実行したトランザクションがそれを読んだ場合、上で述べたトランザクションのどちらかの前に走ったように見えるトランザクションとなってしまいます。最後に実行したように見えるトランザクションが実際には最初にコミットしていた場合、トランザクションの実行順のグラフには循環が容易に出現します。そのような循環が出現する時、完全性のチェックはなにかしらの支援がなければ正しく動作しません。

13.2.3により述べたように、シリアライザブルトランザクションは、危険なパターンの読み取り/書き込み競合のための非ブロッキング監視を加えたリピータブルリードトランザクションです。明白に実行順が循環を引き起こすパターンが検知された場合、含まれていたトランザクションのうちの1つは循環を断ち切るためにロールバックされます。

13.4.1. シリアライザブルトランザクションを用いた一貫性の強化

シリアライザブルトランザクション分離レベルが、データの一貫性を必要とするすべての書き込みおよびすべての読み取りに使用される場合、一貫性を確実にするために必要なことは他にはありません。一貫性を保証するためにシリアライザブルトランザクションを使用するよう書かれている他の環境からのソフトウェアは、PostgreSQLでこの点に関して「正しく動く」べきです。

この技術を使用した場合、アプリケーションソフトウェアが直列化失敗でロールバックしたトランザクションを自動的に再試行するようなフレームワークを備えている場合、アプリケーションプログラマにとって不必要な負担を生み出さないようにするでしょう。default_transaction_isolationをserializableにセットすることはよい考えかもしれません。他のトランザクション分離レベルは使用されないことを保証する処置を講ずる、そうでなければ、不注意に完全位チェックを失わないよう、トリガーでトランザクション分離レベルのチェックをすることも賢明でしょう。

実行に関する提言は13.2.3を参照してください。

> **警告**
>
> シリアライザブルトランザクションを使用する整合性保護レベルは、まだホットスタンバイモード(26.5)には拡張されていません。そのために、ホットスタンバイを使用する場合は、マスタにおけるリピータブルリードと明示的なロック処理の利用が望まれるかもしれません。

13.4.2. 明示的なブロッキングロックを用いた一貫性の強化

非シリアライザブルの書き込みが可能な場合、ある行の現時点の有効性を確実なものとし、同時更新を避けるためには、SELECT FOR UPDATE文やSELECT FOR SHARE文、適切なLOCK TABLE文を使用する必要があり

ます（SELECT FOR UPDATE文およびSELECT FOR SHARE文は返ってきた行のみを同時に起こる更新からロックし、LOCK TABLEはテーブル全体をロックします）。これはPostgreSQLに他の環境からアプリケーションを移植する時に考慮されなければなりません

他の環境から切り替えた場合のさらなる注意点としては、同時実行トランザクションが選択された行を更新しないか削除しないということをSELECT FOR UPDATEが保証しないという事実です。PostgreSQLでそれをするためには、値を変更する必要がなくても、実際に行を更新しなければなりません。SELECT FOR UPDATEは、他のトランザクションが同じロックを獲得すること、または、ロックされた行に影響するUPDATEまたはDELETEを実行することを一時的にブロックします。しかしトランザクションがコミットするかロールバックして一度このロックを獲得すると、ロックが獲得されている間に、行の実際のUPDATEが行なわれなかった場合、ブロックされたトランザクションは、競合した操作を続けることになります。

非シリアライザブルMVCCにおいては全体的な有効性チェックに特別な考慮を払わなければなりません。例えば銀行のアプリケーションで、1つのテーブルにある全ての貸方の合計が、別のテーブルにある借方の合計と同じであることを、二つのテーブルが常に更新されているときに、チェックする必要があるとします。2つの連続するSELECT sum(...)コマンドの結果を比べると、2番目の問い合わせは、おそらく最初の問い合わせによってカウントされなかったトランザクションの結果を含んでいるため、リードコミッティドモードでは信頼のおける処理を実行できないことがわかります。1つのリピータブルリードトランザクションで2つの合計を出力すると、リピータブルリードトランザクションが開始される前にコミットされたトランザクション結果のみの正確な状況を得ることができます。しかし、その結果がもたらされた時点でもなお妥当であるかどうかは、実際には疑わしいかもしれません。整合性チェックを行う前にリピータブルリードトランザクション自身が変更を行った場合、そのチェックの有効性はさらに疑わしくなります。これにより、トランザクション開始後に行われる変更の全てだけでなく、何か別のものが含まれるためです。このような場合、注意深い人であれば、現状を確実に把握するためにチェックに必要な全てのテーブルをロックするでしょう。SHAREモード（もしくはそれ以上）のロックにより、現在のトランザクションでの変更を除き、ロックされたテーブルにコミットされていない変更が存在しないことが保証されます。

同時に、明示的なロック処理を使用して、同時に変更が実行されるのを防ごうとする場合、リードコミッティドモードを使用するか、または、リピータブルリードモードの場合は、問い合わせを実行する前にロックを獲得するよう留意してください。リピータブルリードトランザクションにおいて獲得されたロックは、テーブルに変更をかける他のトランザクションが現在実行されていないことを保証します。しかし、トランザクションが参照しているスナップショットが、ロックの獲得より前に取得されたものであれば、そのスナップショットは現時点においてコミットされている変更より前のテーブルのものである可能性があります。リピータブルリードトランザクションのスナップショットは、実際にはその最初の問い合わせもしくはデータ変更コマンド（SELECT、INSERT、UPDATE、またはDELETE）が開始された時点で取得されます。したがって、スナップショットを取得する前に、明示的にロックを獲得することが可能です。

13.5. 警告

DDLコマンドの中には、現在はTRUNCATEとテーブルを書き換える形のALTER TABLEだけですが、MVCCセーフでないものがあります。これは、DDLコマンドをコミットする前に取得したスナップショットを使っていると、切り詰めまたは書き換えのコミット後に、同時実行トランザクションに対してテーブルが空に見えることを意味しています。該当するテーブルにDDLコマンドが開始する前にアクセスしなかったトランザクションにとってのみ、これは問題となるでしょう—開始前にアクセスしたトランザクションは少なくともACCESS SHAREテーブルロックを保持しており、そのトランザクションが完了するまでDDLコマンドはブロックされるでしょう。ですので、対象のテーブルに対する連続した問い合わせで、このコマンドはテーブルの内容の見かけ

上の不整合の原因とはなりません。しかし、対象のテーブルとデータベース内の他のテーブルの内容の間の可視の不整合の原因となるかもしれません。

シリアライザブルトランザクション分離レベルのサポートは、まだホットスタンバイレプリケーションは対象に加えられていません(26.5で述べます)。ホットスタンバイモードで現在サポートされた最も厳しい分離レベルはリピータブルリードです。マスタ上でシリアライザブルトランザクション中にデータベースに永続的な書き込みを行なえば、スタンバイはすべて最終的に一貫した状態に達するだろうということは保証されるでしょうが、スタンバイ上で実行されたリピータブルリードトランザクションは、時々マスタのトランザクションの任意の連続する実行と一致しない過渡状態を見ることがあるでしょう。

システムカタログへの内部のアクセスは現在のトランザクションの分離レベルを使っては行われません。これは、テーブルのような新しく作られたデータベースオブジェクトが、たとえシステムカタログが含む行が可視でないとしても、並行するリピータブルリードトランザクションやシリアライザブルトランザクションに対して可視であることを意味します。対照的に、明示的にシステムカタログを確認する問い合わせは、より高い分離レベルで並行して作られているデータベースオブジェクトを表す行を見ることはできません。

13.6. ロックとインデックス

PostgreSQLは、テーブルデータへのノンブロック読み込み/書き込みアクセスを備えています。しかし現在、この機能はPostgreSQLで実装されている全てのインデックスアクセスメソッドに対して実装されているわけではありません。各種のインデックスでは下記のように扱われます。

B-treeインデックス、GiSTおよびSP-GiSTインデックス

> 読み込み/書き込みアクセスに短期の共有/排他モードのページレベルロックを使います。ロックは、インデックス行が挿入または取り出されるとただちに解放されます。これらのインデックス種類は、デッドロック状態になることなく、最も高い同時実行性を提供します。

ハッシュインデックス

> 読み込み/書き込みアクセスに共有/排他モードのハッシュバケットレベルロックを使います。ロックは、バケット全体が処理された後に解放されます。バケットレベルロックは、インデックスレベルのロックよりも同時実行性に優れていますが、1つのインデックス操作よりも長くロックが保持されますので、デッドロックに陥りやすくなります。

GINインデックス

> 読み込み/書き込みアクセスに短期の共有/排他モードのページレベルロックを使います。ロックは、インデックス行が挿入または取り出されるとただちに解放されます。しかし、GINによりインデックス付けされた値の挿入は、通常1行当たり複数のインデックスキーの挿入をもたらすことに注意してください。そのため、GINは単一の値を挿入する時に更に多くの作業を行います。

現時点では、B-treeインデックスは同時実行アプリケーションにおいて最善の性能を提供します。これはまた、ハッシュインデックスよりも多くの機能を持つため、スカラデータのインデックスが必要な同時実行アプリケーションで推奨するインデックス型です。非スカラデータを扱う場合、B-treeを使用することができないことは明確です。この場合は代わりにGiST、SP-GiSTもしくはGINインデックスを使用すべきです。

第14章 性能に関するヒント

問い合わせの性能は多くの要因に影響されます。ユーザが制御できるものもありますが、背後にあるシステム設計に起因する根本的な要因もあります。本章ではPostgreSQLの性能を理解し、チューニングするためのヒントを提供します。

14.1. EXPLAINの利用

PostgreSQLは受理した問い合わせから*問い合わせ計画*を作り出します。問い合わせの構造と含まれるデータの性質に適した正しい問い合わせ計画を選択することが、良い性能を得るために非常に重要になります。ですので、システムには優れた計画の選択を試みる複雑な*プランナ*が存在します。EXPLAINコマンドを使えば、任意の問い合わせに対してプランナがどのような問い合わせ計画を作ったのかわかります。問い合わせ計画を読みこなすには、ある程度の経験が必要です。本節ではその基本を提供しようと考えます。

本節の例は、9.3の開発版ソースを用いてVACUUM ANALYZEを実行した後でリグレッションテストデータベースから取り出したものです。実際にこの例を試すと、似たような結果になるはずですが、おそらく推定コストや行数は多少異なることになるでしょう。ANALYZEによる統計情報は厳密なものではなくランダムなサンプリングを行った結果であり、また、コストは本質的にプラットフォームに何かしら依存するためです。

例では、簡潔で人が読みやすいEXPLAINのデフォルトの「text」出力書式を使用します。今後の解析でEXPLAINの出力をプログラムに渡すことを考えているのであれば、代わりに機械読み取りが容易な出力書式（XML、JSON、YAML）のいずれかを使用する必要があります。

14.1.1. EXPLAINの基本

問い合わせ計画は*計画ノード*のツリー構造です。ツリー構造の最下層ノードはスキャンノードで、テーブルから行そのものを返します。シーケンシャルスキャン、インデックススキャン、ビットマップインデックススキャンといったテーブルアクセス方法の違いに応じ、スキャンノードの種類に違いがあります。また、VALUES句やFROM内の集合を返す関数など独自のスキャンノード種類を持つ、テーブル行を元にしないものがあります。問い合わせが結合、集約、ソートなど、行そのものに対する操作を必要としている場合、スキャンノードの上位に更に、これらの操作を行うためのノードが追加されます。これらの操作の実現方法にも通常複数の方法がありますので、異なった種類のノードがここに出現することもあり得ます。EXPLAINには計画ツリー内の各ノードにつき1行の出力があり、基本ノード種類とプランナが生成したその計画ノードの実行に要するコスト推定値を示します。さらに、ノードの追加属性を表示するためにノードの要約行からインデント付けされた行が出力される可能性があります。最初の1行目（最上位ノード）には、計画全体の実行コスト推定値が含まれます。プランナはこの値が最小になるように動作します。

どのような出力となるのかを示すためだけに、ここで簡単な例を示します。

```
EXPLAIN SELECT * FROM tenk1;

                        QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..458.00 rows=10000 width=244)
```

この問い合わせにはWHERE句がありませんので、テーブル行をすべてスキャンしなければなりません。このためプランナは単純なシーケンシャルスキャン計画を使用することを選びました。(左から右に)括弧で囲まれた数値には以下のものがあります。

- 初期処理の推定コスト。出力段階が開始できるようになる前に消費される時間、例えば、SORTノードで実行されるソート処理の時間です。
- 全体推定コスト。これは計画ノードが実行完了である、つまりすべての利用可能な行を受け取ることを前提として示されます。実際には、ノードの親ノードはすべての利用可能な行を読む前に停止する可能性があります(以下のLIMITの例を参照)。
- この計画ノードが出力する行の推定数。ここでも、ノードが実行を完了することを前提としています。
- この計画ノードが出力する行の(バイト単位での)推定平均幅。

コストはプランナのコストパラメータ(19.7.2参照)によって決まる任意の単位で測定されます。取り出すディスクページ単位でコストを測定することが、伝統的な方式です。つまり、seq_page_costを慣習的に1.0に設定し、他のコストパラメータを相対的に設定します。本節の例では、デフォルトのコストパラメータで実行しています。

上位ノードのコストには、すべての子ノードのコストもその中に含まれていることを理解することは重要です。このコストはプランナが関与するコストのみ反映する点もまた重要です。とりわけ、結果の行をクライアントに転送するコストは、実際の処理時間の重要な要因となる可能性があるにもかかわらず、考慮されません。プランナは、計画をいかに変更しようと、どうすることもできないため、これを無視します。(正しい計画はどんなものであれ、すべて同じ行を結果として出力すると信じています。)

rowsの値は、計画ノードによって処理あるいはスキャンされた行数を表しておらず、ノードによって発行された行数を表すので、多少扱いにくくなっています。該当ノードに適用されるすべてのWHERE句条件によるフィルタ処理の結果、スキャンされる行より少ない行数になることがよくあります。理想的には、最上位の行数の推定値は、実際に問い合わせによって返され、更新され、あるいは削除された概算の行数となります。

例に戻ります。

```
EXPLAIN SELECT * FROM tenk1;

                        QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..458.00 rows=10000 width=244)
```

これらの数値はとても素直に導かれます。以下を実行すると、

```
SELECT relpages, reltuples FROM pg_class WHERE relname = 'tenk1';
```

tenk1には358のディスクページと10000の行があることがわかります。推定コストは(ディスクページ読み取り * seq_page_cost)+(スキャンした行 * cpu_tuple_cost)と計算されます。デフォルトでは、seq_page_costは1.0、cpu_tuple_costは0.01です。ですから、推定コストは(358 * 1.0) + (10000 * 0.01) = 458となります。

では、WHERE条件を加えて、問い合わせを変更してみます。

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 7000;
```

```
                        QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=7001 width=244)
   Filter: (unique1 < 7000)
```

EXPLAINの出力が、Seq Scan計画ノードに付随する「フィルタ」条件として適用されるWHERE句を表示していることに注意してください。これは、この計画ノードがスキャンした各行に対してその条件を検査することを意味し、その条件を通過したもののみが出力されます。WHERE句があるため、推定出力行数が小さくなっています。しかし、依然として10000行すべてをスキャンする必要があるため、コストは小さくなっていません。実際には、WHERE条件を検査するためにCPU時間が余計にかかることを反映して、ほんの少し（正確には10000 * cpu_operator_cost）ですがコストが上昇しています。

この問い合わせが選択する実際の行数は7000です。しかし、rowsの推定行数は概算値に過ぎません。この実験を2回実行した場合、おそらく多少異なる推定値を得るでしょう。もっと言うと、これはANALYZEコマンドを行う度に変化することがあり得ます。なぜなら、ANALYZEで生成される統計情報は、テーブルのランダムな標本から取り出されるからです。

では、条件をより強く制限してみます。

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100;

                                QUERY PLAN
---------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=5.07..229.20 rows=101 width=244)
   Recheck Cond: (unique1 < 100)
   ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0)
         Index Cond: (unique1 < 100)
```

ここでは、プランナは2段階の計画を使用することを決定しました。子の計画ノードは、インデックスを使用して、インデックス条件(index condition)に合う行の場所を検索します。そして、上位計画ノードが実際にテーブル自体からこれらの行を取り出します。行を別々に取り出すことは、シーケンシャルな読み取りに比べ非常に高価です。しかし、テーブルのすべてのページを読み取る必要はありませんので、シーケンシャルスキャンより低価になります。（2段階の計画を使用する理由は、別々に行を取り出すコストを最小にするために、上位の計画ノードがインデックスにより識別された行の位置を読み取る前に物理的な順序でソートすることです。ノードで記載されている「bitmap」は、ソートを行う機構の名前です。）

ここでWHERE句に別の条件を付与してみましょう。

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND stringu1 = 'xxx';

                                QUERY PLAN
---------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=5.04..229.43 rows=1 width=244)
   Recheck Cond: (unique1 < 100)
   Filter: (stringu1 = 'xxx'::name)
   ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0)
```

```
         Index Cond: (unique1 < 100)
```

追加されたstringu1 = 'xxx'条件は出力行数推定値を減らしますが、同じ行集合にアクセスしなければなりませんので、コストは減りません。このインデックスがunique1列に対してのみ存在するため、stringu1句をインデックス条件として適用できないことに注意してください。代わりに、インデックスによって取り出される行に対するフィルタとして適用されます。これにより、追加の検査分を反映するため、コストは実際には少し上がります。

場合によってはプランナは「単純な」インデックススキャン計画を選択します。

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 = 42;

                                QUERY PLAN
---------------------------------------------------------------------------
 Index Scan using tenk1_unique1 on tenk1  (cost=0.29..8.30 rows=1 width=244)
   Index Cond: (unique1 = 42)
```

この種の計画では、テーブル行はインデックス順で取り出されます。このため読み取りがより高価になりますが、この場合取り出す行数が少ないため、改めて行位置をソートし直すための追加コストは割に合いません。単一の行のみを取り出す問い合わせでは、この計画種類がよく現れます。また、ORDER BYを満たすために必要となる余分な必要なソート処理がないため、インデックスの順序に一致するORDER BY条件を持つ問い合わせでよく使用されます。この例では、ORDER BY unique1を追加すると、要求された順序がインデックスによってすでに暗黙的に提供されているため、同じ計画が使用されます。

プランナはORDER BY句をいくつかの方法で実装できます。上の例ではこのようなORDER BY句を暗黙的に実装できることを示しています。プランナは明示的なsortステップを追加もします。

```
EXPLAIN SELECT * FROM tenk1 ORDER BY unique1;
                            QUERY PLAN
-------------------------------------------------------------------
 Sort  (cost=1109.39..1134.39 rows=10000 width=244)
   Sort Key: unique1
   ->  Seq Scan on tenk1  (cost=0.00..445.00 rows=10000 width=244)
```

ソートキーで必要な接頭辞の順序がプランの一部で保証されている場合、インクリメンタルソート(incremental sort)ステップを使用することを決定できます。

```
EXPLAIN SELECT * FROM tenk1 ORDER BY four, ten LIMIT 100;
                                          QUERY PLAN
-------------------------------------------------------------------
----------------------------------
 Limit  (cost=521.06..538.05 rows=100 width=244)
   ->  Incremental Sort  (cost=521.06..2220.95 rows=10000 width=244)
         Sort Key: four, ten
         Presorted Key: four
         ->  Index Scan using index_tenk1_on_four on tenk1  (cost=0.29..1510.08 rows=10000
 width=244)
```

通常のソートと比較して、インクリメンタルソートは、結果セット全体がソートされる前にタプルを戻すことができます。これにより、特にLIMITがある問い合わせで最適化が可能になります。また、メモリ使用量が削減され、ソートがディスクにオーバーフローする可能性が減少しますが、結果セットを複数のソートバッチに分割するオーバーヘッドが増加という代償を払うことになります。

WHERE句で参照される複数の列に対して別々のインデックスが存在する場合、プランナはインデックスをANDやORで組み合わせて使用することを選択する可能性があります。

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000;

                                     QUERY PLAN
-------------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=25.08..60.21 rows=10 width=244)
   Recheck Cond: ((unique1 < 100) AND (unique2 > 9000))
   ->  BitmapAnd  (cost=25.08..25.08 rows=10 width=0)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0)
               Index Cond: (unique1 < 100)
         ->  Bitmap Index Scan on tenk1_unique2  (cost=0.00..19.78 rows=999 width=0)
               Index Cond: (unique2 > 9000)
```

しかし、これは両方のインデックスを参照する必要があります。そのため、インデックスを1つ使用し、他の条件についてはフィルタとして扱う方法と比べて常に勝るとは限りません。含まれる範囲を変更すると、それに伴い計画も変わることが分かるでしょう。

以下にLIMITの影響を示す例を示します。

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000 LIMIT 2;

                                     QUERY PLAN
-------------------------------------------------------------------------------------
 Limit  (cost=0.29..14.48 rows=2 width=244)
   ->  Index Scan using tenk1_unique2 on tenk1  (cost=0.29..71.27 rows=10 width=244)
         Index Cond: (unique2 > 9000)
         Filter: (unique1 < 100)
```

これは上と同じ問い合わせですが、すべての行を取り出す必要がないためLIMITを付けています。プランナはどうすべきかについて考えを変えました。インデックススキャンノードの総コストと総行数があたかも実行完了したかのように表示されていることに注意してください。しかしLimitノードが、これらの行の1/5だけを取り出した後で停止することが想定されています。そのため総コストは1/5程度のみとなり、これが問い合わせの実際の推定コストとなります。この計画は、以前の計画にLimitノードを追加することより好まれます。Limitはビットマップスキャンの起動コストを払うことを避けることができないためです。このため総コストはこの方法に25単位ほど増加します。

今まで説明に使ってきたフィールドを使って2つのテーブルを結合してみましょう。

```
EXPLAIN SELECT *
FROM tenk1 t1, tenk2 t2
```

```
WHERE t1.unique1 < 10 AND t1.unique2 = t2.unique2;

                                      QUERY PLAN
--------------------------------------------------------------------------------
 Nested Loop  (cost=4.65..118.62 rows=10 width=488)
   ->  Bitmap Heap Scan on tenk1 t1  (cost=4.36..39.47 rows=10 width=244)
         Recheck Cond: (unique1 < 10)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..4.36 rows=10 width=0)
               Index Cond: (unique1 < 10)
   ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.29..7.91 rows=1 width=244)
         Index Cond: (unique2 = t1.unique2)
```

この計画では、入力または子として2つのテーブルスキャンを持つネステッドループ結合ノードがあります。計画のツリー構造を反映して、ノード要約行はインデント付けされます。結合の先頭、「外部」、子は以前に説明したものと似たビットマップスキャンです。そのコストと行数は、該当ノードにunique1 < 10 WHERE句が適用されるため、SELECT ... WHERE unique1 < 10で得られたものと同じです。この段階ではt1.unique2 = t2.unique2句は関係しておらず、外部スキャンにおける出力行数に影響していません。ネステッドループ結合ノードは、外部の子から得られた行毎に、その2番目または「内部の」子を一回実行します。現在の外部の行からの列の値は内部スキャンに組み込まれます。ここで、外部行からのt1.unique2の値が利用できますので、上述の単純なSELECT ... WHERE t2.unique2 = constantの場合に示したものと似た計画とコストが得られます。(実際、推定コストは、t2に対するインデックススキャンが繰り返される間に発生することが想定されるキャッシュの結果、上で示した値よりわずかに低くなります。) ループノードのコストは、外部スキャンのコストと、各々の外部の行に対して内部スキャンが繰り返されることによるコスト(ここでは10 * 7.91)を加え、さらに結合処理を行うための少々のCPU時間を加えたものになります。

この例では、結合の出力行数は2つのスキャンの出力行数の積に等しくなっていますが、いつもそうなるわけではありません。2つのテーブルに関係するWHERE句は、入力スキャン時ではなく、結合を行う際に適用されるからです。以下が例です。

```
EXPLAIN SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 10 AND t2.unique2 < 10 AND t1.hundred < t2.hundred;

                                      QUERY PLAN
--------------------------------------------------------------------------------------------
 Nested Loop  (cost=4.65..49.46 rows=33 width=488)
   Join Filter: (t1.hundred < t2.hundred)
   ->  Bitmap Heap Scan on tenk1 t1  (cost=4.36..39.47 rows=10 width=244)
         Recheck Cond: (unique1 < 10)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..4.36 rows=10 width=0)
               Index Cond: (unique1 < 10)
   ->  Materialize  (cost=0.29..8.51 rows=10 width=244)
         ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.29..8.46 rows=10 width=244)
               Index Cond: (unique2 < 10)
```

条件t1.hundred < t2.hundredはtenk2_unique2インデックスの中では試験されません。このため結合ノードで適用されます。これは結合ノードの推定出力行数を減らしはしますが、入力スキャンには影響しません。

ここではプランナが、具体化計画ノードをその上に挿入することで、結合の内部リレーションの「具体化」を選択していることに注意してください。これは、たとえネステッドループ結合ノードが外部リレーションから各行につき一度、そのデータを10回読む必要があったとしても、t2インデックススキャンが一度だけ行なわれることを意味します。具体化ノードはそのデータを読んだときにメモリに保存し、その後の読み出しではそのデータをメモリから返します。

外部結合を扱う時、「結合フィルタ」および通常の「フィルタ」の両方が付随する結合計画ノードが現れる可能性があります。結合フィルタ条件は外部結合のON句を元にしますので、結合フィルタ条件に合わない行がNULLで展開された行として発行され続けます。しかし通常のフィルタ条件が外部結合規則の後に適用され、条件に合わない行は削除されます。内部結合では、これらのフィルタ種類の間に意味的な違いはありません。

問い合わせの選択性を少し変更すると、非常に異なる結合計画が得られるかもしれません。

```
EXPLAIN SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;

                                       QUERY PLAN
-----------------------------------------------------------------------------------------
 Hash Join  (cost=230.47..713.98 rows=101 width=488)
   Hash Cond: (t2.unique2 = t1.unique2)
   ->  Seq Scan on tenk2 t2  (cost=0.00..445.00 rows=10000 width=244)
   ->  Hash  (cost=229.20..229.20 rows=101 width=244)
         ->  Bitmap Heap Scan on tenk1 t1  (cost=5.07..229.20 rows=101 width=244)
               Recheck Cond: (unique1 < 100)
               ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0)
                     Index Cond: (unique1 < 100)
```

ここでプランナはハッシュ結合の使用を選択しました。片方のテーブルの行がメモリ内のハッシュテーブルに格納され、もう片方のテーブルがスキャンされた後、各行に対して一致するかどうかハッシュテーブルを探索します。繰り返しますが、インデント付けにより計画の構造が表されます。tenk1に対するビットマップスキャンはハッシュノードへの入力です。外部の子計画から行を読み取り、各行に対してハッシュテーブルを検索します。

他にも、以下に示すようなマージ結合という結合があり得ます。

```
EXPLAIN SELECT *
FROM tenk1 t1, onek t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;

                                       QUERY PLAN
-----------------------------------------------------------------------------------------
 Merge Join  (cost=198.11..268.19 rows=10 width=488)
   Merge Cond: (t1.unique2 = t2.unique2)
   ->  Index Scan using tenk1_unique2 on tenk1 t1  (cost=0.29..656.28 rows=101 width=244)
         Filter: (unique1 < 100)
   ->  Sort  (cost=197.83..200.33 rows=1000 width=244)
```

```
         Sort Key: t2.unique2
         ->  Seq Scan on onek t2  (cost=0.00..148.00 rows=1000 width=244)
```

マージ結合は、結合キーでソートされる入力データを必要とします。この計画では、正確な順序で行をアクセスするためにtenk1データがインデックススキャンを用いてソートされます。しかし、このテーブルの中でより多くの行がアクセスされるため、onekではシーケンシャルスキャンとソートが好まれています。(多くの行をソートする場合、インデックススキャンでは非シーケンシャルなディスクアクセスが必要となるため、シーケンシャルスキャンとソートの方がインデックススキャンより優れています。)

19.7.1に記載したenable/disableフラグを使用して、プランナが最も良いと考えている戦略を強制的に無視させる方法により、異なった計画を観察することができます。(非常に原始的なツールですが、利用価値があります。14.3も参照してください。) 例えば、前の例にてonekテーブルを扱う最善の方法がシーケンシャルスキャンとソートであると納得できなければ、以下を試みることができます。

```
SET enable_sort = off;

EXPLAIN SELECT *
FROM tenk1 t1, onek t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;

                                        QUERY PLAN
------------------------------------------------------------------------------------------
 Merge Join  (cost=0.56..292.65 rows=10 width=488)
   Merge Cond: (t1.unique2 = t2.unique2)
   ->  Index Scan using tenk1_unique2 on tenk1 t1  (cost=0.29..656.28 rows=101 width=244)
         Filter: (unique1 < 100)
   ->  Index Scan using onek_unique2 on onek t2  (cost=0.28..224.79 rows=1000 width=244)
```

これは、プランナが、シーケンシャルスキャンとソートよりインデックススキャンによるonekのソート処理がおよそ12%程高価であるとみなしたことを示します。当然ながら、次の疑問はこれが正しいかどうかでしょう。後で説明するEXPLAIN ANALYZEを使用することで調査することができます。

14.1.2. EXPLAIN ANALYZE

EXPLAINのANALYZEオプションを使用して、プランナが推定するコストの精度を点検することができます。このオプションを付けるとEXPLAINは実際にその問い合わせを実行し、計画ノードごとに実際の行数と要した実際の実行時間を、普通のEXPLAINが示すものと同じ推定値と一緒に表示します。例えば、以下のような結果を得ることができます。

```
EXPLAIN ANALYZE SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 10 AND t1.unique2 = t2.unique2;

                                                           QUERY PLAN
--------------------------------------------------------------------------
-------------------------------------------------------------
 Nested Loop  (cost=4.65..118.62 rows=10 width=488) (actual time=0.128..0.377 rows=10 loops=1)
```

```
  ->  Bitmap Heap Scan on tenk1 t1  (cost=4.36..39.47 rows=10 width=244) (actual
time=0.057..0.121 rows=10 loops=1)
        Recheck Cond: (unique1 < 10)
        ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..4.36 rows=10 width=0) (actual
time=0.024..0.024 rows=10 loops=1)
              Index Cond: (unique1 < 10)
  ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.29..7.91 rows=1 width=244) (actual
time=0.021..0.022 rows=1 loops=10)
        Index Cond: (unique2 = t1.unique2)
Planning time: 0.181 ms
Execution time: 0.501 ms
```

「actual time」値は実時間をミリ秒単位で表されていること、cost推定値は何らかの単位で表されていることに注意してください。ですからそのまま比較することはできません。注目すべきもっとも重要な点は通常、推定行数が実際の値と合理的に近いかどうかです。この例では、推定はすべて正確ですが、現実的にはあまりありません。

問い合わせ計画の中には、何回も副計画ノードを実行する可能性のあるものがあります。例えば、上述のネステッドループの計画では、内部インデックススキャンは外部の行ごとに一度行われます。このような場合、loops値はそのノードを実行する総回数を報告し、表示される実際の時間と行数は1実行当たりの平均です。これで値を表示された推定コストと比較できるようになります。loops値をかけることで、そのノードで実際に費やされた総時間を得ることができます。上の例では、tenk2に対するインデックススキャンの実行のために合計0.220ミリ秒要しています。

場合によっては、EXPLAIN ANALYZEは計画ノードの実行時間と行数以上の実行統計情報をさらに表示します。例えば、ソートとハッシュノードでは以下のような追加情報を提供します。

```
EXPLAIN ANALYZE SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2 ORDER BY t1.fivethous;

                                                                  QUERY PLAN
-----------------------------------------------------------------------------
-------------------------------------------------------------------------------
 Sort  (cost=717.34..717.59 rows=101 width=488) (actual time=7.761..7.774 rows=100 loops=1)
   Sort Key: t1.fivethous
   Sort Method: quicksort  Memory: 77kB
   ->  Hash Join  (cost=230.47..713.98 rows=101 width=488) (actual time=0.711..7.427 rows=100
 loops=1)
         Hash Cond: (t2.unique2 = t1.unique2)
         ->  Seq Scan on tenk2 t2  (cost=0.00..445.00 rows=10000 width=244) (actual
 time=0.007..2.583 rows=10000 loops=1)
         ->  Hash  (cost=229.20..229.20 rows=101 width=244) (actual time=0.659..0.659 rows=100
 loops=1)
               Buckets: 1024  Batches: 1  Memory Usage: 28kB
               ->  Bitmap Heap Scan on tenk1 t1  (cost=5.07..229.20 rows=101 width=244) (actual
 time=0.080..0.526 rows=100 loops=1)
```

```
                      Recheck Cond: (unique1 < 100)
                      ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0)
 (actual time=0.049..0.049 rows=100 loops=1)
                            Index Cond: (unique1 < 100)
 Planning time: 0.194 ms
 Execution time: 8.008 ms
```

ソートノードは使用されるソート方式（具体的にはソートがメモリ内かディスク上か）および必要なメモリまたはディスクの容量を表示します。ハッシュノードでは、ハッシュバケット数とバッチ数、ハッシュテーブルで使用されるメモリのピーク容量が表示されます。（バッチ数が1を超える場合、同時にディスクの使用容量も含まれますが、表示はされません。）

他の種類の追加情報はフィルタ条件によって除外される行数があります。

```
EXPLAIN ANALYZE SELECT * FROM tenk1 WHERE ten < 7;

                                                 QUERY PLAN
-------------------------------------------------------------------------
-------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=7000 width=244) (actual time=0.016..5.107 rows=7000
 loops=1)
   Filter: (ten < 7)
   Rows Removed by Filter: 3000
 Planning time: 0.083 ms
 Execution time: 5.905 ms
```

特に結合ノードで適用されるフィルタ条件ではこれらの数が有用です。「Rows Removed」行は、少なくともスキャンされた1行、結合ノードにおける結合組み合わせの可能性がフィルタ条件によって拒絶された時にのみ現れます。

「非可逆」インデックススキャンはフィルタ条件に似た状況です。例えば、特定の点を含有する多角形の検索を考えてみます。

```
EXPLAIN ANALYZE SELECT * FROM polygon_tbl WHERE f1 @> polygon '(0.5,2.0)';

                                              QUERY PLAN
-------------------------------------------------------------------------
----------------------------------
 Seq Scan on polygon_tbl  (cost=0.00..1.05 rows=1 width=32) (actual time=0.044..0.044 rows=0
 loops=1)
   Filter: (f1 @> '((0.5,2))'::polygon)
   Rows Removed by Filter: 4
 Planning time: 0.040 ms
 Execution time: 0.083 ms
```

プランナは（ほぼ正確に）、インデックススキャンを考慮するには例のテーブルが小さ過ぎるとみなします。このため、フィルタ条件によってすべての行が拒絶される、普通のシーケンシャルスキャンとなります。しかしインデックススキャンの使用を強制するのであれば、以下のようにします。

```
SET enable_seqscan TO off;

EXPLAIN ANALYZE SELECT * FROM polygon_tbl WHERE f1 @> polygon '(0.5,2.0)';

                                                        QUERY PLAN
-----------------------------------------------------------------------------
-------------------------------------------------------
 Index Scan using gpolygonind on polygon_tbl  (cost=0.13..8.15 rows=1 width=32) (actual
 time=0.062..0.062 rows=0 loops=1)
   Index Cond: (f1 @> '((0.5,2))'::polygon)
   Rows Removed by Index Recheck: 1
 Planning time: 0.034 ms
 Execution time: 0.144 ms
```

ここで、インデックスが1つの候補行を返し、それがインデックス条件の再検査により拒絶されることが分かります。多角形の含有試験ではGiSTインデックスが「非可逆」であるため、これは発生します。実際には対象と重なる多角形を持つ行を返し、そしてこれらの行が正確に含有関係であることを試験しなければなりません。

EXPLAINには、より多くの実行時統計情報を取り出すために、ANALYZEに付与できるBUFFERSオプションがあります。

```
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000;

                                                            QUERY PLAN
-----------------------------------------------------------------------------
-------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=25.08..60.21 rows=10 width=244) (actual time=0.323..0.342
 rows=10 loops=1)
   Recheck Cond: ((unique1 < 100) AND (unique2 > 9000))
   Buffers: shared hit=15
   ->  BitmapAnd  (cost=25.08..25.08 rows=10 width=0) (actual time=0.309..0.309 rows=0 loops=1)
         Buffers: shared hit=7
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0) (actual
 time=0.043..0.043 rows=100 loops=1)
               Index Cond: (unique1 < 100)
               Buffers: shared hit=2
         ->  Bitmap Index Scan on tenk1_unique2  (cost=0.00..19.78 rows=999 width=0) (actual
 time=0.227..0.227 rows=999 loops=1)
               Index Cond: (unique2 > 9000)
               Buffers: shared hit=5
 Planning time: 0.088 ms
 Execution time: 0.423 ms
```

BUFFERSにより提供される数は、問い合わせのどの部分がもっとも大きいI/Oであるかを識別する役に立ちます。

EXPLAIN ANALYZEが実際に問い合わせを実行しますので、EXPLAINのデータを出力することを優先して問い合わせの出力が破棄されたとしても、何らかの副作用が通常通り発生することに注意してください。テーブルを変更すること無くデータ変更問い合わせの解析を行いたければ、以下の例のように、実行後コマンドをロールバックしてください。

```
BEGIN;

EXPLAIN ANALYZE UPDATE tenk1 SET hundred = hundred + 1 WHERE unique1 < 100;

                                                           QUERY PLAN
-------------------------------------------------------------------------------
-------------------------------------------------------------
 Update on tenk1  (cost=5.07..229.46 rows=101 width=250) (actual time=14.628..14.628 rows=0
 loops=1)
   ->  Bitmap Heap Scan on tenk1  (cost=5.07..229.46 rows=101 width=250) (actual
 time=0.101..0.439 rows=100 loops=1)
         Recheck Cond: (unique1 < 100)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0) (actual
 time=0.043..0.043 rows=100 loops=1)
               Index Cond: (unique1 < 100)
 Planning time: 0.079 ms
 Execution time: 14.727 ms

ROLLBACK;
```

この例で分かるように、問い合わせがINSERT、UPDATE、DELETEである場合、テーブル変更を行うための実作業は最上位のInsert、Update、Delete計画ノードで行われます。このノード以下にある計画ノードは、古い行の検索、新しいデータの計算、あるいはその両方を行います。このため、前に述べたものと同じ種類のビットマップテーブルスキャンがあり、その出力が更新される行を格納するUpdateノードに渡されることが分かります。データ変更ノードが実行時間の多くを費やす可能性があります（現在これが一番多くの時間を費やしています）が、プランナは現在その作業を考慮してコスト推定に何も加えません。これは、行われる作業がすべての正確な問い合わせ計画の作業と同一であるためであり、このため計画の決定に影響を与えません。

UPDATEもしくはDELETEコマンドが継承階層に影響する場合には、出力は以下のようになるでしょう。

```
EXPLAIN UPDATE parent SET f2 = f2 + 1 WHERE f1 = 101;
                                     QUERY PLAN
-------------------------------------------------------------------------------------
 Update on parent  (cost=0.00..24.53 rows=4 width=14)
   Update on parent
   Update on child1
   Update on child2
   Update on child3
   ->  Seq Scan on parent  (cost=0.00..0.00 rows=1 width=14)
         Filter: (f1 = 101)
   ->  Index Scan using child1_f1_key on child1  (cost=0.15..8.17 rows=1 width=14)
         Index Cond: (f1 = 101)
```

```
   ->  Index Scan using child2_f1_key on child2  (cost=0.15..8.17 rows=1 width=14)
         Index Cond: (f1 = 101)
   ->  Index Scan using child3_f1_key on child3  (cost=0.15..8.17 rows=1 width=14)
         Index Cond: (f1 = 101)
```

この例では、Updateノードは元々言及されている親テーブルに加えて3つの子テーブルを考慮することが必要です。そのため、テーブル毎に1つ、4つの入力スキャン副計画があります。明確にするため、Updateノードには対応する副計画と同じ順に更新される特定の対象テーブルを示す注釈が付けられています。(この注釈はPostgreSQL 9.5からの新しいものです。以前のバージョンでは副計画を調べることで対象テーブルを勘で当てなければなりませんでした。)

EXPLAIN ANALYZEで表示されるPlanning timeは、解析された問い合わせから問い合わせ計画を生成し最適化するのに掛かった時間です。解析と書き換えは含みません。

EXPLAIN ANALYZEで表示されるExecution time(実行時間)にはエクゼキュータの起動、停止時間、発行される何らかのトリガの実行時間も含まれますが、解析や書き換え、計画作成の時間は含まれません。BEFOREトリガがあればその実行時間は関連するInsert、Update、Deleteノード用の時間に含まれます。しかし、AFTERトリガは計画全体が完了した後に発行されますので、AFTERトリガの実行時間は計上されません。また、各トリガ(BEFORE、AFTERのいずれか)で費やされる総時間は別々に表示されます。しかし、遅延制約トリガはトランザクションが終わるまで実行されませんので、EXPLAIN ANALYZEでは考慮されないことに注意してください。

14.1.3. 警告

EXPLAIN ANALYZEにより測定される実行時間が同じ問い合わせを普通に実行する場合と大きくそれる可能性がある、2つの重大な点があります。1つ目は、出力行がクライアントに配信されませんので、ネットワーク転送コストとI/O変換に関するコストが含まれないことです。2つ目は、EXPLAIN ANALYZEによって加わる測定オーバーヘッドが大きくなることが、特にgettimeofday()オペレーティングシステムコールが低速なマシンであり得ることです。pg_test_timingを用いて、使用中のシステムの時間測定にかかるオーバーヘッドを測ることができます。

EXPLAINの結果を試験を行ったものと大きく異なる状況の推定に使ってはいけません。例えば、小さなテーブルの結果は、巨大なテーブルに適用できるとは仮定できません。プランナの推定コストは線形ではなく、そのため、テーブルの大小によって異なる計画を選択する可能性があります。極端な例ですが、テーブルが1ディスクページしか占めない場合、インデックスが使用できる、できないに関係なく、ほとんど常にシーケンシャルスキャン計画を得ることになります。プランナは、どのような場合でもテーブルを処理するために1ディスクページ読み取りを行うので、インデックスを参照するための追加的ページ読み取りを行う価値がないことを知っています。(上述のpolygon_tblの例でこれが起こることを示しています。)

実際の値と推定値がうまく合わないが本当は間違ったものがない場合があります。こうした状況の1つは、LIMITや同様な効果により計画ノードの実行が短時間で終わる時に起こります。例えば、以前に使用したLIMIT問い合わせでは

```
EXPLAIN ANALYZE SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000 LIMIT 2;

                                                      QUERY PLAN
-----------------------------------------------------------------------
-------------------------------------------------------------
```

```
Limit  (cost=0.29..14.71 rows=2 width=244) (actual time=0.177..0.249 rows=2 loops=1)
  ->  Index Scan using tenk1_unique2 on tenk1  (cost=0.29..72.42 rows=10 width=244) (actual
time=0.174..0.244 rows=2 loops=1)
        Index Cond: (unique2 > 9000)
        Filter: (unique1 < 100)
        Rows Removed by Filter: 287
Planning time: 0.096 ms
Execution time: 0.336 ms
```

インデックススキャンノードの推定コストと行数が実行完了したかのように表示されます。しかし現実では、Limitノードが2行を取り出した後に行の要求を停止します。このため実際の行数は2行のみであり、実行時間は提示された推定コストより小さくなります。これは推定間違いではなく、単なる推定値と本当の値を表示する方法における矛盾です。

またマージ結合には、注意しないと混乱を招く測定上の乱れがあります。マージ結合は他の入力が使い尽くされ、ある入力の次のキー値が他の入力の最後のキー値より大きい場合、その入力の読み取りを停止します。このような場合、これ以上一致することはあり得ず、最初の入力の残りをスキャンする必要がありません。この結果、子のすべては読み取られず、LIMITの説明のようになります。また、外部(最初)の子が重複するキー値を持つ行を含む場合、内部(2番目)の子はバックアップされ、そのキー値が一致する行部分を再度スキャンされます。EXPLAIN ANALYZEはこうした繰り返される同じ内部行の排出を実際の追加される行と同様に計上します。外部で多くの重複がある場合、内部の子計画ノードで繰り返される実際の行数は、内部リレーションにおける実際の行数より非常に多くなることがあり得ます。

実装上の制限のため、BitmapAndおよびBitmapOrノードは常に実際の行数をゼロと報告します。

通常EXPLAINはプランナが生成したすべてのプランノードを表示します。しかし、プラン時にパラメータ値が入手できずそのノードが行を生成できないために、エクゼキューターがあるノードが実行不要であると判断できるケースがあります。(今の所、これはパーティションテーブルを走査しているAppendあるいはMergeAppendノードの子ノードでのみ起きることがあります。) これが起きると、これらのプランノードはEXPLAINの出力から削除され、Subplans Removed: Nという注釈が代わりに表示されます。

14.2. プランナで使用される統計情報

14.2.1. 単一列統計情報

前節で説明した通り、問い合わせプランナは、より良い問い合わせ計画を選択するために問い合わせによって取り出される行数の推定値を必要としています。本節では、システムがこの推定に使用する統計情報について簡単に説明します。

統計情報の1つの構成要素は、各テーブルとインデックスの項目の総数と、各テーブルとインデックスが占めるディスクブロック数です。この情報はpg_classのreltuplesとrelpages列に保持されます。以下のような問い合わせによりこれを参照することができます。

```
SELECT relname, relkind, reltuples, relpages
FROM pg_class
WHERE relname LIKE 'tenk1%';
```

```
       relname        | relkind | reltuples | relpages
----------------------+---------+-----------+----------
 tenk1                | r       |     10000 |      358
 tenk1_hundred        | i       |     10000 |       30
 tenk1_thous_tenthous | i       |     10000 |       30
 tenk1_unique1        | i       |     10000 |       30
 tenk1_unique2        | i       |     10000 |       30
(5 rows)
```

ここで、tenk1とそのインデックスには10000行が存在し、そして、(驚くには値しませんが)インデックスはテーブルよりもかなり小さなものであることがわかります。

効率を上げるため、reltuplesとrelpagesは処理の度には更新されず、したがって通常は多少古い値のみ所有しています。これらはVACUUM、ANALYZE、CREATE INDEXなどの一部のDDLコマンドによって更新されます。テーブル全体をスキャンしないVACUUM、ANALYZE操作(一般的な状況です)は、スキャンされたテーブルの部分に基づいてreltuples値を漸次更新し、概算値を生成します。いずれの場合でもプランナは、現在の物理的なテーブルサイズに合わせるためにpg_classから検索した値を調整して、より高精度な近似値を得ます。

ほとんどの問い合わせは、検証される行を制限するWHERE句によって、テーブル内の行の一部のみを取り出します。したがって、プランナはWHERE句の選択性、つまりWHERE句の各条件にどれだけの行が一致するかを推定する必要があります。この処理に使用される情報はpg_statisticシステムカタログ内に格納されます。pg_statistic内の項目は、ANALYZEとVACUUM ANALYZEコマンドによって更新され、また1から更新がかかったとしても常に概算値になります。

統計情報を手作業で確認する場合、pg_statisticを直接参照するのではなく、pg_statsビューを参照する方が良いでしょう。pg_statsはより読みやすくなるように設計されています。さらに、pg_statsは誰でも読み取ることができますが、pg_statisticはスーパーユーザのみ読み取ることができます。(これは、非特権ユーザが統計情報から他人のテーブルの内容に関わる事項を読み取ることを防止します。pg_statsビューは現在のユーザが読み取ることができるテーブルに関する行のみを表示するよう制限されています。) 例えば、以下を行うことができます。

```
SELECT attname, inherited, n_distinct,
       array_to_string(most_common_vals, E'\n') as most_common_vals
FROM pg_stats
WHERE tablename = 'road';

 attname | inherited | n_distinct |          most_common_vals
---------+-----------+------------+------------------------------------
 name    | f         |  -0.363388 | I- 580                        Ramp+
         |           |            | I- 880                        Ramp+
         |           |            | Sp Railroad                       +
         |           |            | I- 580                            +
         |           |            | I- 680                        Ramp
 name    | t         |  -0.284859 | I- 880                        Ramp+
         |           |            | I- 580                        Ramp+
```

```
         |           |           | I- 680                        Ramp+
         |           |           | I- 580                            +
         |           |           | State Hwy 13                  Ramp
(2 rows)
```

同じ列に対して2行が表示されていることに注意してください。1つはroadテーブルが始まる継承階層(inherited=t)全体に相当し、もう1つはroadテーブル自身(inherited=f)のみを含むものです。

ANALYZEによりpg_statisticに格納される情報量、具体的には、それぞれの列に対するmost_common_vals内とhistogram_bounds配列のエントリの最大数は、ALTER TABLE SET STATISTICSコマンドによって列ごとに、default_statistics_target設定パラメータを設定することによってグローバルに設定することができます。現在のデフォルトの上限は100エントリです。この上限を上げることで、特に、少し変わったデータ分布を持つ列でより正確なプランナの推定が行われますが、pg_statisticにより多くの容量が必要になり、多少推定計算にかかる時間が多くなります。反対に上限を下げることは、単純なデータ分布の列に対して順当です。

プランナによる統計情報の使用に関する詳細については第70章を参照してください。

14.2.2. 拡張統計情報

問い合わせ句で使われている複数列に相関性があることにより、悪い実行計画を実行する遅いクエリがしばしば観察されます。プランナは通常複数の条件がお互いに独立であるとみなしますが、列の値に相関性がある場合はそれは成り立ちません。通常の列ごとの統計情報は、それが個々の列ごとであるという性質上、列をまたがる相関性に関する知識を把握することはできません。しかしながら、PostgreSQLは、*多変量統計情報*を計算することができ、それによってそうした情報を把握することができます。

列の組み合わせの数は非常に大きいため、自動的に多変量統計情報を計算するのは現実的ではありません。代わりに、サーバが興味のある列の集合にまたがる統計情報を得るように指示する目的で、*拡張統計情報オブジェクト*（しばしば単に*統計情報オブジェクト*と呼ばれます）を作成することができます。

統計情報オブジェクトはCREATE STATISTICSで作成します。そうしたオブジェクトを作っても、単に統計情報として興味があることを示すカタログエントリが作られるだけです。実際のデータ収集は、ANALYZE（手動のコマンドを起動あるいはバックグラウンドでの自動ANALYZE）が行います。収集したデータは、pg_statistic_ext_dataカタログで参照することができます。

通常の単一列統計情報の計算に使うのと同じテーブル行のサンプルに基づき、ANALYZEは、拡張統計情報を計算します。（前節で述べたように）テーブルあるいはそのテーブルの対象となる列統計情報の増やすと、サンプルのサイズも増えるので、より大きな統計情報の対象を使うと、通常、より精度の高い拡張統計情報を得られますが、計算に費やす時間も増えます。

次の節では、現在サポートしている拡張統計情報の種類を説明します。

14.2.2.1. 関数従属性

もっとも単純な拡張統計情報は、データベースの正規形の定義で使われる考え方である、*関数従属性*を追跡します。aの値に関する知識がbの値を決定するのに十分であるとき、列bは列aに関数的に従属していると言います。これはすなわち、同じaの値を持ちながら、異なるbの値を持つ二つの行は存在しないということで

す。完全に正規化されたデータベースでは、関数従属性は主キーと超キーにのみ存在します。実際には様々な理由でデータの集合は完全には正規化されません。性能上の理由により非正規化するというのが典型的な例です。完全に正規化されたデータベースにおいても、ある列の間に部分的な相関関係が存在することがあり、これは部分的関数従属性として表現されます。

ある問い合わせでは、関数従属性が存在することが見積もりの精度に直接影響を与えます。問い合わせに独立した列と依存する列の両方に関する条件が含まれていると、依存する列に関する条件はそれ以上結果サイズを小さくしません。しかし関数従属性に関する知識がなければ、クエリプランナはそれらの条件が独立であると見なし、結果サイズの過少見積もりすることになります。

プランナに関数従属性について知らせるために、ANALYZEは列をまたがる依存性の強さを収集することができます。すべての列の集合間の依存性度合いを調査するのは、受け入れられないほど高価になります。そこでデータ収集は、dependenciesオプションで定義された統計情報オブジェクトの中に一緒に現れた列のグループに制限されます。ANALYZEおよび後々のクエリプランニングにおける不必要なオーバーヘッドを避けるために、強い相関関係のある列のグループのみを対象に、dependencies統計情報を作成することをお勧めします。

関数従属性統計情報の収集例です。

```
CREATE STATISTICS stts (dependencies) ON city, zip FROM zipcodes;

ANALYZE zipcodes;

SELECT stxname, stxkeys, stxddependencies
  FROM pg_statistic_ext join pg_statistic_ext_data on (oid = stxoid)
  WHERE stxname = 'stts';
 stxname | stxkeys |             stxddependencies
---------+---------+------------------------------------------
 stts    | 1 5     | {"1 => 5": 1.000000, "5 => 1": 0.423130}
(1 row)
```

ここでは、列1（zip code）が完全に列5（city）を決定しているので、係数は1.0です。一方、cityはzip codeを42%しか決定していないので、一つ以上のzip codeで表現されている多くのcity(58%)が存在するということになります。

関数従属する列を伴うクエリの選択率を計算する際、過小評価を行わないように、プランナは依存性係数を使って条件ごとの選択率を調整します。

14.2.2.1.1. 関数従属性の制限事項

今のところ、列と定数を比較する単純な等価条件と、定数のIN句を考慮する際にしか関数従属性は適用されません。二つの列を比較する、あるいは列を式と比較する等価条件、範囲句、LIKEその他の条件の見積もりを改善するのには使われません。

関数従属性を含めた見積もりでは、プランナは関係する列に対する複数の条件が同時に成り立つ、つまり冗長であるとみなします。それらの条件が同時に成り立たなければ、正しい見積もりは0行となりますが、その可能性は考慮されません。たとえば次のクエリを見てください。

```
SELECT * FROM zipcodes WHERE city = 'San Francisco' AND zip = '94105';
```

プランナは、選択率が変わらないという正しい推定に基づきcity句を無視します。しかし、これを満たす行が0行であるにもかかわらず、次の問い合わせでも同じ推測をします。

```
SELECT * FROM zipcodes WHERE city = 'San Francisco' AND zip = '90210';
```

関数従属性統計情報は、これを結論付けるだけの十分な情報を提供しません。

多くの実用的な場合には、この前提は通常満たされます。たとえば、あるアプリケーションには、クエリの中で両立するcityとZIP codeだけを許すGUIが備わっているかもしれません。もしそうでなければ、関数従属性は実行可能なオプションではないかもしれません。

14.2.2.2. 多変量N個別値計数

単一列統計情報は、それぞれの列で異なる値の数を保持します。たとえば、GROUP BY a, bのように、二つ以上の列を組み合わせての異なる値の数の見積もりは、プランナに単一列の統計情報だけしか与えられない場合は、しばしば間違ったものになり、プランナは悪いプランの選択をしてしまいます。

見積もり改善のために、列のグループに対してANALYZEはN個別統計情報を収集することができます。以前述べたのと同様に、可能なすべての列のグループに対してこれを行なうのは現実的ではありません。ndistinctオプションで定義された統計オブジェクト中に一緒に現れる列のグループに対してのみデータを収集します。列リストの中から、可能な二つ以上の列の組み合わせそれぞれに対してデータが収集されます。

先ほどの例の続きで、ZIP codeのテーブルのN個別値計数は次のようになります。

```
CREATE STATISTICS stts2 (ndistinct) ON city, state, zip FROM zipcodes;

ANALYZE zipcodes;

SELECT stxkeys AS k, stxdndistinct AS nd
  FROM pg_statistic_ext join pg_statistic_ext_data on (oid = stxoid)
  WHERE stxname = 'stts2';
-[ RECORD 1 ]----------------------------------------------------------
k  | 1 2 5
nd | {"1, 2": 33178, "1, 5": 33178, "2, 5": 27435, "1, 2, 5": 33178}
(1 row)
```

この例では、33178の異なる値を持つ列の組み合わせが三つあることを示しています。ZIP codeとstate、ZIP codeとcity、cityとstateです。(これらが等しいという事実は、ZIP codeだけがテーブル中でユニークであることから期待されます。) 一方、cityとstateの組み合わせには、27435だけの異なる値があります。

グループ化で実際に使用する列の組み合わせで、かつグループ数の見積もり間違いによって悪いプランをもたらすものに対してだけ、ndistinct統計情報オブジェクトを作ることをお勧めします。さもないと、ANALYZEサイクルは単に無駄になります。

14.2.2.3. 多変量MCVリスト

列ごとに格納される別なタイプの統計情報は最頻値リスト(most-common value list)です。個々の列ごとには非常に正確な推測を可能にしますが、複数列に渡る条件を持つ問い合わせについては重大な誤った推定をもたらすことがあります。

こうした推定を改善するために、列の組み合わせのMCVリストをANALYZEで収集することができます。関数従属性とN個別値係数同様、考えられるすべての列のグループに対してこれを行うのは実用的ではありません。MCVリストでは（関数従属性とN個別値係数と違って）列の頻値を格納するのでなおさらです。ですからmcvオプションで定義された統計情報オブジェクト中に共通して現れる列のグループのデータだけが収集されます。

前述の例を続けましょう。ZIPコードのテーブルのMCVリストは次のようになるでしょう。（単純な形式の統計情報とは違って、MCVの内容を解析する関数が必要になります）

```
CREATE STATISTICS stts3 (mcv) ON city, state FROM zipcodes;

ANALYZE zipcodes;

SELECT m.* FROM pg_statistic_ext join pg_statistic_ext_data on (oid = stxoid),
                pg_mcv_list_items(stxdmcv) m WHERE stxname = 'stts3';

 index |         values         | nulls | frequency | base_frequency
-------+------------------------+-------+-----------+----------------
     0 | {Washington, DC}       | {f,f} |  0.003467 |        2.7e-05
     1 | {Apo, AE}              | {f,f} |  0.003067 |        1.9e-05
     2 | {Houston, TX}          | {f,f} |  0.002167 |       0.000133
     3 | {El Paso, TX}          | {f,f} |     0.002 |       0.000113
     4 | {New York, NY}         | {f,f} |  0.001967 |       0.000114
     5 | {Atlanta, GA}          | {f,f} |  0.001633 |        3.3e-05
     6 | {Sacramento, CA}       | {f,f} |  0.001433 |        7.8e-05
     7 | {Miami, FL}            | {f,f} |    0.0014 |          6e-05
     8 | {Dallas, TX}           | {f,f} |  0.001367 |        8.8e-05
     9 | {Chicago, IL}          | {f,f} |  0.001333 |        5.1e-05
   ...
(99 rows)
```

これによると市と州のもっとも頻度の高い組み合わせはDCのWashingtonで、（サンプルにおける）実際の頻度は約0.35%でした。比較の基準となる組み合わせの頻度（単純な列ごとの頻度から計算されたもの）はたった0.0027%で、2桁の過少見積になっています。

そのグループの誤推定値が間違った計画をもたらしてしまうような、条件の中で実際に一緒に使われる列の組み合わせについてのみMCV統計情報オブジェクトを作成することが望ましいです。さもないと、ANALYZEとプラン処理は単に無駄になってしまいます。

14.3. 明示的なJOIN句でプランナを制御する

明示的なJOIN構文を使って問い合わせプランナをある程度制御できます。どうしてこういうことが問題になるのか、まずその背景を見る必要があります。

単純な問い合わせ、例えば

```
SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;
```

では、プランナは自由に与えられたテーブルを任意の順で結合することができます。例えば、WHERE条件のa.id = b.idを使ってまずAとBを結合し、他のWHERE条件を使ってその結合テーブルにCを結合するといった計画を立てることができます。あるいは、BとCを結合し、その結果にAを結合することもできます。あるいは、AとCを結合し、その結果にBを結合することもできるでしょう。しかし、それでは効率が良くありません。なぜなら、結合の最適化を行うために適用できる条件がWHERE句にないので、AとCの全直積が作られるからです。(PostgreSQLのエクゼキュータでは、結合はすべて2つのテーブルの間で行われるため、このようにして1つひとつ結果を作っていかなければなりません。) 重要なのは、これらの違った結合の方法は意味的には同じ結果なのですが、実行コストは大きく異なる可能性があるということです。ですから、プランナは最も効率の良い計画を探すために可能な計画をすべて検査します。

結合の対象がせいぜい2、3個のテーブルなら心配するほど結合の種類は多くありません。しかし、テーブル数が増えると可能な結合の数は指数関数的に増えていきます。10程度以上にテーブルが増えると、すべての可能性をしらみつぶしに探索することはもはや実用的ではなくなります。6や7個のテーブルでさえも、計画を作成する時間が無視できなくなります。テーブルの数が多過ぎる時は、PostgreSQLのプランナはしらみつぶしの探索から、限られた可能性だけを探索する遺伝的確率的な探索へと切り替わります。(切り替えの閾値はgeqo_threshold実行時パラメータで設定されます。) 遺伝的探索は短い時間で探索を行いますが、必ずしも最適な計画を見つけるとは限りません。

外部結合が含まれるような問い合わせでは、通常の(内部)結合よりプランナの選択の余地が小さくなります。例えば、次のような問い合わせを考えます。

```
SELECT * FROM a LEFT JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.id);
```

この問い合わせの検索条件は前述の例と表面的には似ているように思えますが、BとCの結合結果の行に適合しないAの各行が出力されなければならないため、意味的には異なります。したがって、ここではプランナには結合順に関して選択の余地がありません。まずBとCを結合し、その結果にAを結合しなければならないのです。そういうわけで、この問い合わせでは計画を立てるのに要する時間は前の例よりも短くなります。その他の場合、プランナが安全な結合順を複数決定できる可能性があります。例えば、以下を考えてみます。

```
SELECT * FROM a LEFT JOIN b ON (a.bid = b.id) LEFT JOIN c ON (a.cid = c.id);
```

この場合、Aを先にBと結合してもCと結合しても有効です。現時点では、FULL JOINのみが完全に結合順を制限します。LEFT JOINやRIGHT JOINを含む、ほとんどの実環境では、何らかの拡張に再調整することができます。

明示的な内部結合構文(INNER JOIN、CROSS JOIN、装飾のないJOIN)は、意味的にはFROM内の入力リレーションの列挙と同じです。したがって、結合順を制約しません。

ほとんどの種類のJOINは完全に結合順を制約しませんが、PostgreSQL問い合わせプランナに、すべてのJOIN句に対してとりあえず結合順を制限させることができます。例えば、以下の3つの問い合わせは論理的には同一です。

```
SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;
SELECT * FROM a CROSS JOIN b CROSS JOIN c WHERE a.id = b.id AND b.ref = c.id;
SELECT * FROM a JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.id);
```

しかし、プランナにJOINの順番を守るように伝えた場合、2番目と3番目の問い合わせは最初のものよりも短い時間で計画を立てることができます。この効果はたった3つのテーブルでは気にするほどのものではありませんが、多くのテーブルを結合する際には最後の頼みの綱になるかもしれません。

プランナを強制的に明示的なJOINに潜在する結合順に従わせるには、join_collapse_limit実行時パラメータを1に設定してください。(以下で他の取り得る値について説明します。)

検索時間を節約するために、結合順を完全に束縛する必要はありません。なぜなら、単純なFROMリストの項目内にJOIN演算子を使っても構わないからです。例えば、次の例です。

```
SELECT * FROM a CROSS JOIN b, c, d, e WHERE ...;
```

join_collapse_limit = 1とした場合、プランナは強制的に他のテーブルと結合する前にAとBを結合しますが、それ以外については特に拘束はありません。この例では、結合順の候補は5の階乗分の1に減ります。

こうした方法でプランナの検索に制約を加えることは、計画作成時間の短縮とプランナに対する優れた問い合わせ計画への方向付けの両方のために有用な技法です。プランナが劣った結合順をデフォルトで選択するのであれば、JOIN構文経由でより良い順番を選択するように強制することができます。ただし、より良い順番を理解しているという前提があります。これには実験することを勧めます。

計画作成時間に影響する密接に関連した問題として、副問い合わせをその親問い合わせに折り畳むことがあります。例えば、以下を考えてみます。

```
SELECT *
FROM x, y,
    (SELECT * FROM a, b, c WHERE something) AS ss
WHERE somethingelse;
```

こうした状況は、結合を含むビューを使用する際に現れます。そのビューのSELECTルールはビューを参照するところに挿入され、上のような問い合わせを生成します。通常、プランナは副問い合わせを親問い合わせに折り畳み、以下を生成します。

```
SELECT * FROM x, y, a, b, c WHERE something AND somethingelse;
```

これは通常、副問い合わせの計画を別途作成するより優れた計画を作成します。(例えば、外部のWHERE条件はXをAに結合するようになり、まずAの多くの行が取り除かれます。これにより、副問い合わせの完全な論理的出力が不要になります。) しかし、同時に計画作成時間が増加します。この場合、2つの3通りの結合問題から5通りの結合問題になります。候補数は指数関数的に増加するため、これは大きな違いになります。プランナは大規模な結合検索問題で行き詰まらないように、もしfrom_collapse_limit個のFROM項目が親問い合わせで発生してしまう場合は副問い合わせの折り畳みを抑制します。この実行時パラメータの値を上下に調整することで計画作成時間と計画の質をトレードオフすることができます。

両者はほとんど同じことを行うため、from_collapse_limitとjoin_collapse_limitは似たような名前になっています。片方は副問い合わせの「平坦化」をプランナがいつ行うかを制御し、もう片方は明示的な結合の平

坦化をいつ行うかを制御します。通常、join_collapse_limitをfrom_collapse_limitと同じ値に設定する（明示的な結合と副問い合わせの動作を同じにする）か、join_collapse_limitを1に設定する（明示的な結合で結合順を制御したい場合）かのどちらかを行います。しかし、計画作成時間と実行時間の間のトレードオフを細かく調整するつもりであれば、これらを別の値に設定しても構いません。

14.4. データベースへのデータ投入

データベースにデータを初期投入するために、大量のテーブル挿入操作を行う必要がままあります。本節では、この作業を効率良く行うためのちょっとした提言を示します。

14.4.1. 自動コミットをオフにする

複数回のINSERTを実行するのであれば、自動コミットを無効にして最後に1回だけコミットしてください。（普通のSQLでは、これはBEGINを開始時に、COMMITを最後に発行することを意味します。クライアント用ライブラリの中にはこれを背後で実行するものもあります。その場合は、要望通りにライブラリが行っているかどうかを確認しなければなりません。）各挿入操作で個別にコミットすることを許すと、PostgreSQLは行を追加する度に多くの作業をしなければなりません。1つのトランザクションですべての挿入を行うことによるもう1つの利点は、1つの行の挿入に失敗した場合、その時点までに挿入されたすべての行がロールバックされることです。その結果、一部のみがロードされたデータの対処に困ることはありません。

14.4.2. COPYの使用

単一コマンドですべての行をロードするために一連のINSERTコマンドではなく、COPYを使用してください。COPYコマンドは行を大量にロードすることに最適化されています。このコマンドはINSERTに比べ柔軟性に欠けていますが、大量のデータロードにおけるオーバーヘッドを大きく低減します。COPYコマンドでテーブルにデータを投入する場合、コマンドは1つなので、自動コミットを無効にする必要はありません。

COPYを使用できない場合、準備されたINSERT文をPREPAREを使用して作成し、必要な回数だけEXECUTEを実行する方が良いでしょう。これにより、繰り返し行われるINSERTの解析と計画作成分のオーバーヘッドを省くことになります。この機能のための方法はインタフェースによって異なります。このインタフェースの文書の「準備された文」を参照してください。

COPYを使用した大量の行のロードは、ほとんどすべての場合において、INSERTを使用するロードよりも高速です。たとえ複数の挿入を単一トランザクションにまとめたとしても、またその際にPREPAREを使用したとしても、これは当てはまります。

COPYは、前もって行われるCREATE TABLEまたはTRUNCATEコマンドと同一トランザクションで行った場合に、最速です。この場合、エラーが起きた場合に新しくロードされるデータを含むファイルがとにかく削除されますので、WALを書き出す必要がありません。しかし、wal_levelがminimalに設定されている場合のみにこの方法は当てはまります。この他の場合には、すべてのコマンドをWALに書き出さなければならないためです。

14.4.3. インデックスを削除する

新規に作成したテーブルをロードする時、最速の方法は、テーブルを作成し、COPYを使用した一括ロードを行い、そのテーブルに必要なインデックスを作成することです。既存のデータに対するインデックスを作成する方が、各行がロードされる度に段階的に更新するよりも高速です。

既存のテーブルに大量のデータを追加しているのであれば、インデックスを削除し、テーブルをロード、その後にインデックスを再作成する方がよいかもしれません。もちろん、他のユーザから見ると、インデックスが存在しない間データベースの性能は悪化します。また、一意性インデックスを削除する前には熟考しなければなりません。一意性制約によるエラー検査がその期間行われないからです。

14.4.4. 外部キー制約の削除

インデックスの場合と同様、外部キー制約は一行一行検査するよりも効率的に、「まとめて」検査することができます。従って、外部キー制約を削除し、データをロード、そして、制約を再作成する方法は有用となることがあります。繰り返しますが、データロードの速度と、制約が存在しない間のエラー検査がないという点とのトレードオフがあります。

外部キー制約をすでに持つテーブルにデータをロードする時、新しい行はそれぞれ(行の外部キー制約を検査するトリガを発行しますので)サーバの待機中トリガイベントのリスト内に項目を要求します。数百万の行をロードすると、トリガイベントのキューが利用可能なメモリをオーバーフローさせてしまい、耐えられないほどのスワッピングが発生してしまう、最悪はそのコマンドが完全に失敗してしまう可能性があります。したがって単に好ましいだけでなく、大量のデータをロードする時には外部キーを削除し再度適用することが必要かもしれません。一時的な制約削除が受け入れられない場合に他に取り得る手段は、ロード操作をより小さなトランザクションに分割することだけかもしれません。

14.4.5. maintenance_work_memを増やす

大規模なデータをロードする時maintenance_work_mem設定変数を一時的に増やすことで性能を向上させることができます。これは、CREATE INDEXコマンドとALTER TABLE ADD FOREIGN KEYの速度向上に役立ちます。COPY自体には大して役立ちませんので、この助言は、上述の技法の片方または両方を使用している時にのみ有用です。

14.4.6. max_wal_sizeを増やす

大規模なデータをロードする時max_wal_size設定変数を一時的に増やすことで高速化することができます。大量のデータをPostgreSQLにロードすることで、通常のチェックポイントの頻度（checkpoint_timeout設定変数により指定されます）よりも頻繁にチェックポイントが発生するためです。チェックポイントが発生すると、すべてのダーティページ（ディスクに未書き込みの変更済みメモリページ）はディスクに吐き出されなければなりません。大量のデータロードの際に一時的にmax_wal_sizeを増加させることで、必要なチェックポイント数を減らすことができます。

14.4.7. WALアーカイブ処理とストリーミングレプリケーションの無効化

大量のデータをWALアーカイブ処理またはストリーミングレプリケーションを使用するインストレーションにロードする時、増加する大量のWALデータを処理するより、ロードが完了した後に新しくベースバックアップを取る方が高速です。ロード中のWALログの増加を防ぐためには、wal_levelをminimalに、

archive_modeをoffに、max_wal_sendersをゼロに設定することにより、アーカイブ処理とストリーミングレプリケーションを無効にしてください。しかし、これらの変数を変更するにはサーバの再起動が必要な点に注意してください。

こうすると、WALデータを処理する保管処理またはWAL送信処理にかかる時間がかからないことの他に、実際のところ、特定のコマンドをより高速にします。wal_levelがminimalの場合、これらのコマンドではWALへの書き出しは全く予定されないためです。(これらは最後にfsyncを実行することで、WALへの書き込みより安価にクラッシュした場合の安全性を保証することができます。)

14.4.8. 最後にANALYZEを実行

テーブル内のデータ分布を大きく変更した時は毎回、ANALYZEを実行することを強く勧めます。これは、テーブルに大量のデータをまとめてロードする場合も含まれます。ANALYZE(またはVACUUM ANALYZE)を実行することで、確実にプランナがテーブルに関する最新の統計情報を持つことができます。統計情報が存在しない、または古い場合、プランナは、そのテーブルに対する問い合わせの性能を損なわせる、お粗末な問い合わせ計画を選択する可能性があります。自動バキュームデーモンが有効な場合、ANALYZEが自動的に実行されます。詳細は24.1.3および24.1.6を参照してください。

14.4.9. pg_dumpに関するいくつかの注意

pg_dumpで生成されるダンプスクリプトは自動的に上のガイドラインのいくつかを適用します(すべてではありません)。pg_dumpダンプをできる限り高速にリロードするには、手作業で更に数作業が必要です。(これらは作成時に適用するものではなく、ダンプを復元する時に適用するものです。psqlを使用してテキスト形式のダンプをロードする時とpg_dumpのアーカイブファイルからpg_restoreを使用してロードする時にも同じことが適用できます。)

デフォルトでは、pg_dumpはCOPYを使用します。スキーマとデータのダンプ全体を生成する場合、インデックスと外部キー制約を作成する前にデータをロードすることに注意してください。ですので、この場合、ガイドラインのいくつかは自動的に行われます。残された作業は以下のとおりです。

- maintenance_work_memおよびmax_wal_sizeを適切な(つまり通常よりも大きな)値に設定します。
- WALアーカイブ処理またはストリーミングレプリケーションを使用する場合は、リストア時にこれを無効にすることを検討してください。このためにはダンプをロードする前にarchive_modeをoffに、wal_levelをminimalに、max_wal_sendersをゼロに設定してください。その後それらを正しい値に戻し、新規にベースバックアップを取ってください。
- pg_dumpとpg_restoreで、並列ダンプとリストア方式を実験して、利用する並列なジョブの最適な数を見つけて下さい。-jオプションでダンプとリストアを並列に行なうのは逐次方式よりも大きく性能を向上させるでしょう。
- ダンプ全体を単一トランザクションとしてリストアすべきかどうか検討してください。このためにはpsqlまたはpg_restoreに-1または--single-transactionコマンドラインオプションを指定してください。このモードを使用する場合、たとえ小さなエラーであっても、エラーがあればリストア全体がロールバックされます。データ同士の関連性がどの程度あるかに依存しますが、手作業での整理の際には好まれるかと思います。さもなくばあまり勧めません。単一トランザクションで実行し、WALアーカイブを無効にしている場合、COPYコマンドは最も高速に行われます。

- データベースサーバで複数のCPUが利用できるのであれば、pg_restoreの--jobsオプションの利用を検討してください。これによりデータのロードとインデックスの作成を同時に行うことができます。
- この後でANALYZEを実行してください。

データのみのダンプもCOPYコマンドを使用しますが、インデックスの削除と再作成を行いません。また、通常は外部キー制約を変更しません。[1] したがって、データのみのダンプをロードする時、上の技法を使用したければ自らインデックスと外部キーを削除、再作成しなければなりません。データをロードする時にmax_wal_sizeを増やすことも有用です。しかし、maintenance_work_memを増やすことは考えないでください。これは、後でインデックスと外部キーを手作業で再作成する時に行う方がよいでしょう。また、実行した後でANALYZEを行うことを忘れないでください。詳細は24.1.3および24.1.6を参照してください。

14.5. 永続性がない設定

永続性とは、サーバがクラッシュしたり電源が落ちたりしたとしても、コミットされたトランザクションが記録されていることを保証するデータベースの機能です。しかし、永続性はデータベースに多くのオーバーヘッドを与えます。このためこの保証を必要としないサイトでは、PostgreSQLをかなり高速に実行するように設定することができます。以下に、こうした状況で性能を向上させるために行うことができる設定変更を示します。後述の注意を除き、データベースソフトウェアがクラッシュした場合でも、永続性は保証されています。突然のオペレーティングシステムの停止だけが、この設定を使用した時のデータ損失、破損の危険性を引き起こします。

- データベースクラスタのデータディレクトリをメモリ上のファイルシステム（つまりRAMディスク）に設定します。これはすべてのデータベースによるディスクI/Oを取り除きますが、データ量が利用可能なメモリ（およびスワップも使われるかもしれません）量までに制限されます。
- fsyncを無効にします。データをディスクに吐き出す必要がありません。
- synchronous_commitを無効にします。コミット毎にディスクにWAL書き出しを強制する必要がありません。この設定は、データベースがクラッシュした場合、トランザクション損失（データ破損ではありません）の危険性があります。
- full_page_writesを無効にします。部分的なページ書き出しから保護する必要がありません。
- max_wal_sizeおよびcheckpoint_timeoutを増加させます。これによりチェックポイントの頻度が減少しますが、/pg_walで必要とする容量が増加します。
- WAL書き出しを回避するためには、テーブルがクラッシュに対して安全ではなくなりますが、ログを取らないテーブルを作成してください。

[1] --disable-triggersオプションを使用して、外部キーを無効にさせることができます。しかし、これは外部キー制約を遅らせるのではなく、除去することに注意してください。そのため、これを使用すると不正なデータを挿入することができてしまいます。

第15章 パラレルクエリ

PostgreSQLは、クエリの応答をより速くするために、複数のCPUを活用するクエリプランを生成することができます。この機能は、パラレルクエリとして知られています。多くのクエリはパラレルクエリの恩恵にあずかることができません。その理由は、現在の実装の制限によるもの、あるいは直列にクエリを実行するよりも速いと思われるクエリプランが存在しないため、のどちらかです。しかし、パラレルクエリの恩恵にあずかることのできるクエリでは、パラレルクエリによる高速化は、しばしばかなりのものとなります。多くのクエリではパラレルクエリを使用すると2倍以上速くなり、中には4倍かそれ以上に速くなるものもあります。大量のデータにアクセスするが、返却する行が少ないクエリが典型的には最大の恩恵にあずかります。この章では、パラレルクエリの利用を希望しているユーザが、そこから何が期待できるのかを理解できるようにするために、パラレルクエリの動作の詳細と、どのような状況でユーザがパラレルクエリを使用できるのか説明します。

15.1. パラレルクエリはどのように動くのか

あるクエリの最速の実行戦略がパラレルクエリであるとオプティマイザが決定すると、*Gather*または*Gather Merge*ノードを含むクエリプランを作成します。単純な例を示します。

```
EXPLAIN SELECT * FROM pgbench_accounts WHERE filler LIKE '%x%';
                                     QUERY PLAN
------------------------------------------------------------------------------------
 Gather  (cost=1000.00..217018.43 rows=1 width=97)
   Workers Planned: 2
   ->  Parallel Seq Scan on pgbench_accounts  (cost=0.00..216018.33 rows=1 width=97)
         Filter: (filler ~~ '%x%'::text)
(4 rows)
```

どの場合でも、GatherまたはGather Mergeノードは、正確に一つの子ノードを持ちます。子プランは、プランの中で並列に実行される部分です。GatherまたはGather Mergeノードがプランツリーの中で最上位にある場合は、クエリ全体が並列に実行されます。GatherまたはGather Mergeノードがプランツリーの他の部分にある場合は、その部分だけが並列に実行されます。上の例では、クエリはただ一つのテーブルにアクセスするので、Gatherノード自身以外では、たった一つのプランノードだけが存在します。そのプランノードはGatherノードの子ノードなので、並列に実行されます。

EXPLAINを使って、プランナが選択したワーカーの数を見ることができます。クエリの実行中にGatherノードに到達すると、ユーザのセッションに対応しているプロセスは、プランナが選択したワーカーと同じ数のバックグラウンドワーカープロセスを要求します。プランナが使用を検討するバックグラウンドワーカーの数は、最大でもmax_parallel_workers_per_gatherに制限されます。ある時点で存在できるバックグラウンドワーカーの数は、max_worker_processesとmax_parallel_workersの両方を満たすように制限されます。ですから、あるパラレルクエリが、プラン時よりも少ない数のワーカープロセスによって実行されたり、まったくワーカープロセスなしに実行されることがあり得ます。最適なプランは利用可能なワーカーの数に依存することもあるので、これは低い性能をもたらす結果になるかもしれません。これがしばしば起こるようなら、max_worker_processesとmax_parallel_workersを増やしてより多くのワーカーが同時に実行できるようにするか、max_parallel_workers_per_gatherを減らして、プランナがより少ない数のワーカーを要求するようにすることを考慮してください。

与えられたパラレルクエリから起動されたすべてのバックグラウンドワーカープロセスは、そのプランの一部を実行します。リーダーはそうしたプランの部分を実行するだけでなく、追加の任務が与えられます。つまり、ワーカーが生成したすべてのタプルを読み込まなければなりません。プラン中のパラレル部分が少数のタプルしか生成しない場合は、リーダーは追加のワーカーとほぼ同じように振る舞い、クエリの実行を高速化します。反対にプラン中のパラレル部分が大量のタプルを生成する場合は、リーダーはワーカーが生成したタプルの読み込みと、Gatherノードあるいは`Gather Merge`より上位のプランノードが要求する追加の処理ステップに忙殺されるかもしれません。そのような場合は、リーダーはプランの並列実行部分のごく一部しか処理しません。

プランの並列部分の最上位ノードがGatherではなくて`Gather Merge`なら、プランの並列部分を実行する各プロセスはタプルをソート順に生成し、リーダーはソート順を保存するマージを実行していることを意味します。対照的に、Gatherは、ワーカーから都合の良い順でタプルを読み込むので、ソート順が存在しているとしても、それを壊してしまいます。

15.2. どのような時にパラレルクエリは使用できるのか?

どのような状況においても、プランナにパラレルクエリプランを生成させなくしてしまう設定があります。とにかくパラレルクエリプランを生成させるためには、次に示すように設定しなければなりません。

- max_parallel_workers_per_gatherは0より大きい値に設定しなければなりません。`max_parallel_workers_per_gather`で設定した数以上のワーカーは使用されないという一般原則に含まれる個別のケースです。

加えて、システムはシングルユーザモードで動いていてはいけません。この場合はデータベースシステム全体が一つのプロセスで動いているので、バックグラウンドワーカーが使えません。

一般にパラレルクエリプランが生成可能な場合でも、以下のうち一つでも真であると、プランナはクエリに対するパラレルクエリプランを生成しません。

- クエリがデータを書き込むか、データベースの行をロックする場合。クエリがデータ更新操作をトップレベルあるいはCTE内で含むと、そのクエリに対するパラレルプランは生成されません。例外として、新たなテーブルを作ってデータ投入するコマンド`CREATE TABLE ... AS`、`SELECT INTO`、および、`CREATE MATERIALIZED VIEW`ではパラレルプランを使用できます。

- クエリが実行中に一時停止する場合。クエリの一部あるいは増分の実行が発生するとシステムが判断すると、パラレルプランは生成されません。たとえば、DECLARE CURSORで作られるカーソルは、決してパラレルプランを使用しません。同様に、`FOR x IN query LOOP .. END LOOP`のPL/pgSQLループは、決してパラレルプランを使用しません。パラレルクエリが実行中に、ループの中のコードを実行しても安全かどうか、パラレルクエリシステムが判断できないからです。

- クエリが`PARALLEL UNSAFE`とマーク付されている関数を使っています。ほとんどのシステム定義の関数は`PARALLEL SAFE`です。しかし、ユーザ定義関数はデフォルトで`PARALLEL UNSAFE`とマーク付されます。15.4の議論をご覧ください。

- クエリが、すでにパラレル実行している別のクエリの内部で走っている場合。たとえば、パラレルクエリから呼ばれている関数自身がSQLクエリを発行すると、そのクエリは決してパラレルプランを使用しません。

これは現在の実装の制限によるものですが、この制限を取り外すのは好ましくないかもしれません。なぜなら、単一のクエリが非常に大きな数のプロセスを使用する結果となることがあり得るからです。

あるクエリに対してパラレルクエリプランが生成された場合でも、実行時にプランを並列に実行できないような状況があります。この状況においては、まるでGatherノードが存在しなかったかのように、リーダはGatherノード以下部分のプランのすべてを自分自身で実行します。これは、以下の条件のどれかが当てはまると起こります。

- バックグラウンドワーカー数の合計がmax_worker_processesを超えてはいけない、という制限によってバックグラウンドワーカーが得られない場合。
- パラレルクエリ目的で起動されたバックグラウンドワーカー数の合計がmax_parallel_workersを超えてはいけない、という制限によってバックグラウンドワーカーが得られない場合。
- クライアントが0ではないフェッチカウント付きのExecuteメッセージを送信した場合。拡張問い合わせプロトコルの議論をご覧ください。現在のlibpqにはそのようなメッセージを送る方法がないため、これはlibpqに依存しないクライアントを使った時にだけ起こります。これが頻繁に起こるようなら、順次実行したときに最適ではないプランが生成されるのを防ぐために、それが起こりそうなセッションの中で、max_parallel_workers_per_gatherを0に設定すると良いかもしれません。

15.3. パラレルプラン

各々のワーカーは完了すべきプランのパラレル部分を実行するので、単に通常のクエリプランを適用し、複数のワーカーを使って実行することはできません。それぞれのワーカーが結果セットの全体のコピーを生成するので、クエリは通常よりも決して速くなりませんし、不正な結果を生成してしまいます。そうではなくて、プランのパラレル部分は、クエリオプティマイザの内部で*部分プラン*として知られているものでなくてはなりません。すなわち、プランを実行する各プロセスが、要求される個々の出力行が、協調動作するプロセスの正確に1個だけによって生成されることが保証されているような方法で、出力行の一部だけを生成します。一般に、これはクエリの処理対象のテーブルに対するスキャンは、パラレル対応のスキャンでなければならないことを意味します。

15.3.1. パラレルスキャン

今のところ、次に示すパラレル対応のテーブルスキャンがサポートされています。

- *パラレルシーケンシャルスキャン*では、テーブルのブロックは、協調するプロセスに分割して割り当てられます。ブロックは一度に1個ずつ処理され、テーブルへのアクセスは逐次のままです。
- *パラレルビットマップヒープスキャン*では、一つのプロセスがリーダーに選ばれます。そのプロセスは、一つ以上のインデックスをスキャンし、アクセスする必要のあるブロックを示すビットマップを作成します。次にこれらのブロックは、パラレルシーケンシャルスキャン同様、協調するプロセスに割り当てられます。つまり、ヒープスキャンは並列であるものの、対応するインデックスのスキャンは並列ではありません。
- *パラレルインデックススキャン*あるいは*パラレルインデックスオンリースキャン*では、協調するプロセスは、交代でインデックスからデータを読み込みます。今のところ、パラレルインデックススキャンは、btreeインデックスのみでサポートされています。個々のプロセスは単一のインデックスブロックを要求し、ス

キャンしてそのブロックから参照されているすべてのタプルを返却します。他のプロセスは同時に他のインデックスからタプルを返却することができます。並列btreeスキャンの結果は、ワーカー内におけるソート順の結果で返却されます。

btree以外のインデックススキャンのような他のスキャンタイプは、将来パラレルスキャンをサポートするかもしれません。

15.3.2. パラレルジョイン

非パラレルプランと同様、処理対象のテーブルは、1個以上の他のテーブルとネステッドループ、ハッシュ結合、マージ結合で結合することができます。結合の内側は、パラレルワーカー中で実行しても安全だという条件下で、プランナがサポートするどのような非パラレルプランであっても構いません。結合タイプによっては内側がパラレルプランであってもよいです。

- ネステッドループ結合では、内側は常に非パラレルです。外側タプルとこのようなインデックスで値を探すループは共同するプロセス間で分割されるので、全体で実行されても内側がインデックススキャンであるなら、これは効率的です。

- マージ結合では、内側は常に非パラレルプランで、それゆえに全体で実行されます。特にソート実行を要する場合、全ての共同プロセスで処理と結果データが重複するので、これは非効率的と考えられます。

- (parallelが付かない)ハッシュ結合では、内側は全ての共同プロセスがハッシュテーブルの同じコピーを作ることで、全体で実行されます。ハッシュテーブルが大きかったり、そのプランが高価である場合、これは非効率的と考えられます。パラレルハッシュ結合では、内側は共有ハッシュテーブルの構築処理を共同プロセス間で分割するパラレルハッシュです。

15.3.3. パラレル集約

PostgreSQLは、ふたつのステージで集約処理を行うことによってパラレル集約処理をサポートします。まず、クエリのパラレル部分に参加している個々のプロセスが集約ステップを実行し、それぞれのプロセスが認識しているグループに対する部分的な結果を生成します。これはPartial Aggregateノードとしてプラン中に反映されています。次に、GatherまたはGather Mergeノードを通じて部分的な結果がリーダーに転送されます。最後に、リーダーは、すべてのワーカーにまたがる結果を再集約して、最終的な結果を生成します。これは、Finalize Aggregateノードとしてプラン中に反映されています。

Finalize Aggregateノードはリーダープロセスで実行されるので、入力行数の割には、比較的多数のグループを生成するクエリは、クエリプランナはあまり好ましくないものとして認識します。たとえば最悪の場合、Finalize Aggregateノードが認識するグループ数は、Partial Aggregateですべてのワーカープロセスが認識する入力行数と同じだけの数になります。こうした場合には、明らかにパラレル集約を利用する性能上の利点がないことになります。クエリプランナはプラン処理中にこれを考慮するので、このシナリオでパラレル集約を採用することはまずありません。

どんな状況でもパラレル集約がサポートされているわけではありません。個々の集約は並列処理安全で、結合関数(combine function)を持っていなければなりません。その集約がinternal型の遷移状態を持っているならば、シリアライズ関数とデシリアライズ関数を持っていなければなりません。更なる詳細はCREATE AGGREGATEをご覧ください。パラレル集約は、集約関数呼び出しがDISTINCTあるいはORDER BY句を含む場合、また 順序集合集約、あるいはクエリがGROUPING SETSを実行する場合にはサポートされません。パラレ

ル集約は、クエリの中で実行されるすべての結合が、プラン中の並列実行部分の一部であるときにのみ利用できます。

15.3.4. パラレルアペンド

PostgreSQLが複数のソースから一つの結果セットへの行の連結を必要とするときはいつでも、AppendまたはMergeAppendプランノードが使われます。これは一般にUNION ALLを実施するときや、パーティションテーブルをスキャンするときに発生します。他のプランと同様にこのようなノードをパラレルプランで使うことができます。しかしながら、パラレルプランではプランナは代わりにParallel Appendノードを使ってもよいです。

Appendノードがパラレルプランで使われるとき、各プロセスは子プランをそれらの出現順に実行します。そのため、全ての参加しているプロセスは共同して最初の子プランを完了するまで実行して、その後、一斉に次プランに移ります。代わりにParallel Appendが使われるときには、エクゼキュータは逆に参加しているプロセスを各子プランにできるだけ均等に分散させます。そのため、複数の子プランは同時並行に実行されます。これは競合を回避し、また、プランを実行することのないプロセスで子プランの開始コストが生じることも回避します。

また、パラレルプランの中で使われるときだけ部分的な子プランを持てる、通常のAppendノードと違い、Parallel Appendノードは部分的、非部分的のどちらの子プランも持つことができます。複数回のスキャンは重複した結果をもたらすため、非部分的な子プランは単一プロセスのみからスキャンされます。複数の結果セットの連結に関わるプランは、効率的なパラレルプランが不可能なときでも、それゆえ粗い並列性を実現できます。例えば、パラレルスキャンをサポートしないインデックスを使うことでのみ効率的に実行できるパーティションテーブルに対する問い合わせを考えてください。プランナは通常のIndex Scanプランの Parallel Appendを選ぶことができます。個々のインデックススキャンは単一プロセスで最後まで実行しなければなりませんが、別のスキャンは同時に別プロセスで実行することができます。

本機能を無効にするためにenable_parallel_appendを使用できます。

15.3.5. パラレルプランに関するヒント

パラレルプランを生成すると期待していたクエリがそうならない場合には、parallel_setup_costまたはparallel_tuple_costを減らしてみてください。もちろん、このプランは結局のところ、プランナが選択した順次実行プランよりも遅いということもあり得ますが、いつもそうだとは限りません。これらの設定値を非常に小さく(つまり両方とも0に)したにも関わらずパラレルプランを得られない場合、あなたのクエリのためにクエリプランナがパラレルプランを生成できない何か理由があるのかもしれません。そうしたケースに該当しているかどうかを、15.2と15.4を参照して確認してください。

パラレルプランを実行する際には、EXPLAIN (ANALYZE, VERBOSE)を使って個々のプランノードに対するワーカーごとの状態を表示することができます。これは、すべてのプランノードに均等に仕事が分散されているかどうかを確認すること、そしてもっと一般的には、プランの性能特性を理解するのに役に立つかもしれません。

15.4. パラレル安全

プランナは、クエリ中に実行される操作をパラレル安全(parallel safe)、パラレル制限(parallel restricted)、パラレル非安全(parallel unsafe)に分類します。パラレル安全操作は、パラレルクエリとコン

フリクトしない操作です。パラレル制限操作は、パラレルクエリを利用中に、パラレルワーカー中では実行できないが、リーダーによって実行できる操作です。したがって、パラレル制限操作は、GatherあるいはGather Mergeノードより下では決して実行されませんが、Gatherノードを含むプランの別の場所では実行されるかもしれません。パラレル非安全操作は、パラレルクエリ利用中に、リーダも含めて実行できない操作です。クエリがパラレル非安全なものを含む場合は、クエリ中でのパラレルクエリの利用は全くできなくなります。

次の操作は常にパラレル制限です。

- 共通テーブル式(CTE)のスキャン
- 一時テーブルのスキャン
- 外部テーブルのスキャン。外部データラッパがIsForeignScanParallelSafeAPIを持ち、パラレル安全を返す場合を除く。
- InitPlanが付加されたプランノード
- 関連するSubPlanを参照するプランノード

15.4.1. 関数と集約のためのパラレルラベル付け

プランナは、自動的にはユーザ定義関数や集約がパラレル安全か、パラレル制限か、あるいはパラレル非安全かを決定することはできません。この関数が潜在的に実行する可能性のあるすべての操作を予測することが、このために要求されるからです。一般的には、これは停止性問題と同等で、それ故に不可能です。おそらく終了できると思われる単純な関数においてさえ、私達は試みません。なぜなら、そうした予測は高価でエラーを起こしやすいからです。その代わりに、そうではないとマークされない限り、すべてのユーザ定義関数は、パラレル非安全と見なされます。CREATE FUNCTIONあるいはALTER FUNCTIONを使用するときは、適当なPARALLEL SAFE、PARALLEL RESTRICTED、PARALLEL UNSAFEを指定することによってマーキングを行うことができます。CREATE AGGREGATEを利用するときは、対応する値にしたがって、SAFE、RESTRICTED、UNSAFEのどれかをPARALLELオプションに指定します。

データベースに書き込むか、シーケンスにアクセスするか、あるいはトランザクションの状態を一時的にであっても変更する(たとえばエラーを捕捉するためにEXCEPTIONブロック確立するPL/pgSQL関数)、恒久的な設定変更を行う関数あるいは集約は、PARALLEL UNSAFEとマークされなければなりません。同様に、一時テーブル、クライアントの接続状態、カーソル、準備された文、システムがワーカーの間で同期できないその他のバックエンドローカルな状態にアクセスする関数あるいは集約は、PARALLEL RESTRICTEDとマークされなければなりません。たとえば、setseedとrandomは、最後の理由により、パラレル制限です。

一般的に制限あるいは非安全な関数が安全とラベル付されたり、実際には非安全なのに制限付きとラベル付されると、パラレルクエリの中で使用される際に、エラーを生じたり、間違った結果を生成するかもしれません。誤ったラベル付をされると、C言語関数は理論的にはまったく未定義の振る舞いを示すことがあります。システムは任意のCコードから身を守るすべがないからです。しかしもっとも起こりえる可能性としては、他の関数のよりも悪いということはなさそうです。もし自信がないなら、たぶんその関数をUNSAFEとラベル付するのが最善でしょう。

パラレルワーカーの中で実行される関数がリーダーが獲得していないロックを獲得する場合、たとえばクエリ中で参照されていないテーブルに対して問い合わせを実行する場合などは、これらのロックはトランザクションが終了した時点ではなく、ワーカーが終了する際に解放されます。もしあなたがこれを行う関数を作成し、

こうした振る舞いの違いがあなたにとって重要ならば、関数がリーダーの中だけで実行されることを保証するために、関数をPARALLEL RESTRICTEDとマーク付けしてください。

より良いプランを得るために、プランナがクエリの中で実行されるパラレル制限な関数や集約の評価の遅延を考慮することはないことに注意してください。したがって、たとえばあるテーブルに適用されるWHERE句がパラレル制限であるときに、クエリプランナはプランの並列実行部分中のテーブルに対してスキャンを実行をすることを考慮しません。ある場合においては、クエリ中のパラレル部分におけるテーブルのスキャンを含むようにして、WHERE句の評価を遅らせ、Gatherノード上で実行されるようにすることも可能でしょう(そしてその方が効率が良いことさえあります)。しかし、プランナはそうしたことは行いません。

パート III. サーバの管理

ここでは、PostgreSQLデータベース管理者にとって関心のある事項を扱います。これには、ソフトウェアのインストール、サーバの設定や構成、ユーザやデータベースの管理、および保守作業が含まれます。PostgreSQLサーバを個人的に使用している場合もそうですが、特に業務で使用している場合は、ここで扱う事項に精通している必要があります。

ここに記載された情報は、大体において新規ユーザが読み進めるべき順番に並べられています。ただ、章ごとに内容が独立していますので、必要に応じて個々の章を読むこともできます。ここでの情報は、項目単位で限定して説明されています。特定のコマンドについて完全な説明を知りたい場合は、パート VIを参照してください。

最初の数章は、これからサーバを構築する新規ユーザにも読めるように、前提知識がなくても理解できるようになっています。残りの部分では調整や管理について記されていますが、その資料は読者がPostgreSQLデータベースシステムの一般的な使用について理解していることを前提としています。詳細な情報についてはパート Iやパート IIを参照することをお勧めします。

目次

第16章 ソースコードからインストール

この章では、PostgreSQLのソースコード配布物を使用したインストール方法について説明します。RPMやDebianパッケージなどパッケージ済みの配布物をインストールしている場合は、この章を無視し、代わりにパッケージの手順を読んでください。

PostgreSQLをMicrosoft Windows向けに構築する場合、MinGWやCygwinで構築するつもりなら、この章を読んでください。MicrosoftのVisual C++で構築するつもりなら、代わりに第17章を参照してください。

16.1. 簡易版

```
./configure
make
su
make install
adduser postgres
mkdir /usr/local/pgsql/data
chown postgres /usr/local/pgsql/data
su - postgres
/usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
/usr/local/pgsql/bin/pg_ctl -D /usr/local/pgsql/data -l logfile start
/usr/local/pgsql/bin/createdb test
/usr/local/pgsql/bin/psql test
```

本章の残りで詳細を説明します。

16.2. 必要条件

通常、最近のUnix互換プラットフォームならばPostgreSQLを動作させることができるはずです。リリース時点で明示的なテストを受けていたプラットフォームを以下の16.6に示します。

PostgreSQLを構築するには、以下のソフトウェアパッケージが必要です。

- GNU makeのバージョン3.80以上が必要です。他のmakeや古いGNU makeでは動作しません。(GNU makeはときどきgmakeという名前でインストールされます。) GNU makeの試験を行うためには以下を実行してください。

  ```
  make --version
  ```

- ISO/ANSI Cコンパイラ(最低限C99-準拠)が必要です。GCC の最近のバージョンをお勧めしますが、PostgreSQLは異なるベンダの、様々なコンパイラを使用して構築できることで知られています。

- 配布物を展開するために、tarおよびgzipかbzip2のどちらかが必要です。

- GNU Readlineライブラリは、デフォルトで使用されます。これによりpsql（PostgreSQLコマンドラインSQLインタプリタ）は入力したコマンドの記憶、さらに、カーソルキーを使用した過去のコマンドの再実行や編集ができるようになります。これは非常に役に立ちますので、強く推奨します。使用したくない場合は、configureに--without-readlineオプションを指定する必要があります。その代わりとして、BSDライセンスのlibeditライブラリを使用することもできます。このライブラリは元々NetBSDで開発されていました。libeditライブラリはGNUのReadlineと互換性があり、libreadlineを認識できなかった場合やconfigureのオプションに--with-libedit-preferredが使用された場合に使用されます。パッケージベースのLinuxディストリビューションを使用し、そのディストリビューションの中でreadlineとreadline-develパッケージが別個に存在していた場合、両方とも必要ですので注意してください。

- zlib圧縮ライブラリはデフォルトで使用されます。これを使用したくなければ、configureに対し--without-zlibを指定しなければなりません。このオプションを使用すると、pg_dumpおよびpg_restore内の圧縮アーカイブサポートが無効になります。

以下のパッケージはオプションです。これらはデフォルトの設定では必要ありませんが、下記のように特定の構築オプションを有効とする場合に必要となります。

- サーバプログラム言語であるPL/Perlを構築するには、libperlライブラリとヘッダファイルを含む完全なPerlのインストレーションが必要です。必要なバージョンはPerl 5.8.3以上です。PL/Perlは共有ライブラリですので、ほとんどのプラットフォームにおいてlibperlライブラリも共有ライブラリでなければなりません。これは最近のバージョンのPerlではデフォルトのようですが、以前のバージョンではデフォルトではありませんでした。とにかく、これはPerlをサイトにインストールした人により決定されます。PL/Perlを構築することを選択したのに共有のlibperlが見つからなければ、configureは失敗するでしょう。その場合には、PL/Perlを構築できるようにするために手動でPerlを再構築してインストールしなければならないでしょう。Perlの構成プロセスには共有ライブラリが必要です。

  もし、PL/Perlを意図的に使用するつもりであるのなら、Perlのインストレーションがusemultiplicityオプションを有効にして実施されているかを確認すべきです(perl -Vにより有効かどうかを確認できます)。

- PL/Pythonサーバプログラム言語を構築するには、ヘッダファイルとdistutilsモジュールを含むPythonのインストレーションが必要です。Pythonバージョン2.6が最低でも必要です。バージョン3.1以降であればPython 3もサポートされます。しかしPython 3を使用する場合は45.1を参照してください。

  PL/Pythonは共有ライブラリになりますので、ほとんどのプラットフォームでは、libpythonもまた共有ライブラリである必要があります。ソースから構築したPythonのインストレーションでは、これはデフォルトではありませんが、共有ライブラリは多くのオペレーティングシステムのディストリビューションで入手可能です。PL/Pythonを構築することを選択したのに共有のlibpythonが見つからなければ、configureは失敗するでしょう。それは、この共有ライブラリを提供するために追加のパッケージをインストールするか、Pythonのインストレーション(の一部)を再構築しなければならないということを意味しているかもしれません。ソースから構築する場合、--enable-sharedフラグを付けてPythonのconfigureを実行してください。

- PL/Tcl手続き言語の構築には、もちろんTclのインストレーションが必要です。要求される最小のバージョンはTcl 8.4です。

- 各国語サポート（NLS）、つまり、英語以外の言語によるプログラムメッセージの表示機能を有効にするには、Gettext APIの実装が必要です。オペレーティングシステムの中には（例えば、Linux、NetBSD、Solarisなど）、組み込み済みのものがあります。他のシステムでは、追加パッケージをhttp://www.gnu.org/software/gettext/からダウンロードすることができます。GNU Cライブラリのgettextの実

装を使用する場合、さらにいくつかのユーティリティプログラムのためにGNU Gettextパッケージが必要となります。他の実装の場合には必要ありません。

- 暗号化されたクライアント接続をサポートする場合にはOpenSSLが必要です。OpenSSLは、/dev/urandomのないプラットフォーム(Windowsを除く)での乱数生成のためにも必要です。要求される最小のバージョンは1.0.1です。
- Kerberos、OpenLDAP、PAMが、そのサービスを使用した認証をサポートする場合には必要です。
- PostgreSQLの文書を構築するためには必要なセットは別途記載します。J.2を参照してください。

リリースされたソースパッケージではなくGitツリーからの構築の場合、またはサーバ開発を行いたい場合には、以下のパッケージも必要となります。

- FlexおよびBisonは、Gitチェックアウトから構築する場合や、実際のスキャナとパーサの定義ファイルを変更した場合に必要となります。それらが必要な場合は、Flex 2.5.31以降とBison 1.875以降を使うようにしてください。他のlexとyaccプログラムは使用できません。
- Gitからチェックアウトしたものから構築する場合や、構築時にPerlスクリプトを使用して作成されるファイルの入力となるファイルを変更する場合にはPerl5.8.3以降が必要です。Windows上で構築する場合、いずれにしてもPerlは必要です。Perlはテストスイートのいくつかを実行するのにも必要です。

GNUパッケージの入手が必要な場合、近くのGNUミラーサイトから探してください(ミラーサイトの一覧はhttps://www.gnu.org/prep/ftpにあります)。または、ftp://ftp.gnu.org/gnu/から探してください。

また、十分なディスク領域があることも確認してください。コンパイル中、ソースツリーのために350メガバイト、インストールディレクトリに60メガバイトほど必要となります。空のデータベースクラスタだけでも約40メガバイト必要です。データベースは、同じデータのフラットテキストファイルと比べて5倍ほどの領域が必要になります。リグレッションテストを実行する場合は、一時的に最大で300メガバイトの領域がさらに必要になります。ディスクの空き容量を確認するためにはdfコマンドを使います。

16.3. ソースの入手

PostgreSQL 13.1のソースは、Webサイトhttps://www.postgresql.org/download/のダウンロードページから入手することができます。postgresql-13.1.tar.gzまたはpostgresql-13.1.tar.bz2という名前のファイルを入手してください。ファイルを入手したら、展開します。

```
gunzip postgresql-13.1.tar.gz
tar xf postgresql-13.1.tar
```

(.bz2ファイルをダウンロードした場合はgunzipではなくbunzip2を使用してください。また、tarの最新のバージョンでは圧縮されたアーカイブを直接展開できますので、gunzipやbunzip2の個別のステップは実際は必要ないことに注意してください。) これにより、カレントディレクトリ以下にpostgresql-13.1というディレクトリが作成され、PostgreSQLのソースが展開されます。この後のインストール手順を行うために、このディレクトリに移動してください。

またバージョン管理システムのリポジトリから直接ソースを入手することができます。付録Iを参照してください。

16.4. インストール手順

1. **構成**

   インストール手順の最初のステップは、システムに合わせてソースツリーを設定し、使用するオプションを選択することです。configureスクリプトを実行することでこれを行います。デフォルトのインストールを行う場合は、単に以下を入力してください。

   ```
   ./configure
   ```

   このスクリプトは、各種のシステムに依存した変数の値を決定するために多くの試験を行い、使用中のオペレーティングシステムが持つどんなクセでも検出し、最終的に構築用ツリーに結果を記録するためのファイルをいくつか作成します。

   構築用のディレクトリを別の場所にしたい場合は、ソースツリーの外のディレクトリでconfigureを実行することもできます。この処理は*VPATH*構築と呼ばれます。どのように行うかは下記を参照して下さい。

   ```
   mkdir build_dir
   cd build_dir

   /path/to/source/tree/configure [オプションはここに]
   make
   ```

   デフォルトの構成では、サーバ、ユーティリティの他に、Cコンパイラだけを必要とするクライアントアプリケーションやインタフェースを構築します。デフォルトでは、全てのファイルは/usr/local/pgsql以下にインストールされます。

   configureにコマンドラインオプションを1つ以上指定することで、構築処理やインストール処理を変更することができます。よくあるのは、インストールの位置や構築するオプションの機能の設定を変更することでしょう。configureには数多くのオプションがあり、それは16.4.1に書かれています。

   また、configureは、16.4.2に書かれているように特定の環境変数に対応しています。これは設定を変更する追加の方法を提供します。

2. **構築**

   構築作業を開始するには、以下のいずれかを入力してください。

   ```
   make
   make all
   ```

   （GNU makeを使用することを忘れないでください。）ハードウェアに依存しますが、構築作業には数分かかります。最後に以下のような行が表示されるはずです。

   ```
   All of PostgreSQL successfully made. Ready to install.
   ```

もし、ドキュメント(HTMLやman)や追加モジュール(contrib)を含め、構築可能なものすべてを構築したい場合、次のように実施します。

```
make world
```

最終行は次のように表示されるはずです。

```
PostgreSQL, contrib, and documentation successfully made. Ready to install.
```

手動で指定するのではなく、別のMakefileから構築をしたい場合には、例えば以下のようにMAKELEVELを削除するか、0に設定しなければなりません。

```
build-postgresql:
        $(MAKE) -C postgresql MAKELEVEL=0 all
```

上記に失敗すると、通常はヘッダファイルが見つからないという奇妙なエラーメッセージが出る場合があります。

3. **リグレッションテスト**

インストールを行う前に、新しく構築したサーバをテストしたい場合、この時点でリグレッションテストを実行できます。リグレッションテストとは、使用するマシンにおいてPostgreSQLが、開発者の想定通りに動作することを検証するためのテストのまとまりです。次のように入力します。

```
make check
```

(これは root では動作しません。非特権ユーザとして実行してください。) 第32章にはテスト結果の解釈に関する詳しい情報があります。同じコマンドを入力することで、後にいつでもテストを繰り返すことができます。

4. **ファイルのインストール**

> **注記**
>
> もし既存のシステムのアップグレードをする場合、DBクラスタのアップグレードの解説が記載されている18.6を参照してください。

PostgreSQLをインストールするには、以下を入力してください。

```
make install
```

これは、ファイルをステップ 1で指定されたディレクトリにインストールします。その領域に書き込むための権限を持っていることを確認してください。通常はこのステップはrootで行う必要があります。代わりに対象とするディレクトリを前もって作成し、適切に権限を調整することも可能です。

ドキュメント(HTMLやman)をインストールするには、以下を入力して下さい。

```
make install-docs
```

先にすべてを(worldを付けて)構築していた場合には、代わりに以下を実行してください。

```
make install-world
```

これによりドキュメントもインストールされます。

make installの代わりにmake install-stripを使用することで、インストール時に実行可能ファイルやライブラリをストリップ(strip)することができます。これにより、多少の容量を節約できます。デバッグをサポートするように構築している場合でも、ストリップするとデバッグのサポートは実質、除去されてしまいます。したがって、これはデバッグが必要なくなった場合にのみ実行すべきです。install-stripは容量を節約するために適切な作業を行おうとしますが、実行可能ファイルから全ての不必要なバイトを完全にストリップすることはできません。可能な限りのディスク容量をすべて節約したい場合は、手動で作業を行う必要があります。

この標準的なインストール方法では、クライアントアプリケーションの開発に必要なヘッダファイルと、Cで独自の関数やデータ型を作成するといったサーバ側のプログラムの開発用のヘッダファイルが用意されます（PostgreSQL 8.0より前まででは、後で別途make install-all-headersコマンドが必要でした。しかし、この手順は標準のインストールに含まれるようになりました）。

クライアント側のみのインストール: クライアントアプリケーションとインタフェースライブラリのみをインストールしたい場合、下記のコマンドを使います。

```
make -C src/bin install
make -C src/include install
make -C src/interfaces install
make -C doc install
```

src/binにはサーバ用の数個のバイナリがあります。これらは小さなものです。

アンインストール: インストールを取り消すには、make uninstall コマンドを使います。しかし、作成済みのディレクトリは削除されません。

クリーニング: インストールが終わったら、make clean コマンドを使ってソースツリーから構築用のファイルを削除し、ディスク領域を解放できます。configureプログラムが作るファイルは保持されますので、後でmakeコマンドですべてを再構築できます。ソースツリーを配布された時の状態に戻したい場合は、make distcleanコマンドを使います。同じソースツリー内で複数のプラットフォーム向けに構築する場合、これを実行して、それぞれのプラットフォームに対し再構成しなければなりません。（または、未変更のソースツリーを維持するために、各プラットフォームで別々の構築用ツリーを使用してください。）

構築作業を行った後でconfigure用オプションが間違っていることに気付いた場合や、configureの調査結果に何らかの変更を加えた場合（例えば、ソフトウェアのアップグレードなど）、再設定と再構築の前にmake distcleanを行うことをお勧めします。さもないと、設定選択肢の変更は、必要なところ全てには反映されない可能性があります。

16.4.1. configureオプション

configureのコマンドラインオプションを以下で説明します。この一覧は完全なものではありません(完全なものを得るには./configure --helpを使ってください)。ここで取り上げていないオプションは黒須コンパイルのような高度なユースケースのためのもので、標準のAutoconfのドキュメントに書かれています。

16.4.1.1. インストレーションの位置

このオプションはmake installがファイルをどこに置くかを制御します。たいていの場合--prefixオプションで十分です。特別な必要性があるのであれば、この節に書かれた他のオプションを使用して個々のインストレーションサブディレクトリを変更できます。しかし、異なるサブディレクトリの相対的な位置を変更した場合、インストレーションは再配置不能になります。つまり、インストールの後にディレクトリを移動できないことに注意してください。(manとdocの場所はこの制限の影響を受けません。) 再配置可能インストールのために、後述の--disable-rpathを使用しようと考えるかもしれません。

--prefix=PREFIX

/usr/local/pgsqlではなく、PREFIXディレクトリ以下に全てのファイルをインストールします。ファイルは実際には様々なサブディレクトリにインストールされ、PREFIXディレクトリの直下にインストールされるファイルはありません。

--exec-prefix=EXEC-PREFIX

アーキテクチャ依存のファイルをPREFIXの設定とは別の接頭辞EXEC-PREFIX以下にインストールすることができます。ホスト間でアーキテクチャ非依存のファイルを共有する場合に便利です。省略した場合、EXEC-PREFIXはPREFIXと同じに設定され、アーキテクチャに依存するファイルも非依存なファイルも同じツリー以下にインストールされます。ほとんどの場合、これが望まれています。

--bindir=DIRECTORY

実行可能プログラム用のディレクトリを指定します。デフォルトではEXEC-PREFIX/binであり、通常/usr/local/pgsql/binとなります。

--sysconfdir=DIRECTORY

各種設定ファイル用のディレクトリを設定します。デフォルトではPREFIX/etcです。

--libdir=DIRECTORY

ライブラリや動的ロード可能モジュールをインストールする場所を設定します。デフォルトはEXEC-PREFIX/libです。

--includedir=DIRECTORY

CおよびC++のヘッダファイルをインストールするディレクトリを設定します。デフォルトはPREFIX/includeです。

--datarootdir=DIRECTORY

いろいろな種類の読み取り専用データファイル用のルートディレクトリを設定します。これは後述のオプションの一部についてのデフォルトを設定するだけです。デフォルトはPREFIX/shareです。

--datadir=DIRECTORY

インストールプログラムが使用する読み取り専用のディレクトリを設定します。デフォルトはDATAROOTDIRです。これはインストールするデータベースファイルがどこに設置されるかとは関係ないことを覚えておいてください。

--localedir=DIRECTORY

特にメッセージ翻訳カタログファイルのロケールデータをインストールするディレクトリを設定します。デフォルトはDATAROOTDIR/localeです。

--mandir=DIRECTORY

PostgreSQL付属のマニュアルページがこのディレクトリ以下の、対応するmanxサブディレクトリにインストールされます。デフォルトはDATAROOTDIR/manです。

--docdir=DIRECTORY

「man」ページを除いた、ドキュメント一式ファイルをインストールするルートディレクトリを設定します。これは以下のオプションのデフォルトのみを設定します。このオプションのデフォルト値はDATAROOTDIR/doc/postgresqlです。

--htmldir=DIRECTORY

PostgreSQLのHTML形式化文書一式はこのディレクトリの下にインストールされます。デフォルトはDATAROOTDIRです。

注記

(/usr/local/includeといった)共用のインストール場所に、システムの他の名前空間に影響を与えることなくPostgreSQLをインストールすることができるような配慮がなされています。まず、完全に展開したディレクトリ名に「postgres」か「pgsql」という文字列が含まれていない場合、「/postgresql」という文字列が自動的にdatadir、sysconfdir、docdirに追加されます。例えば、接頭辞として/usr/localを使用する場合、文書は/usr/local/doc/postgresqlにインストールされますが、接頭辞が/opt/postgresの場合は/opt/postgres/docにインストールされます。クライアントインタフェース用の外部向けCヘッダファイルはincludedirにインストールされ、名前空間の問題はありません。内部向けヘッダファイルやサーバ用ヘッダファイルは、includedir以下の非公開ディレクトリにインストールされます。各インタフェース用のヘッダファイルにアクセスする方法についての情報は、そのインタフェースの文書を参照してください。最後に、適切であれば、動的ロード可能モジュール用にlibdir以下にも非公開用のサブディレクトリが作成されます。

16.4.1.2. PostgreSQLの機能

この節に書かれたオプションは、デフォルトでは構築されないPostgreSQLの様々な機能を構築できるようにするものです。この多くがデフォルトでないのは、16.2で書かれているように追加のソフトウェアを必要とするからだけです。

`--enable-nls[=LANGUAGES]`

各国語サポート(NLS)、つまり、英語以外の言語によるプログラムメッセージの表示機能を有効にします。LANGUAGESはオプションであり、サポートさせたい言語コードを空白で区切ったリストを指定します。例えば、`--enable-nls='de fr'`などとします。(指定したリストと実際に用意された翻訳との論理積が自動的に計算されます。) リストに何も指定しなかった場合、利用可能な翻訳すべてがインストールされます。

このオプションを使用するためには、gettext APIの実装が必要です。

`--with-perl`

PL/Perlサーバサイド言語を構築します。

`--with-python`

PL/Pythonサーバサイド言語を構築します。

`--with-tcl`

PL/Tclサーバサイド言語を構築します。

`--with-tclconfig=DIRECTORY`

Tclは、Tclへのインタフェースモジュールを構築するために必要な設定情報を含む`tclConfig.sh`ファイルをインストールします。このファイルは通常、自動的に一般的に知られている場所にありますが、もしTclの別のバージョンを使いたい場合は、検索対象のディレクトリを指定することができます。

`--with-icu`

ICUライブラリのサポートを有効にして構築します。これによりICU照合機能が使用できるようになります。(23.2を参照してください。) これには、ICU4Cパッケージがインストールされていることが必要です。ICU4Cの要求される最小のバージョンは現在4.2です。

デフォルトでは、pkg-configが必要なコンパイルオプションを見つけるのに使われます。これはICU4Cバージョン4.6またはそれ以降でサポートされています。より古いバージョンの場合やpkg-configが使えない場合には、以下の例のように、変数ICU_CFLAGSとICU_LIBSをconfigureに指定できます。

```
./configure ... --with-icu ICU_CFLAGS='-I/some/where/include' ICU_LIBS='-L/some/where/lib -licui18n -licuuc -licudata'
```

(ICU4Cがコンパイラのデフォルトの検索パスにあるのなら、pkg-configの使用を避けるため、例えばICU_CFLAGS=' 'のような空でない文字列を指定することも必要です。)

`--with-llvm`

LLVMに基づいたJITコンパイル (第31章を参照)のサポート有効にして構築します。これには、LLVMライブラリがインストールされていることが必要です。LLVMの要求される最小のバージョンは現在3.9です。

要求されるコンパイルオプションを見つけるために`llvm-config`が使われます。`llvm-config`、それからサポートされるバージョンすべての`llvm-config-$major-$minor`を`PATH`で探します。それで正しいバイナリが見つからなければ、正しい`llvm-config`へのパスを指定するためにLLVM_CONFIGを使ってください。例えば、以下の通りです。

```
./configure ... --with-llvm LLVM_CONFIG='/path/to/llvm/bin/llvm-config'
```

LLVMサポートはclang互換のコンパイラ(必要なら環境変数CLANGで指定してください)と動作するC++コンパイラ(必要なら環境変数CXXで指定してください)を要求します。

--with-openssl

SSL(暗号化)接続のサポートを有効にして構築します。これには、OpenSSLパッケージがインストールされていることが必要です。configureは、処理を進める前にOpenSSLのインストールを確認するために、必要なヘッダファイルとライブラリを検査します。

--with-gssapi

GSSAPI認証のサポートを構築します。多くのシステムでは、GSSAPIシステム(通常Kerberosインストレーションの一部)はデフォルトの検索場所(例えば/usr/includeや/usr/lib)にインストールされていません。そのため、--with-includesと--with-librariesオプションをさらに追加して使わなければいけません。configureは、処理を進める前にGSSAPIが正しくインストールされていることを確認するために、必要とされるヘッダファイルとライブラリを検査します。

--with-ldap

認証および接続パラメータ検索用のLDAPサポートを有効にして構築します。(詳細は33.17および20.10を参照してください。) Unixでは、OpenLDAPパッケージがインストールされていることが必要です。WindowsではデフォルトのWinLDAPライブラリが使用されます。configureは、処理を進める前にOpenLDAPのインストールが十分されているかどうかを確認するために、必要なヘッダファイルとライブラリを検査します。

--with-pam

PAM(プラガブル認証モジュール)のサポートを有効にして構築します。

--with-bsd-auth

BSD認証のサポートを有効にして構築します。(BSD認証フレームワークは今のところOpenBSDだけで利用可能です。)

--with-systemd

systemdサービス通知のサポートを有効にして構築します。サーババイナリがsystemdの元で開始する場合には、これは統合を改善しますが、それ以外は影響はありません。詳細は18.3を参照してください。このオプションを使えるようにするには、libsystemdと関連するヘッダファイルをインストールすることが必要です。

--with-bonjour

Bonjour自動サービス検出のサポートを有効にして構築します。これには、オペレーティングシステムがBonjourをサポートしていることが必要です。macOSでは推奨します。

--with-uuid=LIBRARY

指定されたUUIDライブラリを使用して(UUIDを生成する関数を提供する)uuid-osspモジュールをビルドします。LIBRARYは以下のいずれかでなければなりません。

- bsdはFreeBSD、NetBSD、その他のBSD派生システムにあるUUID関数を使います。
- e2fsはe2fsprogsプロジェクトで作られたUUIDライブラリを使います。このライブラリはたいていのLinuxシステムとmacOSにあり、また、その他のプラットフォームでも入手可能です。
- osspはOSSP UUIDライブラリ[1]を使用します。

`--with-ossp-uuid`

`--with-uuid=ossp`に相当する古いものです。

`--with-libxml`

libxml2を使用して構築し、SQL/XMLサポートを有効にします。この機能のためにはlibxml2バージョン2.6.23以降が必要です。

pkg-configがインストールされていて、かつそれがlibxml2について知っているようであれば、必要なコンパイラオプション、リンカオプションを検出するために、PostgreSQLは`pkg-config`に問い合わせます。そうでなければ、libxml2がインストールするプログラム`xml2-config`を見つけられれば使用します。複数アーキテクチャのインストレーションをよりうまく扱えますので、`pkg-config`を使用する方が好ましいです。

通常以外の場所にインストールしたlibxml2インストレーションを使用するためには、`pkg-config`関連の環境変数を設定するか(そのドキュメントを参照してください)、環境変数`XML2_CONFIG`がそのインストレーション用の`xml2-config`プログラムを指し示すように設定するか、変数`XML2_CFLAGS`と`XML2_LIBS`を設定します。(`pkg-config`がインストールされていて、libxml2がどこにあるかについてのその認識を覆したいのであれば、`XML2_CONFIG`を、もしくは`XML2_CFLAGS`と`XML2_LIBS`の両方を空でない文字列に設定しなければなりません。)

`--with-libxslt`

XMLのXSL変換を行うためにxml2モジュールを有効にしてlibxsltを構築します。`--with-libxml`も指定しなければなりません。

16.4.1.3. 機能の無効化

この節に書かれたオプションは、デフォルトでは構築されますが、必要なソフトウェアやシステムの機能が利用可能でない場合にオフにする必要があるPostgreSQLの特定の機能を無効にします。本当に必要でない限りは、ここのプションの使用は勧められません。

`--without-readline`

Readlineライブラリ(およびlibedit)の使用を防止します。このオプションはpsqlでのコマンドライン編集および履歴を無効にします。

`--with-libedit-preferred`

GPLライセンスのReadlineではなくBSDライセンスのlibeditライブラリを優先して使用します。このオプションは両方のライブラリがインストールされている場合にのみ重要です。その場合デフォルトでReadlineが使用されます。

[1] http://www.ossp.org/pkg/lib/uuid/

--without-zlib

Zlibライブラリの使用を抑制します。これは、pg_dumpとpg_restoreにおける圧縮アーカイブのサポートを無効にします。

--disable-spinlocks

PostgreSQLがそのプラットフォーム用のCPUスピンロックをサポートしない場合でも、構築に成功するようにします。スピンロックのサポートの欠落により、性能は悪化します。したがって、このオプションは、構築が失敗し、その原因が使用するプラットフォームでスピンロックサポートが欠落している場合にのみ使用してください。使用するプラットフォームにおけるPostgreSQLの構築にこのオプションが必要とされた場合は、PostgreSQL開発者にその問題を報告してください。

--disable-atomics

CPU不可分操作の使用を無効にします。このオプションはそのような操作のないプラットフォームでは何もしません。そのような操作のあるプラットフォームでは、これにより性能が低下するでしょう。このオプションはデバッグや性能比較をする場合にのみ有用です。

--disable-thread-safety

クライアントライブラリのスレッドセーフを無効にします。これにより、libpqやECPGプログラム内の同時実行スレッドは、安全にその固有の接続ハンドルを制御できなくなります。スレッドのサポートに欠陥があるプラットフォームでのみ、これを使ってください。

16.4.1.4. 構築プロセスの詳細

--with-includes=DIRECTORIES

DIRECTORIESには、コンパイラがヘッダファイルを検索するディレクトリのリストをコロンで区切って指定します。(GNU Readlineなどの)オプションのパッケージが非標準的な場所にインストールされている場合、このオプションと、おそらく対応する--with-librariesオプションを使用する必要があります。

例: --with-includes=/opt/gnu/include:/usr/sup/include

--with-libraries=DIRECTORIES

DIRECTORIESには、ライブラリを検索するディレクトリのリストをコロンで区切って指定します。パッケージが非標準的な場所にインストールされている場合は、おそらくこのオプション(と対応する--with-includesオプション)を使用する必要があります。

例: --with-libraries=/opt/gnu/lib:/usr/sup/lib

--with-system-tzdata=DIRECTORY

PostgreSQLは、日付時刻に関する操作で必要な、独自の時間帯データベースを持ちます。実際のところ、この時間帯データベースはFreeBSD、Linux、Solarisなどの多くのオペレーティングシステムで提供されるIANA時間帯データベースと互換性があります。このため、これを再びインストールすることは冗長です。このオプションが使用されると、DIRECTORYにあるシステムが提供する時間帯データベースがPostgreSQLソース配布物に含まれるものの代わりに使用されます。DIRECTORYは絶対パスで指定しなければなりません。/usr/share/zoneinfoがオペレーティングシステムの一部でよく使われます。イン

ストール処理が時間帯データが一致しない、またはエラーがあることを検知しないことに注意してください。このオプションを使用する場合、指定した時間帯データがPostgreSQLで正しく動作するかどうかを検証するためにリグレッションテストを実行することが推奨されています。

このオプションは、対象オペレーティングシステムを熟知しているパッケージ配布者を主な対象としたもの。このオプションを使用する大きな利点は、多くの局所的な夏時間規則の変更があってもPostgreSQLパッケージを更新する必要がないことです。他の利点として、時間帯データベースファイルをインストール時に構築する必要がありませんので、PostgreSQLのクロスコンパイルをより簡単に行うことができます。

`--with-extra-version=STRING`

PostgreSQLバージョン番号に`STRING`を追加します。これは、例えば、リリースされていないGitスナップショットからビルドしたバイナリや、`git describe`識別子やディストリビューションパッケージリリース番号のような追加のバージョン文字列のあるカスタムパッチを含むバイナリに印をつけるために使えます。

`--disable-rpath`

PostgreSQLの実行ファイルがインストレーションのライブラリディレクトリ(`--libdir`を参照してください)にある共有ライブラリを探すよう指示する印を付けません。ほとんどのプラットフォームでは、この印付けはライブラリディレクトリへの絶対パスを利用しますので、後でインストレーションを再配置したときには役に立たないでしょう。ですので、実行ファイルが共有ライブラリを見つける他の方法を提供する必要があるでしょう。通常は、オペレーティングシステムの動的リンカがライブラリディレクトリを探すよう設定することが必要です。詳細は16.5.1を参照してください。

16.4.1.5. その他

デフォルトのポート番号を`--with-pgport`で調整することは、特にテスト構築のためには、かなり良くあることです。この節の他のオプションは上級ユーザにのみ勧められます。

`--with-pgport=NUMBER`

サーバとクライアントのデフォルトのポート番号を`NUMBER`に設定します。デフォルトは5432です。このポートは後でいつでも変更することができますが、ここで指定した場合、サーバとクライアントはコンパイル時に同じデフォルト値を持つようになります。これは非常に便利です。通常、デフォルト以外の値を選択すべき唯一の理由は、同じマシンで複数のPostgreSQLを稼働させることです。

`--with-krb-srvnam=NAME`

GSSAPIで使用されるKerberosのサービスプリンシパルのデフォルトの名前です。デフォルトでは`postgres`です。これを変える理由はWindows環境のために構築しているのでない限り、特にありません。Windows環境のために構築している場合は大文字の`POSTGRES`に設定する必要があります。

`--with-segsize=SEGSIZE`

セグメントサイズをギガバイト単位で指定します。大規模なテーブルはこのセグメントサイズと同じサイズの複数のオペレーティングシステムのファイルに分割されます。これにより多くのプラットフォームで存在するファイルサイズ上限に関する問題を防ぎます。デフォルトのセグメントサイズは1ギガバイトで、サポートされるすべてのプラットフォームで安全です。使用するオペレーティングシステムが「ラー

ジファイル」をサポートしていれば（最近はほとんどサポートしています）、より大きなセグメントサイズを使用することができます。非常に大規模なテーブルで作業する時のファイル記述子の消費数を減らすために、これが役に立つでしょう。しかし、プラットフォーム、または使用予定のファイルシステムでサポートされる値以上に大きな値を指定しないように注意してください。tarなどの、使用したいその他のツールにも使用できるファイルサイズに制限があることがあります。絶対に必要ではありませんが、この値を2のべき乗にすることを勧めます。この値を変更するとディスク上でのデータベースの互換性を壊すことに注意してください。すなわち、pg_upgradeを使ってセグメントサイズの異なるビルドにはアップグレードできません。

--with-blocksize=BLOCKSIZE

キロバイト単位で*ブロック容量*を設定します。これはテーブル内でのストレージとI/Oの単位です。8キロバイトのデフォルトはほとんどの場合適切ですが、特別な場合は他の値が役立ちます。値は1から32（キロバイト）の範囲の2のべき乗でなければなりません。この値を変更するとディスク上でのデータベースの互換性を壊すことに注意してください。すなわち、pg_upgradeを使ってブロック容量の異なるビルドにはアップグレードできません。

--with-wal-blocksize=BLOCKSIZE

キロバイト単位で*WALブロック容量*を設定します。これはWALログ内でのストレージとI/Oの単位です。8キロバイトのデフォルトはほとんどの場合適切ですが、特別な場合は大きめの値が役立ちます。値は1から64（キロバイト）の範囲の2のべき乗でなければなりません。この値を変更するとディスク上でのデータベースの互換性を壊すことに注意してください。すなわち、pg_upgradeを使ってWALブロック容量の異なるビルドにはアップグレードできません。

16.4.1.6. 開発者向けオプション

この節のオプションのほとんどは、PostgreSQLを開発したりデバッグしたりするために重要なものです。--enable-debugを除いて、実運用での構築には勧められません。--enable-debugはバグに出くわすという不幸な出来事の時に詳細なバグレポートが得られるので有用かもしれません。DTraceをサポートするプラットフォームでは、--enable-dtraceを実運用で使うことも適当かもしれません。

サーバ内でコードの開発に使われるインストレーションを構築する場合には、少なくともオプション--enable-debugと--enable-cassertを使うことをお勧めします。

--enable-debug

すべてのプログラムとライブラリをデバッグシンボル付きでコンパイルします。これは、問題を解析するためにデバッガ内でプログラムを実行できることを意味します。これはインストールする実行形式ファイルのサイズをかなり大きくし、また、GCC以外のコンパイラでは、通常はコンパイラによる最適化が行われなくなりますので、低速になります。しかし、デバッグシンボルが利用できるということは、発生した問題に対応する時に非常に便利です。現在のところ、GCC を使用している場合にのみ、稼働用のインストレーションにこのオプションを使用することを推奨します。しかし、開発作業時やベータ版を実行する時は、常にこれを有効にすべきです。

--enable-cassert

サーバにおける、多くの「あり得ない」状態をテストする*アサーション*チェックを有効にします。これは、プログラムの開発のためには測り知れない価値がありますが、このテストによりサーバはかなり低速になり

ます。また、このテストを有効にしても、サーバの安定性が向上するとは限りません! アサーションチェックは、重要度によって分類されていませんので、比較的害がないようなバグでも、アサーション失敗をトリガとした、サーバの再起動が行われてしまいます。稼働用にこのオプションを使用することは推奨されませんが、開発作業時やベータ版を実行する場合は、これを有効にすべきです。

--enable-tap-tests

Perl TAPツールを使ったテストを有効にします。これにはPerlのインストールとPerlモジュールIPC::Runが必要です。詳細は32.4を参照してください。

--enable-depend

自動依存関係追跡を有効にします。このオプションを使用すると、ヘッダファイルが変更された場合に、影響を受ける全てのオブジェクトファイルが再構築されるように、makefile が設定されます。これは開発作業時には有用ですが、単に一度コンパイルしインストールするだけであれば、これは無駄なオーバーヘッドです。現在のところ、GCC でのみ、このオプションは動作します。

--enable-coverage

GCCを使用している場合、すべてのプログラムとライブラリはコードカバレッジ試験機構付きでコンパイルされます。実行すると、これらは構築用ディレクトリ内にコードカバレッジメトリックを持ったファイルを生成します。詳細は32.5を参照してください。このオプションはGCC専用であり、また、開発作業中に使用するためのものです。

--enable-profiling

GCCを使用する場合、すべてのプログラムとライブラリがプロファイリング可能状態でコンパイルされます。バックエンドの終了時、プロファイリングに使用するgmon.outファイルを含むサブディレクトリが作成されます。このオプションはGCCを使用する場合のみ使用でき、開発作業を行う時に使用します。

--enable-dtrace

動的追跡ツールDTraceのサポートを有効にしてPostgreSQLをコンパイルします。より詳細な情報は27.5を参照してください。

dtraceプログラムを指し示すためにDTRACE環境変数を設定することができます。dtraceは通常、検索パス内に存在しない可能性がある/usr/sbin以下にインストールされていますので、この設定はよく必要になります。

さらにdtraceプログラム用のコマンドラインオプションをDTRACEFLAGS環境変数で指定できます。Solarisで64ビットバイナリでDTraceをサポートするには、DTRACEFLAGS="-64"をconfigureに指定してください。例えばGCCコンパイラを使用する場合は以下のようにします。

```
./configure CC='gcc -m64' --enable-dtrace DTRACEFLAGS='-64' ...
```

Sunのコンパイラを使用する場合は以下のようにします。

```
./configure CC='/opt/SUNWspro/bin/cc -xtarget=native64' --enable-dtrace DTRACEFLAGS='-64' ...
```

16.4.2. configure環境変数

上記の通常のコマンドラインオプションに加えて、configureは数多くの環境変数に対応します。次のようにして、configureコマンドラインに環境変数を指定できます。

```
./configure CC=/opt/bin/gcc CFLAGS='-O2 -pipe'
```

この使い方では、環境変数はコマンドラインオプションとは少し異なります。あらかじめそのような変数を設定しておくこともできます。

```
export CC=/opt/bin/gcc
export CFLAGS='-O2 -pipe'
./configure
```

多くのプログラムの設定スクリプトは似たようにこの変数に対応しますので、この使い方は便利でしょう。

configureが選ぶものと違うCコンパイラを使いたいという場合には、CC 環境変数をその使用したいプログラムに設定することができます。デフォルトでは、configureは利用できるのであればgccを、利用できなければプラットフォームのデフォルト（通常cc）を選択します。同様に、デフォルトのコンパイラフラグは必要に応じてCFLAGS変数で上書きすることもできます。

以下は、この方式で設定可能な重要な環境変数の一覧です。

BISON

Bisonプログラム。

CC

Cコンパイラ。

CFLAGS

Cコンパイラに渡すオプション。

CLANG

--with-llvmでコンパイルされた場合、ソースコードのインライン展開を処理するために使われるclangプログラムへのパス。

CPP

Cプリプロセッサ。

CPPFLAGS

Cプリプロセッサに渡すオプション。

CXX

C++コンパイラ。

CXXFLAGS

C++コンパイラに渡すオプション。

DTRACE

dtraceプログラムの場所。

DTRACEFLAGS

dtraceプログラムに渡すオプション。

FLEX

Flexプログラム。

LDFLAGS

実行ファイルや共有ライブラリにリンクする場合に使用するオプション。

LDFLAGS_EX

実行ファイルのリンク時のみに追加されるオプション。

LDFLAGS_SL

共有ライブラリのリンク時のみに追加されるオプション。

LLVM_CONFIG

LLVMインストレーションの場所を特定するために使用するllvm-configプログラム。

MSGFMT

多言語サポート(NLS)用のmsgfmtプログラム。

PERL

Perlインタプリタプログラム。これは、PL/Perl構築に関する依存性を決定するために使用されます。デフォルトはperlです。

PYTHON

Pythonインタプリタプログラム。これは、PL/Python構築に関する依存性を決定するために使用されます。またここでPython 2または3を指定するかどうかで(あるいは暗黙的に選択されます)、どちらのPL/Python言語が利用可能になるかも決まります。詳細は45.1を参照してください。設定されていなければ、python python3 python2の順で調べられます。

TCLSH

Tclインタプリタプログラム。これは、PL/Tcl構築に関する依存性を決定するために使用され、Tclスクリプト内を置き換えます。設定されていなければ、以下の順で調べられます。tclsh tcl tclsh8.6 tclsh86 tclsh8.5 tclsh85 tclsh8.4 tclsh84

XML2_CONFIG

libxml2インストレーションの場所を特定するために使用するxml2-configプログラムです。

configureが選んだコンパイラフラグに対して、事後にフラグを追加することが有用な場合があります。重要な例は、configureに渡すCFLAGSにgccの-Werrorオプションを含められないことです。なぜなら、そうするとconfigureの組み込みテストの多くが失敗するからです。そのようなフラグを追加するには、makeを実行する時にCOPT環境変数に含めてください。configureで設定されたCFLAGSオプションとLDFLAGSオプションの両方に、COPTの内容が追加されます。例えば、以下のようにします。

```
make COPT='-Werror'
```

または

```
export COPT='-Werror'
make
```

注記

GCCを使う場合、少なくとも-O1レベルの最適化で構築することがベストです。なぜなら、何の最適化もしない(-O0)と、重要なコンパイル警告(初期化されていない変数の使用など)が無効になるからです。しかし、最適化を行うことでソースコードとコンパイルされたコードのステップは1対1とはならなくなるため、デバッグは複雑になるかもしれません。最適化されたコードのデバッグに悩まされてしまう場合は、関心のある特定のファイルに対して-O0で再コンパイルしてください。これを実行するための簡単な方法は、make PROFILE=-O0 file.oのように、make経由でオプションを渡すことです。

COPTとPROFILEの環境変数は、PostgreSQLのmakefileでは実際には全く同一に扱われます。どちらを使うかは好みの問題ですが、開発者の一般的な習慣では、一時的にフラグを調整するにはPROFILEを使い、永続的に保持するものにはCOPTを使います。

16.5. インストール後の設定作業

16.5.1. 共有ライブラリ

共有ライブラリを持つ何らかのシステムの中には、新しくインストールされた共有ライブラリを探す場所をシステムに通知する必要があるものがあります。これが必要ではないシステムはFreeBSD、HP-UX、Linux、NetBSD、OpenBSD、およびSolarisです。

共有ライブラリの検索パスを設定する方法は、プラットフォームによって異なります。しかし、最もよく使用される方法はLD_LIBRARY_PATHといった環境変数を以下のように設定することです。Bourne シェル（sh、ksh、bash、zsh）では、

```
LD_LIBRARY_PATH=/usr/local/pgsql/lib
export LD_LIBRARY_PATH
```

cshまたはtcshでは、以下のように設定します。

```
setenv LD_LIBRARY_PATH /usr/local/pgsql/lib
```

/usr/local/pgsql/libはステップ 1で--libdirに設定したものに置き換えてください。/etc/profileや~/.bash_profileといったシェルの起動ファイルにこれらのコマンドを追加してください。この方法に関する警告についての優れた情報がhttp://xahlee.org/UnixResource_dir/_/ldpath.htmlにあります。

システムによっては、構築作業の前にLD_RUN_PATH環境変数を設定した方が良い場合があります。

Cygwinでは、ライブラリディレクトリをPATHに追加するか、.dllファイルをbinディレクトリに移動します。

もし確信が持てない場合は、システムのマニュアルページ(おそらくld.soかrld)を参照してください。もし後に下記のようなメッセージが出たら、このステップが必要だったということです。

```
psql: error in loading shared libraries
libpq.so.2.1: cannot open shared object file: No such file or directory
```

この場合は処置を行ってください。

もしLinuxを使用していて、root権限があれば、

```
/sbin/ldconfig /usr/local/pgsql/lib
```

(または同等のディレクトリ)をインストール後に実行して、実行時リンカが共有ライブラリを素早く検索できるようにできます。より詳細についてはldconfigのマニュアルページを参照してください。FreeBSD、NetBSDおよびOpenBSDの場合のコマンドは以下の通りです。

```
/sbin/ldconfig -m /usr/local/pgsql/lib
```

同様なコマンドを持つ他のシステムは知られていません。

16.5.2. 環境変数

もし/usr/local/pgsqlか、もしくはデフォルトでプログラムが検索されない場所にインストールした場合、/usr/local/pgsql/bin(もしくはステップ 1で--bindirに設定した場所)をPATHに追加する必要があります。厳密に言えば、これは必要ではありません。しかし、これによってPostgreSQLの使用がずっと便利になります。

これを行うためには、以下を~/.bash_profile(もしくは、もし全てのユーザに反映したい場合は/etc/profile)のようなシェルの起動ファイルに追加してください。

```
PATH=/usr/local/pgsql/bin:$PATH
export PATH
```

cshやtcshを使用している場合は、以下のコマンドを使用してください。

```
set path = ( /usr/local/pgsql/bin $path )
```

デフォルトで検索される場所にインストールした場合を除き、システムがmanドキュメントを検索できるようにするためには、以下の行をシェルの起動ファイルに追加する必要があります。

```
MANPATH=/usr/local/pgsql/share/man:$MANPATH
export MANPATH
```

環境変数PGHOSTとPGPORTは、クライアントアプリケーションにデータベースサーバのホストとポートを指定し、コンパイル時に決定されたデフォルト値を無効にします。クライアントアプリケーションをリモートで実行する場合、データベースを使用する予定の全てのユーザがPGHOSTを設定していると便利です。しかしこれは必須ではありません。この設定は、ほとんどのクライアントプログラムのコマンドラインオプションでも設定することができます。

16.6. サポートされるプラットフォーム

プラットフォーム（CPUアーキテクチャとオペレーティングシステムの組合せ）は、そのプラットフォーム上で動作する仕組みがコード内に存在し、かつ、そのプラットフォーム上で構築およびリグレッションテストに合格することが最近検証できた場合に、PostgreSQL開発者コミュニティによってサポートされたものとみなされます。現在、プラットフォームの互換性に関するほとんどの試験はPostgreSQLビルドファーム[2]中の試験マシンによって自動的に行われます。ビルドファームに存在しないが、コードが動作するあるいは動作させることができたプラットフォームにおけるPostgreSQLの使用に興味のあるかたは、継続した互換性を確実にするために、ビルドファームのメンバマシンとして設定することを強く勧めます。

一般的に、PostgreSQLは、次のCPUアーキテクチャで動作することを期待できます。x86, x86_64, IA64, PowerPC, PowerPC 64, S/390, S/390x, Sparc, Sparc 64, ARM, MIPS, MIPSEL, PA-RISC。M68K、M32R、VAXをサポートするコードは存在しますが、これらのアーキテクチャで試験が行われたという報告は最近ありません。--disable-spinlocksを付けることで、未サポートの種類のCPUでも構築することがしばしばできますが、性能は低下します。

PostgreSQLは次のオペレーティングシステムで動作することを期待できます。Linux (最近のディストリビューションすべて), Windows (XP以降), FreeBSD, OpenBSD, NetBSD, macOS, AIX, HP/UX, Solaris。他のUnixに似たシステムでも動作するかもしれませんが、最近試験されていません。ほとんどの場合、指定されたオペレーティングシステムでサポートされるCPUアーキテクチャはすべて動作するでしょう。特に古めのシステムを使用している場合、以下の16.7を参照し、使用するオペレーティングシステム固有の情報がないか確認してください。

最近のビルドファームの結果でサポートしているものとされているプラットフォームでインストールに問題があった場合は、<pgsql-bugs@lists.postgresql.org>に報告してください。新しいプラットフォームへのPostgreSQLの移植に興味があるのならば、<pgsql-hackers@lists.postgresql.org>がその議論に適しています。

16.7. プラットフォーム特有の覚書

本節はPostgreSQLのインストールと設定に関する追加のプラットフォーム固有の問題について説明します。インストール手順、特に16.2を注意して読んでください。またリグレッションテスト結果の解釈については第32章を確認してください。

[2] https://buildfarm.postgresql.org/

ここで触れられていないプラットフォームは、インストールに関してプラットフォーム特有の問題がありません。

16.7.1. AIX

AIX上でPostgreSQLは動作しますが、6.1より前のAIXのバージョンには様々な問題がありますので、勧められません。GCCまたは在来のIBMコンパイラ`xlc`が使用できます。

16.7.1.1. メモリ管理

AIXはメモリ管理手法の観点から見ると多少独特です。ギガバイト単位のRAMが空いているサーバがあっても、アプリケーションを実行している時にメモリ不足やアドレス空間エラーが発生することがあります。こうした例の1つが、見慣れないエラーによる拡張のロードの失敗です。例えば、PostgreSQLインストレーションの所有者として実行してみます。

```
=# CREATE EXTENSION plperl;
ERROR:  could not load library "/opt/dbs/pgsql/lib/plperl.so": A memory address is not in the
 address space for the process.
```

PostgreSQLインストレーションの処理グループ内の所有者以外として実行してみます。

```
=# CREATE EXTENSION plperl;
ERROR:  could not load library "/opt/dbs/pgsql/lib/plperl.so": Bad address
```

他の実例は、PostgreSQLサーバログ中のメモリ不足エラーで、256 MB以上もしくはその近辺で全てのメモリ割り当てが失敗します。

これら問題のすべての総合原因は、サーバプロセスで使用されるデフォルトのビット割当とメモリモデルです。デフォルトでは、AIXで構築されたすべてのバイナリは32ビットです。これは使用中のハードウェアの種類やカーネルに依存しません。これらの32ビットプロセスは、数個のモデルの1つを使用して256メガバイトのセグメントで割りつけられた4ギガバイトメモリに制限されます。デフォルトでは、スタックで1つのセグメントとして共有されるものとしてヒープ内の256メガバイト未満の領域が許されます。

上記、`plperl`の例の場合において、PostgreSQLインストレーションにおけるバイナリのumaskとパーミッションをチェックしてください。例に関与したバイナリは32-ビットであり、755ではなく750モードでインストールされました。このような形式で設定されたパーミッションのため、所有者もしくはグループ所有のメンバーのみライブラリを読み込めます。それは誰もが読み取り可能ではないため、ローダは、そうでない場合に配置される共有ライブラリセグメントにではなく、オブジェクトをプロセスのヒープに配置します。

これに対しての「理想的な」解決策はPostgreSQLの64-ビットビルドを使うことですが、32-ビットプロセッサのシステムでは64-ビットバイナリをビルドできますが実行できないので、常に実務的ではありません。

32-ビットバイナリを要求する場合、PostgreSQLサーバを起動する前に`LDR_CNTRL`を`MAXDATA=0xn0000000`に設定します。ここで、1 <= n <= 8です。そして異なる値と`postgresql.conf`設定で満足に稼動する構成を見つけ出します。このように`LDR_CNTRL`を使用すると、AIXに対してサーバがヒープにかかわらず、256 MBセグメントに割り当てられた`MAXDATA`バイトセットを持つようにさせたい意図を表明します。稼動する構成を見つけたとき、意図したヒープ容量をデフォルトで使用するように`ldedit`を使用してバイナリを変更することができま

す。同じ効果を得るため、configure LDFLAGS="-Wl,-bmaxdata:0xn0000000"を渡してPostgreSQLを再構築することもできます。

64-ビット構築に対し、OBJECT_MODEを64に設定し、configureにCC="gcc -maix64"とLDFLAGS="-Wl,-bbigtoc"を渡します。(xlcに対するオプションは異なるかもしれません。) OBJECT_MODEのexportを省略すると、構築はリンカエラーで失敗することがあります。OBJECT_MODEが設定された場合、ar、as、およびldのようなAIXの構築ユーティリティにどの種類のオブジェクトがデフォルトで対応されるのかを伝えます。

デフォルトで、ページングスペースのオーバーコミットが起こることがあります。これが起こることを経験したことはありませんが、AIXはメモリを使い切って、オーバーコミットがアクセスされたときにプロセスをkillします。システムが別のプロセスに対する十分なメモリがないことを判断したためにフォークが失敗するという、これとよく似たことは経験したことがあります。多くの他のAIX部分のように、ページングスペース割り当て方式とメモリ不足によるプロセス停止は、これが問題となるのであれば、システム全体またはプロセス全体を基準として設定可能です。

16.7.2. Cygwin

Windowsに対するLinux的環境である、Cygwinを使ってPostgreSQLを構築することが可能です。しかし、この手法はWindowsネイティブビルド(第17章を参照)には及ばないので、もはや推奨されません。

ソースから構築する場合、以下のCygwin特有の差異に注意し、Unix形式のインストール手順に従って進めます(つまり、./configure;make; など)。

- Windowsユーティリティの前に使用するCygwinのbinディレクトリのパスを設定します。コンパイルにおける問題を回避する助けになります。
- adduserコマンドはサポートされていません。Windows NT、2000、またはXP上の適切なユーザ管理アプリケーションを使用してください。そうでなければ、この手順を飛ばします。
- suコマンドはサポートされていません。Windows NT、2000、またはXP上でsuをシミュレートするため、sshを使用します。そうでなければ、この手順を飛ばします。
- OpenSSLはサポートされていません。
- 共有メモリサポートのためにcygserverを開始します。これを行うためには、コマンド/usr/sbin/cygserver&を入力します。このプログラムはPostgreSQLサーバを起動するとき、または(initdbで)データベースクラスタを初期化するときはいつでも必要です。システム資源が欠けていることによるPostgreSQLの失敗を避けるため、デフォルトのcygserver設定を(例えばSEMMNSを増やすなど)変更する必要があるかもしれません。
- いくつかのシステムでは、Cロケール以外を使っている場合に構築が失敗するかもしれません。これに対処するためには、構築前にexport LANG=C.utf8を実施してロケールをCに設定し、PostgreSQLのインストール後に以前の設定に戻してください。
- 並行リグレッションテスト(make check)は、接続拒絶エラーやハングアップを引き起こすlisten()バックログキューのオーバーフローにより、誤ったリグレッションテストの失敗を生成する可能性があります。make 変数MAX_CONNECTIONSを使用して、最大接続数を制限できます。つまり次のようにします。

```
make MAX_CONNECTIONS=5 check
```

（いくつかのシステムでは、同時接続を10まで広げられます。）

Windows NTサービスとしてcygserverとPostgreSQLサーバをインストールすることができます。これを実現する方法は、CygwinのPostgreSQLバイナリパッケージに含まれるREADME文書を参照してください。それは/usr/share/doc/Cygwinディレクトリにインストールされます。

16.7.3. macOS

最近のmacOSのリリースでは、システムヘッダファイルを見つけるために使われるインクルードスイッチに「sysroot」のパスを埋め込むことが必要です。これにより、configureでどのSDKのバージョンが使われたかに依存して、configureスクリプトの出力が変わることになります。これは簡単なシナリオでは問題を引き起こさないでしょうが、サーバのコードが構築されたのとは異なるマシンで拡張を構築するなどのようなことを試みているのだとしたら、異なるsysrootのパスを利用するように強制することが必要です。そうするには、PG_SYSROOTを設定してください。例えば以下のようにです。

```
make PG_SYSROOT=/desired/path all
```

自分のマシンでの適切なパスを見つけるには、以下のようにしてください。

```
xcodebuild -version -sdk macosx Path
```

コアサーバを構築するのに使われたのとは異なるsysrootのバージョンを使って拡張を構築することは、実のところ勧められないことに注意してください。最悪の場合、デバッグの難しいABIの不一致を招くかもしれません。

configureにPG_SYSROOTを指定することで、configureの時にデフォルトでないsysrootのパスを選ぶこともできます。

```
./configure ... PG_SYSROOT=/desired/path
```

macOSの「System Integrity Protection」(SIP)機能は、DYLD_LIBRARY_PATHの必要な設定をテスト対象の実行ファイルに渡すのを妨げますので、make checkを壊します。make checkの前にmake installとすることで回避できます。ですが、PostgreSQLの開発者はほとんど単にSIPをオフにしています。

16.7.4. MinGW/ネイティブWindows

Windows用PostgreSQLは、Microsoftオペレーティングシステム用のUnixに似た構築環境であるMinGW、またはMicrosoftのVisual C++コンパイラ一式を使って構築できます。MinGW版の構築は本章で記述されている通常の構築システムを使用します。Visual C++構築は、第17章で記述するようにまったく異なった動作をします。

ネイティブに移植されたWindows版ではWindows 2000以降の32ビットまたは64ビット版が必要です。これより前のオペレーティングシステムには充分な構造基盤がありません（ただし、Cygwinはそれら上で使える可能性があります）。Unixに似た構築ツールであるMinGWと、configureのようなシェルスクリプトを実行す

るために必要なUnixツール群であるMSYSは、http://www.mingw.org/からダウンロード可能です。作成されたバイナリの実行にはいずれも必要ありません。バイナリの作成のためのみ必要です。

MinGWを使って64ビット版バイナリをビルドするためには、https://mingw-w64.org/から64ビット用のツールを入手してインストールし、PATHにあるbinディレクトリへそれらを入れ、そして--host=x86_64-w64-mingw32オプション付きでconfigureを実施します。

MSYSコンソールはバッファリングに問題があるので、すべてをインストールした後にCMD.EXE下でpsqlを実行することを推奨します。

16.7.4.1. Windows上でのクラッシュダンプの収集

もしWindows上のPostgreSQLがクラッシュした場合、Unixにおけるコアダンプと似た、クラッシュの原因を追跡するために使用できるminidumpsを生成することができます。このダンプはWindows Debugger ToolsやVisual Studioを使うことで解析できます。Windowsにてダンプを生成できるように、crashdumpsという名前のサブディレクトリをデータベースクラスタディレクトリの中に作成します。ダンプは、クラッシュ時の現在時間と原因となったプロセスの識別子を元にした一意な名前としてこのディレクトリの中に生成されます。

16.7.5. Solaris

PostgreSQLはSolaris上でとても良くサポートされています。オペレーティングシステムが更新されればされる程、問題点の遭遇は少なくなります。

16.7.5.1. 必要なツール

GCCもしくはSunのコンパイラー式により構築できます。より良いコード最適化のため、SPARCアーキテクチャではSunのコンパイラを強く推奨します。Sunのコンパイラを使用するのであれば、/usr/ucb/ccを選択せず、/opt/SUNWspro/bin/ccを使用するように注意してください。

https://www.oracle.com/technetwork/server-storage/solarisstudio/downloads/からSun Studioをダウンロードできます。数多くのGNUツールがSolaris 10に統合、もしくはSolaris companion CDの中にあります。Solarisのより古いバージョンに対するパッケージが必要であれば、それらのツールはhttp://www.sunfreeware.comにあります。ソースの方が良いという方はhttps://www.gnu.org/prep/ftpを参照してください。

16.7.5.2. 失敗したテストプログラムについてconfigureが出すエラー

もしconfigureが失敗したテストプログラムについてエラーを出す場合、おそらく実行時のリンカがlibz、libreadline、またはlibsslのような非標準のライブラリを見つけ出せないことによります。それを正しい場所に指し示すため、configureコマンドラインでLDFLAGS環境変数を例えば以下のように設定します。

```
configure ... LDFLAGS="-R /usr/sfw/lib:/opt/sfw/lib:/usr/local/lib"
```

より詳細な情報はldマニュアルページを参照ください。

16.7.5.3. 最適性能を得るためのコンパイル

SPARCアーキテクチャにおけるコンパイルでは、Sun Studioを強く推奨します。特筆するような速さのバイナリを生成するため、-x05最適化フラグを使用してみてください。浮動小数点演算と、(-fastのような)errno演算を修正するようなフラグはすべて使ってはいけません。

SPARCで64ビットバイナリを使用する理由がないのであれば、32ビット版を選択してください。64ビット操作はより遅く、64ビットバイナリは32ビット版より遅いのです。一方で、AMD64 CPUファミリ上の32ビットコードはネイティブではありません。そのため、このCPUファミリでは32ビットコードは非常に低速です。

16.7.5.4. PostgreSQLをトレースするためのDTrace使用

そのとおりです。DTraceを使うことができます。より詳細な情報は27.5を参照してください。

以下のようなエラーメッセージでpostgres実行形式のリンクが中断することを体験した場合、そのDTraceインストールが静的関数におけるプローブを扱うには古すぎると言うことです。DTraceを使うには、Solaris 10u4もしくはそれより新しいものが必要です。

```
Undefined                       first referenced
 symbol                             in file
AbortTransaction                   utils/probes.o
CommitTransaction                  utils/probes.o
ld: fatal: Symbol referencing errors. No output written to postgres
collect2: ld returned 1 exit status
make: *** [postgres] Error 1
```

第17章 Windowsにおけるソースコードからのインストール

ほとんどのユーザには、PostgreSQLウェブサイトからグラフィカルインストーラパッケージとして入手可能なWindows用のバイナリ配布物をダウンロードすることを推奨します。ソースからの構築は、PostgreSQLそのもの、もしくはその拡張の開発者のみを対象としています。

WindowsでPostgreSQLを構築する方法は複数存在します。Microsoftのツールを使用した最も単純な構築方法は、Visual Studio 2019をインストールし、それに含まれるコンパイラを使用することです。また完全なMicrosoft Visual C++ 2013から2019までを使用しても構築することができます。コンパイラの他にWindows SDKのインストールが必要となる場合があります。

また、MinGWで提供されるGNUコンパイラツール、または、古めのWindowsではCygwinを使用してPostgreSQLを構築することができます。

MinGWまたはCygwinを使用した構築では、通常の構築システムを使用します。第16章、16.7.4および16.7.2にある固有の注記を参照してください。これらの環境で64ビットバイナリを生成するためにはMinGW-w64のツールを使用します。また、これらのツールは32ビットと64ビットWindows上で、LinuxやmacOSなどの他のホストを対象としたクロスコンパイルをする際にも使用されます。Cygwinは商用サーバでの稼動は推奨されません。これは、ネイティブな構築ができない古めのバージョンのWindowsでの使用に限定してください。公式のバイナリはVisual Studioを使用して構築しています。

psqlのネイティブな構築はコマンドライン編集をサポートしていません。Cygwinによる構築はコマンドライン編集をサポートしているので、Windows上でインタラクティブなpsqlの振る舞いが必要になる場合はこちらを使うべきでしょう。

17.1. Visual C++またはMicrosoft Windows SDKを使用した構築

Microsoftが提供するVisual C++コンパイラスイートを使用してPostgreSQLを構築することができます。これらのコンパイラはVisual Studio、Visual Studio Express、いくつかのバージョンのMicrosoft Windows SDKにあります。すでにVisual Studio環境が設定されているのでなければ、Microsoftから無料でダウンロードできるVisual Studio 2019のコンパイラやWindows SDK 10のものを使用することが最も簡単な方法です。

32ビットも64ビットもマイクロソフトのコンパイラー式で構築が可能です。32ビットPostgreSQLの構築はVisual Studio 2013からVisual Studio 2019までに付属のコンパイラをサポートします。単体のWindows SDKの8.1aから10までについても同様です。64ビットPostgreSQLの構築はMicrosoft Windows SDKのバージョン8.1aから10、もしくはVisual Studio 2013かそれ以上のバージョンでサポートしています。Visual Studio 2013からVisual Studio 2019での構築は、Windows 7やWindows Server 2008 R2 SP1までサポートされています。

Visual C++またはPlatform SDKを使用して構築するためのツールが`src/tools/msvc`ディレクトリに存在します。構築する際に、MinGWやCygwinに付属するツールがシステムPATHに存在しないことを確認してく

ださい。また、PATH上で必要なVisual C++ツールがすべて利用可能になっていることを確認してください。Visual Studioでは、Visual Studio コマンドプロンプトを起動してください。64ビット版を構築したい場合には、それぞれ64ビット版のコマンドを使用する必要がありますし、逆の場合も同様です。Visual Studio 2017からは、これはコマンドラインからVsDevCmd.batを使ってできます。利用可能なオプションとそのデフォルト値については-helpを参照してください。Visual Studio 2015とそれ以前のバージョンでは、同じ目的のためにvsvars32.batが利用可能です。Visual Studioコマンドプロンプトから、vcvarsall.batコマンドを使うことで対象のCPUアーキテクチャ、構築種類、対象OSを変更することができます。例えば、vcvarsall.bat x64 10.0.10240.0は対象をWindows 10の64ビットリリース版として構築します。その他のvcvarsall.batのオプションについては-helpを参照してください。すべてのコマンドはsrc\tools\msvcディレクトリから実行しなければなりません。

構築する前に、config.plファイルを編集して、変更したい設定オプションや使用する何らかのサードパーティ製のライブラリのパスを反映させる必要があるかもしれません。まずconfig_default.plファイルを読み取り、解析することから始まり、そしてconfig.pl内の何らかの変更が適用されて、すべての設定が決定されます。例えば、Pythonインストレーションの場所を指定する場合、以下をconfig.plに記載します。

```
$config->{python} = 'c:\python26';
```

config_default.pl内の指定と異なるパラメータのみを指定する必要があります。

何か他に環境変数を設定する必要があれば、buildenv.plという名前のファイルを作成し、そこに必要なコマンドを記載してください。たとえば、PATHにbison用のパスを追加したいのであれば、以下を含むファイルを作成してください。

```
$ENV{PATH}=$ENV{PATH} . ';c:\some\where\bison\bin';
```

Visual Studioビルドコマンド(msbuildまたはvcbuild)に追加のコマンドライン引数を渡すには次のようにします。

```
$ENV{MSBFLAGS}="/m";
```

17.1.1. 必要条件

PostgreSQLを構築するには以下の追加製品が必要です。config.plファイルを使用してライブラリを利用できるディレクトリを指定してください。

Microsoft Windows SDK

もしあなたの構築環境がサポートされているMicrosoft Windows SDKのバージョンを搭載していないのであれば、入手可能な最新版(現在はバージョン10)までアップグレードすることを推奨します。https://www.microsoft.com/downloadからダウンロードできます。

SDKのWindows Headers and Librariesを常にインクルードしなければなりません。Visual C++ Compilersに含まれるWindows SDKをインストールしている場合、構築のためにVisual Studioは必要ありません。バージョン8.0aでは、Windows SDKは完全なコマンドライン構築環境を提供していないことに注意してください。

ActiveState Perl

ActiveState Perlが構築生成スクリプトを実行するために必要です。MinGWまたはCygwinのPerlでは動作しません。また、PATH内に含められていなければなりません。https://www.activestate.comからバイナリをダウンロードできます（注意：バージョン5.8.3以降が必要です。フリー版の標準配布で十分です）。

使用できるようになることが目的であれば以下の追加製品は必要ありませんが、完全なパッケージを構築する場合には必要です。`config.pl`を使用してライブラリが利用できるディレクトリを指定してください。

ActiveState TCL

PL/Tclを構築する時に必要です（注意：バージョン8.4が必要です。フリー版の標準配布で十分です）。

BisonおよびFlex

Gitから構築する場合はBisonおよびFlexが必要です。しかしリリースファイルから構築する場合は不要です。Bison 1.875またはバージョン2.2以降のみで動作します。Flexはバージョン2.5.31以降でなければなりません。

BisonおよびFlexの両方が、MinGWコンパイラー式の一部としてhttp://www.mingw.org/wiki/MSYSから入手できる、msysツール一式に含まれています。

すでにPATHが通っていない場合は、`buildenv.pl`の中で`flex.exe`および`bison.exe`があるディレクトリをPATH環境変数に追加する必要があります。MinGWの場合、このディレクトリはMinGWのインストールディレクトリの`\msys\1.0\bin`サブディレクトリです。

注記

GnuWin32からのBisonディストリビューションでは、`C:\Program Files\GnuWin32`の様に名前に空白を持つディレクトリにインストールされると正常に機能しないというバグがあります。代わりに`C:\GnuWin32`へのインストール、または、PATH環境設定におけるGnuWin32へのNTFSショートネームパスの使用（例えば`C:\PROGRA~1\GnuWin32`）を検討してください。

注記

PostgreSQL FTPサイトで配布され、古い文書で参照していた古い`winflex`は、64ビットWindowsホストでは「flex: fatal internal error, exec failed」で失敗します。代わりにMSYSからのflexを使用してください。

Diff

リグレッションテストを実行するにはdiffが必要です。http://gnuwin32.sourceforge.netからダウンロードできます。

Gettext

NLSサポート付きで構築する場合はgettextが必要です。http://gnuwin32.sourceforge.netからダウンロードできます。バイナリ、依存物、開発用ファイルすべてが必要であることに注意してください。

MIT Kerberos

GSSAPI認証をサポートする場合に必要です。MIT Kerberosはhttps://web.mit.edu/Kerberos/dist/index.htmlからダウンロードできます。

libxml2およびlibxslt

XMLサポートのために必要です。バイナリはhttps://zlatkovic.com/pub/libxmlから、ソースはhttp://xmlsoft.orgからダウンロードできます。libxml2はiconvを必要とすることに注意してください。同じ場所からダウンロードできます。

OpenSSL

SSLサポートのために必要です。バイナリはhttps://slproweb.com/products/Win32OpenSSL.htmlから、ソースはhttps://www.openssl.orgからダウンロードできます。

ossp-uuid

UUID-OSSPサポート（contribのみ）で必要です。ソースはhttp://www.ossp.org/pkg/lib/uuid/にあります。

Python

PL/Pythonを構築する場合に必要です。バイナリはhttps://www.python.orgからダウンロードできます。

zlib

pg_dumpおよびpg_restoreにおける圧縮をサポートするために必要です。バイナリはhttps://www.zlib.netからダウンロードできます。

17.1.2. 64ビット版のWindowsにおける特別な考慮事項

64ビット版Windowsにおいてx64アーキテクチャのみでPostgreSQLを構築することができます。Itaniumプロセッサをサポートしていません。

同じ構築用ツリーで32ビット版と64ビット版を混在させることはサポートされません。構築システムは32ビット環境で動作しているか64ビット環境で動作しているかを自動的に検出し、それにしたがってPostgreSQLを構築します。このため構築作業を始める前に正しいコマンドプロンプトを開始することが重要です。

pythonやOpenSSLなどのサーバサイドのサードパーティ製ライブラリを使用するためには、ライブラリも64ビット版である必要があります。64ビット版のサーバで32ビット版のライブラリをロードすることはサポートされていません。PostgreSQLがサポートするサードパーティ製のライブラリで32ビット版しか利用できないものが複数あります。こうした場合、64ビット版のPostgreSQLで使用することはできません。

17.1.3. 構築

リリース条件（デフォルト）でPostgreSQLをすべて構築するためには、以下のコマンドを実行してください。

```
build
```

デバッグ条件でPostgreSQLをすべて構築するためには、以下のコマンドを実行してください。

```
build DEBUG
```

単一のプロジェクトのみを構築するためには、たとえばpsqlであれば、以下のコマンドを実行してください。

```
build psql
build DEBUG psql
```

デバッグのためにデフォルトの構築条件を変更するためには、以下をbuildenv.plファイルに記載してください。

```
$ENV{CONFIG}="Debug";
```

また、Visual Studio GUI内から構築することも可能です。この場合はコマンドプロンプトから以下を実行しなければなりません。

```
perl mkvcbuild.pl
```

その後に、生成されたpgsql.sln（ソースツリーのトップディレクトリに存在します。）をVisual Studioで開いてください。

17.1.4. 整理およびインストール

ほとんどの場合、Visual Studioの持つ自動依存関係追跡により変更されたファイルが扱われます。しかし、大規模な変更が行われた場合、インストレーションを整理する必要があるかもしれません。このためには、clean.batコマンドを実行してください。これにより、生成されたファイルがすべて自動的に消去されます。また、distパラメータを使用して実行することも可能です。この場合、**make distclean**のように振舞い、flex/bisonの出力ファイルも削除します。

デフォルトで、すべてのファイルはdebugまたはreleaseディレクトリ以下のサブディレクトリに書き出されます。これらのファイルを標準レイアウトでインストールし、データベースの初期化や使用に必要なファイルを生成するためには、以下のコマンドを実行してください。

```
install c:\destination\directory
```

クライアントアプリケーションとインタフェースライブラリだけをインストールしたいのであれば、以下のコマンドが使えます。

```
install c:\destination\directory client
```

17.1.5. リグレッションテストの実行

リグレッションテストを実行するためには、まず必要なすべての部品の構築が完了していることを確認してください。また、システムのすべての部品で必要とするDLL（手続き言語用のPerlのDLLやPythonのDLLなど）

がシステムパスに含まれていることを確認してください。もし含まれていなければ、buildenv.plファイルを介して設定してください。試験を実行するためには、以下のコマンドのいずれかをsrc\tools\msvcディレクトリから実行してください。

```
vcregress check
vcregress installcheck
vcregress plcheck
vcregress contribcheck
vcregress modulescheck
vcregress ecpgcheck
vcregress isolationcheck
vcregress bincheck
vcregress recoverycheck
vcregress upgradecheck
```

使用するスケジュール（デフォルトはparallelです）を変更するためには、コマンドラインに以下のように追加してください。

```
vcregress check serial
```

リグレッションテストの詳細については第32章を参照してください。

クライアントプログラムでvcregress bincheckによりリグレッションテストを実行したり、vcregress recoverycheckによりリカバリテストを実行したりするには、追加のPerlモジュールをインストールしておかなければなりません。

IPC::Run

これを書いている時点では、IPC::RunはActiveState Perlインストレーションにも ActiveState Perl Package Manager(PPM)ライブラリにも含まれていません。インストールするためには、https://metacpan.org/release/IPC-RunでCPANからIPC-Run-<version>.tar.gzソースアーカイブをダウンロードして、展開してください。buildenv.plを編集して、取り出されたアーカイブからlibサブディレクトリを指すように変数PERL5LIBを追加してください。例えば以下の通りです。

```
$ENV{PERL5LIB}=$ENV{PERL5LIB} . ';c:\IPC-Run-0.94\lib';
```

第18章 サーバの準備と運用

本章では、データベースサーバの設定と実行方法、そしてオペレーティングシステムとの相互作用について説明します。

本章で説明する手順は、追加の基盤を必要とせずに、単純なPostgreSQLを使用していることを前提としています。例えば、前の章で説明した手順に従ってソースからビルドしてコピーした等です。PostgreSQLのパッケージ化された版またはベンダー提供版で作業している場合は、パッケージャがシステムの規約に従ってデータベースサーバをインストールし、開始するための特別な準備をしている場合があります。詳細についてはパッケージレベルのドキュメントを参照してください。

18.1. PostgreSQLユーザアカウント

外部へアクセスできるサーバデーモンと同じように、PostgreSQLを独立したユーザアカウントで実行することをお勧めします。このユーザアカウントは、サーバによって管理されるデータのみを所有する必要があります。また、他のデーモンとアカウントを共有しない方が良いです。(例えば、nobodyユーザの使用はお勧めできません。) このユーザによって所有される実行プログラムをインストールすることも好ましくありません。特に、侵害されたサーバプロセスがこれらの実行可能ファイルを変更できないようにするために、このユーザアカウントは、PostgreSQL実行可能ファイルを所有しないことをお勧めします。

パッケージ化された版のPostgreSQLは、通常、パッケージのインストール中に自動的に適切なユーザアカウントを作成します。

システムにUnixのユーザアカウントを追加するためには、コマンドuseraddかadduserを使用してください。postgresというユーザ名がよく使われ、本書全体でも使用していますが、好みの名前を使用しても構いません。

18.2. データベースクラスタの作成

まず最初に、ディスク上にデータベース格納領域を初期化する必要があります。この格納領域を*データベースクラスタ*と呼びます。(標準SQLではカタログクラスタという用語が使用されています)。データベースクラスタはデータベースの集合で、稼働しているデータベースサーバのただ一つのインスタンスを通して管理されます。初期化が終わると、データベースクラスタにはpostgresという名前のデータベースが含まれています。このデータベースは、ユーティリティ、ユーザ、サードパーティ製アプリケーションが使用するデフォルトデータベースになります。データベースサーバ自身はこのpostgresデータベースの存在を必要としていませんが、多くの外部ユーティリティはその存在を想定しています。初期化中に他にもtemplate1というデータベースが各クラスタ内に作成されます。その名前から推測できるように、これはその後に作成されるデータベースのテンプレートとして使われます。したがって、実際の作業に使用しない方がよいです。(クラスタ内における新しいデータベースの作成については第22章を参照してください。)

ファイルシステムの観点から見ると、データベースクラスタというのは、すべてのデータが格納される1つのディレクトリということになります。これは*データディレクトリ*もしくは*データ領域*と呼ばれます。どこにデータを格納するかは完全にユーザの自由です。特にデフォルトの領域はありませんが、一般的によく使われるのは/usr/local/pgsql/dataや/var/lib/pgsql/dataです。データディレクトリは、使用前にPostgreSQLと一緒にインストールされるコマンドinitdbを使用して初期化する必要があります。

パッケージ化された版のPostgreSQLを使用している場合は、データディレクトリを配置する場所について特別な規則がある場合があります。また、データディレクトリを作成するためのスクリプトが提供されている場合もあります。その場合は、initdbを直接実行するのではなくそのスクリプトを使用する必要があります。詳細についてはパッケージレベルのドキュメントを参照してください。

データベースクラスタを手動で初期化するには、-Dオプションを使用してデータベースクラスタのファイルシステムの場所を指定しinitdbを実行します。例えば次のようにします。

```
$ initdb -D /usr/local/pgsql/data
```

このコマンドは、前節で説明したPostgreSQLユーザアカウントでログインしている間に実行する必要があることに注意してください。

> **ヒント**
>
> -Dオプションを使う代わりにPGDATA環境変数を設定することもできます。

他にも以下のようにpg_ctlプログラム経由でinitdbを実行することができます。

```
$ pg_ctl -D /usr/local/pgsql/data initdb
```

pg_ctlがデータベースサーバインスタンスの管理に使用する単一のコマンドになりますので、サーバの起動や停止にpg_ctlを使用している場合(18.3参照)はこちらの方がより直感的かもしれません。

もし指定したディレクトリが存在しない場合は、initdbはその新しいディレクトリを作成しようとします。もちろん、その親ディレクトリに書き込み権限がない場合initdbは失敗します。PostgreSQLユーザがデータディレクトリだけでなく、親ディレクトリも所有することを一般的に推奨します。このようにすると問題になることはありません。目的の親ディレクトリが存在しない場合は、まずそのディレクトリを作成する必要があります。親の親ディレクトリが書き込み可能でない場合は、root権限を使用して作成します。そのため、手順は下記のようになります。

```
root# mkdir /usr/local/pgsql
root# chown postgres /usr/local/pgsql
root# su postgres
postgres$ initdb -D /usr/local/pgsql/data
```

データディレクトリが存在し、すでにファイルが含まれている場合は、initdbは実行を拒否します。これは、誤って既存のインストールを上書きしないようにするためです。

データディレクトリにはデータベースの中のすべてのデータが保持されるため、権限を持たない人からのアクセスを確実に制限することが不可欠です。ですから、initdbはPostgreSQLユーザ、更にオプションでグループ以外からのアクセス権を剥奪します。許可されている場合には、グループアクセスは読み出し専用になります。これにより、クラスタの所有者と同じグループに所属する非特権ユーザが、そのクラスタのデータをバックアップすることや、読み出し権限だけが必要なその他の操作を実行することが可能になります。

既存のクラスタに対してグループアクセスを有効にする、あるいは無効にするには、PostgreSQLを再起動する前に、クラスタが停止済みの状態で、すべてのディレクトリとファイルに適切なモードが設定されている必要があることに注意してください。そうでないと、データディレクトリ内に異なるモードが混在してしまうかも

しれません。所有者のみにアクセスを許可するクラスタでは、適切なディレクトリのモードは0700で、ファイルモードは0600です。加えてグループに対して読み出しを許可するクラスタでは、適切なディレクトリのモードは0750で、ファイルモードは0640です。

しかし、ディレクトリの内容は安全ですが、デフォルトのクライアント認証の設定では、すべてのローカルユーザはデータベースに接続でき、データベーススーパーユーザになることさえ可能です。他のローカルユーザを信用しない場合、initdbの-W、--pwprompt、--pwfileオプションのいずれか1つを使用して、データベーススーパーユーザにパスワードを付与することを推奨します。また、デフォルトのtrust認証モードを使用しないように、-A md5もしくは-A passwordを指定してください。もしくは、initdbの後、初回のサーバの起動の前に、生成済みのpg_hba.confファイルを変更してください。(他の穏当な方法として、peer認証やファイルシステムの権限を使用して、接続を制限することもできます。詳細については第20章を参照してください。)

initdbはまた、データベースクラスタのデフォルトのロケールを初期化します。通常は、環境のロケール設定を初期化されたデータベースにそのまま適用します。データベースに異なるロケールを指定することも可能です。詳細については23.1を参照してください。特定のデータベースクラスタ内で使用されるデフォルトのソート順はinitdbで設定されます。異なるソート順を使用する新しいデータベースを作成することもできますが、initdbが作成するテンプレートデータベースで使用される順は削除して再作成しない限り変更することができません。また、CやPOSIX以外のロケールを使用する場合には性能上の影響もあります。ですので初回にこれを正しく選択することが重要です。

またinitdbは、データベースクラスタのデフォルトの文字セット符号化方式も設定します。通常これは、ロケールの設定と合うものが選ばれなければなりません。詳細は23.3を参照してください。

非Cおよび非POSIXのロケールでは、文字セットのソート順はオペレーティングシステムの照合ライブラリに依存しています。これは、インデックスに格納されているキーの順序を制御します。このためにクラスタは、スナップショットのリストア、バイナリストリーミングレプリケーション、異なるオペレーティングシステム、またはオペレーティングシステムのアップグレードのいずれでも互換性のない照合ライブラリバージョンに切り替えることは出来ません。

18.2.1. セカンダリファイルシステムの使用

多くのインストールでは、マシンの「ルート」ボリューム以外のファイルシステム(ボリューム)上にデータベースクラスタを作成します。この選択をした場合、セカンダリボリュームの最上位ディレクトリ(マウントポイント)をデータディレクトリとして使用することはお勧めできません。最善の方法はマウントポイントディレクトリ内にPostgreSQLユーザが所有するディレクトリを作成し、その中にデータディレクトリを作成することです。これにより、権限の問題、特にpg_upgradeなどの操作での問題を避けることができ、またセカンダリボリュームがオフラインになったときに、確実にきれいなエラーを起こすようになります。

18.2.2. ファイルシステム

一般的にはPOSIXのセマンティクスを備えたすべてのファイルシステムがPostgreSQLで利用できます。ユーザはベンダーのサポート、性能、慣れ親しんでいるかどうかなどの様々な理由で異なるファイルシステムを選択します。経験が示すところによると、これ以外の要素が同じなら、単にファイルシステムを変更したり、ファイルシステムの設定を少し変えただけで大きな性能の違いや挙動の違いがあるとは思わないほうが良いでしょう。

18.2.2.1. NFS

PostgreSQLのデータディレクトリを格納するためにNFSファイルシステムが使えます。PostgreSQLはNFSファイルシステムのために何ら特別なことはしません。つまりNFSがローカルに接続されたドライブと完全に同じように振る舞うものとみなします。PostgreSQLは、ファイルのロックなど、NFS上で非標準の振る舞いをすると知られている機能は使いません。

NFSをPostgreSQLで使う上での必須要件はhardオプションを使ってファイルシステムをマウントすることです。hardオプションでは、ネットワークに問題があればプロセスは永久に「ハング」する可能性があります。ですからこの設定では注意深い監視が必要になります。softオプションはネットワークに問題があるとシステムコールに割り込みますが、PostgreSQLはこの方法で割り込まれたシステムコールを再発行しません。ですからそのような割り込みに対してはI/Oエラーの発生が報告されることとなります。

syncマウントオプションを使う必要はありません。asyncオプションの動作で十分です。なぜならPostgreSQLは書き込みキャッシュを吐き出すために適切な時にfsync呼び出しを発行するからです。(これはローカルファイルシステム上での動作と同様です。) しかし、syncエクスポートオプションがあるシステム(主にLinux)上のNFSサーバでは、そのオプションを使うことを強くお勧めします。さもないとNFSクライアント上のfsync、あるいは同等ものは実際にはサーバ上の永続ストレージに到達することが保証されず、fsyncパラメータをオフにして実行するのと同じような破壊をもたらす可能性があります。これらのマウントオプションとエクスポートオプションのデフォルトはベンダーとバージョンによって違います。ですから曖昧さを避けるためにこれらのオプションをチェックし、また常に明示的にオプションを指定したほうが良いでしょう。

場合によっては外部ストレージ製品は、NFSあるいはiSCSIのような低レベルのプロトコルのどちらでもアクセスできます。後者の場合にはストレージはブロックデバイスとして扱われ、利用可能などのようなファイルシステムもその上に作ることができます。このアプローチはNFSの特異性に対処することからDBAを解放するかも知れません。もちろんリモートストレージを管理する複雑さが別のレベルで起こってしまいますが。

18.3. データベースサーバの起動

データベースにアクセスするためには、まずデータベースサーバを起動しなくてはいけません。データベースサーバプログラムはpostgresという名前です。

パッケージ化された版のPostgreSQLを使用している場合は、オペレーティングシステムの規則に従って、サーバをバックグラウンドタスクとして実行するための提供がほぼ確実に含まれています。パッケージの基盤を使用してサーバを起動させるほうが、自分でこれをおこなう方法を理解するよりもはるかに作業量が少なくなります。詳細についてはパッケージレベルのドキュメントを参照してください。

サーバを手動で起動するための必要最低限の方法は、-Dオプションを使用してデータディレクトリの場所を指定postgresを直接呼び出すことです。次に例を示します。

```
$ postgres -D /usr/local/pgsql/data
```

上記のコマンドはサーバをフォアグラウンドで実行させます。これは、PostgreSQLユーザアカウントでログインしている間に実行されなくてはいけません。-Dオプションが指定されていない場合、サーバはPGDATA環境変数で指定されたデータディレクトリを使用しようと試みます。どちらの変数も指定されていなければ失敗します。

通常はバックグラウンドでpostgresを起動することをお勧めします。そのためには以下のように通常のUnixシェルの構文を使います。

```
$ postgres -D /usr/local/pgsql/data >logfile 2>&1 &
```

この例のように、サーバの標準出力と標準エラー出力をどこかに保管しておくことが重要です。これは監査目的と問題の原因究明に役立ちます。(ログファイルの取り扱いについての全体的な説明については24.3を参照してください。)

postgresプログラムには、この他にも多くのコマンドラインオプションを指定することができます。詳細はpostgresマニュアルページと後述の第19章を参照してください。

こうしたシェル構文は長くなりがちです。そのため、pg_ctl ラッパプログラムが提供されていて、いくつかのタスクを単純化しています。以下に例を示します。

```
pg_ctl start -l logfile
```

これは、サーバをバックグラウンドで起動し、出力を指定されたログファイルに書き出します。-Dオプションは、ここでもpostgresの場合と同じ意味を持ちます。pg_ctlによってサーバを停止させることもできます。

通常、コンピュータが起動された時にデータベースサーバも一緒に起動したい場合が多いと思われます。自動起動スクリプトはオペレーティングシステム固有のものです。いくつかのスクリプトの例はPostgreSQLのcontrib/start-scriptsディレクトリに同梱されています。このインストールにはおそらくroot権限が必要となります。

起動時にデーモンを開始する方法はシステムによって異なります。多くのシステムには/etc/rc.localファイルや/etc/rc.d/rc.localファイルがあります。他のシステムではinit.dやrc.dディレクトリが使用されます。何を実行するにしても、サーバはPostgreSQLユーザアカウントで起動させなければなりません。*root*であってはいけませんし、他のユーザでもいけません。したがって、su postgres -c '...'を使用してコマンドを実行する必要があるでしょう。以下に例を示します。

```
su postgres -c 'pg_ctl start -D /usr/local/pgsql/data -l serverlog'
```

さらにいくつかのオペレーティングシステム固有の提案を挙げます。(ここでは一般的な値で説明していますので、各項目において適切なインストールディレクトリとユーザ名に置き換えて読んでください。)

- FreeBSDでは、PostgreSQLのソース配布物の中にあるcontrib/start-scripts/freebsdファイルを参照してください。
- OpenBSDでは、以下の数行を/etc/rc.localファイルに追加してください。

```
if [ -x /usr/local/pgsql/bin/pg_ctl -a -x /usr/local/pgsql/bin/postgres ]; then
    su -l postgres -c '/usr/local/pgsql/bin/pg_ctl start -s -l /var/postgresql/log -D /usr/
local/pgsql/data'
    echo -n ' postgresql'
fi
```

- Linuxシステムでは、

```
/usr/local/pgsql/bin/pg_ctl start -l logfile -D /usr/local/pgsql/data
```

を/etc/rc.d/rc.localや/etc/rc.localに追加してください。または、PostgreSQLのソース配布物の中にあるcontrib/start-scripts/linuxファイルを参照してください。

systemdを使用する場合は以下のサービスユニットファイルを（例えば/etc/systemd/system/postgresql.serviceとして）使用できます。

```
[Unit]
Description=PostgreSQL database server
Documentation=man:postgres(1)

[Service]
Type=notify
User=postgres
ExecStart=/usr/local/pgsql/bin/postgres -D /usr/local/pgsql/data
ExecReload=/bin/kill -HUP $MAINPID
KillMode=mixed
KillSignal=SIGINT
TimeoutSec=0

[Install]
WantedBy=multi-user.target
```

Type=notifyを使うには、サーバのバイナリがconfigure --with-systemdでビルドされている必要があります。

タイムアウトの設定について慎重に検討してください。この文書を書いている時点で、systemdのデフォルトのタイムアウトは90秒で、その時間内に準備ができたことを通知しないプロセスは終了させられます。しかし、PostgreSQLサーバは起動時にクラッシュリカバリを実行せねばならないことがあり、準備ができるまでにそれよりずっと長い時間を要することがあります。ここで提案されている0という値は、そのタイムアウトの仕組みを無効にします。

- NetBSDでは、FreeBSDかLinuxの好きな方の起動スクリプトを使用してください。
- Solarisでは、/etc/init.d/postgresqlというファイルを作成し、そこに以下の1行を記述してください。

```
su - postgres -c "/usr/local/pgsql/bin/pg_ctl start -l logfile -D /usr/local/pgsql/data"
```

そして、/etc/rc3.d以下にS99postgresqlとしてそのファイルに対するシンボリックリンクを作成してください。

サーバが実行している間は、そのPIDはデータディレクトリの中のpostmaster.pidファイルに記述されています。これは同じデータディレクトリで複数のサーバインスタンスが実行されるのを防止し、また、サーバの停止にも使うことができます。

18.3.1. サーバ起動の失敗

サーバの起動が失敗する理由として代表的なものがいくつかあります。サーバのログファイルを点検するか、(標準出力や標準エラーをリダイレクトせずに)手動で起動して、どのようなエラーメッセージが出ているか確認してください。以下に、よく発生するエラーメッセージのいくつかをより詳細に説明します。

```
LOG:  could not bind IPv4 address "127.0.0.1": Address already in use
HINT:  Is another postmaster already running on port 5432? If not, wait a few seconds and retry.
FATAL:  could not create any TCP/IP sockets
```

これはたいていの場合メッセージが示す通りの意味です。既にサーバが動いているポートで別のサーバを起動しようとしたことを示しています。しかし、カーネルエラーメッセージがAddress already in useやそれに類似したものではない場合は、別の問題の可能性もあります。例えば、予約済みのポート番号でサーバを起動しようとすると下記のようなメッセージが出るかもしれません。

```
$ postgres -p 666
LOG:  could not bind IPv4 address "127.0.0.1": Permission denied
HINT:  Is another postmaster already running on port 666? If not, wait a few seconds and retry.
FATAL:  could not create any TCP/IP sockets
```

次のようなメッセージが表示された場合、

```
FATAL:  could not create shared memory segment: Invalid argument
DETAIL:  Failed system call was shmget(key=5440001, size=4011376640, 03600).
```

これは、おそらくカーネルによる共有メモリのサイズの上限がPostgreSQLが作ろうとしている作業領域(この例では4011376640バイト)よりも小さいことを示しています。これはshared_memory_typeをsysvに設定した場合にのみ発生する可能性があります。その場合は、サーバを通常よりも少ないバッファ数(shared_buffers)で起動するか、カーネルを再設定して許容される共有メモリサイズを増やすこともできます。このメッセージは、同じマシン上で複数のサーバを起動しようとした時に、要求された領域の合計がカーネルの上限を超えた場合にも表示されます。

下記のようなエラーの場合:

```
FATAL:  could not create semaphores: No space left on device
DETAIL:  Failed system call was semget(5440126, 17, 03600).
```

ディスクの空き容量がなくなったということを示しているわけではありません。これはカーネルのSystem Vセマフォの上限が、PostgreSQLが作成しようとしている数よりも小さいということを意味しています。上記のように、許可される接続の数(max_connections)を減らしてサーバを起動させることで問題は回避できるかもしれませんが、最終的にはカーネルの設定を変えてセマフォの上限を増やした方が良いでしょう。

System V IPC設備の設定についての詳細は18.4.1を参照してください。

18.3.2. クライアント接続の問題

クライアント側で起こり得るエラー状態はきわめて多様で、アプリケーションに依存します。その中のいくつかはサーバが起動された方法と直接関係するかもしれません。以下で説明する以外の状態については各々のクライアントアプリケーションの資料を参照してください。

```
psql: could not connect to server: Connection refused
        Is the server running on host "server.joe.com" and accepting
        TCP/IP connections on port 5432?
```

これは一般的な「接続するサーバが見つけられませんでした」という失敗です。TCP/IP通信を試みた時に上記のように表示されます。よくある間違いはサーバにTCP/IPを許可する設定を忘れていることです。

代わりに、ローカルのサーバにUnixソケット通信を試みると下記のような表示が出ます。

```
psql: could not connect to server: No such file or directory
        Is the server running locally and accepting
        connections on Unix domain socket "/tmp/.s.PGSQL.5432"?
```

最後の行は、クライアントが正しいところに接続しようとしていることを実証するのに役立ちます。もしそこに動いているサーバがない場合、典型的なカーネルエラーメッセージは、表示されているようにConnection refusedもしくはNo such file or directoryとなります。(この場合のConnection refusedはサーバが接続要求を受け付けた後に拒否したわけではないということを理解しておくことが大切です。もしそうだった場合は20.15で示されるような別のメッセージが表示されます。) Connection timed outのような他のメッセージは、例えばネットワーク接続の欠如のようなもっと根本的な問題を表しています。

18.4. カーネルリソースの管理

PostgreSQLは、特に同一システム上で複数のサーバコピーを実行している場合や非常に大規模なインストレーションでは、オペレーティングシステムの様々なリソース制限を超えてしまうことがあります。本節では、PostgreSQLで使用されるカーネルリソース、およびカーネルリソース消費に関連した問題を解消する時に取ることができる手順について説明します。

18.4.1. 共有メモリとセマフォ

PostgreSQLはオペレーティングシステムが、プロセス間通信(IPC)特に共有メモリとセマフォ機能を提供することを要求します。Unix(派生)システムでは、「System V」IPCや、「POSIX」IPC、またはその両方を提供します。Windowsは、これらの機能を独自で実装しているため、ここでは説明しません。

デフォルトではPostgreSQLは通常、非常に少量のSystem V共有メモリと、もっと大量の無名mmap共有メモリを割り当てます。代替方法として、単一の大きなSystem Vメモリリージョンも利用できます(shared_memory_type参照)。さらに、System V又はPOSIXスタイルのどちらかのセマフォがサーバの起動時に作成されます。現在、LinuxとFreeBSDシステムではPOSIXセマフォが使用され、それ以外のプラットフォームではSystem Vセマフォが使用されます。

System V IPC機能は、通常システム全体の割り当て制限に制約されます。PostgreSQLがこれらの制限のいずれかを超えると、サーバは起動を拒否し、問題および何をすべきかを説明するエラーメッセージを残します。(18.3.1 も参照してください。) 関係するカーネルパラメータは別々のシステム上でも統一して名付けられています。表 18.1で概略がわかります。しかしこれらを設定するための方法は異なります。以下に、いくつかのプラットフォームへの提案を挙げます。

表18.1 System V IPCパラメータ

名前	説明	一つのPostgreSQLインスタンスに必要な値
SHMMAX	共有メモリセグメントの最大サイズ(バイト)	最小でも1キロバイト(ただしデフォルトはもっと多くなっています)
SHMMIN	共有メモリセグメントの最小サイズ(バイト)	1
SHMALL	使用可能な共有メモリの総量(バイトまたはページ)	バイト指定の場合はSHMMAXと同じ。ページ指定の場合はceil(SHMMAX/PAGE_SIZE)。+他のアプリケーション用の空間
SHMSEG	プロセスごとの共有メモリセグメントの最大数	必要なのは1セグメントのみ(ただしデフォルトはもっと多くなっています)
SHMMNI	システム全体の共有メモリセグメントの最大数	SHMSEGと同様+他のアプリケーション用の空間
SEMMNI	セマフォ識別子の最大数(つまりセット)	最低ceil((max_connections + autovacuum_max_workers + max_worker_processes + 5) / 16)+他のアプリケーション用の空間
SEMMNS	システム全体のセマフォの最大数	ceil((max_connections + autovacuum_max_workers + max_worker_processes + 5) / 16) * 17+他のアプリケーション用の空間
SEMMSL	セットごとのセマフォの最大数	最低17
SEMMAP	セマフォマップの中の項目の数	本文を参照
SEMVMX	セマフォの最大値	最低1000(デフォルトはしばしば32767ですが、必要がなければ変更しないでください)

PostgreSQLは、サーバのコピー毎にSystem V共有メモリの数バイト(64ビットプラットフォームでは通常48バイト)を必要とします。最近のほとんどのオペレーティングシステムでは、このくらいの量は簡単に割り当てられます。しかし複数のサーバのコピーを実行している場合やSystem V共有メモリを使用する他のアプリケーションを実行している場合(shared_memory_typeおよびdynamic_shared_memory_typeを参照)は、システム全体のSystem V共有メモリであるSHMALLを増加させる必要があるかもしれません。多くのシステムではSHMALLをバイト単位ではなくページ単位で測ることに注意してください。

問題が少ないのは共有メモリセグメントの最小サイズ(SHMMIN)で、PostgreSQLでは最大でもおよそ32バイトのはずです(通常では1です)。システム全体のセグメントの最大数(SHMMNI)もしくはプロセスごとのセグメントの最大数(SHMSEG)に関して、使用しているシステムで0に設定されていない限り、問題が起きることはほぼありません。

System V セマフォを使用している場合、PostgreSQLは、許可した接続(max_connections)、許可したオートバキュームワーカプロセス(autovacuum_max_workers)、許可したバックエンドプロセス(max_worker_processes)ごとに1つのセマフォを使用し、16個のセマフォをセットとして扱います。それぞれそのようなセットは、他のアプリケーションに使われているセマフォセッ

トとの衝突を検出するための「マジックナンバー」が含まれている17個目のセマフォを持っています。システム内のセマフォの最大数はSEMMNSによって設定され、その結果としてその値は少なくともmax_connections＋autovacuum_max_workers＋max_worker_processesと同じ、ただし、許可された接続とワーカ16個ごとに余分な1個を加えた値以上はなければいけません（表 18.1の公式を参照してください）。SEMMNIパラメータはシステム上に同時に存在できるセマフォ集合の数の上限を決定します。ですから、このパラメータは少なくともceil((max_connections + autovacuum_max_workers + max_worker_processes + 5) / 16)以上はなくてはいけません。一時的な失敗の回避策としては許可される接続の数を下げることができますが、「No space left on device」という紛らわしい言葉がsemget関数から表示されます。

場合によってはSEMMAPを少なくともSEMMNSと同程度に増やすことが必要になる場合があるかもしれません。システムにこのパラメータがあるなら（ないかもしれません）、このパラメータはセマフォリソースマップのサイズを定義し、その中では有効なセマフォのそれぞれの隣接したブロックの項目が必要です。セマフォ集合が解放されると、解放されたブロックに隣接する既に存在する項目に追加されるか、もしくは新しいマップの項目の下に登録されます。もしマップが一杯だった場合、解放されたセマフォは（再起動するまで）失われます。セマフォ空間の断片化により時間が経つごとに、有効なセマフォがあるべき量よりも少なくなる可能性があります。

SEMMNUとSEMUMEのような、その他の様々な「semaphore undo」に関する設定はPostgreSQLには影響を与えません。

POSIXセマフォを使用している場合、System Vと同じ数のセマフォを必要とします。つまり、許可した接続（max_connections）、許可したオートバキュームワーカプロセス（autovacuum_max_workers）、許可したバックエンドプロセス(max_worker_processes)ごとに1つのセマフォを使用します。このオプションが優先されるプラットフォームでは、POSIXセマフォの数に特定のカーネル制限はありません。

AIX

SHMMAXなどのパラメータに対して特別な設定は必要ありません。これは、すべてのメモリを共有メモリとして使用できるように設定されているためです。これはDB/2などの他のデータベースでも使用される、一般的な設定方法です。

しかし、/etc/security/limits内の大域的なulimit情報は変更しなければならないかもしれません。デフォルトのファイルサイズ（fsize）とファイル数（nofiles）用のハードリミットは低過ぎるかもしれないためです。

FreeBSD

shared_memory_typeをsysvに設定していない限り、通常はデフォルトの共有メモリ設定で十分です。System V セマフォはこのプラットフォームでは使用しません。

デフォルトのIPC設定はsysctlまたはloaderインタフェースを使用して変更を行うことができます。以下ではsysctlを使用してパラメータを変更しています。

```
# sysctl kern.ipc.shmall=32768
# sysctl kern.ipc.shmmax=134217728
```

これらの設定を再起動しても永続化するには、/etc/sysctl.confを変更します。

shared_memory_typeをsysvに設定している場合は、System V共有メモリをRAM上に固定して、スワップによってページアウトされるのを避けるために、カーネルを設定することもできます。これはsysctlを使用してkern.ipc.shm_use_physを設定することで実現できます。

FreeBSD jailを実行している場合、sysvshmパラメータをnewに設定して、独自のSystem V共有メモリ名前空間を有するようにする必要があります。(FreeBSD 11.0以前は、jailからIPC名前空間への共有アクセスを可能にし、衝突を避けるための対策を講じる必要がありました。)

NetBSD

shared_memory_typeをsysvに設定していない限り、通常はデフォルトの共有メモリ設定で十分です。通常は、kern.ipc.semmniとkern.ipc.semmnsを増やす必要があります。これは、NetBSDのデフォルト設定は不適切に小さいためです。

以下の例のようにIPCパラメータをsysctlを用いて調整することができます。

```
# sysctl -w kern.ipc.semmni=100
```

これらの設定を再起動しても永続化するには、/etc/sysctl.confを変更します。

shared_memory_typeをsysvに設定している場合は、System V共有メモリをRAM上に固定して、スワップによってページアウトされるのを避けるために、カーネルを設定することもできます。これはsysctlを使用してkern.ipc.shm_use_physを設定することで実現できます。

OpenBSD

shared_memory_typeをsysvに設定していない限り、通常はデフォルトの共有メモリ設定で十分です。通常は、kern.seminfo.semmniとkern.seminfo.semmnsを増やす必要があります。これは、OpenBSDのデフォルト設定は不適切に小さいためです。

以下の例のようにIPCパラメータをsysctlを用いて調整することができます。

```
# sysctl kern.seminfo.semmni=100
```

これらの設定を再起動しても永続化するには、/etc/sysctl.confを変更します。

HP-UX

通常のインストールでは、既定の設定で十分です。

IPCパラメータはシステム管理マネージャ(SAM)からKernel Configuration→Configurable Parametersの下で、設定することができます。終わったらCreate A New Kernelを選択してください。

Linux

shared_memory_typeをsysvに設定していて、低いデフォルトで出荷されたより古いカーネルバージョンでない限り、通常はデフォルトの共有メモリ設定で十分です。System V セマフォはこのプラットフォームでは使用しません。

共有メモリサイズの設定はsysctlインタフェースを使用して変更可能です。例えば16ギガバイトまで許すには以下のようにします。

```
$ sysctl -w kernel.shmmax=17179869184
$ sysctl -w kernel.shmall=4194304
```

これらの設定を再起動しても永続化するには、/etc/sysctl.confを参照してください。

macOS

shared_memory_typeをsysvに設定していない限り、通常はデフォルトの共有メモリとセマフォ設定で十分です。

macOSにおける共有メモリの推奨設定方法は、以下のような変数代入文からなる/etc/sysctl.confという名称のファイルを作成することです。

```
kern.sysv.shmmax=4194304
kern.sysv.shmmin=1
kern.sysv.shmmni=32
kern.sysv.shmseg=8
kern.sysv.shmall=1024
```

一部のバージョンのmacOSでは/etc/sysctl.conf内に共有メモリパラメータ*5つすべて*を設定しなければならないという点に注意してください。さもなくば値が無視されます。

SHMMAXは4096の倍数のみ設定できます。

このプラットフォームではSHMALLは4キロバイトページ単位です。

SHMMNI以外の変更は、sysctlを用いることにより、その場でおこなうことができます。しかしいずれにせよ/etc/sysctl.conf経由で望む値に設定することが最善です。再起動を行っても値が保持されるからです。

Solaris
illumos

大抵のPostgreSQLアプリケーションではデフォルトの共有メモリとセマフォ設定で十分です。Solarisのデフォルトの SHMMAXはシステムのRAMの1/4になりました。さらにこの設定を調整するためには、postgresユーザに関するプロジェクト設定を使用しなければなりません。例えば以下をroot権限で実行してください。

```
projadd -c "PostgreSQL DB User" -K "project.max-shm-memory=(privileged,8GB,deny)" -U postgres
 -G postgres user.postgres
```

このコマンドはuser.postgresプロジェクトを追加し、postgresユーザの共有メモリの最大サイズを8GBに設定します。この影響は次にこのユーザがログインした時、またはPostgreSQLを再起動した時（再読み込み時ではありません）に有効になります。上ではPostgreSQLはpostgresグループに属するpostgresユーザにより実行されていることを前提としています。サーバの再起動は不要です。

多くの接続を受け付けるデータベースサーバにおいて推奨するカーネル設定にはこの他に以下があります。

```
project.max-shm-ids=(priv,32768,deny)
project.max-sem-ids=(priv,4096,deny)
project.max-msg-ids=(priv,4096,deny)
```

さらに、ゾーン内でPostgreSQLを実行している場合、ゾーンのリソース使用上限も上げる必要があるかもしれません。projectsとprctlについては*System Administrator's Guide*の第2章 プロジェクトとタスクを参照してください。

18.4.2. systemd RemoveIPC

systemdが使用されている場合、(共有メモリを含む)IPCリソースがオペレーティングシステムによって時期尚早に削除されないように注意する必要があります。これはPostgreSQLをソースからインストールした場合に特に重要です。PostgreSQLのディストリビューションパッケージのユーザーは、通常postgresユーザーがシステムユーザーで作成されるため、影響を受けにくいでしょう。

logind.confのRemoveIPCの設定はユーザが完全にログアウトしたときにIPCオブジェクトを削除するかどうかを制御します。システムユーザは免除されます。この設定のデフォルトはsystemdですが、いくつかのオペレーティングシステムではデフォルトでオフになっています。

この設定が有効になっている時の典型的な影響は、並列問い合わせの実行で使われる共有メモリオブジェクトが見かけ上ランダムな時間に削除され、共有メモリオブジェクトをオープンしようとしたり、削除しようとしたりした時に以下のようなエラーや警告が出ることです。

```
WARNING:  could not remove shared memory segment "/PostgreSQL.1450751626": No such file or
 directory
```

IPCオブジェクトの違い(共有メモリ vs. セマフォ、System V vs. POSIX)はsystemdによって若干扱いが異なるため一部のIPCは他のものと違って削除されないことがあります。しかし、これらの微妙な違いに依存することはお勧めできません。

「ユーザーログアウト」は、メンテナンスジョブの一環として、又は手動で、管理者がpostgresユーザーや類似のユーザでログインする可能性があるため、一般的に防止することは困難です。

「システムユーザー」は、/etc/login.defsのSYS_UID_MAXの設定によりsystemdのコンパイル時に決定されます。

パッケージとデプロイスクリプトは、useradd -r, adduser --system又は同等のコマンドを使用してpostgresユーザを作成するように注意する必要があります。

また、ユーザアカウントが誤って作成されて変更出来ないような場合は、以下を設定することを推奨します。

```
RemoveIPC=no
```

/etc/systemd/logind.conf又はその他の設定ファイルで上記を入れます。

> **注意**
>
> これらの2つのうち少なくとも1つが保証されてないとなりません。そうでないとPostgreSQLサーバは非常に信頼性が低くなります。

18.4.3. リソースの制限

Unixライクなオペレーティングシステムでは PostgreSQL サーバの操作と関係する可能性のある様々な種類のリソース制限があります。特に重要なのは、ユーザごとのプロセス数の制限、プロセスごとのオープンファイルの数、プロセスごとの利用可能なメモリの量です。これらのそれぞれが「ハード」と「ソフト」の2つの制限を持っています。ソフト制限が実際に有効な制限ですが、ユーザによってハード制限まで変えることが可能です。ハード制限はrootユーザによってのみ変えることができます。setrlimitシステムコールがこれらのパラメータの設定を行います。シェルの組み込みコマンドulimit（Bourne シェル）もしくはlimit（csh）は、コマンドラインからリソース制限を制御するために使われます。BSD派生システム上では/etc/login.confファイルが、ログイン時に設定される様々なリソース制限を制御します。詳細はオペレーティングシステムの文書を参照してください。関連するパラメータはmaxproc、openfiles、datasizeです。以下に例を示します。

```
default:\
...
        :datasize-cur=256M:\
        :maxproc-cur=256:\
        :openfiles-cur=256:\
...
```

（-curはソフト制限です。ハード制限を設定するためには-maxを付けてください。）

カーネルはいくつかのリソースに対して、システム全体の制限も持つことができます。

- Linuxでは、/proc/sys/fs/file-maxが、カーネルがサポートするオープンファイル数の最大を決定します。この数を変えるためには、そのファイルに別の数を書き込むか、あるいは/etc/sysctl.confに代入式を追加します。プロセスごとのファイルの最大制限はカーネルがコンパイルされた時に固定されます。詳しい情報については/usr/src/linux/Documentation/proc.txtを参照してください。

PostgreSQLサーバは接続ごとに1つのプロセスを使うので、少なくとも許可された接続の数だけのプロセスに残りのシステムで必要な分を追加したものが必要になります。通常はこれは問題ではありませんが、1つのマシン上でいくつかのサーバを起動している場合は厳しい状況になるかもしれません。

オープンファイルの制限の出荷時のデフォルトは、しばしば大多数のユーザはマシン上でシステムリソースの不正使用をしないという前提に立った「社会的に友好的な」値を設定してしまいます。もし1つのマシン上で複数のサーバを起動する場合はそれが必要でしょうが、専用サーバではこの制限を上げたいかもしれません。

反対に、個々のプロセスが多数のファイルをオープンすることを許可するシステムもあります。そのようなプロセスが数個以上あれば、システム全体の制限は簡単に超えてしまいます。この発生を検知し、システム全体の制限の変更を望まない場合は、PostgreSQLのmax_files_per_process設定パラメータを設定し、オープンファイルの消費を制限することができます。

18.4.4. Linuxのメモリオーバーコミット

Linuxでのデフォルトの仮想メモリの動作はPostgreSQLには最適ではありません。カーネルがメモリオーバーコミットを実装する方法のため、カーネルは、PostgreSQLや他のプロセスのメモリ要求がシステムの仮

想メモリを枯渇させた場合、PostgreSQL postmaster（マスタサーバプロセス）を終了させる可能性があります。

これが発生した場合、以下のようなカーネルメッセージが現れます（こうしたメッセージを検索する場所についてはシステム文書と設定を参照してください）。

```
Out of Memory: Killed process 12345 (postgres).
```

これは、postgresプロセスがメモリ不足のために終了してしまったことを示します。起動中のデータベース接続は正常に動作しますが、新しい接続は受け付けられません。復旧するには、PostgreSQLを再起動しなければなりません。

この問題を防止する1つの方法として、PostgreSQLを他のプロセスがそのマシンのメモリを枯渇させないことが確実なマシンで起動するというものがあります。物理メモリとスワップ領域が消費尽くされた時のみにメモリ不足（OOM）killerが発生するため、メモリが不足する場合、オペレーティングシステムのスワップ領域を増やすことが問題解決の役にたちます。

PostgreSQL自体が実行中のシステムのメモリ不足を引き起こした場合、設定を変更することで問題を防止することができます。場合によっては、メモリ関連の設定パラメータ、shared_buffers、work_memおよびhash_mem_multiplierを低くすることで回避できる場合もあります。この他にもデータベースサーバ自体への接続を多く許可しすぎることで問題が引き起こされる場合もあります。多くの場合、max_connectionsを減らし、外部のコネクションプールソフトウェアを使用することで改善されます。

メモリを「オーバーコミット」させないようにカーネルの動作を変更することができます。この設定は完全にOOM killer[1]の発生を防ぐことはできませんが、その発生頻度をかなり軽減しますので、システム動作の堅牢性をより高めます。これは、以下のようにsysctlを使用して厳密なオーバーコミットモードを選択することで実施されます。

```
sysctl -w vm.overcommit_memory=2
```

また、関連するvm.overcommit_ratio設定を変更した方が良いでしょう。詳細はカーネル文書ファイルhttps://www.kernel.org/doc/Documentation/vm/overcommit-accountingを参照してください。

vm.overcommit_memoryの変更と関係なく使用できるその他の方法は、プロセス固有の*OOM*スコア調整値をpostmasterプロセス向けに-1000に設定することです。これによりOOM killerの対象とならないことが保証されます。このための最も簡単な方法は以下をpostmasterの起動スクリプト内でpostmasterを実行する直前に実行することです。

```
echo -1000 > /proc/self/oom_score_adj
```

この作業をrootで実行しなければならないことに注意して下さい。さもないと効果がありません。このためrootが所有する起動スクリプトが、これを行うための最も簡単な場所です。その場合には、スタートアップスクリプトのpostmasterの起動前に以下の環境変数を設定することも推奨します。

```
export PG_OOM_ADJUST_FILE=/proc/self/oom_score_adj
```

[1] https://lwn.net/Articles/104179/

```
export PG_OOM_ADJUST_VALUE=0
```

これらの設定は、いざという時にpostmasterの子プロセスをOOM killerのターゲットに出来るようにOOMスコア調整を通常のゼロで実行します。子プロセスを他のOOMスコア調整で実行したい場合には、PG_OOM_ADJUST_VALUEにより別の値にすることが出来ます。(PG_OOM_ADJUST_VALUEは省略することが出来て、その場合はデフォルトのゼロになります。) PG_OOM_ADJUST_FILEを設定しない場合、子プロセスはpostmasterと同じOOMスコア調整で実行されますが、postmasterが優先される設定にすることが肝心なので、それは賢明とは言えません。

18.4.5. LinuxのHugePages

PostgreSQLのように、メモリの大きな連続チャンクを使用するとき、特にshared_buffersの値が大きい場合に、huge pagesを使用するとオーバーヘッドが減少します。PostgreSQLでこの機能を有効にするには、CONFIG_HUGETLBFS=yおよびCONFIG_HUGETLB_PAGE=yとしたカーネルが必要です。またカーネル設定vm.nr_hugepagesを調整する必要もあるでしょう。必要なhuge pages数を見積もるには、huge pagesを有効にせずにPostgreSQLを起動し、/procファイルシステムを使用してpostmasterの無名共有セグメントサイズとシステムのhuge pageサイズの値をチェックします。これは以下のような感じになるでしょう。

```
$ head -1 $PGDATA/postmaster.pid
4170
$ pmap 4170 | awk '/rw-s/ && /zero/ {print $2}'
6490428K
$ grep ^Hugepagesize /proc/meminfo
Hugepagesize:       2048 kB
```

6490428 / 2048はおよそ3169.154ですので、この例では少なくとも3170のhuge pagesが必要で、それは以下のようにして設定できます。

```
$ sysctl -w vm.nr_hugepages=3170
```

同じマシン上で他にもhuge pagesが必要なプログラムがあるなら、もっと大きな設定が適切でしょう。再起動のときにこの設定が適用されるように、これを/etc/sysctl.confに追加するのを忘れないで下さい。

時には、カーネルは求められた数のhuge pagesを割り当てることができないことがあるので、そのコマンドを繰り返すか、再起動する必要があるかもしれません。(再起動の直後は、マシンのメモリの大部分はhuge pagesへの変更が可能なはずです。) huge pagesの割り当ての状況を確認するには、次のようにします。

```
$ grep Huge /proc/meminfo
```

sysctlを使ってvm.hugetlb_shm_groupを設定する、あるいはulimit -lでメモリをロックする権限を与えることで、データベースサーバのOSユーザにhuge pagesを使用する権限を与える必要もあるかもしれません。

PostgreSQLのhuge pagesのデフォルトの動作は、可能な場合はhuge pagesを使用し、失敗した場合は通常のページを使用します。postgresql.confでhuge_pagesをonに設定することで、huge pagesの使用を強制することができます。この設定の場合、十分なhuge pagesが確保できなければ、PostgreSQLの起動に失敗することに注意してください。

Linuxのhuge pages機能の詳細はhttps://www.kernel.org/doc/Documentation/vm/hugetlbpage.txtを参照してください。

18.5. サーバのシャットダウン

データベースサーバをシャットダウンする方法は複数あります。内部的には、これらはすべてスーパーバイザのpostgresプロセスにシグナルを送信することになります。

パッケージ化された版のPostgreSQLを使用していて、サーバの起動にその規定を使用した場合は、サーバの停止にもその規定を使用する必要があります。詳細についてはパッケージレベルのドキュメントを参照してください。

サーバを直接管理する場合は、postgresプロセスに異なるシグナルを送信することで、シャットダウンのタイプを制御できます。

SIGTERM

これはスマートシャットダウンモードです。SIGTERMを受け取った後で、サーバは新しい接続を禁止しますが、既に存在するセッションは通常通り動作させます。すべてのセッションが通常に終了するまではシャットダウンしません。サーバがオンラインバックアップモードである場合、オンラインバックアップモードが活動しなくなるまでさらに待ちます。バックアップモードが活動している間、新規接続は依然として許可されますが、スーパーユーザに対してだけです（この例外はスーパーユーザがオンラインバックアップモード停止のための接続を許可します）。スマートシャットダウンが要求された時にサーバがリカバリ状態である場合、すべての通常のセッションが終了した後のみでリカバリとストリーミングレプリケーションが停止します。

SIGINT

これは高速シャットダウンモードです。サーバは新しい接続を禁止しすべての存在するサーバプロセスにSIGTERMを送り、この結果サーバプロセスは現在のトランザクションをアボートし、即座に終了します。そして サーバはすべてのサーバプロセスの終了を待って、最後にシャットダウンします。サーバがオンラインバックアップモードである場合、バックアップモードは終了しますので、そのバックアップは使用することができなくなります。

SIGQUIT

これは即時シャットダウンモードです。サーバは、すべての子プロセスにSIGQUITを送信し、それらが終了するのを待ちます。5秒以内に終了しないものには、SIGKILLが送られます。すべての子プロセスが終了したら、マスタサーバプロセスはすぐに終了しますが、このとき通常のデータベースのシャットダウン処理を実行しません。これは次の起動時に（WALログを再実行することで）リカバリをすることになります。これは緊急の時にのみ使うことを勧めます。

pg_ctlプログラムは、サーバをシャットダウンさせるシグナルを送信するための便利なインタフェースを提供します。他にも、Windows以外のシステムではkillを使用して直接シグナルを送信することもできます。postgresプロセスのPIDは、psプログラム、もしくはデータディレクトリの中のpostmaster.pidファイルを使用して見つけることができます。例えば、高速シャットダウンをするためには下記のようになります。

```
$ kill -INT `head -1 /usr/local/pgsql/data/postmaster.pid`
```

重要

サーバをシャットダウンするためにSIGKILLを使わない方が良いでしょう。これによってサーバが共有メモリとセマフォを解放できなくなります。さらに、SIGKILLは、子プロセスにシグナルを中継することなくpostgresを停止させます。このため、個々の子プロセスを停止させるために、同じ作業を手作業で行わなければならなくなります。

他のセッションを継続させながら個別のセッションを停止するにはpg_terminate_backend()(表 9.84を参照)を使用するか、そのセッションに関連する子プロセスにSIGTERMシグナルを送ります。

18.6. PostgreSQLクラスタのアップグレード処理

本節ではPostgreSQLリリースからより新しいリリースにデータベースデータをアップグレードする方法を説明します。

現在のPostgreSQLのバージョン番号はメジャーバージョンとマイナーバージョンのバージョン番号で構成されます。例えばバージョン番号10.1は、10がメジャーバージョンで、1がマイナーバージョンです。メジャーリリース10の最初のマイナーリリースを意味します。PostgreSQLの10.0より前のバージョンは, 3つの番号で構成されています。例えば9.5.3です。この場合は、メジャーバージョンが最初の2つのグループのバージョン番号、例えば9.5で構成されています。そしてマイナーバージョンは3つ目の番号で例えば3です。これはメジャーリリース9.5の3番めのマイナーリリースを意味します。

マイナーリリースでは内部格納書式が変わることは決してありませんので、同じメジャーバージョンにおける前後のマイナーリリースとの間で常に互換性があります。例えばバージョン10.1はバージョン10.0やバージョン10.6と互換性があります。同様に、例えば9.5.3は9.5.0、9.5.1、9.5.6と互換性があります。互換性があるバージョンとの間で更新するためには、サーバを停止させ、実行ファイルを置き換え、サーバを再起動させるだけです。データディレクトリはまったく変更されません。マイナーリリースのアップグレードは簡単です。

PostgreSQLのメジャーリリースでは、内部データ格納書式は変更されがちです。したがって、アップグレードは複雑になります。新しいメジャーバージョンにデータを移行する伝統的な方法は、遅くなることがありますが、データベースをダンプしてリロードすることです。より速い方法については、pg_upgradeを参照してください。以下で説明するようにレプリケーションを使用する方法もあります。(パッケージ化された版のPostgreSQLを使用している場合は、主要バージョンのアップルグレードを支援するスクリプトが提供される場合があります。詳細についてはパッケージレベルのドキュメントを参照してください。)

新しいメジャーバージョンは通常、ユーザにも影響する非互換性がいくつか導入されます。このためアプリケーションのプログラム変更が必要になる可能性があります。ユーザに影響する変更はすべてリリースノート(付録E)に列挙されています。「移行」という名前の節に特に注意してください。複数のメジャーバージョンをまたいでアップグレードする場合は、関連するバージョンそれぞれのリリースノートを確認してください。

用心深いユーザは、完全に切り替える前に新しいバージョンにおける自身のクライアントアプリケーションを試験したいと考えるでしょう。このため古いバージョンと新しいバージョンを並行してインストールさせるというのは、よく良い考えとなります。PostgreSQLメジャーアップグレードを試験する時、以下に示す変更があり得る分野を検討してください。

管理

各メジャーリリースにおいて、管理者が利用できるサーバの監視、制御機能はよく変更、向上されます。

SQL

通常、これには新しいSQLコマンド機能が含まれます。リリースノートに特に記載がない限りその動作には変更はありません。

ライブラリAPI

繰り返しになりますが、リリースノートに記載がない場合のみですが、通常libpqのようなライブラリには新しい機能が追加されるだけです。

システムカタログ

システムカタログの変更は通常データベース管理用ツールにのみ影響します。

サーバC言語API

ここには、Cプログラム言語で作成されたバックエンド関数APIにおける変更が含まれます。こうした変更は、サーバ内部深くにあるバックエンド関数を参照するコードに影響します。

18.6.1. pg_dumpallを介したデータのアップグレード

PostgreSQLのアップグレードの一つの方法は、PostgreSQLの1メジャーバージョンからデータをダンプし、別のバージョンにリロードすることです - これを行うには、pg_dumpallのような論理バックアップツールを使用しなければなりません。ファイルシステムレベルのバックアップ方法は動作しません。(あるデータディレクトリで間違ったバージョンのサーバを起動しようとして、大きな損害が起こることがないように、適所に互換性がないバージョンのPostgreSQLのデータディレクトリが使用されないようにするための検査があります。)

新しいバージョンのPostgreSQLのpg_dumpとpg_dumpallを使用することを勧めます。これらのプログラムで拡張された機能を利用する可能性があるためです。現在のリリースのダンププログラムは7.0以降のバージョンのサーバからデータを読み取ることができます。

以下の手順では、既存のインストレーションが/usr/local/pgsql以下にあり、そのデータ領域が/usr/local/pgsql/dataにあることを前提としています。使用しているパスに適切に置き換えてください。

1. バックアップを作成する場合、使用しているデータベースが確実に更新されないようにしてください。これはバックアップの整合性には影響しませんが、当然ながら変更されたデータがバックアップに含まれません。必要に応じて、/usr/local/pgsql/data/pg_hba.conf(またはこれと等価なファイル)における権限を変更して、バックアップを行うユーザ以外からのアクセスを禁止してください。アクセス制御に関する情報は第20章を参照してください。

   データベースインストレーションをバックアップするためには以下を入力してください。

   ```
   pg_dumpall > outputfile
   ```

   バックアップを作成するために、現在起動中のバージョンのpg_dumpallコマンドを使用することができます。詳細は25.1.2 を参照してください。しかし最善の結果を得るためには、PostgreSQL 13.1のpg_dumpallコマンドを試してください。このバージョンでは、過去のバージョンに対して、不具合の修正や改良が含まれているからです。新しいバージョンをまだインストールしていませんので、この勧告は奇異に思えるかもしれませんが、古いバージョンと並行して新しいバージョンをインストールすることを

計画しているのであれば、これに従うことを推奨します。この場合、インストールを普通に完了させてからデータを移行することができます。これは同時に停止時間を短縮します。

2. 古いサーバを停止します。

```
pg_ctl stop
```

起動時にPostgreSQLを実行させるようにしているシステムではおそらく、同じことを達成する起動ファイルがあります。例えばRed Hat Linuxシステムでは、以下が動作することが分かります。

```
/etc/rc.d/init.d/postgresql stop
```

サーバの起動と停止については第18章を参照してください。

3. バックアップからリストアする場合、名前を変更、またはバージョン固有でない場合は古いインストレーションディレクトリを削除してください。問題があった場合に戻さなければならない場合に備え、削除するよりディレクトリの名前を変更する方を勧めます。このディレクトリが多くのディスク容量を占めている可能性があることに注意してください。ディレクトリの名前を変更するためには、以下のようなコマンドを使用してください。

```
mv /usr/local/pgsql /usr/local/pgsql.old
```

（相対パスが維持されるように確実にディレクトリ単位で移動してください。）

4. 概要を16.4で示すように、新しいバージョンのPostgreSQLをインストールしてください。

5. 必要に応じて新しいデータベースクラスタを作成してください。（アップグレードの場合はすでに存在している）特別なデータベースユーザアカウントでログインして、このコマンドを実行しなければならないことに注意してください。

```
/usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
```

6. 以前のpg_hba.confとpostgresql.confに加えた何らかの変更を戻してください。

7. 繰り返しになりますが、特別なデータベースユーザアカウントを使用して、データベースサーバを起動してください。

```
/usr/local/pgsql/bin/postgres -D /usr/local/pgsql/data
```

8. 最後に、バックアップからデータをリストアしてください。

```
/usr/local/pgsql/bin/psql -d postgres -f outputfile
```

新しいサーバを異なるディレクトリにインストールし、古いサーバと新しいサーバを別のポートで並行して実行させることで、停止時間を最小にすることができます。この場合、データを移行するために以下のようなコマンドを使用することができます。

```
pg_dumpall -p 5432 | psql -d postgres -p 5433
```

18.6.2. pg_upgradeを使用したアップグレード方法

pg_upgradeモジュールにより、PostgreSQLのあるバージョンから次のバージョンにインストレーションをその場で移行することができます。特に--linkオプションを使用することで、アップグレードは数分で行うことができます。これは、pg_dumpallと同様の工程を必要とします。例えば、initdbを実行し、サーバの起動／停止をおこないます。pg_upgradeドキュメントで必要な手順を説明します。

18.6.3. レプリケーション経由のアップグレード

論理レプリケーションを使って更新対象のバージョンのPostgreSQLをスタンバイサーバとして作成することもできます。論理レプリケーションが異なるメジャーバージョンのPostgreSQLの間でレプリケーションすることができるため、これが実現できます。スタンバイは同じコンピュータで作成することも異なるコンピュータで作成することもできます。(古いバージョンのPostgreSQLで実行している)マスタサーバと同期した後、マスタを切り替え、スタンバイをマスタにし、古いデータベースインスタンスを停止することができます。このようなスイッチオーバの結果、数秒の停止時間でアップグレードされます。

この方法によるアップグレードは、組み込みの論理レプリケーション機能か、あるいはpglogical、Slony、Londiste、Bucardoなどの外部の論理レプリケーションシステムを使うことで実施できます。

18.7. サーバのなりすまし防止

サーバが稼動中、悪意のあるユーザが通常のデータベースサーバに取って代わることはできません。しかし、サーバが停止している時、ローカルユーザに対し、独自のサーバを起動させることで正常なサーバになりすますことは可能です。なりすましたサーバで、クライアントから送信されたパスワードを読み取ることも問い合わせを読み取ることも可能です。しかし、PGDATAディレクトリの安全性はディレクトリの権限により維持されていますので、データを返すことはできません。誰もがデータベースサーバを起動させることができるため、なりすましは可能です。特殊な設定がなされていなければ、クライアントは無効なサーバであることを識別できません。

local接続に対してなりすましを防ぐ、ひとつの方法は、信頼できるローカルユーザのみに書き込み権限を付与したUnixドメインソケットディレクトリ(unix_socket_directories)を使用することです。これにより、悪意のあるユーザがそのディレクトリに独自のソケットを作成することを防ぐことができます。一部のアプリケーションがソケットファイルのために/tmpを参照し、なりすましに対して脆弱であるかもしれないと気にするならば、オペレーティングシステムの起動時に、再割り当てされたソケットファイルを指し示す/tmp/.s.PGSQL.5432というシンボリックリンクを作成してください。また、このシンボリックリンクが削除されることを防ぐために、/tmpを整理するスクリプトを変更する必要があるかもしれません。

local接続についての別の選択肢は、クライアントがrequirepeerを使用して、ソケットに接続しているサーバプロセスの必要な所有者を指定することです。

TCP接続のなりすましを防ぐためには、SSL証明書を使用してクライアントにサーバの証明書を確実に検査させるか、GSSAPI暗号化を使用します。(あるいはそれらが別々の接続上にあるなら、その両方を使います。)

SSLでなりすましを防ぐためには、サーバはhostssl接続(20.1)のみを受け付け、SSLキーと証明書ファイル(18.9)を持つ必要があります。TCPクライアントはsslmode=verify-caもしくはverify-fullを使用して接続し、また、適切なルート証明書ファイルをインストールしなければなりません(33.18.1)。

GSSAPIでなりすましを防ぐためには、サーバはhostgssenc接続(20.1)のみを受け付け、gss認証をその接続で使います。TCPクライアントはgssencmode=requireを使用して接続しなければなりません。

18.8. 暗号化オプション

PostgreSQLは、複数レベルの暗号化を備え、データベースサーバ自身、管理者の注意不足、安全ではないネットワークを原因とした漏洩からデータを柔軟に保護することができます。また、医療記録や金融取引など高セキュリティが求められるデータを格納する場合に暗号化が必要な場合もあります。

パスワードの暗号化

データベースユーザのパスワードは(password_encryptionによって決定される)ハッシュとして格納されます。ですので、ユーザに割り当てられているパスワードは管理者でも分かりません。SCRAM又はMD5暗号化がクライアント認証に使用されている場合、平文のパスワードはサーバ上に一時的にも存在することはありません。ネットワーク上に流れる前にクライアントが暗号化を行うからです。SCRAMは、インターネット標準で、PostgreSQL特有なMD5認証プロトコルよりセキュアであるため、より望ましいです。

特定の列に関する暗号化

pgcryptoモジュールにより、あるフィールドを暗号化して保存することができます。データの一部が極秘データであるような場合に有用です。クライアントが提供した復号化用のキーで、サーバ上のデータを復号化し、クライアントに返されます。

復号化されたデータと復号化用のキーは、復号処理中およびクライアントサーバ間の通信中サーバ上に存在している時間があります。このため、データベースサーバへのすべてのアクセス権限を持つユーザ(例えばシステム管理者)によって、データとキーが盗まれるわずかな時間があります。

データパーティションに関する暗号化

ストレージの暗号化は、ファイルシステムレベルまたはブロックレベルで行うことができます。Linuxファイルシステムの暗号化オプションには、eCryptfsとEncFSがあります。一方、FreeBSDではPEFSを使用します。ブロックレベルまたはフルディスクの暗号化オプションとして、Linuxにはdm-crypt + LUKS、FreeBSDにはGEOMモジュールのgeliとgbdeがあります。これにより、ファイルシステムパーティション全体をディスク上で暗号化することができます。他の多くのオペレーティングシステムは、Windowsを含め、この機能をサポートしています。

この機構により、ディスク装置やコンピュータ全体が盗まれた場合でも、ディスクから平文のデータが読み取られることを防止できます。ファイルシステムがマウントされている時は、この機構による保護は働きません。マウント時にはオペレーティングシステムが復号化したデータを提供するからです。しかし、ファイルシステムをマウントするためになんらかの方法で暗号化キーをオペレーティングシステムに渡さなければなりません。そのディスク装置をマウントするホストのどこかに暗号化キーを格納している場合もあります。

ネットワーク越しのデータ暗号化

SSL接続により、ネットワーク越しに送信されるデータ(パスワード、問い合わせ、結果のデータ)はすべて暗号化されます。pg_hba.confファイルを使用して、管理者はどのホストは暗号化しない接続を使用し

(host)、どのホストがSSLで暗号化された接続を必要とするか(hostssl)を指定することができます。また、SSL経由のサーバとの接続のみを使用するようにクライアントで指定することもできます。

GSSAPI暗号化接続は、問い合わせ及び返却されるデータを含めてネットワーク上に転送されるすべてのデータを暗号化します。(パスワードはネットワーク上に送信されません。) 管理者はpg_hba.confファイルを使ってどのホストが非暗号化接続を使うか(host)、どのホストがGSSAPI暗号化接続を要求するか(hostgssenc)を指定できます。クライアントはGSSAPI暗号化接続のみを使ってサーバに接続することも指定できます(gssencmode=require)。

StunnelやSSHを使用して暗号化転送を行うこともできます。

SSLホスト認証

クライアントとサーバの両方で証明書を互いに提供することができます。これには両方で追加の設定を行わなければなりませんが、これにより単なるパスワードの使用よりも強力な身元の検証を行うことができます。クライアントから送信されたパスワードを盗聴する偽装サーバからコンピュータを保護します。また、クライアントとサーバとの間にあるコンピュータがサーバになりすまし、クライアントとサーバ間で流れるデータを読み取り中継する、「中間者」攻撃から保護することもできます。

クライアントサイドの暗号化

サーバマシンのシステム管理者を信頼できない場合、クライアント側でデータを暗号化する必要があります。この場合、平文のデータはデータベースサーバ上に存在しません。データはサーバに送信される前にクライアント上で暗号化されます。また、使用する前にデータベースからの結果をクライアントで復号化しなければなりません。

18.9. SSLによる安全なTCP/IP接続

PostgreSQLは標準でSSL接続をサポートし、クライアント/サーバの通信がさらに安全になるよう暗号化します。そのためにはOpenSSLがクライアントとサーバシステムの両方にインストールされ、構築時にPostgreSQLにおけるそのサポートが有効になっている必要があります(第16章を参照してください)。

18.9.1. 基本的な設定

SSLサポートを有効にしてコンパイルされた場合、PostgreSQLサーバは、postgresql.confにおいてsslパラメータをonにすることで、SSLサポートを有効にして起動することができます。サーバは同じTCPポートで通常の接続とSSL接続の両方を待ち受け、クライアントとの接続にSSLを使用するかどうかを調停します。デフォルトでは、これはクライアント側の選択肢です。一部またはすべての接続でSSLの使用を必要とさせるためのサーバ側の設定方法に関しては20.1を参照してください。

SSLモードで起動するには、サーバ証明書と秘密鍵を含むファイルが存在していなければなりません。デフォルトでは、これらのファイルはserver.crtおよびserver.keyという名前で、それぞれがサーバのデータディレクトリに存在していることが想定されていますが、設定パラメータのssl_cert_fileとssl_key_fileによって他の名前、他の場所を指定することもできます。

Unixシステムでは、server.keyの権限は所有者以外からのアクセスを許可してはなりません。これはchmod 0600 server.keyというコマンドで実現できます。あるいは、このファイルの所有者をrootにして、グループ

に読み取りアクセス権を与える（つまり、パーミッションを0640にする）ということもできます。この設定は、証明書と鍵ファイルがオペレーティングシステムによって管理されるインストレーションのためのものです。PostgreSQLサーバを実行するユーザは、証明書と鍵ファイルにアクセス権のあるグループのメンバーにする必要があります。

データディレクトリがグループアクセスを許可している場合、証明書ファイルは上記のセキュリティ上の要求を満たすためにデータディレクトリ外に置く必要があるかも知れません。一般に、グループアクセスは権限を持たないユーザがデータベースをバックアップできるように有効化されます。この場合、バックアップソフトウェアは証明書を読むことができず、おそらくエラーとなるでしょう。

秘密鍵がパスフレーズで保護されている場合、サーバはパスフレーズの入力を促し、入力されるまでは起動しません。パスフレーズを使用すると、サーバを再起動せずにサーバのSSL設定を変更する機能はデフォルトで無効になりますが、ssl_passphrase_command_supports_reloadを参照してください。さらに、パスフレーズで保護された秘密鍵は、Windowsではまったく使用できません。

server.crtの最初の証明書は、サーバ証明書になり、秘密鍵とマッチしなければなりません。「中間」認証局の証明書をファイルに追加することもできます。これにより、ルートと中間証明書がv3_ca拡張により作成されていることが前提になりますが、中間証明書をクライアントに保存する必要が無くなります。（これにより、CAの証明書の基本制約がtrueに設定されます。）これは、中間証明書の有効期限の扱いをより簡単にします。

server.crtにルート証明書を追加する必要はありません。代わりに、クライアントはサーバ証明書のチェーンのルート証明書を持っていなければなりません。

18.9.2. OpenSSLの設定

PostgreSQLはシステム全体のOpenSSL設定ファイルを読み込みます。デフォルトでは、このファイルはopenssl.cnfという名前で、openssl version -dが報告するディレクトリに設置されます。このデフォルトはOPENSSL_CONF環境変数を設定することによって希望の名前に置き換えることができます。

OpenSSLは様々な強度を持つ、多様な暗号と認証アルゴリズムをサポートしています。暗号のリストはOpenSSLの設定ファイルで指定できますが、使用するデータベースサーバ用にpostgresql.confのssl_ciphersで指定することができます。

> **注記**
>
> NULL-SHAあるいはNULL-MD5暗号を使って暗号化のオーバーヘッドがない認証を行うことができます。しかし、中間者がクライアントとサーバの間のコミュニケーションを読んで転送することができます。また、認証のオーバーヘッドに比べると暗号化のオーバーヘッドは最小限です。これらの理由から、NULL暗号はお勧めできません。

18.9.3. クライアント証明書の使用

信頼できる証明書をクライアントに要求するには、信頼するルート認証局（CA）の証明書をデータディレクトリ内のファイルに置き、postgresql.confのssl_ca_fileパラメータを設定し、認証オプションclientcert=verify-caまたはclientcert=verify-fullをpg_hba.confの適切なhostssl行に追加しま

す。そうすると、SSL接続の開始時にクライアントへ証明書が要求されます。(クライアント上での証明書の設定方法については33.18を参照してください。)

clientcert=verify-ca指定付きのhostsslエントリでは、サーバは、クライアントの証明書が信頼する認証局のいずれかにより署名されていることを検証します。clientcert=verify-fullが指定されていると、サーバは証明書チェーンを検証するだけでなく、ユーザ名あるいはユーザ名のマッピングが提供された証明書のcn (Common Name)に一致しているかどうかも検証します。cert認証メソッドが使われている場合は認証チェーンの検証は常に行われることに注意してください。(20.12参照。)

既存のルート証明書に連鎖する中間証明書は、クライアントに保存することを避けたい場合にssl_ca_fileに含めることができます(ルート証明書と中間証明書がv3_ca拡張で作成されている場合)。ssl_crl_fileパラメータが設定されている場合、証明書失効リスト(CRL)項目も検査されます。

認証オプションclientcertはすべての認証方式について利用可能ですが、pg_hba.confのhostsslとして指定された行でのみ有効です。clientcertが指定されていない、またはno-verifyと設定されている場合でも、認証局のリストが設定されていれば、サーバはその認証局に対してクライアント証明書の検証を行いますが、クライアント証明書を提示することを要求しません。

ユーザに対してログイン中に証明書を提供するように強制する二つのアプローチがあります。

最初のアプローチはpg_hba.confのhostsslにcert認証メソッドを使うことです。そうすることによりSSL接続によるセキュリティが提供されるとともに、証明書自身が認証に使われます。詳細は20.12をご覧ください。(cert認証メソッドを使う際には明示的にclientcertオプションを指定する必要はありません。) この場合証明書が提供するcn (Common Name)がユーザ名あるいは適用可能なマッピングに対して検証されます。

二番目のアプローチは、clientcert認証オプションにverify-caあるいはverify-fullを設定することによってhostsslエントリの認証メソッドにクライアント証明書の検証を組み合わせることです。前者のオプションは証明書が有効であることだけを強制し、後者は更に証明書のcn (Common Name)がユーザ名あるいは適用可能なマッピングと一致することを強制します。

18.9.4. サーバにおけるSSL関連ファイルの利用

表 18.2にて、サーバにおけるSSLの設定に関連するファイルをまとめます。(表示されているファイル名はデフォルトまたは一般的な名前です。異なる名前を個別に設定することもできます。)

表18.2 SSLサーバファイルの使用方法

ファイル	内容	影響
ssl_cert_file ($PGDATA/server.crt)	サーバ証明書	サーバの身元を示すためにクライアントに送信します
ssl_key_file ($PGDATA/server.key)	サーバの秘密鍵	サーバ証明書が所有者によって送られたことを証明します。証明書所有者が信頼できることを意味しません。
ssl_ca_file	信頼できる認証局	信頼する認証局により署名されたクライアント証明書か検査します。
ssl_crl_file	認証局により失効された証明書	クライアント証明書はこの一覧にあってはいけません。

サーバは、サーバ起動時及びサーバ設定がリロードされるたびに、これらのファイルを読み取ります。Windowsシステム上では新しいクライアント接続のために新しいバックエンドプロセスが生成されるたびに再読み込みされます。

サーバ起動時にこれらのファイルのエラーが検出された場合、サーバは起動を拒否します。ただし、設定のリロード中にエラーが検出された場合、ファイルは無視され、古いSSL設定が引き続き使用されます。Windowsシステム上ではバックエンドの開始時にこれらのファイルのエラーが検出された場合、そのバックエンドはSSL接続を確立出来ません。これらのすべてのケースでは、エラー状態がサーバログに記録されます。

18.9.5. 証明書の作成

365日有効なサーバ用の自己署名証明書を簡単に作るためには下記のOpenSSLコマンドを実行してください(dbhost.yourdomain.comをサーバのホスト名に置き換えてください)。

```
openssl req -new -x509 -days 365 -nodes -text -out server.crt \
  -keyout server.key -subj "/CN=dbhost.yourdomain.com"
```

続けて以下も実行します。

```
chmod og-rwx server.key
```

サーバの秘密鍵および証明書を作成するための詳しい方法についてはOpenSSLの文書を参照してください。

テストには自己署名証明書を使用できますが、運用時は認証局(CA)(通常は事業全体のCA)により署名された証明書を使用する必要があります。

クライアントが身元を検証できるサーバ証明書を作成するには、まず最初に証明書署名要求(CSR)と公開／秘密鍵を作成します。

```
openssl req -new -nodes -text -out root.csr \
  -keyout root.key -subj "/CN=root.yourdomain.com"
chmod og-rwx root.key
```

その後、鍵を使用して署名要求に署名しルート証明書を作成します(Linux上のデフォルトのOpenSSL設定ファイルの場所を使用)。

```
openssl x509 -req -in root.csr -text -days 3650 \
  -extfile /etc/ssl/openssl.cnf -extensions v3_ca \
  -signkey root.key -out root.crt
```

最後に、新しいルート証明書によって署名されるサーバ証明書を作成します。

```
openssl req -new -nodes -text -out server.csr \
  -keyout server.key -subj "/CN=dbhost.yourdomain.com"
chmod og-rwx server.key
```

```
openssl x509 -req -in server.csr -text -days 365 \
  -CA root.crt -CAkey root.key -CAcreateserial \
  -out server.crt
```

server.crtとserver.keyをサーバに格納し、root.crtをクライアントに格納します。クライアントはサーバのリーフ証明書が信頼されたルート証明書によって署名されたことを確認できます。root.keyは将来の証明書の作成に使用するために、オフラインで保存する必要があります。

中間証明書が含まれる信頼の連鎖を作成することも可能です。

```
# root
openssl req -new -nodes -text -out root.csr \
  -keyout root.key -subj "/CN=root.yourdomain.com"
chmod og-rwx root.key
openssl x509 -req -in root.csr -text -days 3650 \
  -extfile /etc/ssl/openssl.cnf -extensions v3_ca \
  -signkey root.key -out root.crt

# intermediate
openssl req -new -nodes -text -out intermediate.csr \
  -keyout intermediate.key -subj "/CN=intermediate.yourdomain.com"
chmod og-rwx intermediate.key
openssl x509 -req -in intermediate.csr -text -days 1825 \
  -extfile /etc/ssl/openssl.cnf -extensions v3_ca \
  -CA root.crt -CAkey root.key -CAcreateserial \
  -out intermediate.crt

# leaf
openssl req -new -nodes -text -out server.csr \
  -keyout server.key -subj "/CN=dbhost.yourdomain.com"
chmod og-rwx server.key
openssl x509 -req -in server.csr -text -days 365 \
  -CA intermediate.crt -CAkey intermediate.key -CAcreateserial \
  -out server.crt
```

server.crtとintermediate.crtは証明書ファイルに束ねて連結し、サーバに格納する必要があります。server.keyもまたサーバに格納される必要があります。サーバのリーフ証明書が信頼されたルート証明書にリンクされた一連の証明書によって署名されていることをクライアントが確認できるように、root.crtをクライアントに格納する必要があります。root.keyとintermediate.keyは将来の証明書を作成に使用するためにオフラインで格納する必要があります。

18.10. GSSAPIによる安全なTCP/IP接続

セキュリティを強化する目的でクライアント／サーバの通信を暗号化するためにPostgreSQLもまたGSSAPIの利用を直接サポートしています。このサポートにはGSSAPIの実装（MIT krb5など）がクライアントとサーバ

システムの両方にインストールされていて、PostgreSQLの構築時にそのサポートが有効になっていること（第16章参照）が必要です。

18.10.1. 基本的な設定

PostgreSQLサーバは通常の接続とGSSAPIによる暗号化接続の両方を同じTCPポートで待ち受け、接続しようとするクライアントとGSSAPIによる暗号化（そして認証）を使うかどうかを交渉します。デフォルトではこの決定はクライアントに任されます（これは攻撃者によってダウングレードできることを意味します）。サーバが一部あるいはすべての接続でGSSAPIを使うことを要求する設定に関しては20.1をご覧ください。

この交渉の挙動を設定すること以外にはGSSAPI暗号化では、GSSAPI認証に必要な設定はこれ以上ありません。（設定のより詳細に関しては20.6をご覧ください。）

18.11. SSHトンネルを使った安全なTCP/IP接続

クライアントとPostgreSQLサーバ間のネットワーク接続を暗号化するためにSSHを使うことができます。正しく行えば、SSL機能がクライアントになくても、これで十分に安全なネットワーク接続を行うことができます。

まずSSHサーバがPostgreSQLサーバと同じマシン上で正しく起動していて、sshを使ってログインできるユーザが存在することを確かめてください。リモートサーバへ安全なトンネルを確立することができます。安全なトンネルは、ローカルポートをリッスンし、すべてのトラフィックをリモートマシン上のポートに転送します。リモートポートに送信されたトラフィックは、localhostアドレスまたは必要に応じて別のバインドアドレスに到達することができ、ローカルマシンからのトラフィックとは表示されません。次のコマンドは、クライアントマシンからリモートマシンfoo.comへの安全なトンネルを作成します。

```
ssh -L 63333:localhost:5432 joe@foo.com
```

-L引数の1番目の数字（63333）はトンネルのローカル側のポート番号で、未使用のポートを選択することが可能です。（IANAは49152から65535までのポートを私的使用のため予約しています。）この後の名前かIPアドレスは、接続先のリモート側のバインドアドレス（デフォルトはlocalhost）です。2番目の数字（5432）は、トンネルのリモート側のサーバが使用しているポート番号です。このトンネルを使ってデータベースサーバに接続するためには、ローカルマシンのポート63333に接続します。

```
psql -h localhost -p 63333 postgres
```

データベースサーバにとっては、ユーザがホストfoo.com上のユーザjoeであり、localhostバインドアドレスに接続しているように見え、そのバインドアドレスに対するそのユーザの接続向けに設定された認証手続きが使用されます。実際、SSHサーバとPostgreSQLサーバとの間は暗号化されていないため、サーバはこの接続がSSLで暗号化されているとみなさないことに注意してください。それらが、同じマシン上にあるため、セキュリティ上の危険性が増すことはありません。

トンネルの確立が成功するためには、sshを使用して端末セッションを作成したのと同様に、joe@foo.comユーザがsshを通して接続することが許可されていなければいけません。

以下に示すようにポートフォワードを設定することができます。

```
ssh -L 63333:foo.com:5432 joe@foo.com
```

しかしそうすると、データベースサーバはそのfoo.comバインドアドレスから接続が来たように判断し、デフォルトの設定であるlisten_addresses = 'localhost'では開かれません。通常これは好ましいことではありません。

どこかのログインホスト経由でデータベースサーバに「跳躍」しなければならない場合、以下のようにすることが可能です。

```
ssh -L 63333:db.foo.com:5432 joe@shell.foo.com
```

shell.foo.comからdb.foo.comへのこのような接続はSSHトンネルで暗号化されません。SSHはいろいろな方法でネットワークが制約されているとき、かなりの数の設定可能性を提供しています。詳細はSSHの文書を参照してください。

> **ヒント**
>
> ここで説明してきたものと似た概念の手続きを使用して、安全なトンネルを提供可能なアプリケーションが他にもいくつか存在します。

18.12. WindowsにおけるEvent Logの登録

Windowsの OSのevent logライブラリに登録するには、以下のコマンドを発行します:

```
regsvr32 pgsql_library_directory/pgevent.dll
```

このコマンドは、PostgreSQLというデフォルトのイベントソース名で、イベントビューアが使用するレジストリエントリを作成します。

異なるイベントソース名（event_source参照）を指定するには、/nおよび/iオプションを使ってください:

```
regsvr32 /n /i:event_source_name pgsql_library_directory/pgevent.dll
```

OSからevent logライブラリを削除するには、以下のコマンドを発行します:

```
regsvr32 /u [/i:event_source_name] pgsql_library_directory/pgevent.dll
```

> **注記**
>
> データベースサーバにおけるイベントロギングを有効にするには、eventlogを含むようにpostgresql.confのlog_destinationを変更してください。

第19章 サーバの設定

データベースシステムの動作に影響を与える数多くのパラメータがあります。この章の最初の節で、どのように設定パラメータを操作するのかについて説明します。引き続く節で、それぞれのパラメータの詳細を説明します。

19.1. パラメータの設定

19.1.1. パラメータ名とその値

全てのパラメータの名前は大文字と小文字を区別しません。それぞれのパラメータは、論理値、整数、浮動小数点、文字列、またはenum(列挙型)の5つの型のいずれかの値を取ります。型はパラメータをセットするための記法を定義します。

- 論理型: 値はon、off、true、false、yes、no、1、0(すべて大文字小文字の区別なし)、あるいは、曖昧でなければ、これらの先頭から数文字を省略形として使うこともできます。

- 文字列型: 一般に、単一引用符の中に値を入れます。単一引用符を値に含める場合は単一引用符を2つ続けます。なお、値が単純な数字や識別子である場合は、通常は引用符を省略できます。(使用する場所によっては、SQLキーワードと一致する値に引用符が必要になることがあります。)

- 数値型(整数型と浮動小数点型): 数値パラメータには通常の整数と浮動小数点型が使用できます。パラメータが整数型なら、小数値はもっとも近い整数に丸められます。加えて整数型パラメータは16進数入力(0xで始まります)と8進数入力(0で始まります)を受け付けます。しかし、これらの形式では小数点以下は使えません。1000の位取りの区切り文字は使わないでください。16進数入力を除き引用符は必要ありません。

- 単位付きの数値: 数値型のパラメータによっては暗黙的な単位を持つことがあります。メモリの量や時間について記述するからです。単位はバイト、キロバイト、ブロック(通常8キロバイト)、ミリ秒、秒、分などです。修飾無しの数値によるこれらの設定においては、pg_settings.unit からデフォルト値が採用されます。使い勝手を考えて、たとえば'120 ms'のように単位を明示的に指定することもできます。この場合は、実際の単位に変換が行われます。なお、この機能を使う場合は、引用符付きの文字列として値を指定しなければならないことに注意してください。単位の名称は大文字小文字を区別します。また、数値と単位の間に空白があっても構いません。

  - 有効なメモリの単位はB (バイト) kB (キロバイト)、MB (メガバイト)、GB (ギガバイト), TB (テラバイト)です。メモリ単位の乗数は1024です。1000ではありません。

  - 有効な時間の単位は us (マイクロ秒)、ms (ミリ秒)、s (秒)、min (分)、h (時間)、d (日数) です。

  単位に添えて小数点以下が指定された場合、より小さな単位が存在すれば、値はその小さな単位の積に丸められます。たとえば、30.1 GBは32319628902 Bではなく30822 MBに変換されます。整数型のパラメータでは、単位変換の後で最終的な整数への丸めが行われます。

- 列挙型: 列挙型のパラメータは文字列パラメータと同じように記述します。ただ、使用できる文字列の種類が決まっているだけです。使用できる文字列は pg_settings.enumvals で定義されています。列挙型の値は大文字小文字を区別しません。

19.1.2. 設定ファイルによるパラメータ操作

これらのパラメータを設定する最も基本的な方法は、postgresql.confファイルを編集することで、これは通常 data ディレクトリに格納されています。デフォルトのコピーはデータベースクラスタディレクトリが初期化されるときそこにインストールされます。このファイルがどういったものかの例を示します。

```
# This is a comment
log_connections = yes
log_destination = 'syslog'
search_path = '"$user", public'
shared_buffers = 128MB
```

1つの行毎に1つのパラメータが指定されます。名前と値の間の等号は省略可能です。(引用符付きのパラメータ値内を除き)空白は特に意味を持たず、何もない行は無視されます。ハッシュ記号(#)はその行の後の表記がコメントであることを意味します。単純でない識別子、または数値でないパラメータ値は単一引用符で括られなければなりません。パラメータ値の中に単一引用符を埋め込むには、引用符を2つ(推奨)もしくはバックスラッシュ-引用符を使います。ファイル中、同じパラメータに対して複数のエントリが指定されている場合は、最後のエントリ以外は無視されます。

この方法によりクラスタに対してデフォルト値が設定されます。上書きされない限り、アクティブなセッションが見るのはこの値です。次節以降では、管理者やユーザがこれらのデフォルト値を上書きする方法を説明します。

設定ファイルは、メインサーバプロセスがSIGHUPシグナルを受け取るたびに再読み込みされます。このシグナルを手っ取り早く送信するには、コマンドラインからpg_ctl reloadを実行するか、SQL関数のpg_reload_conf()を呼び出します。メインサーバプロセスは同時にこのシグナルを、現存のセッションが同様に新しい値を入手できるように、全ての現在実行しているサーバプロセスに伝播します(これは現在実行中のクライアントコマンドの処理を完了してから行われます)。他の手段として、直接単一のサーバプロセスにシグナルを送ることも可能です。一部のパラメータはサーバの起動時のみ設定されまするので、設定ファイル中のそれらのエントリの変更はすべて、サーバが再起動されるまで無視されます。設定ファイル内で無効なパラメータが設定された場合も、同じように(ログには残りますが)SIGHUP 処理中は無視されます。

postgresql.confに加え、PostgreSQLのデータディレクトリには postgresql.auto.confというファイルがあります。このファイルは postgresql.conf と同じフォーマットですが、手動ではなく自動で編集されることを意図しています。このファイルは ALTER SYSTEM コマンドを使った設定値を保存します。このファイルはpostgresql.conf が読み込まれるときはいつでも自動的に読み込まれ、同じように設定が反映されます。postgresql.auto.confは、postgresql.confの設定を上書きします。

外部ツールもpostgresql.auto.confを変更するかも知れません。ALTER SYSTEMが変更を上書きする可能性があるので、サーバが稼働中は外部ツールによる変更は推奨されません。そのようなツールは、単に新しい設定を最後に追加するか、重複した設定あるいはコメント(ALTER SYSTEMが行います)を削除することを選択するかも知れません。

システムビューのpg_file_settingsは、設定ファイルへの変更を前もってテストしたい場合や、SIGHUPシグナルで望み通りの効果がなかった場合に問題を調査する際に役立ちます。

19.1.3. SQLを通じたパラメータ操作

PostgreSQLは3つのSQLコマンドでデフォルト値を設定します。すでに説明したALTER SYSTEMコマンドは、SQLによってグローバルな設定値を変更する方法を提供します; postgresql.confを編集するのと等価です。これに加え、データベース単位あるいはロール単位で設定するためのコマンドがあります:

- ALTER DATABASEコマンドはデータベース単位でグローバルな設定値を上書きします。
- ALTER ROLEコマンドはグローバルと、データベース単位の両方をユーザ固有の設定値で上書きします。

ALTER DATABASEとALTER ROLEによる設定値は新しくデータベースセッションを開始した時にのみ適用されます。これらのコマンドは設定ファイルやサーバへのコマンド引数による設定値を上書きし、セッションの以後の状態に適用します。なお、一部の設定はサーバを起動した後では変更できず、これらのコマンドを使っては設定できません(以下に記述するコマンドでも同じことが言えます)。

クライアントがデータベースに接続すると、PostgreSQLでは更に2つのSQL(そして同等の関数)を使ってセッションローカルの設定変更を行うことができます。

- SHOWコマンドを使ってすべてのパラメータの現在の値を調べることができます。対応する関数はcurrent_setting(setting_name text)です。
- SETでセッション内でローカルに変更できるパラメータの値を変更することができます。対応する関数はset_config(setting_name, new_value, is_local)です。

更にシステムビューのpg_settingsを使ってセッションローカルな値を参照したり変更することができます。

- このビューを問い合わせるのは、SHOW ALLを使うのと同じですが、更に詳細な情報を提供します。フィルター条件を指定したり他のリレーションと結合ができるので、より柔軟です。
- このビューに対してUPDATEを実行する、具体的にはsetting列を更新することは、SETコマンドを実行するのと同等です。たとえば、

  ```
  SET configuration_parameter TO DEFAULT;
  ```

  は

  ```
  UPDATE pg_settings SET setting = reset_val WHERE name = 'configuration_parameter';
  ```

  と同じです。

19.1.4. シェルによるパラメータ操作

グローバルなデフォルト値を設定したりデータベース、ロール単位で上書きを行えるだけでなく、シェル機能を使ってPostgreSQLに設定値を渡すことができます。サーバもlibpqクライアントライブラリもシェル経由でパラメータ値を受けとることができます。

- サーバ起動時に、-cコマンドラインパラメータを使ってパラメータ設定値をpostgresに渡すことができます。たとえば、

```
postgres -c log_connections=yes -c log_destination='syslog'
```

このようにして渡された設定値は、postgresql.confやALTER SYSTEMによる設定を上書きします。したがってサーバを再起動しない限りこれらの設定値をグローバルに変更することはできません。

- libpqを使ってクライアントセッションを開始するときにPGOPTIONS環境変数を使って設定値を指定できます。このようにして渡された設定値はセッションのデフォルトとなりますが、他のセッションには影響を与えません。歴史的な理由により、PGOPTIONSの形式はpostgresを起動するときのものと似ています。たとえば、-cフラグを指定しなければならない点です。

```
env PGOPTIONS="-c geqo=off -c statement_timeout=5min" psql
```

他のクライアントやライブラリではそれぞれ固有の方法でシェルなどを経由して、SQLコマンドを直接使わずにセッションの設定を変更することができるかもしれません。

19.1.5. 設定ファイルの内容の管理

PostgreSQLは複雑なpostgresql.confファイルを複数の小さなファイルに分割する複数の方法を提供しています。これは、とりわけお互いに関連しているものの設定が同じではない複数のサーバを管理する際に有用です。

パラメータ設定に加え、postgresql.confファイルに*include*ディレクティブを入れることができます。このようにすると、別のファイルがあたかも設定ファイルのその場所に挿入されているかのごとく読み込まれ、処理されるように指定されます。この機能により、設定ファイルを物理的に異なる複数のパーツに分解することができます。Includeディレクティブは単に次のような形式になります。

```
include 'ファイル名'
```

ファイル名が絶対パスでない場合、参照する設定ファイルを含むディレクトリからの相対パスであると受け取られます。Includeコマンドは入れ子にすることができます。

include_if_existsディレクティブもあります。これは参照ファイルが存在しないか、または読み込むことができない場合の動作を除き、includeディレクティブと同一の動作をします。通常のincludeはこれをエラーと解釈しますが、include_if_existsはただ単にメッセージをログ出力し、そして参照している設定ファイルの処理を続けます。

includeする設定ファイルを含むディレクトリ全体を指定するinclude_dirディレクティブを、postgresql.confファイルに含めることもできます。このような感じです。

```
include_dir 'ディレクトリ名'
```

絶対パスではないディレクトリ名はその設定ファイルがあるディレクトリへの相対パスと見なされます。指定したディレクトリの中で、ディレクトリではないファイルで末尾が.confで終わるファイルだけがincludeされます。また、文字. で開始するファイル名は一部のプラットフォームでは隠しファイルとされるので、間違いを防止するため無視されます。includeされるディレクトリにある複数ファイルはファイル名順に処理されます(ファイル名はCロケール規則で順序付けされます。つまり、文字より数字、小文字より大文字が先になります)。

includeされるファイルもしくはディレクトリは、大きな単一のpostgresql.confファイルを使う代わりに、データベース設定の一部分を論理的に分離するために使用することが可能です。異なるメモリー容量を持つ二つのデータベースサーバを所有する会社を考えてみてください。例えばログ取得のように、二つが共有する設定の要素があると思われます。しかし、サーバ上のメモリに関連したパラメータは二つの間では異なります。更に、サーバ特有のカスタマイズも存在することがあります。この状況に対処する一つの方法として、そのサイトに対するカスタマイズされた設定の変更を三つのファイルにすることです。それらをincludeするためにはpostgresql.confファイルの最後に以下を追加します。

```
include 'shared.conf'
include 'memory.conf'
include 'server.conf'
```

全てのシステムは同一のshared.confを所有する様になるでしょう。特定のメモリー容量を所有するそれぞれのサーバは同じmemory.confを共有できます。RAMが8GBのすべてのサーバには共通のmemory.confを1つ使い、16GBのサーバ群には別のものを使う、ということもできるでしょう。そして最後のserver.confには、本当にサーバ固有となる設定情報を記載します。

別の方法として、設定ファイルディレクトリを作成し、この情報をそこのファイルに格納することができます。たとえば、conf.dディレクトリをpostgresql.confの最後で参照するようにできます。

```
include_dir 'conf.d'
```

そして、conf.dの中のファイルを以下のような名前にすることができます。

```
00shared.conf
01memory.conf
02server.conf
```

この命名規則により、これらのファイルが読み込まれる順序が明確になります。サーバが設定を読み込んでいるときに各パラメータについて最後にあった設定だけが使用されるので、このことは重要です。この例では、conf.d/02server.confでされた指定はconf.d/01memory.confの設定値よりも優先します。

代わりに以下の方法を使って、ファイルにわかりやすい名前をつけることもできます。

```
00shared.conf
01memory-8GB.conf
02server-foo.conf
```

こういった工夫で、設定ファイルのバリエーションに対して固有の名前を付与することができます。また、バージョン管理リポジトリのリポジトリに複数のサーバの設定ファイルを置く場合に生じる曖昧さを排除することができます。(データベース設定ファイルをバージョン管理することは、これもまた検討に値するやり方です)。

19.2. ファイルの場所

すでに説明したpostgresql.confファイルに加え、PostgreSQLは、クライアント認証の管理を行うために、他の2つの手作業で編集される設定ファイルを使用します(これらの使用方法は第20章で説明します)。全て

の3つの設定ファイルは、デフォルトではデータベースクラスタのdataディレクトリに格納されます。本節で説明するパラメータにより、設定ファイルを他の場所に置くことが可能になります。(そのようにすると管理がしやすくなります。とりわけ、設定ファイルを分けて保存することで、設定ファイルの適切なバックアップを確実に行うことがしばしば容易になります。)

data_directory (string)

データ格納に使用するディレクトリを指定します。このパラメータはサーバ起動時のみ設定可能です

config_file (string)

メインサーバ設定ファイルを指定します(通例postgresql.confと呼ばれます)。このパラメータはpostgresコマンドライン上でのみ設定可能です。

hba_file (string)

ホストベース認証(HBA)用のファイルを指定します(通例pg_hba.confと呼ばれます)。このパラメータはサーバ起動時のみ設定可能です。

ident_file (string)

ユーザ名マッピングの設定ファイルを指定します(通例pg_ident.confと呼ばれます)。このパラメータはサーバ起動時のみ設定可能です。20.2もご覧ください。

external_pid_file (string)

サーバ管理プログラムで使用するためにサーバが作成する、追加のプロセス識別子(PID)ファイルの名前を指定します。このパラメータはサーバ起動時のみ設定可能です。

デフォルトのインストールでは、上記のいかなるパラメータも明示的に設定されません。その代わり、data ディレクトリは-Dコマンドラインオプション、またはPGDATA環境変数で指定され、設定ファイル全てはその data ディレクトリ内に格納されます。

dataディレクトリ以外の場所に設定ファイルを格納したいのであれば、postgresの-Dコマンドラインオプション、またはPGDATA環境変数で設定ファイルの場所を指し示し、そしてdataディレクトリが実際どこに存在するのかを示すため、postgresql.confの(もしくはコマンドライン上で)data_directoryパラメータを設定しなければなりません。data_directoryは、設定ファイルの場所ではなく、data ディレクトリの位置に関して、-DおよびPGDATAを上書きすることに注意してください。

必要に応じて、パラメータconfig_file、hba_file、ident_fileを使用し、設定ファイルの名前と場所を個別に指定することができます。config_fileはpostgresコマンドラインによってのみ指定されますが、その他は主設定ファイル内で設定できます。全ての3つのパラメータとdata_directoryが明示的に設定されていれば、-DまたはPGDATAを指定する必要はありません。

これらのパラメータのどれを設定する場合でも、相対パスは、postgresが起動されるディレクトリから見た相対パスとして解釈されます。

19.3. 接続と認証

19.3.1. 接続設定

listen_addresses (string)

クライアントアプリケーションからの接続をサーバが監視する TCP/IP アドレスを指定します。この値は、ホスト名をコンマで区切ったリスト、そして/もしくは、数値によるIPアドレスです。*という特別なエントリは利用可能な全てのIPインタフェースに対応します。エントリ0.0.0.0は全てのIPv4アドレスの監視を、そしてエントリ::は全てのIPv6アドレスの監視を許容します。リストが空の場合、サーバはいかなるIPインタフェースも全く監視しないで、Unixドメインソケットのみを使用して接続が行われます。デフォルトの値はlocalhostで、ローカルなTCP/IP「loopback」接続のみ許可します。クライアント認証 (第20章)は誰がサーバにアクセス可能かをきめ細かく制御するのに対し、listen_addressesはどのインタフェースが接続を試みるかを制御します。これにより、安全でないネットワークインタフェース上において繰り返して行われる悪意のある接続要求の防止に役立ちます。このパラメータはサーバ起動時のみ設定可能です。

port (integer)

サーバが監視するTCPポートで、デフォルトは 5432です。サーバが監視する全てのIPアドレスに対し、同じポート番号が使用されることを覚えておいてください。このパラメータはサーバ起動時のみ設定可能です。

max_connections (integer)

データベースサーバに同時接続する最大数を決定します。デフォルトは典型的に100接続ですが、カーネルの設定が（initdbの過程で）それをサポートしていない場合、もっと少なくなることがあります。このパラメータはサーバ起動時のみに設定可能です。

スタンバイサーバを運用している場合、このパラメータはマスターサーバでの設定と同じ、もしくはより高い値に設定しなければなりません。そうしないと問い合わせがスタンバイサーバ内で受け入れられません。

superuser_reserved_connections (integer)

PostgreSQLのスーパーユーザによる接続のために予約されている接続「開口部（スロット）」の数を決定します。最大、max_connectionsの数までの接続を同時に有効にすることができます。有効な接続数がmax_connectionsからsuperuser_reserved_connectionsを差し引いた数以上のときは、新規接続はスーパーユーザのみが許可され、新たなレプリケーション接続は受け入れられません。

デフォルトの値は3接続です。この値は max_connectionsより小さくなくてはなりません。このパラメータはサーバ起動時のみ設定可能です。

unix_socket_directories (string)

サーバがクライアントアプリケーションからの接続要求を監視するUnixドメインソケットのディレクトリを指定します。複数ソケットはコンマで区切られた複数ディレクトリをリストすることで作成できます。項目間の空白文字は無視されます。ディレクトリ名に空白文字もしくはコンマを使用する必要がある場合、ディレクトリ名を二重引用符で括ります。空の値はいかなるUnixドメインソケットも監視しないようにしま

す。この場合、TCP/IPソケットのみがサーバとの接続に使用されます。デフォルト値は通常/tmpですが、ビルド時に変更できます。Windowsではデフォルトは空文字で、これはつまりUnixドメインソケットがデフォルトでは作成されないことを意味します。このパラメータはサーバ起動時のみ設定可能です。

.s.PGSQL.nnnnという名前のソケットファイル（nnnnはポート番号）のほかに、.s.PGSQL.nnnn.lockという通常ファイルがそれぞれのunix_socket_directoriesディレクトリの中に作成されます。いずれのファイルも手作業で削除してはいけません。

unix_socket_group (string)

Unixドメインソケット（複数も）を所有するグループを設定します（ソケットを所有するユーザは常にサーバを起動するユーザです）。unix_socket_permissionsパラメータとの組合せで、Unixドメインソケット接続の追加的アクセス管理機構として使うことができます。デフォルトでは空文字列で、サーバユーザのデフォルトグループを使用します。このパラメータはサーバ起動時のみ設定可能です。

このパラメータはWindowsではサポートされていません。すべての設定は無視されます。

unix_socket_permissions (integer)

Unixドメインソケット（複数も）のアクセスパーミッションを設定します。Unixドメインソケットは通常のUnixファイルシステムパーミッション設定の一式を使用します。パラメータ値は、chmodおよびumaskシステムコールが受け付ける数値形式での指定を想定しています。（通常使われる8進数形式を使用するのであれば、0（ゼロ）で始まらなければなりません。）

デフォルトのパーミッションは、誰でも接続できる0777になっています。変更するならば0770（ユーザとグループのみです。unix_socket_groupも参照してください）や0700（ユーザのみ）が適切です。（Unixドメインソケットでは書き込み権限だけが問題になるため、読み込みや実行のパーミッションを設定または解除する意味はありません。）

このアクセス制御機構は 第20章 で記述されたものとは別個のものです。

このパラメータはサーバ起動時のみ設定可能です。

このパラメータはSolaris 10の時点でのSolarisなど、ソケットのパーミッションを完全に無視するシステムでは無関係です。こうしたシステムでは、許可したいユーザだけが検索パーミッションを持つディレクトリをunix_socket_directoriesで指すようにすることによって同じような効果を得ることができます。

bonjour (boolean)

Bonjourによりサーバの存在を公表することを可能にします。デフォルトはoffです。このパラメータはサーバ起動時のみ設定可能です。

bonjour_name (string)

Bonjourサービス名を指定します。このパラメータが空文字列''（デフォルトです）に設定されていると、コンピュータ名が使用されます。サーバがBonjourサポート付でコンパイルでされていない場合は無視されます。このオプションはサーバ起動時のみに設定可能です。

tcp_keepalives_idle (integer)

クライアントとのやり取りがなくなった後、オペレーティングシステムがTCPのkeepaliveパケットをクライアントに送信するまでの時間を指定します。この値が単位なしで指定された場合は、秒単位であるとみなします。0（デフォルトです）の場合はオペレーティングシステムのデフォルト値を使用します。このパ

ラメータはTCP_KEEPIDLEまたは同等のソケットオプションをサポートするシステムと、Windowsでのみサポートされます。その他のシステムではゼロでなければなりません。Unixドメインソケット経由で接続されたセッションでは、このパラメータは無視され、常にゼロとして読み取られます。

> **注記**
>
> Windowsでは0を指定するとこのパラメータを2時間に設定します。なぜなら、Windowsはシステムデフォルト値を読む手段を提供していないからです。

tcp_keepalives_interval (integer)

TCPのkeepaliveメッセージに対してクライアントから応答がない場合に、再送を行うまでの時間を指定します。この値が単位なしで指定された場合は、秒単位であるとみなします。0（デフォルトです）の場合はシステムのデフォルト値を使用します。このパラメータはTCP_KEEPINTVLまたは同等のソケットオプションをサポートするシステムと、Windowsでのみサポートされます。その他のシステムではゼロでなければなりません。Unixドメインソケット経由で接続されたセッションでは、このパラメータは無視され、常にゼロとして読み取られます。

> **注記**
>
> Windowsでは0を指定すると、このパラメータを1秒に設定します。なぜなら、Windowsはシステムデフォルト値を読む手段を提供していないからです。

tcp_keepalives_count (integer)

サーバのクライアントへの接続が切れたと判断されるまでのTCP keepaliveメッセージの数を指定します。0（デフォルトです）の場合はオペレーティングシステムのデフォルト値を使用します。このパラメータはTCP_KEEPCNTまたは同等のソケットオプションをサポートするシステムでのみサポートされます。その他のシステムではゼロでなければなりません。Unixドメインソケット経由で接続されたセッションでは、このパラメータは無視され、常にゼロとして読み取られます。

> **注記**
>
> このパラメータはWindowsではサポートされておらず、ゼロでなければなりません。

tcp_user_timeout (integer)

未確認のデータが残ったままの接続が強制的に閉じられるまでの時間を指定します。この値が単位なしで指定された場合は、ミリ秒単位であるとみなします。0（デフォルトです）の場合はオペレーティングシステムのデフォルト値を使用します。このパラメータは、TCP_USER_TIMEOUTをサポートするシステムでのみ使用できます。他のシステムでは、0にする必要があります。UNIXドメインソケットで接続しているセッションではこのパラメータは無視され、常に0として扱われます。

> **注記**
>
> このパラメータはWindowsではサポートされておらず、ゼロでなければなりません。

19.3.2. 認証

authentication_timeout (integer)

クライアント認証を完了するまでの最大時間です。もし、この時間内に自称クライアントが認証プロトコルを完了しない場合、サーバは接続を閉じます。これはハングしたクライアントが接続を永久に占有することを防ぎます。この値が単位なしで指定された場合は、秒単位であるとみなします。デフォルトは1分(1m)です。このパラメータはpostgresql.confファイル、またはサーバのコマンドラインでのみ設定可能です。

password_encryption (enum)

CREATE ROLEあるいはALTER ROLEでパスワードを設定する際に、このパラメータはパスワードを暗号化するアルゴリズムを指定します。デフォルト値はmd5で、パスワードをMD5ハッシュとして格納します(onもmd5の別名として受付けます)。このパラメータをscram-sha-256とすると、SCRAM-SHA-256でパスワードを暗号化します。

古いクライアントはSCRAM認証機構をサポートしていない可能性があり、したがってSCRAM-SHA-256による暗号化は動作しないかもしれないことに注意してください。さらなる詳細については20.5をご覧ください。

krb_server_keyfile (string)

Kerberosサーバーキーファイルの場所を設定します。詳細は20.6をご覧ください。このパラメータはpostgresql.confファイル、またはサーバのコマンドラインでのみ設定可能です。

krb_caseins_users (boolean)

GSSAPIユーザ名を大文字小文字の区別なく取り扱うかどうかを設定します。デフォルトはoff(大文字小文字を区別する)です。このパラメータはpostgresql.confファイル、またはサーバのコマンドラインでのみ設定可能です。

db_user_namespace (boolean)

このパラメータはデータベース毎のユーザ名を可能にします。デフォルトはオフです。このパラメータはpostgresql.confファイル、またはサーバのコマンドラインでのみ設定可能です。

これがオンの場合、username@dbnameの様にしてユーザを作成しなければなりません。usernameが接続中のクライアントより渡された時、@およびデータベース名がユーザ名に付加され、そのデータベース特有のユーザ名をサーバが見に行きます。SQL環境下で@を含む名前のユーザを作成する場合、そのユーザ名は引用符で括られなければならないことに注意してください。

このパラメータを有効にしていても通常の広域ユーザを作成することができます。クライアントにユーザ名を指定する時に、たとえばjoe@のように単に@を付け加えてください。@はサーバがユーザ名を検索する以前に取り去られます。

db_user_namespaceはクライアントとサーバのユーザ名の表示を区別することができます。認証検査は常にサーバのユーザ名で行われるので、認証方式はクライアントのではなくサーバのユーザ名で構成されなければなりません。md5では、クライアントおよびサーバの両方でユーザ名をソルトとして使用するので、md5をdb_user_namespaceと一緒に使用することはできません。

> **注記**
>
> この機能は完全な解決方法が見つかるまでの一時的なものです。完全な解決方法が見つかったら、このオプションは削除される予定です。

19.3.3. SSL

SSLの設定に関するさらなる情報については18.9をご覧ください。

ssl (boolean)

SSLによる接続を有効にします。このパラメータは、postgresql.confファイルか、サーバのコマンドラインでのみ設定可能です。デフォルトはoffです。

ssl_ca_file (string)

SSLサーバ認証局(CA)が入っているファイル名を設定します。相対パスの場合は、データディレクトリからの相対パスになります。このパラメータは、postgresql.confファイルか、サーバのコマンドラインでのみ設定可能です。デフォルトは空で、この場合CAファイルは読み込まれず、クライアントのサーバ検証は行われません。

ssl_cert_file (string)

SSLサーバ証明書が入っているファイル名を設定します。相対パスの場合は、データディレクトリからの相対パスになります。このパラメータは、postgresql.confファイルか、サーバのコマンドラインでのみ設定可能です。デフォルトは server.crt です。

ssl_crl_file (string)

SSLサーバ証明書失効リスト(CRL)が入っているファイル名を設定します。相対パスの場合は、データディレクトリからの相対パスになります。このパラメータは、postgresql.confファイルか、サーバのコマンドラインでのみ設定可能です。デフォルトは空で、この場合CRLファイルは読み込まれません。

ssl_key_file (string)

SSLサーバの秘密鍵が入っているファイル名を設定します。相対パスの場合は、データディレクトリからの相対パスになります。このパラメータは、postgresql.confファイルか、サーバのコマンドラインでのみ設定可能です。デフォルトは server.key です。

ssl_ciphers (string)

SSL接続で使うことのできるSSL暗号スイートのリストを指定します。設定構文と使用可能な値のリストについてはOpenSSLパッケージの ciphersマニュアルをご覧ください。TLSバージョン1.2あるいはそれ以下のバージョンを使用する接続のみが影響を受けます。今の所、TLSバージョン1.3接続で使用される暗号の選択を制御する設定はありません。デフォルト値はHIGH:MEDIUM:+3DES:!aNULLです。特別なセキュリティ要件でなければ通常これが適当です。

このパラメータは、postgresql.confファイルか、サーバのコマンドラインでのみ設定可能です。

デフォルト値の説明:

HIGH

HIGHグループ(たとえばAES, Camellia, 3DES)を使用する暗号スイート

MEDIUM

MEDIUMグループ(たとえば RC4, SEED)を使用する暗号スイート

+3DES

OpenSSLのHIGHに対するデフォルトの並び順には問題があります。3DESがAES128より高いとしているからです。3DESはAES128よりもセキュアではなく、またずっと遅いので、これは間違っています。+3DESではそれを他のすべてのHIGHとMEDIUM暗号よりも後に位置づけます。

!aNULL

認証を行わない無名暗号スイートを無効にします。そういった暗号スイートは中間者攻撃に対して脆弱で、使用すべきではありません。

OpenSSLのバージョンにより、利用可能な暗号スイートの詳細は異なります。`openssl ciphers -v 'HIGH:MEDIUM:+3DES:!aNULL'` コマンドを使って現在インストールされているOpenSSLのバージョンに関する詳細情報を得てください。ここで得られるリストは、サーバキータイプにより実行時にフィルターされることに注意してください。

ssl_prefer_server_ciphers (boolean)

サーバのSSL暗号設定をクライアントに優先して使うかどうかを指定します。このパラメータは、postgresql.confファイルか、サーバのコマンドラインでのみ設定可能です。デフォルトはonです。

古いバージョンのPostgreSQLにはこの設定がなく、常にクライアントの設定を使います。この設定は、主に古いバージョンとの互換性のために設けられています。通常サーバの設定に従うほうが良いです。大抵の場合、サーバはより適切に設定されているからです。

ssl_ecdh_curve (string)

ECDHキー交換で使われる曲線の名前を指定します。接続するすべてのクライアントがこの設定をサポートしている必要があります。サーバの楕円曲線キーで使用されるのと同じ曲線である必要はありません。このパラメータは、postgresql.confファイルか、サーバのコマンドラインでのみ設定可能です。デフォルト値はprime256v1です。

OpenSSLはよく使われる曲線に名前を付けています。prime256v1 (NIST P-256), secp384r1 (NIST P-384), secp521r1 (NIST P-521). 利用できる曲線の完全なリストは`openssl ecparam -list_curves`で得られます。ただし、TLSではこのすべてが利用できるわけではありません。

ssl_min_protocol_version (enum)

使用するSSL/TLSプロトコルバージョンの最小値を設定します。今の所使用できる値はTLSv1、TLSv1.1、TLSv1.2、TLSv1.3です。古いバージョンのOpenSSLライブラリはすべての値をサポートしません。サポートしていない値が設定されるとエラーが発生します。TLS 1.0より前のプロトコルバージョン、すなわちSSLバージョン2あるいは3は常に無効となります。

デフォルトはTLSv1.2で、本稿執筆時点では業界のベストプラクティスを満たしています。

ssl_max_protocol_version (enum)

使用するSSL/TLSプロトコルバージョンの最大値を設定します。使用できる値はssl_min_protocol_versionと、すべてのプロトコルバージョンを許可する空文字です。デフォルトはすべてのプロトコルバージョンを許可する設定です。最大プロトコルバージョンの設定は主にテスト、あるいは新しいプロトコルを使った時にコンポーネントのどこかに問題がある時に有用です。

ssl_dh_params_file (string)

いわゆる短命DH系SSL暗号で使用するディフィー・ヘルマンパラメータを格納するファイル名を指定します。デフォルトは空で、この場合はコンパイル時に決められたデフォルトのDHパラメータが使用されます。攻撃者が、よく知られたコンパイル時設定のDHパラメータを解読しようとしている場合には、カスタムDHパラメータを使うことでその危険性を低減できます。openssl dhparam -out dhparams.pem 2048を使って、独自のDHパラメータファイルを作ることができます。

このパラメータは、postgresql.confファイルか、サーバのコマンドラインでのみ設定可能です。

ssl_passphrase_command (string)

秘密鍵などのSSLファイルを復号する際に、パスフレーズの入手が必要な時に起動される外部コマンドを設定します。デフォルトではこのパラメータは空文字で、組み込みのプロンプト機構が使用されます。

このコマンドは、パスフレーズを標準出力に書き出し、コード0で終了しなければなりません。パラメータの値の%pはプロンプト文字列に置き換えられます。(%を使いたい場合は%%としてください。) プロンプト文字列はおそらく空白文字を含むので、適切に引用符付けするように注意してください。出力の最後に一個の改行があれば、削除されます。

このコマンドは実際にはパスフレーズ用にユーザにプロンプトを表示する必要はありません。ファイルからパスフレーズが読めるなら、キーチェーン機構やその他から取得します。選択された仕組みが適切にセキュアかどうかを確認するのはユーザ次第です。

このパラメータはpostgresql.confファイル内、またはサーバのコマンドラインのみで設定可能です。

ssl_passphrase_command_supports_reload (boolean)

このパラメータは、キーにパスフレーズが必要な場合、設定ファイルの再読み込み中にssl_passphrase_commandで設定されたパスフレーズコマンドも呼び出されるかどうかを設定します。このパラメータがoff(デフォルト)なら、ssl_passphrase_commandは再読込の際に無視され、パスフレーズが必要な場合、SSL設定は再読込されません。この設定は、サーバ実行中は存在しないかもしれないTTYがプロンプトに必要なコマンドに適しています。たとえばパスフレーズがファイルから読み込める場合には、この設定をonにするのが適切です。

このパラメータはpostgresql.confファイル内、またはサーバのコマンドラインのみで設定可能です。

19.4. 資源の消費

19.4.1. メモリ

shared_buffers (integer)

データベースサーバが共有メモリバッファのために使用するメモリ量を設定します。デフォルトは一般的に128メガバイト(128MB)です。しかし、稼働中のカーネルの設定がこの値をサポートしていない場合、より少なくなることがあります(initdbの過程で決定されます)。この設定は最低限128キロバイトなければなりません。しかし、良い性能を引き出すためには、最小値よりかなり高い値の設定が通例必要です。この値が単位なしで指定された場合は、ブロック単位であるとみなします。すなわち、BLCKSZバイト、一般的には8kBです。(BLCKSZがデフォルト値と異なる場合、この最小値も異なる値になります。) このパラメータはサーバ起動時にのみ設定可能です。

1GB以上のRAMを載せた専用データベースサーバを使用している場合、shared_buffersに対する妥当な初期値はシステムメモリの25%です。shared_buffersをこれよりも大きな値に設定することが有効なワークロードもあります。しかし、PostgreSQLはオペレーティングシステムキャッシュにも依存するため、shared_buffersにRAMの40%以上を割り当てても、それより小さい値の時より動作が良くなる見込みはありません。shared_buffersをより大きく設定する場合は、大抵max_wal_sizeも合わせて増やす必要があります。これは、新規または変更された多量のデータを書き出す処理をより長い時間に渡って分散させるためです。

1GB未満のRAMのシステムでは、オペレーティングシステムに十分な余裕を残すために、RAMに対してより小さい割合を設定することが適切です。

huge_pages (enum)

主共有メモリ領域に対してhuge pageを要求するかどうかを管理します。可能な値はtry (デフォルト)、on、offです。huge_pagesをtryに設定すると、サーバはhuge pageの要求を試み、失敗したらデフォルトに戻します。onにすると、要求に失敗した場合にサーバの起動ができなくなることになります。offならhuge pageの要求は行いません。

今のところこの機能はLinuxとWindowsでのみサポートされています。他のシステムではtryと設定しても無視されます。

huge pageを使うと、ページテーブルが小さくなり、メモリ管理に使用されるCPU時間が少なくなり、性能が向上します。詳細は、18.4.5を見てください。

huge pageはWindowsではlarge pageとして知られています。それを使用するには、PostgreSQLを実行するWindowsユーザアカウントにメモリ中のロックページ権限を与える必要があります。ユーザにメモリ中のロックページ権限を与えるには、Windowsのグループポリシーツール(gpedit.msc)を利用できます。Windowsサービスとしてではなく、スタンドアロンプロセスとしてデータベースサーバをコマンドプロンプトで起動するには、コマンドプロンプトを管理者として実行するか、ユーザアクセス管理(UAC)を無効にしておかなければなりません。UACが有効だと、通常のコマンドプロンプトは起動時にユーザのメモリ中のロックページ権限を剥奪します。

この設定は主共有メモリ領域にのみ影響することに注意してください。Linux、FreeBSD、Illumosのようなオペレーティングシステムでは、PostgreSQLからの明示的な要求なしにhuge page(「super」 pageあるいは「large」pageとしての知られています)が通常のメモリ獲得の際に使用できます。Linuxでは、

これは「transparent huge pages」(THP)と呼ばれています。この機能は、あるLinuxバージョンのあるユーザにおいてPostgreSQLの性能低下をもたらすことが知られています。ですから、この機能の利用は(huge_pagesの明示的な利用と違って)今の所推奨されていません。

temp_buffers (integer)

それぞれのデータベースセッションが使用する一時バッファの最大メモリ量を設定します。一時バッファは、一時テーブルにアクセスする時にのみ使用されるセッションローカルのバッファです。この値が単位なしで指定された場合は、ブロック単位であるとみなします。すなわち、BLCKSZバイト、一般的には8kBです。デフォルトは8メガバイト(8MB)です。(BLCKSZが8kBでなければ、それに比例して増減します。) 設定はそれぞれのセッション内で変更できますが、そのセッション内で一時テーブルが最初に使用されるまでになります。それより後に値の変更を試みても、そのセッションでは効果がありません。

セッションは、temp_buffersを上限として、必要に応じて一時バッファを確保します。多くの一時バッファを実際に必要としないセッションで大きな値を設定するコストとは、temp_buffersの増分毎に、1つのバッファ記述子、約64バイトだけです。しかし、バッファが実際に使用されると、それに対して追加の8192バイト(汎用的に言えばBLCKSZバイト)が消費されます。

max_prepared_transactions (integer)

同時に「プリペアド」状態にできるトランザクションの最大数を設定します(PREPARE TRANSACTIONを参照してください)。このパラメータをゼロ(これがデフォルトです)に設定すると、プリペアドトランザクション機能が無効になります。このパラメータはサーバ起動時にのみ設定可能です。

プリペアドトランザクションの使用を意図しないのであれば、このパラメータはプリペアドトランザクションが偶然に作成されないようゼロに設定すべきです。プリペアドトランザクションを使用する場合、全てのセッションがプリペアドトランザクションを保留できるように、max_prepared_transactionsを少なくともmax_connectionsと同じ大きさに設定するのが良いでしょう。

スタンバイサーバを運用している場合、このパラメータはマスターサーバ上の設定よりも同等かもしくはより高水準に設定しなければなりません。そうしないと問い合わせがスタンバイサーバ内で受け入れられません。

work_mem (integer)

一時ディスクファイルに書き込むようになる前に、問い合わせ操作(たとえば並べ替えとハッシュテーブル操作)が使用する基本的な最大のメモリ容量を指定します。この値が単位なしで指定された場合は、キロバイト単位であるとみなします。デフォルト値は4メガバイト(4MB)です。複雑な問い合わせの場合、いくつかの並び替えもしくはハッシュ操作が並行して実行されることに注意してください。一時ファイルへの書き込み開始の前に、それぞれの操作にこの値が指定するのと同じメモリ容量の使用が許容されます。さらに、いくつかの実行中のセッションはこれらの動作を同時に行います。したがって、使用されるメモリの合計は、work_memの数倍になります。値を選択する時には、この事実に留意することが必要です。並び替え操作はORDER BY、DISTINCT、およびマージ結合に対して使われます。ハッシュテーブルはハッシュ結合、ハッシュに基づいた集約、およびIN副問い合わせのハッシュに基づいた処理で使用されます。

一般的に、ハッシュに基づく操作はソートに基づく操作よりも利用可能なメモリに敏感です。ハッシュテーブルが利用可能なメモリはwork_memにhash_mem_multiplierを掛けて計算します。これにより、ハッシュに基づく操作では通常のwork_memに基づく量を超える量のメモリが使用される可能性があります。

hash_mem_multiplier (floating point)

ハッシュに基づく操作が利用できる最大のメモリ量を計算するために使用します。最終的な制限はwork_memにhash_mem_multiplierを掛けて決定されます。デフォルト値は1.0で、ソートに基づく操作と同じように単にwork_memが最大となります。

問い合わせ操作によって日常的にメモリ不足になるような環境、とりわけ単にwork_memを増やしたことによってメモリ逼迫(メモリ逼迫が典型的には間欠的なメモリ不足エラーの発生の形で起こる)が起きる場合にはhash_mem_multiplierを増やすことを考慮してください。1.5あるいは2.0が色々なワークロードが混在している場合には効果的かも知れません。より大きな2.0から8.0、あるいはそれ以上の設定はwork_memがすでに40MB以上に増やしてあるような環境で効果的かも知れません。

maintenance_work_mem (integer)

VACUUM、CREATE INDEX、およびALTER TABLE ADD FOREIGN KEYの様な保守操作で使用されるメモリの最大容量を指定します。この値が単位なしで指定された場合は、キロバイト単位であるとみなします。デフォルト値は64メガバイト(64MB)です。1つのデータベースセッションでは、一度に1つしか上記操作はできませんし、通常インストレーションでこうした操作が同時に非常に多く発生することはありませんので、これをwork_memよりもかなり多めの値にしても安全です。大きい値を設定することでvacuum処理と、ダンプしたデータベースのリストア性能が向上します。

自動バキュームが稼動すると、最大でこのメモリのautovacuum_max_workers倍が配分されるので、デフォルトの値をあまり高く設定しないよう注意してください。別の設定項目autovacuum_work_memで制御するのが良いかもしれません。

autovacuum_work_mem (integer)

個々の自動バキュームワーカプロセスが使用する最大のメモリ量を指定します。この値が単位なしで指定された場合は、キロバイト単位であるとみなします。デフォルトは-1で、maintenance_work_memが代わりに使われる設定になります。別の文脈で実行されるVACUUMにはこの設定は影響しません。

logical_decoding_work_mem (integer)

デコードされた更新がローカルディスクに書かれる前にロジカルデコーディングが使用する最大のメモリ量を指定します。これにより、ロジカルストリーミングレプリケーションの接続が使用する最大メモリが制限されます。デフォルトは64メガバイト(64MB)です。個々のレプリケーション接続がここで指定した単一のバッファだけを使用し、インストールは通常たくさんの接続を並行して使わないので(max_wal_sendersで制限されます)、この値をwork_memよりもずっと大きくしても安全で、それによってデコードされた更新がディスクに書かれる量が削減されます。

max_stack_depth (integer)

サーバの実行スタックの最大安全深度を指定します。このパラメータの理想的な設定はカーネルにより強要される実際のスタック容量の(ulimit -sもしくはそれと同等の機能で設定された)限界から、1メガバイト程度の安全余裕度を差し引いたものです。この安全余裕度は、サーバがすべてのルーチンではスタック深度を検査をせず、再帰を行う可能性のある重要なルーチンでのみ検査をするために必要となるものです。この値が単位なしで指定された場合は、キロバイト単位であるとみなします。デフォルト設定は2メガバイト(2MB)で、かなり控え目、かつクラッシュの危険がなさそうな設定です。しかし、複雑な関数の実行を許容するには小さ過ぎるかも知れません。スーパーユーザのみがこの設定を変更することができます。

max_stack_depthを実際のカーネルの制限よりも高い値に設定した場合、暴走した再帰関数により、個々のバックエンドプロセスがクラッシュするかもしれません。PostgreSQLがカーネルの制限を決定することができるプラットフォームでは、この変数を危険な値に設定させません。しかし、すべてのプラットフォームがこの情報を提供できるわけではありません。このため、値を選ぶ時には注意が必要です。

shared_memory_type (enum)

PostgreSQLの共有バッファおよび他の共有データを保持する主共有メモリ領域のためにサーバが使用すべき共有メモリの実装を指定します。可能な値はmmap（mmapを使って獲得した無名共有メモリ）、sysv（shmgetを使って獲得したSystem V共有メモリ）、windows（Windows共有メモリ）です。すべての値がすべてのプラットフォームでサポートされているわけではありません。サポートされている最初のオプションがそのプラットフォームのデフォルトです。どのプラットフォームでもデフォルトになっていないsysvオプションの利用は一般に推奨されません。通常、デフォルトではないカーネルの設定が大きなアロケーションでは必要になるからです。（18.4.1参照。）

dynamic_shared_memory_type (enum)

サーバが使う動的共有メモリの実装を指定します。可能な値はposix (shm_openで獲得するPOSIX共有メモリ)、sysv(shmgetで獲得するSystem V共有メモリ)、windows (Windows共有メモリ)、mmap(データディレクトリ内のメモリマップファイルを使ってシミュレートする共有メモリ)です。すべての値がすべてのプラットフォームでサポートされているわけではありません。そのプラットフォームでの推奨実装がデフォルトになります。どのプラットフォームでもデフォルトになっていないmmapは、オペレーティングシステムが変更されたページをディスクに継続的に書き込み、I/O負荷を増加させるので一般的には利用が推奨されていません。しかし、デバッグ目的のためにpg_dynshmemディレクトリがRAMディスク上にある場合や、他の共有メモリ機能が使えない場合は有用かもしれません。

19.4.2. ディスク

temp_file_limit (integer)

あるプロセスが一時ファイルとして使用できるディスクの最大容量を設定します。例えば、ソートやハッシュの一時ファイルであったり、カーソルを保持する格納ファイルです。この制限値を超えようとするトランザクションはキャンセルされます。この値が単位なしで指定された場合は、キロバイト単位であるとみなします。-1（デフォルトです）の場合は制限がありません。この設定はスーパーユーザのみ変更可能です。

この設定により、ある PostgreSQL セッションによって使用される一時ファイルの合計の容量が常に制約されることになります。なお、問い合わせの実行において暗黙的に使用される一時ファイルとは異なり、一時テーブルとして明示的に使用されるディスク容量は、この制限には含まれません。

19.4.3. カーネル資源使用

max_files_per_process (integer)

それぞれのサーバ子プロセスが同時にオープンできるファイル数の最大値をセットします。デフォルトは1000ファイルです。もしもカーネルがプロセス毎の安全制限を強要している場合、この設定を気にかける必要はありません。しかし、いくつかのプラットフォーム（特にほとんどのBSDシステム）では、もし多

くのプロセス全てがそれだけ多くのファイルを開くことを試みたとした場合、実際にサポートできるファイル数より多くのファイルを開くことを許しています。もしも「Too many open files」エラーが発生した場合、この設定を削減してみてください。このパラメータはサーバ起動時にのみ設定可能です。

19.4.4. コストに基づくVacuum遅延

VACUUM および ANALYZE コマンドの実行中、実行される各種I/O操作の予測コストを追跡し続ける内部カウンタをシステムが保守します。累積されたコストが（vacuum_cost_limitで指定された）限度に達すると、操作を実行しているプロセスはvacuum_cost_delayで指定されたちょっとの間スリープします。その後、カウンタをリセットし、実行を継続します。

この機能の目的は、同時に実行されているデータベースの活動に対するこれらコマンドによるI/Oへの影響を、管理者が軽減できるようにすることです。VACUUM および ANALYZEの様な保守用コマンドが即座に終了することが重要ではない事態が数多くあります。しかし、他のデータベースの操作を行うに当たって、これらのコマンドがシステムの能力に多大な阻害を与えないことは通常とても重要です。コストに基づいたvacuum遅延はこれを実現するための方法を管理者に提供します。

手動で実行したVACUUMコマンドについては、デフォルトでこの機能は無効になっています。有効にするには、vacuum_cost_delay変数をゼロでない値に設定します。

vacuum_cost_delay (floating point)

コストの限度を越えた場合、プロセスがスリープする時間の長さです。この値が単位なしで指定された場合は、ミリ秒単位であるとみなします。デフォルトの値は0で、コストに基づいたvacuum遅延機能を無効にします。正の整数はコストに基づいたvacuumを有効にします。

コストに基づいたバキューム処理を使用する場合、vacuum_cost_delayの適切な値は通常かなり小さくなり、おそらく1ミリ秒以下になります。バキュームによるリソース消費の調整は、他のバキュームのコストパラメータを変更して行うことが最善です。vacuum_cost_delayを1ミリ秒以下に設定することは可能ですが、そうした遅延は古いプラットフォームでは正確には計測されないかも知れません。そうしたプラットフォームでは、VACUUMのリソース消費制限を1ミリ秒のときに得られる値以上にするには、他のVACUUMコストパラメータの変更が必要となるでしょう。とは言うものの、使用するプラットフォームで常に計測できる範囲でvacuum_cost_delayをできるだけ小さくするようにしてください。大きな遅延は助けになりません。

vacuum_cost_page_hit (integer)

共有バッファキャッシュの中のバッファにvacuumを掛ける予測コストです。バッファプールのロック、共有ハッシュテーブルの検索、およびページ内容走査のコストを示します。デフォルトの値は1です。

vacuum_cost_page_miss (integer)

ディスクから読み込まれなければならないバッファにvacuumを掛ける予測コストです。これが示すものは、バッファプールロックの試み、共有ハッシュテーブルの参照、ディスクから目的ブロックの読み込み、そしてその内容走査です。デフォルトの値は10です。

vacuum_cost_page_dirty (integer)

vacuumが、先だって掃除したブロックを変更するのに必要な見積コストです。ダーティブロックを再度ディスクに吐き出すのに必要な余分なI/Oを表します。デフォルトの値は20です。

vacuum_cost_limit (integer)

vacuumを掛けるプロセスをスリープさせることになる累計されたコストです。デフォルトの値は200です。

> **注記**
>
> 重要なロックを保有し可能なかぎり早急に完了しなければならないある種の操作があります。コストに基づいたvacuum遅延はこの様な操作では起こりません。したがって、コストの累計が指定された限度をかなり高く越える可能性があります。このような場合無駄な長い遅延を防止するため、実際の遅延はvacuum_cost_delay * 4 を上限として、以下のように計算されます。vacuum_cost_delay * accumulated_balance / vacuum_cost_limit

19.4.5. バックグラウンドライタ

バックグラウンドライタと呼ばれる個別のサーバプロセスがあり、その機能は（新規または更新された）「ダーティ」な共有バッファの書き込みを行うことです。ユーザの問い合わせを処理するサーバプロセスが、書き込みが起きるまで滅多に待つ必要がない、あるいは決して待つ必要がないように、共有バッファの書き込みを行います。しかし、バックグラウンドライタは正味の全体的I/O負荷の増加を引き起こします。その理由は、繰り返しダーティ化されるページは、バックグラウンドライタを使わなければチェックポイント間隔で一度だけ書き出されれば十分なのに対し、バックグラウンドライタは同じ間隔内で何度もダーティ化されると、それを複数回書き出すかもしれないからです。本節で説明する各パラメータは、サイト独自の必要に応じて動作を調整することに使用できます。

bgwriter_delay (integer)

バックグラウンドライタの動作周期間の遅延を指定します。それぞれの周期でライタは、（以下のパラメータで管理される）一部のダーティバッファの書き込みを行います。そしてbgwriter_delayの長さスリープした後、これを繰りかえします。しかし、バッファプールにダーティバッファが存在しない場合、bgwriter_delayに係わらずより長くスリープします。この値が単位なしで指定された場合は、ミリ秒単位であるとみなします。デフォルトの値は200ミリ秒（200ms）です。多くのシステムで、スリープ遅延の実精度は10ミリ秒です。bgwriter_delayの値の設定を10の倍数としない場合、次に大きい10の倍数に設定した結果と同一になるかもしれないことを覚えておいてください。このパラメータはpostgresql.confファイル、または、サーバのコマンドラインで設定可能です。

bgwriter_lru_maxpages (integer)

それぞれの周期で、この数以上のバッファはバックグラウンドライタにより書き込まれません。ゼロに設定することでバックグラウンド書き込みは無効になります。（分離し、そして専用の補助プロセスにより管理されるチェックポイントは影響を受けません。）デフォルト値は100バッファです。このパラメータはpostgresql.confファイル内、または、サーバのコマンドラインでのみで設定可能です。

bgwriter_lru_multiplier (floating point)

各周期で書き出されるダーティバッファ数は、最近の周期でサーバプロセスが必要とした新しいバッファ数を基にします。次の周期で必要となるバッファ数を推定するために、最近必要とされた平均がbgwriter_lru_multiplierと掛け合わせられます。ダーティバッファの書き出しは、同数の整理済み、再利用可能なバッファが利用できるようになるまで行われます。（しかし1周期

にbgwriter_lru_maxpagesを越えるバッファ数を書き出しません。) したがって、1.0と設定することは、必要と予想されるバッファ数の書き込みについて「必要なときに必要なだけ」というポリシーを表します。より大きな値は突発的な要求に対する多少の緩衝材を提供します。より小さな値はサーバプロセスでなされる書き込みを意図的に残します。デフォルトは2.0です。このパラメータはpostgresql.confファイル、または、サーバのコマンドラインでのみで設定可能です。

bgwriter_flush_after (integer)

バックグラウンドライタがこの値より多く書く度に、OSが記憶装置に書き込むことを強制しようとします。このことにより、カーネルのページキャッシュが持つダーティデータの量を一定量に制限し、チェックポイントの最後にfsyncが実行される際、あるいはOSがバックグラウンドでデータを大きな塊で書き出す際に性能の急激な低下を招く可能性を減らします。多くの場合これによってトランザクションの遅延が大幅に少なくなりますが、あるケース、特にワークロードがshared_buffersよりも大きく、OSのページキャッシュよりも小さい時には性能が低下するかもしれません。この設定が無効なプラットフォームがあります。この値が単位なしで指定された場合は、ブロック単位であるとみなします。すなわち、BLCKSZバイト、一般的には8kBです。有効な設定値は、この強制書き込み機能が無効になる0から、2MBまでです。デフォルト値は、Linuxでは512kBで、それ以外は0です。(BLCKSZが8kBでなければ、この設定のデフォルト値と最大値がBLCKSZに比例して変更されます。) このパラメータはpostgresql.confファイル、または、サーバのコマンドラインでのみで設定可能です。

bgwriter_lru_maxpagesおよびbgwriter_lru_multiplierの値がより少ないと、バックグラウンドライタで引き起こされる追加のI/O負荷を軽減しますが、サーバプロセスが自分自身で行わなければならない書き込みが増加することになり、会話型問い合わせを遅らせることになります。

19.4.6. 非同期動作

effective_io_concurrency (integer)

PostgreSQLが同時実行可能であると想定する同時ディスクI/O操作の数を設定します。この値を大きくすると、あらゆる個別のPostgreSQLセッションが並行して開始を試みるI/O操作の数が増加します。設定可能な範囲は1から1000まで、または非同期I/Oリクエストの発行を無効にするゼロです。現在、この設定はビットマップヒープスキャンのみに影響します。

磁気ディスクドライブにおいては、データベースに使用されるRAID 0ストライプ、RAID 1ミラーを構成する個々のドライブ数から始めると良いでしょう。(RAID 5ではパリティ用のドライブを数に含めません) しかし、同時実行セッションで発行される複数の問い合わせでデータベースが頻繁にビジーとなる場合、小さめの値で十分ディスクアレイがビジーになるかもしれません。ディスクをビジーにするのに必要な値より大きな値を設定しても、余計なCPUオーバーヘッドを発生させるだけです。SSDやそれ以外のメモリーベースの記憶装置は、多くの同時リクエストをこなすことができるので、最適な値は数百になるかもしれません。

非同期I/Oは実質的にposix_fadvise関数に依存します。これは一部のオペレーティングシステムには存在しません。この関数が存在しない場合、この値をゼロ以外に設定するとエラーとなります。一部のオペレーティングシステム (例えばSolaris) では存在するけれども、実際何も行わないものもあります。

デフォルトは、サポートされているシステムでは1、そうでなければ0です。この値は、テーブルスペースパラメータの同じ名前のパラメータを設定することで、特定のテーブルスペース内のテーブルに対して上書きできます。(ALTER TABLESPACEを参照ください)。

maintenance_io_concurrency (integer)

effective_io_concurrencyと似ていますが、多くのクライアントセッションのために行われる保守作業で適用されるところが異なります。

サポートしているシステムではデフォルトは10で、それ以外は0です。この値は同じ名前のテーブル空間パラメータを使って、特定のテーブル空間にあるテーブルのために置き換えることができます。(ALTER TABLESPACEを参照してください。)

max_worker_processes (integer)

システムがサポートするバックグラウンドプロセスの最大数を指定します。このパラメータはサーバ起動時にのみ設定できます。デフォルトは8です。

スタンバイサーバを起動しているときは、このパラメータを、マスタサーバの設定値と同じかそれ以上にしなければなりません。さもなければ、スタンバイサーバで問い合わせの実行ができなくなります。

この値を変更する際は、max_parallel_workers、max_parallel_maintenance_workersとmax_parallel_workers_per_gatherを変更することも考慮してください。

max_parallel_workers_per_gather (integer)

一つのGatherまたはGather Mergeノードに対して起動できるワーカー数の最大値を設定します。パラレルワーカーは、max_parallel_workersで上限が決まるmax_worker_processesで確立されたプロセスのプールから取得されます。実行時には、要求された数のワーカーは取得できないかもしれないことに注意してください。そうなると、実行プランは期待していたよりも少ない数のワーカーで実行されることになり、効率は悪化するかもしれません。デフォルト値は2です。この設定値を0にすると、パラレルクエリの実行は行われません。

パラレルクエリの実行により、パラレルクエリではない場合に比べて非常に多くのリソースが使用されるかもしれないことに注意してください。これは、個々のワーカープロセスは完全に別個のプロセスであり、システムに対してユーザセッションが追加されたのと大体同じくらいの影響があるからです。この設定値を選択する際には、他のリソースの消費量を制御する他の設定値、たとえばwork_memを設定するときと同様に、この点を考慮しておく必要があります。work_memのような設定値によるリソース制限は、個々のワーカーに対して個別に適用されます。つまり、ひとつのプロセスに対するよりも、すべてのプロセスの全体のリソース消費はずっと多いかもしれないということです。たとえば、あるパラレルクエリが4つのワーカーを使っているとすると、ワーカーを使わない場合に比べて、最大5倍のCPU時間、メモリ、I/Oバンド幅、その他を使うかもしれません。

パラレルクエリに関する更なる情報については、第15章をご覧ください。

max_parallel_maintenance_workers (integer)

単一のユーティリティコマンドで使用されるパラレルワーカーの最大数を設定します。今の所、パラレルワーカーの利用をサポートしているパラレルユーティリティコマンドは、CREATE INDEXがB-treeインデックスを構築するときと、FULLオプションなしのVACUUMです。パラレルワーカーは、max_worker_processesで確立したプロセスのプールから取得され、max_parallel_workersによって制限されます。要求したワーカー数は、実行時に実際には利用可能でないかも知れないことに注意してください。この場合は、ユーティリティ操作は期待したよりも少ない数のワーカーにより実行されます。デフォルト値は2です。0に設定すると、ユーティリティコマンドはパラレルワーカーを使用しません。

パラレルユーティリティコマンドは同等の非パラレル操作よりもかなり多くのメモリを消費すべきでないことに留意してください。この戦略は、一般的にワーカー毎にリソース制限を適用するパラレルクエリとは異なります。パラレルワーカープロセスの数にかかわらず、パラレルユーティリティコマンドは、その全体でリソース制限maintenance_work_memが適用されるとみなします。しかし、パラレルユーティリティコマンドは、依然としてかなり多くのCPUリソースとI/Oバンド幅を消費するかも知れません。

max_parallel_workers (integer)

パラレルクエリ操作用にシステムがサポートできる最大のワーカー数を設定します。デフォルト値は8です。この値を増減するときは、max_parallel_maintenance_workersとmax_parallel_workers_per_gatherを調整することを考慮してください。また、この設定値をmax_worker_processesよりも高い値にしても効果がないことに注意してください。max_worker_processesで決まるワーカープロセスのプールから、パラレルワーカーが使われるからです。

backend_flush_after (integer)

この量を超えるデータが単一のバックエンドによって書き込まれたときはいつでも、OSが記憶装置に書き込むことを強制します。このことにより、カーネルのページキャッシュが持つダーティデータの量を一定量に制限し、チェックポイントの最後にfsyncが実行される際、あるいはバックグラウンドで実行される大きなバッチの中でOSがデータを書き出す際に性能の急激な低下を招く可能性を減らします。多くの場合これによってトランザクションの遅延が大幅に少なくなりますが、あるケース、特にワークロードがshared_buffersよりも大きく、OSのページキャッシュよりも小さい時には性能が低下するかもしれません。この設定が無効なプラットフォームがあります。この値が単位なしで指定された場合は、ブロック単位であるとみなします。すなわち、BLCKSZバイト、一般的には8kBです。有効な設定値は、この強制書き込み機能が無効になる0から、2MBまでです。デフォルト値は0です(すなわち書き出し制御を行いません)。(BLCKSZが8kbでなければ、最大値がBLCKSZに比例して変更されます。)

old_snapshot_threshold (integer)

スナップショットを使用する際に、「snapshot too old」エラーが起こるリスク無しに問い合わせスナップショットが利用できる最小の期間を設定します。この制限値を越えてデッド状態のままになったデータはバキュームしてしまうことが許可されます。これにより、長い間残っていたスナップショットによりデータが溢れてしまうのを防ぐことができます。スナップショットから見えるデータが消えることによる不正な結果を防ぐため、スナップショットがこの制限値よりも古く、かつこのスナップショットが作られた以降に変更されたページを読むためにスナップショットが使用されるときはエラーが発生します。

この値が単位なしで指定された場合は、分単位であるとみなします。-1(デフォルトです)を設定するとこの機能が無効になり、実質的にスナップショットの寿命を無限にします。このパラメータはサーバ起動時にのみ設定可能です。

実際の環境でのおすすめの値はおそらく数時間から2, 3日の間となるでしょう。小さな値(たとえば0や1min)は、テストの際に有用だということで許可されています。60dのような大きな値の設定もできますが、多くのワークロードにおいて、大きなデータ溢れやトランザクションIDの周回がそれよりはずっと短い期間で起こる可能性があることに注意してください。

この機能が有効であると、リレーションの終端部にあるフリースペースはオペレーティングシステムには返却されません。そうしないと、「snapshot too old」の条件の検出に必要な情報を削除してしまうことになるからです。明示的に解放されない限り(たとえばVACUUM FULLによって)、リレーションに割り当てられた領域は、そのリレーションの中での再利用に限定して紐付けられます。

この設定は、どのような状況でもエラーが検出されることを保証するものではありません。(たとえば)マテリアライズされた結果集合を持つカーソルから正しい結果を得ることができるのであれば、たとえ参照している元のテーブルからVACUUMによって行が削除されたとしてもエラーにはなりません。ある種のテーブルでは、早期にVACUUMできないので、この設定の影響を受けません。例としては、システムカタログが挙げられます。このようなテーブルにおいては、この設定によってデータ溢れを防ぐことも、スキャンの際に「snapshot too old」エラーを起こす可能性を作り出すこともできません。

19.5. ログ先行書き込み(WAL)

これらの設定をチューニングする追加情報は29.4を参照してください。

19.5.1. 諸設定

wal_level (enum)

wal_levelはどれだけの情報がWALに書かれるかを決定します。デフォルト値はreplicaで、WALアーカイビングおよびレプリケーションをサポートするために十分なデータを書き出し、これにはスタンバイサーバで読み取り専用の問い合わせを実行することも含みます。minimalはクラッシュまたは即時停止から回復するのに必要な情報を除き、すべてのログを削除します。最後に、logicalは、更にロジカルデコーディングをサポートするのに必要な情報を追加します。それぞれのレベルは、下位のレベルのログ出力を含んでいます。このパラメータはサーバ起動時のみ設定可能です。

minimalレベルでは、永続リレーションを作成あるいは書き換えるトランザクションの差分に関する情報はログされません。これにより、それらの操作が大幅に高速になります(14.4.7を参照してください)。この最適化が適用される操作には以下のものがあげられます。

```
ALTER ... SET TABLESPACE
CLUSTER
CREATE TABLE
REFRESH MATERIALIZED VIEW (CONCURRENTLYなし)
REINDEX
TRUNCATE
```

しかしminimal WALはベースバックアップとWALログからデータを再構築するための充分な情報を持ち合わせていません。したがって、WALアーカイビング(archive_mode)とストリーミングレプリケーションを有効にするには、replica以上を使用しなければなりません。

logicalレベルでは、replicaと同じ情報がログされるのに加え、ロジカルチェンジセットをWALから取り出すのに必要な情報が追加されます。logicalを使うとWALの量が増えます。とりわけ、多数のテーブルがREPLICA IDENTITY FULLと設定されていて(訳注: ALTER TABLE参照)、多くのUPDATEとDELETE文が実行される場合はこのことが言えます。

9.6よりも前のリリースでは、このパラメータはarchiveとhot_standbyという設定値も可能でした。引き続きこれらも受け付けられますが、replicaへとマップされます。

fsync (boolean)

このパラメータがオンの場合、PostgreSQLサーバはfsync()システムコールを発行するか、もしくはこれに相当する方法(wal_sync_methodを参照)で、更新が物理的にディスクに確実に書き込まれるよう

に試みます。これは、オペレーティングシステムもしくはハードウェアがクラッシュした後、データベースクラスタを一貫した状態に復旧させることを確実にします。

fsyncを停止することはしばしば性能上の利益になるとは言っても、停電やクラッシュの際に回復不可能なデータ破壊になることがあります。従って外部データから全てのデータベースを簡単に再構築できる場合のみfsyncを停止してください。

fsyncを停止しても安全な状況の例としては、以下があげられます。バックアップファイルから新しいデータベースクラスタにデータの初期読み込みを行う場合、バッチデータの処理のためにデータベースクラスタを使用し、その後データベースを削除して再構築する場合、読み込み専用のデータベースのクローンを頻繁に再作成するが、それをフェイルオーバーに使用しない場合、などです。高性能なハードウェアであるからと言って、fsyncを停止することは正当性を主張する十分な理由とはなりません。

fsyncを無効(off)から有効(on)に変更したときの信頼できるリカバリのためには、カーネル内の全ての変更されたバッファを恒久的ストレージに強制的に吐き出させることが必要です。これは、クラスタがシャットダウンしている間、またはfsyncが有効のときに、initdb --sync-onlyを実行する、syncを実行する、ファイルシステムをアンマウントする、またはサーバを再起動することによって可能となります。

多くの場合、重要でないトランザクションに対してsynchronous_commitを無効にすることにより、データ破壊という付随的危険性を伴うことなく、fsyncを無効にすることで得られるであろう性能上のメリットの多くを得ることができます。

fsync はpostgresql.confファイル、または、サーバのコマンドラインでのみ設定可能です。このパラメータを無効にする場合、full_page_writesも同時に無効にすることを検討してください。

synchronous_commit (enum)

トランザクションのコミットがクライアントに「成功」の報告を返す前に、どれだけのWAL処理を完了しなければならないかを指定をします。有効な値はremote_apply、on(デフォルト)、remote_write、local、 offです。

synchronous_standby_namesが空文字なら、意味のある設定はonとoffだけです。remote_apply、 remote_write、localはすべてonと同じ同期レベルを提供します。offモード以外のローカルの振る舞いは、WALがディスクにローカルに吐き出されるのを待ちます。offモードでは待ちはありません。ですから、クライアントに成功が報告されてから、トランザクションが後でサーバクラッシュに対して安全が保証されるまでの間に遅延が生じる可能性があります。(遅延は最大でwal_writer_delayの3倍です。) fsyncと違って、このパラメータをoffにすることでデータベースの一貫性が損なわれるリスクはありません。オペレーティングシステムやデータベースのクラッシュにより最近コミットされたということになっているトランザクションの一部が失われる可能性がありますが、これらのトランザクションが正常にアボートされた時とデータベースの状態は変わりません。したがって、synchronous_commitのオフによる調整は、トランザクションの耐障害性を確実にするよりも性能が重要な場合の有用な代替案となるかも知れません。詳細は29.3を参照してください。

synchronous_standby_namesが空文字でない場合は、synchronous_commitは、WALレコードが、スタンバイサーバに複製されるまでトランザクションコミットを待機するか否かも制御します。

remote_applyに設定すると、現在の同期スタンバイがトランザクションのコミットレコードを受け取って適用し、スタンバイ上で発行されたクエリから見えるようになり、スタンバイ上の永続的な記憶装置に書き込まれたことを報告するまでコミットは待機します。WALのリプレイを待つので、今までの設定に比べるとこの設定によってずっと大きなコミットの遅延が発生します。onに設定すると、現在の同期スタンバイ

がトランザクションのコミットレコードを受け取り、永続的な記憶装置に吐き出したことを報告するまでコミットは待機します。このモードでは、プライマリおよびすべての同期スタンバイがデータベース記憶装置の故障を被った場合を除いて、トランザクションが失われないことが保証されます。remote_writeに設定すると、現在の同期スタンバイがトランザクションのコミットレコードを受け取り、スタンバイのファイルシステムに書き出したことを報告するまでコミットは待機します。この設定は仮にPostgreSQLのスタンバイインスタンスがクラッシュしたとしても、データ保護を保証するのに充分です。しかし、スタンバイがオペレーティングシステムのレベルでクラッシュした場合はこの限りではありません。データが必ずしもスタンバイの永続的な記憶装置に到達したとは言えないからです。最後に、local設定は、コミットがローカルにディスクに吐出されるまで待機しますが、レプリケーションされるまでは待機しません。これは通常同期レプリケーションが使用されている場合は望ましい設定ではありませんが、完全さのために提供されています。

このパラメータはいつでも変更可能です。どのトランザクションの動作も、コミット時に有効であった設定によって決まります。したがって、一部のトランザクションのコミットを同期的に、その他を非同期的にすることが可能で、かつ、有用です。例えば、デフォルトが同期コミットの場合に複数文トランザクションを一つだけ非同期にコミットさせるためには、トランザクション内でSET LOCAL synchronous_commit TO OFFを発行します。

synchronous_commit設定の機能のまとめが表 19.1にあります。

表19.1 synchronous_commitモード

synchronous_commit設定	localの永続的なコミット	PGがクラッシュした後のスタンバイの永続的なコミット	OSがクラッシュした後のスタンバイの永続的なコミット	スタンバイのクエリの一貫性
remote_apply	•	•	•	•
on	•	•	•	
remote_write	•	•		
local	•			
off				

wal_sync_method (enum)

WALの更新をディスクへ強制するのに使用される方法です。fsyncがオフの場合この設定は役に立ちません。と言うのはWALファイルの更新が全く強制されないからです。取り得る値は以下のものです。

- open_datasync（open()のオプションO_DSYNCでWALファイルに書き込む）
- fdatasync（コミット毎にfdatasync()を呼び出す）
- fsync（コミット毎にfsync()を呼び出す）
- fsync_writethrough（すべてのディスク書き込みキャッシュをライトスルーさせるため、コミット毎にfsync()を呼び出す）
- open_sync（open()のオプションO_SYNCでWALファイルに書き込む）

可能ならopen_*オプションもO_DIRECTを使用します。全てのプラットフォームでこれら全ての選択肢が使えるわけではありません。デフォルトは、上のリストのプラットフォームでサポートされるものの最初に

列挙されているものです。ただしLinuxではfdatasyncがデフォルトです。デフォルトは必ずしも理想的なものではありません。クラッシュに適応した構成にする、あるいはアーカイブの最適性能を導くためには、この設定あるいはシステム構成の他の部分を変更することが必要かもしれません。これらの側面は29.1で解説されます。このパラメータはpostgresql.confファイル、または、サーバのコマンドラインでのみ設定可能です。

full_page_writes (boolean)

このパラメータが有効の場合、PostgreSQLサーバは、チェックポイントの後にそのページが最初に変更された過程で、ディスクページの全ての内容をWALに書き込みます。オペレーティングシステムがクラッシュした時に進行中のページ書き込みは途中までしか終わっていない可能性があり、ディスク上のページが古いデータと新しいデータが混在する状態になるため、この機能が必要です。通常WAL内に保存される行レベルの変更データは、クラッシュ後のリカバリ時にこうしたページを完全に復旧させるには不十分です。完全なページイメージを保存することにより、ページを正しく復旧できることを保証しますが、その代わりに、WALに書き込まなければならないデータ量が増加することになります。(WAL再生は常にチェックポイントから始まるため、チェックポイント後のそれぞれのページの最初の変更時にこれを行えば十分です。従って、完全ページ書き出しのコストを低減する方法の1つは、チェックポイント間隔パラメータを大きくすることです。)

このパラメータを無効にすると、通常の操作速度が上がりますが、システム障害後に、回復不能なデータ破損、あるいは警告なしのデータ損壊をもたらすかもしれません。このリスクは小さいながらfsyncを無効にした場合と似ています。そしてそのfsyncに対して推奨されている同一の状況に基づく限りにおいて停止されなければなりません。

このパラメータを無効にしてもポイントインタイムリカバリ(PITR)用のWALアーカイブの使用に影響ありません(25.3を参照してください)。

このパラメータはpostgresql.confファイル内、または、サーバのコマンドラインでのみ設定可能です。デフォルトはonです。

wal_log_hints (boolean)

このパラメータがonの場合、PostgreSQLサーバはチェックポイント後にはじめてページを変更する際に、ディスクページの全内容をWALに書き出します。これは、あまり重要でない、ヒントビットと呼ばれるものに対する変更にさえ当てはまります。

データチェックサムが有効であると、ヒントビットの更新は常にWALにログされ、この設定パラメータは無視されます。この設定パラメータを使って、データチェックサムが有効なときにどれだけのWALログは余計に書きだされるかをテストすることができます。

このパラメータはサーバ起動時のみ設定可能です。デフォルト値はoffです。

wal_compression (boolean)

このパラメータがonなら、full_page_writesがonあるいはベースバックアップの際、PostgreSQLサーバはWALに書き出すフルページイメージを圧縮します。圧縮されたページイメージは、WALリプレイのときに伸張されます。デフォルト値はoffです。スーパーユーザだけがこの設定を変更できます。

このパラメータを有効にすると、回復不可能なデータ破壊のリスクを増やすこと無しにWALの量を減らすことができます。しかし、WALロギングの際の圧縮のため、またWALリプレイの際には伸張のために余分なCPUを使用するというコストが発生します。

wal_init_zero (boolean)

このオプションがon(デフォルト)に設定されると、新しいWALファイルはゼロで初期化されます。システムによっては、このことによってWALレコードを書く必要が出てくる前にスペースが割り当てられることを保証します。しかし、*Copy-On-Write* (COW)ファイルシステムではこの技術は利点がないかもしれず、このオプションはスキップできるようになっています。offに設定すると、ファイルを作る時に期待する大きさになるように最後のバイトだけが書かれます。

wal_recycle (boolean)

このオプションがon(デフォルト)に設定されると、新しくファイルを作るのを避けるために、名前を変えてWALファイルが再利用されます。COWファイルシステムでは、新しいファイルを作るほうが速いかも知れないので、この挙動を無効にできるようにオプションとなっています。

wal_buffers (integer)

未だディスクに書き込まれていないWALデータに対して使用される共有メモリ容量です。デフォルトの設定である-1は、shared_buffersの1/32(約3%)の容量に等しい大きさを選択します。しかし、64kB未満ではなく、かつ典型的に16MBであるWALセグメントの大きさを越えることはありません。もし、自動設定による選択が大きすぎたり、小さすぎる場合この値は手作業で設定可能です。しかし、32kB未満のどんな正の値であっても、32kBとして取り扱われます。この値が単位なしで指定された場合は、WALブロック単位であるとみなします。すなわち、XLOG_BLCKSZバイト、一般的には8kBです。このパラメータはサーバ起動時のみ設定可能です。

WALバッファの内容はトランザクションのコミット毎にディスクに書き込まれます。したがって、極端に大きな値は有意な効果を期待できません。しかし、この値を数メガバイトに設定することにより、多くのクライアントが同時にコミットするトラフィック量の多いサーバでは書き込み性能が向上します。デフォルト設定の-1で選択される自動チューニングによると、ほとんどの場合妥当な結果が得られます。

wal_writer_delay (integer)

WALライタがWALを吐き出す頻度を時間で指定します。WALを吐き出したとあと、非同期コミットしているトランザクションに起こされない限り、wal_writer_delayミリ秒待機します。最後の吐き出しが過去wal_writer_delay以内に行われ、かつそれ以降wal_writer_flush_after相当のWALが生成されている場合は、WALはオペレーティングシステムに書き込まれますが、ディスクには吐出されません。この値が単位なしで指定された場合は、ミリ秒単位であるとみなします。デフォルト値は200ミリ秒(200ms)です。多くのシステムでは、待機間隔の実質的な分解能は10ミリ秒です。10の倍数以外の値をwal_writer_delayに設定しても、その次に大きい10の倍数を設定した場合と同じ結果となります。このパラメータはpostgresql.confファイル内またはサーバのコマンドラインでのみ設定可能です。

wal_writer_flush_after (integer)

WALライタがWALを吐き出す頻度を量で指定します。最後の吐き出しが過去wal_writer_delay以内に起こなわれ、かつそれ以降wal_writer_flush_after相当のWALが生成されている場合は、WALはオペレーティングシステムに書き込まれますが、ディスクには吐出されません。wal_writer_flush_afterが0に設定されている場合は、WALが書かれるたびにWALが吐出されます。この値が単位なしで指定された場合は、WALブロック単位であるとみなします。すなわち、XLOG_BLCKSZバイト、一般的には8kBです。デフォルト値は1MBです。このパラメータはpostgresql.confファイル内またはサーバのコマンドラインでのみ設定可能です。

wal_skip_threshold (integer)

wal_levelがminimalで、トランザクションのコミットが永続リレーションを作るかあるいは書き換えた場合に、この設定は新しいデータをどのように永続させるかを決定します。データがこの設定よりも少なければ、WALログに書きます。そうでなければ、影響のあるファイルに対してfsyncを使用します。そのようなコミットが現在のトランザクションを低速化しているようであれば、使用する記憶装置の特性によってはこの値を増減することが役に立つかも知れません。この値が単位なしに指定されるとキロバイトであると見なします。デフォルトは2メガバイト(2MB)です。

commit_delay (integer)

commit_delayを設定することにより、WALフラッシュを開始する前の時間遅延が追加されます。このことにより、もし追加のトランザクションが与えられた時間間隔内でコミットが可能になるほどシステム負荷が充分に高い場合、一回のWALフラッシュでより多くの数のトランザクションをコミットできるようになり、コミット群のスループットを改善できます。とは言っても、それぞれのWALフラッシュに対して最大commit_delayの待ち時間の増加をきたします。コミットの準備が完了したトランザクションが他に存在しない場合、遅延は無駄になるため、遅延はフラッシュが開始されようとしている時点で少なくともcommit_siblingsだけのトランザクションが活動している場合にだけ機能します。同様に、fsyncが無効の場合も遅延は機能しません。この値が単位なしで指定された場合は、マイクロ秒単位であるとみなします。デフォルトのcommit_delayはゼロ(遅延無し)です。この設定はスーパーユーザのみ変更可能です。

9.3より前のPostgreSQLでは、commit_delayの振る舞いは異なっており、あまり効果がありませんでした。全てのWALフラッシュではなく、コミットだけに影響していました。また、そしてWALフラッシュが早めに完了しても設定された遅延分待機していました。PostgreSQL 9.3以降では、フラッシュの準備が整った最初のプロセスが設定値分待機し、後続のプロセスは最初のプロセスがフラッシュ操作を完了するまでの間だけ待機をします。

commit_siblings (integer)

commit_delayの遅延を実行するときに必要とされる同時に開いているトランザクションの最小数です。より大きい値にすると、遅延周期の間に、少なくとも1つの他のトランザクションのコミットの準備が整う確率が高くなります。デフォルトは5トランザクションです。

19.5.2. チェックポイント

checkpoint_timeout (integer)

自動的WALチェックポイント間の最大間隔を指定します。この値が単位なしで指定された場合は、秒単位であるとみなします。有効な範囲は、30秒から1日の間です。デフォルトは5分(5min)です。このパラメータを増やすと、クラッシュリカバリで必要となる時間が増加します。このパラメータはpostgresql.confファイル、または、サーバのコマンドラインでのみ設定可能です。

checkpoint_completion_target (floating point)

チェックポイントの完了目標をチェックポイント間の総時間の割合として指定します。デフォルトは0.5です。このパラメータはpostgresql.confファイル、または、サーバのコマンドラインでのみ設定可能です。

checkpoint_flush_after (integer)

チェックポイント実行中にこのデータ量よりも多く書く度に、OSが記憶装置に書き込むことを強制しようとします。このことにより、カーネルのページキャッシュが持つダーティデータの量を一定量に制限し、チェックポイントの最後にfsyncが実行される際、あるいはOSがバックグラウンドでデータを大きな塊で書き出す際に性能の急激な低下を招く可能性を減らします。多くの場合これによってトランザクションの遅延が大幅に少なくなりますが、あるケース、特にワークロードがshared_buffersよりも大きく、OSのページキャッシュよりも小さい時には性能が低下するかもしれません。この設定が無効なプラットフォームがあります。この値が単位なしで指定された場合は、ブロック単位であるとみなします。すなわち、BLCKSZバイト、一般的には8kBです。有効な設定値は、この強制書き込み機能が無効になる0から、2MBまでです。デフォルト値は、Linuxでは256kBで、それ以外は0です。(BLCKSZが8kbでなければ、この設定のデフォルト値と最大値がBLCKSZに比例して変更されます。) このパラメータはpostgresql.confファイル、または、サーバのコマンドラインでのみで設定可能です。

checkpoint_warning (integer)

WALセグメントファイルが溢れることが原因で起きるチェックポイントが、ここで指定した時間よりも短い間隔で発生したとき、サーバログにメッセージを書き出します（これは、max_wal_sizeを増やす必要があることを示唆しています）。この値が単位なしで指定された場合は、秒単位であるとみなします。デフォルトは30秒（30s）です。零の場合は警告を出しません。checkpoint_timeoutがcheckpoint_warningより小さい場合は警告を出しません。このパラメータはpostgresql.confファイル、または、サーバのコマンドラインでのみ設定可能です。

max_wal_size (integer)

自動WALチェックポイントの際にWALが増加する最大サイズです。これはソフトリミットです。特別な状況下、たとえば高負荷、archive_commandの失敗、wal_keep_sizeが大きな値に設定されている、などの時には、WALサイズはmax_wal_sizeを超えることがあります。この値が単位なしで指定された場合は、メガバイト単位であるとみなします。デフォルトは1GBです。このパラメータを大きくすると、クラッシュリカバリに必要な時間が長くなります。このパラメータは、postgresql.confファイルで設定するか、サーバのコマンドラインでのみ指定できます。

min_wal_size (integer)

この設定以下にWALのディスク使用量が保たれる限り、古いWALファイルは、消去されることなく今後のチェックポイントで使用するために常にリサイクルされます。この設定は、たとえば大きなバッチジョブを走らせる際のWALの利用スパイクを取り扱うために、十分なWALのスペースが予約されていることを保証するために使用できます。この値が単位なしで指定された場合は、メガバイト単位であるとみなします。デフォルトは80MBです。このパラメータは、postgresql.confファイルで設定するか、サーバのコマンドラインでのみ指定できます。

19.5.3. アーカイビング

archive_mode (enum)

archive_modeが有効な場合、archive_commandを設定することにより、完了したWALセグメントはアーカイブ格納領域に送信されます。無効にするためのoffに加え、2つのモードがあります。onとalwaysです。通常の運用ではこの2つのモードには違いはありませんが、alwaysに設定すると、アーカイブリカバリおよびスタンバイモードでWALアーカイバが有効になります。alwaysモードでは、

アーカイブからリストアされたファイルや、ストリーミングレプリケーションでストリームされたファイルもすべて(再び)アーカイブされます。詳細は26.2.9を参照してください。

アーカイブモードを抜けることなくarchive_commandを変更できるように、archive_modeとarchive_commandは分離されました。このパラメータはサーバ起動時のみ設定可能です。wal_levelがminimalに設定されている場合、archive_modeは有効になりません。

archive_command (string)

完了したWALファイルセグメントのアーカイブを実行するローカルのシェルコマンドです。文字列内のすべての%pは、格納されるファイルのパスで置き換えられ、そして、%fはファイル名のみ置換します。(このパス名はサーバの作業用ディレクトリ、つまり、クラスタのデータディレクトリからの相対パスです。) コマンド内に%文字そのものを埋め込むには%%を使用します。コマンドが成功した場合にのみ終了ステータスゼロを返すことが重要です。詳しくは25.3.1を参照ください。

このパラメータはpostgresql.confファイル、または、サーバのコマンドラインでのみ設定可能です。サーバ起動時にarchive_modeが有効でなければ、これは無視されます。archive_commandが空文字列(デフォルト)、かつ、archive_modeが有効な場合、WALアーカイブ処理は一時的に無効になりますが、コマンドが後で提供されることを見越して、サーバはWALセグメントの蓄積を続けます。例えば、/bin/true(WindowsではREM)のように、真を返すだけで何もしないコマンドをarchive_commandに設定すると、実質的にアーカイブ処理が無効になりますが、アーカイブからの復帰に必要なWALファイルの連鎖も同時に断ち切るため、特別な場合のみ使用するようにしなければなりません。

archive_timeout (integer)

archive_commandは完了したWALセグメントに対してのみ呼び出されます。従って、サーバのWAL転送量が少ししかない(あるいは処理が少ないなぎの期間がある)場合、トランザクションの完了とアーカイブ格納領域への安全な記録との間に長期にわたる遅延があることになります。データが未アーカイブのままでいられる期間を制限するために、archive_timeoutを設定して、強制的にサーバを新しいWALセグメントに定期的に切り替えるようにすることができます。このパラメータが0より大きければ、サーバは前回のセグメントファイル切り替えから指定時間経過し、かつ単一のチェックポイントを含む何らかのデータベース操作が行われた場合、新しいセグメントファイルに切り替えます。(データベースが活動していなければ、チェックポイントはスキップされます。) 強制切り替えにより早期にクローズされたアーカイブ済みファイルは、完全に完了したファイルと同じ大きさを持つことに注意してください。そのため、非常に小さなarchive_timeoutを使用することは賢明ではなく、格納領域を膨張させてしまいます。1分程度のarchive_timeout設定が通常は妥当です。もしそれより高速にデータをマスターサーバからコピーをしてしまいたいのであれば、アーカイブするよりストリーミングレプリケーションの選択を検討すべきです。この値が単位なしで指定された場合は、秒単位であるとみなします。このパラメータはpostgresql.confファイル、または、サーバのコマンドラインでのみで設定可能です。

19.5.4. Archive Recovery

この節では、リカバリの間だけ適用する設定について説明します。その後実施するリカバリでは、その設定はリセットしなければなりません。

「リカバリ」は、サーバをスタンバイとして使用するとき、あるいはターゲットを指定したリカバリで適用されます。通常、スタンバイモードは、高可用性または読み出しスケーラビリティ、あるいはその両方を提供するために使用します。一方ターゲットを指定したリカバリは失なわれたデータを回復するために使用します。

スタンバイモードでサーバを起動するには、standby.signalと呼ばれるファイルをデータディレクトリに作ります。サーバはリカバリモードに入り、アーカイブWALの終端に到着してもリカバリを止めず、primary_conninfoの設定で指定された送信サーバに接続するか、restore_commandを使って新しいWALセグメントを取得するか、あるいはその両方によってリカバリを継続しようとします。このモードでは、この節と19.6.3で説明するパラメータが関係します。19.5.5のパラメータも適用できますが、このモードでは通常有用ではありません。

サーバをターゲットリカバリモードで起動するには、recovery.signalという名前のファイルをデータディレクトリに作ります。standby.signalとrecovery.signalの両方が作られた場合は、スタンバイモードが優先します。ターゲットリカバリモードはアーカイブWALが完全に再生されるか、recovery_targetに到達した時に終了します。このモードでは、この節と19.5.5のパラメータの両方が使用されます。

restore_command (string)

一連のWALファイルからアーカイブセグメントを取り出すために実行するローカルのシェルコマンドです。このパラメータはアーカイブリカバリでは必須ですが、ストリーミングレプリケーションではオプションです。文字列中の%fはアーカイブから取り出すファイルの名前に置換され、%pはサーバ上のコピー先のパス名に置換されます。(パス名は現在の作業ディレクトリの相対パス、つまりクラスタのデータディレクトリです。) %rは有効な最後のリスタートポイントを含むファイル名に置換されます。これはリストアが再開可能であるために維持しなければならない最古のファイルで、現在のリストアからの再開をサポートするのに最小限必要なアーカイブを残して切り詰めるのに必要な情報として利用できます。%rは通常ウォームスタンバイ構成でのみ使用されます。(26.2参照。) %文字自体を埋め込むには%%と書いてください。

コマンドは、成功した時のみ終了コードのゼロを返却することが重要です。コマンドはアーカイブにないファイル名を聞かれることになります。その場合には、非ゼロの値を返却しなければなりません。以下に例を示します。

```
restore_command = 'cp /mnt/server/archivedir/%f "%p"'
restore_command = 'copy "C:\\server\\archivedir\\%f" "%p"'  # Windows
```

例外は、データベースサーバのシャットダウンの一部として、SIGTERM以外のシグナルでコマンドが終了させられたり、シェルによってエラーが発生した(コマンドが見つからない場合など)場合で、その場合はリカバリは中断され、サーバはスタートアップしなくなります。

このパラメータはサーバ起動時のみ設定可能です。

archive_cleanup_command (string)

オプションのパラメータは、すべてのリスタートポイントで実行されるシェルコマンドを指定します。archive_cleanup_commandの目的は、スタンバイサーバにとって必要とされない古いアーカイブWALファイルをクリーンアップする仕組みを提供することです。%rは最後の有効なリスタートポイントを含むファイル名に置換されます。これはリストアが再開可能であるために保持しなければならない最古のファイルで、%rよりも前のすべてのファイルは安全に削除できます。これは現在のリストアからの再開をサポートするのに最小限必要なアーカイブを残して切り詰めるのに必要な情報として利用できます。単一のスタンバイ構成用のarchive_cleanup_commandで、たとえば

```
archive_cleanup_command = 'pg_archivecleanup /mnt/server/archivedir %r'
```

のようにして、しばしばpg_archivecleanupモジュールが使われます。だたし、複数のスタンバイサーバが同じアーカイブディレクトリからリストアしている場合は、どのサーバにおいてももはや必要がなくなるまでWALファイルが削除されることのないようにする必要があることに留意してください。archive_cleanup_commandは通常ウォームスタンバイ構成で使用されます。(26.2参照。) %文字自体を埋め込むには%%と書いてください。

コマンドが非ゼロの終了ステータスを返した場合、警告ログメッセージが出力されます。例外は、コマンドがシグナルで終了されたとき、あるいはシェルがエラーを起こしたとき(コマンドが見つからないなど)で、その場合致命的エラーが生じます。

このパラメータは postgresql.confファイル、または、サーバのコマンドラインでのみで設定可能です。

recovery_end_command (string)

このパラメータは、リカバリの終了時に一度だけ起動されるシェルコマンドを指定します。このパラメータはオプションです。recovery_end_commandの目的はレプリケーションあるいはリカバリの後のクリーンアップのための機構を提供することにあります。%rは、archive_cleanup_commandと同じように、有効な最後のリスタートポイントを含むWALファイル名に置換されます。

コマンドが非ゼロの終了ステータスを返した場合、警告ログメッセージが出力されますが、データベースはスタートアップ処理を続けます。例外は、コマンドがシグナルによって終了させられたか、シェルによってエラーが発生した(そのようなコマンドは見つからない)場合で、その場合はデータベースはスタートアップ処理を継続させません。

このパラメータは postgresql.confファイル、または、サーバのコマンドラインでのみで設定可能です。

19.5.5. リカバリターゲット

デフォルトではWALログの最後までリカバリを行います。次のパラメータがそれより前の時点でリカバリを停止するために利用できます。次のrecovery_target、recovery_target_lsn、recovery_target_name、recovery_target_time、recovery_target_xidのどれか一つが使えます。設定ファイルの中で2つ以上指定するとエラーとなります。これらのパラメータはサーバ起動時のみ設定可能です。

recovery_target = 'immediate'

このパラメータは、リカバリが一貫した状態になり次第、すなわちできるだけ早く終了することを指定します。オンラインバックアップからリストアした場合、これはバックアップが終了した時点を意味します。

技術的にはこれは文字列型のパラメータですが、現時点では'immediate'だけが許容されている値です。

recovery_target_name (string)

このパラメータは、指定した(pg_create_restore_point()により作成された)名前付きリストアポイントまでリカバリを進行させます。

recovery_target_time (timestamp)

このパラメータは、指定したタイムスタンプまでリカバリを進行させます。正確な停止点はrecovery_target_inclusiveにも影響されます。

(timezone_abbreviations変数が設定ファイルで先に設定されていない限り)時間帯略語を使えないことを除けば、このパラメータの値はtimestamp with time zoneデータ型が受け付けるのと同じ形式の

タイムスタンプです。おすすめの形式はUTCからのオフセットか、EESTではなくEurope/Helsinkiのような完全な時間帯名です。

recovery_target_xid (string)

このパラメータは、指定したトランザクションIDまでリカバリを進行させます。トランザクションIDはトランザクション起動時に順番に割り当てられますが、トランザクションは数字順によらず完了することがあることに留意してください。リカバリされるトランザクションは、指定されたものよりも前(オプションによっては指定されたものも含まれる)にコミットされたものになります。正確な停止点はrecovery_target_inclusiveにも影響されます。

recovery_target_lsn (pg_lsn)

このパラメータは、指定した先行書き込みログ(WAL)の場所のLSNまでリカバリを進行させます。正確な停止点は、recovery_target_inclusiveの影響も受けます。このパラメータは、システムデータ型pg_lsnを使用して解析されます。

以下のオプションはリカバリ対象をより詳細に指定し、リカバリが対象に達した時の動作に影響を与えます。

recovery_target_inclusive (boolean)

指定したリカバリ対象のちょうど後に停止するか(on)、ちょうどその前に停止するか(off)を指定します。recovery_target_lsn、recovery_target_time、又はrecovery_target_xidが指定されている場合は適用されます。この設定は、指定した対象のWALの場所(LSN)、コミット時刻、あるいはトランザクションIDが、それぞれ正確に一致するトランザクションをリカバリに含めるかどうかを制御します。デフォルトはonです。

recovery_target_timeline (string)

リカバリが作成する個別のタイムラインを指定します。値は数値のタイムラインIDか、特殊な値です。currentでは、ベースバックアップが取得されたときにカレントだったタイムラインに沿ってリカバリします。latestでは、アーカイブ時に見つけた最新のタイムラインにリカバリします。これはスタンバイサーバで有用です。デフォルトはlatestです。

通常このパラメータの設定が必要となるのは、ポイントインタイムリカバリの実施後に到達した状態に戻す場合など、複雑なリカバリの状況のみです。この論考については25.3.5を参照してください。

recovery_target_action (enum)

リカバリ対象に到達した場合に、サーバがする動作を指定します。デフォルトはpauseで、リカバリを休止することを意味します。promoteは、リカバリの過程が終われば、サーバは接続の受け付けを始めることを意味します。最後に、shutdownは、リカバリ対象に到達した後にサーバを停止します。

pauseの設定の意図した使い方は、このリカバリ対象がリカバリのための最も望ましいポイントかどうかチェックするために、データベースに対して問い合わせを実行できるようにすることです。休止された状態は、pg_wal_replay_resume()(表 9.87参照)の使用により再開することができます。その後、それはリカバリを終了させます。このリカバリ対象が希望の止まるポイントでない場合、サーバをシャットダウンし、リカバリ対象の設定をより後の対象に変更し、リカバリを継続するために再起動してください。

shutdownの設定はインスタンスを正確に望ましい再生ポイントで準備するのに有用です。インスタンスはさらに多くのWALレコードを再生できます(実際、次に起動するときには最後のチェックポイントからWALレコードを再生しなければなりません)。

recovery.signalはrecovery_target_actionがshutdownに設定されていると削除されないことに留意してください。設定が変更されるか、recovery.signalが手動で削除されない限り、以降起動しても直ちに停止されてしまいます。

この設定はリカバリ対象が設定されていない場合には効果がありません。hot_standbyが有効になっていない場合、pauseの設定はshutdownと同じように動作します。昇格中にリカバリ対象に到達した場合は、pauseはpromoteと同じように働きます。

どのような場合でも、リカバリ対象が設定されていて、アーカイブリカバリがそのリカバリ対象に到達する前に終了すると、サーバはフェイタルエラーで停止します。

19.6. レプリケーション

これらの設定は組み込みのストリーミングレプリケーション機能の動作を制御します（26.2.5を参照ください）。サーバ群のサーバはマスターかスタンバイのいずれかです。マスターはデータを送出する一方、複数のスタンバイは複製されたデータを常に受け取ります。カスケードレプリケーション（26.2.7を参照）が使用されている場合、スタンバイサーバ群は受け取り手でもあり、送り手でもあります。パラメータは主として送出サーバとスタンバイサーバ用ですが、いくつかのパラメータはマスターサーバのみに効力を発します。必要とあればクラスタに渡って問題なく設定を変化させることができます。

19.6.1. 送出サーバ群

これらのパラメータはレプリケーションデータを1つ、またはそれ以上複数のスタンバイサーバに送るいかなるサーバ上で設定することができます。マスターは常に送出サーバであるため、パラメータは常にマスター上に設定されなければなりません。これらのパラメータの役割と意味はスタンバイが後にマスターに昇格しても変わりません。

max_wal_senders (integer)

複数のスタンバイサーバあるいは、ストリーミングを使ったベースバックアップクライアントからの同時接続を受ける接続最大値を設定します（つまり、同時に稼動するWAL送信プロセスの最大値です）。デフォルトは10です。0ならば、レプリケーションは無効であるという意味になります。ストリーミングクライアントの突然の切断により、タイムアウトになるまで親のない接続スロットが残ることがあります。ですから、このパラメータは想定されるクライアント数の最大値よりも少し大きめにして、切断されたクライアントが直ちに再接続できるようにした方が良いでしょう。このパラメータはサーバ起動時のみ設定可能です。また、スタンバイサーバからの接続を許可するには、wal_levelをreplica以上に設定しておかなければなりません。

スタンバイサーバを実行する際は、このパラメータをマスタサーバと同じか高い値にしなければなりません。さもなければ、スタンバイサーバでクエリを実行できなくなります。

max_replication_slots (integer)

サーバが使用できるレプリケーションスロット(26.2.6参照)の最大数を指定します。デフォルトは10です。このパラメータはサーバ起動時のみ設定可能です。現在存在しているレプリケーションスロットの数よりも少ない値を設定すると、サーバは起動しません。また、レプリケーションスロットが使用できるためには、wal_levelをreplica以上に設定しなければなりません。

wal_keep_size (integer)

ストリーミングレプリケーションにおいて、スタンバイサーバが過去のファイルセグメントを取得する必要がある場合に備え、pg_walディレクトリに保持しておくファイルセグメントの最小サイズを指定します。もし送出サーバに接続しているスタンバイサーバがwal_keep_sizeメガバイトを越えて遅延した場合、送出サーバはスタンバイサーバが今後とも必要とするWALセグメントを削除する可能性があります。この場合、レプリケーション接続は終了させられます。結果として下流に対する接続も結局は終了されることがあります。(しかし、WALアーカイブが使用されていれば、スタンバイサーバはアーカイブからセグメントを取り出し、復旧することができます。)

pg_walに保持され続けるセグメントの最小値のみを設定します。システムはWALアーカイブのため、またはチェックポイントからの復旧のため、より多くのセグメント保持が必要となることがあります。もしwal_keep_sizeが(デフォルトの)ゼロの場合、システムはスタンバイサーバのために追加セグメントを保持することはしません。従って、スタンバイサーバが使用できる古いWALセグメントの数は、直前のチェックポイントの場所とWALアーカイブの状況によって算出されます。この値が単位無しで指定されると、メガバイトであると見なします。このパラメータは、postgresql.confファイル、もしくはサーバコマンドラインでのみ設定可能です。

max_slot_wal_keep_size (integer)

チェックポイント時にレプリケーションスロットがpg_walディレクトリに残すことのできる 最大のWALファイルのサイズを指定します。max_slot_wal_keep_sizeが-1 (デフォルトです)なら、レプリケーションスロットは無制限のWALファイルを残すかも知れません。そうでなければ、レプリケーションスロットのrestart_lsnが現在のLSNよりも与えられたサイズ分遅れると、そのスロットを使っているスタンバイは必要なWALファイルが削除されたためにレプリケーションを継続できなくなります。レプリケーションスロットのWALが存在するかどうかはpg_replication_slotsを見て確認できます。

wal_sender_timeout (integer)

指定された時間より長く非活動であるレプリケーション接続を停止します。スタンバイサーバのクラッシュ、またはネットワークの停止を送出サーバが検出することにこれが役立ちます。この値が単位なしで指定された場合は、ミリ秒単位であるとみなします。デフォルトの値は60秒です。値ゼロはこのタイムアウト機能を無効にします。

地理的に複数の場所に分散したクラスタでは、場所によって異なる値を使うことでクラスタ管理がより柔軟にできるようになります。より小さな値は低遅延ネットワーク接続上のスタンバイの障害検知をより高速にするのに役立ちます。遅延の大きなネットワーク接続に設置されたスタンバイの健全性の判断にはより大きな値が助けになるでしょう。

track_commit_timestamp (boolean)

トランザクションのコミットタイムを記録します。このパラメータは、postgresql.confファイル、もしくはサーバコマンドラインでのみ設定可能です。デフォルトはoffです。

19.6.2. マスターサーバ

これらのパラメータはレプリケーションデータを1つ、またはそれ以上複数のスタンバイサーバに送るマスター/プライマリサーバ上で設定することができます。これらパラメータに加え、wal_levelはマスターサーバ

上で適切に設定される必要があり、オプションとしてWALアーカイブを有効にしてもかまいません（19.5.3を参照してください）。スタンバイサーバがマスターサーバになるかもしれない状況に備え、それらのパラメータをスタンバイサーバで設定したいと考えたとしても、スタンバイサーバ上でのパラメータの値は意味をなしません。

synchronous_standby_names (string)

26.2.8で説明されているように、同期レプリケーションをサポート可能なスタンバイサーバのリストを指定します。活動中の同期スタンバイサーバは1つまたはそれ以上です。コミットを待機しているトランザクションは、このスタンバイサーバがそのデータの受信を確認してから処理の継続が許可されます。同期スタンバイサーバはこのリストに名前が挙げられていており、現時点で接続され、そしてデータをリアルタイムでストリーミングしているものです（pg_stat_replicationビューにおいてstreaming状態として示されています）。このリストの後の方に記載されているその他のスタンバイサーバは潜在的に同期スタンバイサーバになることを示しています。二つ以上の同期スタンバイサーバ名を指定することで、かなりの高可用性とデータ損失に対する保護が得られます。

この目的のためのスタンバイサーバ名は、スタンバイの接続情報で指定された、スタンバイのapplication_name設定です。物理レプリケーションスタンバイでは、primary_conninfo設定です。デフォルトはcluster_nameの設定で、さもなければwalreceiverです。論理レプリケーションでは、サブスクリプションの接続情報で設定でき、デフォルトはサブスクリプション名です。それ以外のレプリケーションストリームコンシューマーについては、それぞれのドキュメントをご覧ください。

このパラメータは、以下の構文のいずれかを用いてスタンバイサーバのリストを指定します。

```
[FIRST] num_sync ( standby_name [, ...] )
ANY num_sync ( standby_name [, ...] )
standby_name [, ...]
```

ここで、num_syncは、トランザクションが応答を待機する必要のある同期スタンバイの数です。standby_nameは、スタンバイサーバの名前です。FIRSTとANYは、リスト中のサーバーから同期スタンバイを選ぶ方法を指定します。

キーワードFIRSTをnum_syncと組み合わせると、優先度に基づく同期レプリケーションを指定し、優先度に基づいて選ばれたnum_sync個の同期スタンバイにWALレコードがレプリケーションされるまで、トランザクションのコミットは待機します。たとえばFIRST 3 (s1, s2, s3, s4)とすると、s1、s2、s3、s4の中から選ばれた優先順位の高い3つのスタンバイサーバが応答を返すまでコミットは待機します。リストの中で前の方に名前が出現するスタンバイには高い優先度が与えられ、同期と見なされます。それ以外のリストの中で後の方に名前が上がっているスタンバイサーバーは、潜在的な同期スタンバイであることを表しています。どんな理由であれ、現在の同期スタンバイが切断されると、次に高い優先度を持つスタンバイに直ちに取って代わられます。キーワードFIRSTはオプションです。

キーワードANYをnum_syncと組み合わせると、クォーラムに基づく同期レプリケーションを指定し、列挙されたスタンバイのうち少なくともnum_sync個の同期スタンバイにWALレコードがレプリケーションされるまで、トランザクションのコミットを待たせます。たとえばANY 3 (s1, s2, s3, s4)とすると、s1、s2、s3、s4のうちの少なくとも3つが応答を返した時点でコミットが進行します。

FIRSTとANYは、大文字小文字を区別しません。もしこれらのキーワードをスタンバイサーバの名前に使う場合は、standby_nameは二重引用符で囲わなければなりません。

3番目の構文は、PostgreSQL9.6よりも前のバージョンで用いられていたもので、依然としてサポートされています。最初の構文で、FIRST、num_syncを1とした時と同じです。たとえば、FIRST 1 (s1, s2)とs1, s2は同じ意味です。s1かs2が同期スタンバイとして選ばれます。

特別なエントリ*は、すべてのスタンバイ名に一致します。

スタンバイの一意性を強要する仕組みはありません。重複があった場合、一致したスタンバイは優先順位が高いと見なされますが、どれが選ばれるかは非決定的です。

> **注記**
>
> 各々のstandby_nameは、*である場合を除き、SQL識別子の形式を取らなければなりません。二重引用符を用いることもできます。しかし、二重引用符の有無に関わらず、standby_nameとスタンバイのアプリケーション名の比較は、大文字小文字の区別なしに行われることに注意してください。

ここに同期スタンバイ名が指定されていない場合、同期レプリケーションは有効とはならず、トランザクションコミットはレプリケーションを待機しません。これがデフォルトの設定です。同期レプリケーションが有効であっても、synchronous_commitパラメータをlocal または offに設定することにより、個別のトランザクションをレプリケーションに対して待機しないように設定できます。

このパラメータは、postgresql.confファイル、もしくはサーバコマンドラインでのみ設定可能です。

vacuum_defer_cleanup_age (integer)

VACUUM および HOT更新が不要行バージョンの回収を決めるのを延期するトランザクションの数を指定します。デフォルトはゼロトランザクションで、つまり、開いている全てのトランザクションから不可視になった不要行バージョンは速やかに削除されうることを意味します。26.5に記載されているように、ホットスタンバイサーバをサポートしている場合、プライマリサーバ上で非ゼロ値に設定したい場合があります。そうすることで、スタンバイ上での問い合わせが、行の早期回収によるコンフリクトを被ることなく完了するように時間を与えます。しかし、値はプライマリーサーバ上で発生している書き込みトランザクションの観点から計測されるため、スタンバイの問い合わせにたいして猶予時間がどのくらい有効となるかは予測できません。このパラメータは、postgresql.confファイル、もしくはサーバコマンドラインでのみ設定可能です。

このパラメータを使う代わりにスタンバイサーバ上に hot_standby_feedbackの設定を考慮する必要もあります。

old_snapshot_thresholdで指定された年齢に達した無効行のクリーンアップを妨げることはありません。

19.6.3. スタンバイサーバ

これらの設定はレプリケーションデータを受け取るスタンバイサーバの動作を管理します。マスターサーバ上のこれらの値は無意味です。

primary_conninfo (string)

スタンバイサーバが送信サーバに接続するための接続文字列を指定します。この文字列は、33.1.1で説明されている書式で記述されます。この文字列に何のオプションも指定されていない場合、これに対応する環境変数 (33.14 参照) が確認されます。環境変数も設定されていなければデフォルトの値が使われます。

接続文字列では、プライマリサーバのホスト名（またはアドレス）、スタンバイサーバのデフォルトと異なるのであればポート番号も指定する必要があります。また、送信サーバ上で適切な権限を保有するロールのユーザを指定しなければなりません (26.2.5.1 参照)。送信サーバが要求するのであれば、パスワードも記述される必要があります。パスワードは primary_conninfoに記述することもできますし、スタンバイサーバ上の分離されたファイル~/.pgpassに記述することもできます (データベース名には replication を使います)。primary_conninfo文字列の中には、データベース名を指定しないでください。

このパラメータはpostgresql.confか、サーバのコマンドラインでのみ設定可能です。WAL受信プロセスが実行中にこのパラメータが変更されると、そのプロセスにシグナルが送られ、新しい設定で再起動するために停止します（primary_conninfoが空文字の場合を除きます。） サーバがスタンバイモードでなければこの設定は無効となります。

primary_slot_name (string)

上流ノードのリソース削除を制御するためにストリーミングレプリケーション経由でプライマリに接続した場合、既存のレプリケーションスロットを使うように、必要に応じて指定します（26.2.6を参照）。このパラメータはpostgresql.confか、サーバのコマンドラインでのみ設定可能です。WAL受信プロセスが実行中にこのパラメータが変更されると、そのプロセスにシグナルが送られ、新しい設定で再起動するために停止します。primary_conninfoが設定されていない場合、この設定は無効です。

promote_trigger_file (string)

スタンバイサーバにおいて、リカバリの完了を示すトリガファイルを指定します。この値が設定されていなくても、pg_ctl promoteあるいはpg_promote()を使ってスタンバイサーバを昇格させることができます。このパラメータは、postgresql.confファイル、もしくはサーバコマンドラインでのみ設定可能です。

hot_standby (boolean)

26.5に記載されている通り、リカバリの最中に接続し、そして問い合わせを実行できるか否かを設定します。デフォルト値はonです。このパラメータはサーバ起動時のみ設定可能です。これは、アーカイブリカバリ期間、又はスタンバイモードにある場合にのみ効果をもたらします。

max_standby_archive_delay (integer)

ホットスタンバイが稼動している場合、このパラメータは26.5.2で記載されているように、まさに適用されようとしているWALエントリと衝突するスタンバイサーバの問い合わせをキャンセルするにはどれだけ待機しなければならないかを設定します。max_standby_archive_delayはWALデータをWALアーカイブ（すなわち最新ではありません）から読み込んでいる時に適用されます。この値が単位なしで指定された場合は、ミリ秒単位であるとみなします。デフォルトは30秒です。値-1は衝突する問い合わせが完了するまでスタンバイサーバが待ち続けられるようにします。このパラメータは、postgresql.confファイル、もしくはサーバコマンドラインでのみ設定可能です。

max_standby_archive_delayはキャンセル前に問い合わせが実行できる最大の時間の長さと同じでないことに注意してください。むしろ、任意の1つのWALセグメントのデータの適用のために許される最大合計時間です。従って、ある問い合わせによりWALセグメント内の前の部分で大幅な遅延となった場合、その後の衝突する問い合わせの猶予時間はずっと短くなります。

max_standby_streaming_delay (integer)

ホットスタンバイが稼動している場合、このパラメータは26.5.2で記載されているように、まさに適用されようとしているWALエントリと衝突するスタンバイサーバの問い合わせをキャンセルするにはどれだけ待機しなければならないかを設定します。max_standby_streaming_delayはWALデータをストリーミングレプリケーションから受け取っている時に適用されます。特に指定が無ければ単位はミリ秒です。デフォルトは30秒です。値-1は衝突する問い合わせが完了するまでスタンバイサーバが待ち続けられるようにします。このパラメータは、postgresql.confファイル、もしくはサーバコマンドラインでのみ設定可能です。

max_standby_streaming_delayはキャンセル前に問い合わせが実行できる最大の時間の長さと同じでないことに注意してください。むしろ、プライマリサーバから一度受け取られたWALデータを適用するために許される最大合計時間です。従って、ある問い合わせが大幅な遅延を起こした場合、その後の衝突する問い合わせは、スタンバイサーバがふたたび遅れを取り戻すまでの間、猶予時間はずっと短くなります。

wal_receiver_create_temp_slot (boolean)

永続レプリケーションスロットが(primary_slot_nameを使って)作成されない設定になっている時に、WAL受信プロセスがリモートインスタンス上に一時レプリケーションスロットを作るかどうかを指定します。このパラメータはpostgresql.confか、サーバのコマンドラインでのみ設定可能です。WAL受信プロセスが実行中にこのパラメータが変更されると、そのプロセスにシグナルが送られ、新しい設定で再起動するために停止します。

wal_receiver_status_interval (integer)

スタンバイサーバ上のWAL受信プロセスがプライマリー、または上位サーバに対してレプリケーションの進捗情報を送信する最小頻度を指定します。送信された進捗情報はpg_stat_replicationビューにより確認することが可能です。スタンバイサーバは書き込みがされた直近の先行書き込みログ位置、ディスクにフラッシュされた直近のログ位置、およびリカバリ適用された直近のログ位置を報告します。このパラメータの値がそれぞれの報告間における最大の時間間隔です。書き込み、またはフラッシュ位置が変更される毎、あるいは最低でもこのパラメータで設定された頻度で更新情報が送信されます。従って、適用位置は真の位置よりも少し後ろにずれることがあります。この値が単位なしで指定された場合は、秒単位であるとみなします。デフォルトの値は10秒です。このパラメータをゼロに設定すると、ステータスの更新を完全に無効化します。このパラメータは、postgresql.confファイル、もしくはサーバコマンドラインでのみ設定可能です。

hot_standby_feedback (boolean)

ホットスタンバイがスタンバイサーバ上で現在処理を行っている問い合わせについて、プライマリーまたは上位サーバにフィードバックを送るか否かを指定します。このパラメータはレコードの回収に起因する問い合わせの取り消しを排除するために使用することができます。しかし、いくつかのワークロードに対してはプライマリーサーバ上でのデータベース肥大の原因となります。フィードバックメッセージはwal_receiver_status_interval毎に、2回以上送信されません。デフォルトの値はoffです。このパラメータは、postgresql.confファイル、もしくはサーバコマンドラインでのみ設定可能です。

カスケードレプリケーションが使用されている場合、フィードバックは最終的にプライマリーに到達するまで上位サーバに転送されます。スタンバイは上位に転送する以外、受け取ったフィードバックを他に使用しません。

この設定は、プライマリのold_snapshot_thresholdの挙動を上書きしません。プライマリの年齢上限を超えたスタンバイ上のスナップショットは無効となる可能性があり、その場合スタンバイにおいてトランザクションのキャンセルを引き起こします。これは、old_snapshot_thresholdが、無効行によってデータ溢れが起こる時刻の絶対的な制限を提供することを意図しているからです。そうでなければ、スタンバイの設定によってこの目的は成立しなくなってしまいます。

wal_receiver_timeout (integer)

指定された時間より長い間、活動していないレプリケーション接続は停止します。このことは受信するスタンバイサーバがプライマリノードの機能停止、またはネットワーク停止を検出するのに便利です。この値が単位なしで指定された場合は、ミリ秒単位であるとみなします。デフォルト値は60秒です。値ゼロは時間切れメカニズムを無効にします。このパラメータはpostgresql.confファイルまたはサーバのコマンドラインのみで設定可能です。

wal_retrieve_retry_interval (integer)

WALデータがソース(ストリーミングレプリケーション、ローカルのpg_wal、またはWALアーカイブ)から取得できない時に、スタンバイサーバがWALデータ受信をリトライするまでにどの位の時間待つべきかを指定します。このパラメータは、postgresql.confファイル、もしくはサーバコマンドラインでのみ設定可能です。この値が単位なしで指定された場合は、ミリ秒単位であるとみなします。デフォルト値は5秒です。このパラメータは、postgresql.confファイルで設定するか、サーバのコマンドラインでのみ指定できます。

このパラメータは、リカバリ対象のノードにおいて、新しいWALデータが読み込み可能になるまでの待ち時間を制御する必要のある時に有用です。たとえば、アーカイブリカバリにおいては、このパラメータの値を小さくすることにより、新しいWALログファイルを検出する際にリカバリの応答を早くすることができます。WALの生成頻度が少ないシステムでは、この値を大きくすることにより、WALアーカイブへのアクセス頻度を減らすことができます。これは、たとえば基盤へのアクセス回数が課金対象になるクラウド環境において、有用です。

recovery_min_apply_delay (integer)

デフォルトでは、スタンバイサーバは可能な限り早くプライマリからWALレコードをリストアします。時間遅れのデータのコピーを持つことで、データ損失エラーを修正する機会を提供するのは有用かもしれません。このパラメータを使う事で、決まった時間だけリカバリを遅らせることができます。例えば、パラメータに5minと指定した場合、各トランザクションについて、スタンバイのシステム時刻が、マスターから報告されたコミット時刻より5分以上経過している場合のみ、スタンバイサーバはコミットを再生します。単位を指定しない場合、ミリ秒として扱われます。デフォルトは0で、遅延を与えません。

サーバ間のレプリケーション遅延はパラメータの値を上回る可能性があり、その場合には遅延は追加されません。遅延は、マスタサーバで書かれたWALのタイムスタンプと、スタンバイサーバの現在時刻を使って計算されていることに注意してください。ネットワークの遅延やカスケーディングレプリケーション構成によるデータ転送の遅延は、実際の待ち時間を大幅に減らすかもしれません。もし、マスタサーバとスタンバイサーバのシステムクロックが同期されていない場合、期待値よりも早くレコードのリカバリを始めるかもしれません。しかし、このパラメータの有用な設定値は典型的なサーバ間の時間のずれよりもずっと大きいので、それらは大きな問題ではありません。

遅延はトランザクションコミットのWALレコードだけで発生します。他のレコードは可能な限り早く再生されるでしょう。対応する(トランザクション)コミットレコードが適用されるまではその効果が不可視であることがMVCCの可視ルールによって保証されているため、他のレコードが可能な限り早く再生されることは問題にはなりません。

ひとたびリカバリ中のデータベースが整合性のとれた状態になれば、スタンバイサーバが昇格またはトリガーになるまで、遅延が発生します。その後、スタンバイサーバはそれ以上待たずにリカバリを終了します。

このパラメータはストリーミングレプリケーション配信で使われることを目的としていますが、パラメータが指定されていると、クラッシュリカバリを除くすべてのケースで使用されます。この機能を使うことによってhot_standby_feedbackが遅延され、マスタサーバの肥大化に繋がる可能性があります。両方同時に使う場合には注意して使ってください。

警告

synchronous_commitがremote_applyに設定されていれば、同期レプリケーションは、この設定に影響を受けます。各COMMITは適用されるのを待つことが必要です。

このパラメータは、postgresql.confファイル、もしくはサーバコマンドラインでのみ設定可能です。

19.6.4. サブスクライバー

これらの設定項目は、論理レプリケーションのサブスクライバーの挙動を制御します。パブリッシャーにおける設定値とは無関係です。

wal_receiver_timeout、wal_receiver_status_intervalそしてwal_retrieve_retry_interval設定パラメータは、論理レプリケーションワーカーにも影響することに注意してください。

max_logical_replication_workers (int)

論理レプリケーションワーカーの最大数を指定します。適用ワーカーと、テーブル同期ワーカーの両方が含まれます。

論理レプリケーションワーカーは、max_worker_processesで定義されたプールから取得されます。

デフォルト値は4です。

max_sync_workers_per_subscription (integer)

サブスクリプションごとの同期ワーカーの最大数です。このパラメータは、サブスクリプションの初期化中、あるいは新しいテーブルが追加されたときの初期データコピーの並列度を制御します。

今のところ、一つのテーブルにつき、同期ワーカーは一つだけです。

同期ワーカーは、max_logical_replication_workersで定義されたプールから取得されます。

デフォルト値は2です。

19.7. 問い合わせ計画

19.7.1. プランナメソッド設定

以下の設定パラメータは、問い合わせオプティマイザが選択する問い合わせ計画に影響する大雑把な手法を提供します。ある問い合わせに対してオプティマイザが選択したデフォルト計画が最適でない場合、暫定的な解決策は、これらの設定パラメータの1つを使用し、オプティマイザに異なる計画を選択するように仕向けることです。オプティマイザが選択する計画の品質を改善するためのより良い方法には、プランナコスト定数を調節する(19.7.2を参照)、ANALYZEを手作業で実行する、default_statistics_target設定パラメータの値を大きくする、ALTER TABLE SET STATISTICSを使用して、特定の列に対して収集される統計情報を増やす、などがあります。

enable_bitmapscan (boolean)

問い合わせプランナがビットマップスキャン計画型を選択することを有効もしくは無効にします。デフォルトはonです。

enable_gathermerge (boolean)

問い合わせプランナがギャザーマージ計画型を選択することを有効もしくは無効にします。デフォルトはonです。

enable_hashagg (boolean)

問い合わせプランナがハッシュ集約計画型を選択することを有効もしくは無効にします。デフォルトはonです。

enable_hashjoin (boolean)

問い合わせプランナがハッシュ結合計画型を選択することを有効もしくは無効にします。デフォルトはonです。

enable_incremental_sort (boolean)

クエリプランナが増分ソートステップを使うのを有効あるいは無効にします。デフォルトはonです。

enable_indexscan (boolean)

問い合わせプランナがインデックス走査計画型を選択することを有効もしくは無効にします。デフォルトはonです。

enable_indexonlyscan (boolean)

問い合わせプランナがインデックスオンリースキャン計画型を選択することを有効もしくは無効にします。(11.9を参照してください) デフォルトはonです。

enable_material (boolean)

問い合わせプランナの具体化の使用を有効、または無効にします。全体にわたって具体化を差し止めることはできませんが、この値をoffにすることにより、正確性が要求される場合を除いて、具体化ノードをプランナが挿入することを防止します。デフォルトはonです。

enable_mergejoin (boolean)

問い合わせプランナがマージ結合計画型を選択することを有効もしくは無効にします。デフォルトはonです。

enable_nestloop (boolean)

問い合わせプランナがネステッドループ結合計画を選択することを有効もしくは無効にします。ネステッドループ結合を完全に禁止することは不可能ですが、この変数をオフにすると、もし他の方法が利用できるのであれば、プランナはその使用を行わないようになります。デフォルトはonです。

enable_parallel_append (boolean)

問い合わせププランナによるパラレル認識なアペンド計画型の利用を有効あるいは無効にします。デフォルトはonです。

enable_parallel_hash (boolean)

クエリププランナによる、パラレルハッシュを使ったハッシュジョイン計画型の利用を有効あるいは無効にします。デフォルトはonです。

enable_partition_pruning (boolean)

問い合わせププランナが、クエリプランからパーティション化テーブルのパーティションを除く機能を有効あるいは無効にします。これはまた、クエリエクゼキュータがクエリ実行中にパーティションを削除（無視）することができるプランを生成するプランナの機能を制御します。デフォルトはonです。詳細は5.11.4をご覧ください。

enable_partitionwise_join (boolean)

パーティション同士の結合(partitionwise join)を問い合わせプランナが使用するのを有効あるいは無効にします。パーティション同士の結合は、適合するパーティションを結合することによって実行される、パーティション化テーブルにまたがる結合を可能にします。今の所、パーティション同士の結合は、すべてのパーティションキーを含んでいて、かつパーティションキーの型が同じで、一対一で一致する子パーティションの集合を持つ結合条件のときだけに適用されます。パーティション同士の結合のプランニングには大量のCPU時間とメモリが使用されることがあるので、デフォルトはoffです。

enable_partitionwise_aggregate (boolean)

パーティション同士（partitionwise）のグループ化、あるいは集約を、問い合わせプランナが使用するのを有効あるいは無効にします。パーティション同士のグループ化、あるいは集約は、個々のパーティション毎にパーティション化されたテーブルに対して、グループ化あるいは集約を実行するのを可能にします。GROUP BY句がパーティションキーを含まない場合は、パーティション単位で部分集約だけが実行でき、最終結果の算出は後で行われます。パーティション同士のグループ化あるいは集約には大量のCPU時間とメモリが使用されることがあるので、デフォルトはoffです。

enable_seqscan (boolean)

問い合わせプランナがシーケンシャル走査計画を選択することを有効もしくは無効にします。シーケンシャル走査を完全に禁止することは不可能ですが、この変数をオフにすると、もし他の方法が利用できるのであれば、プランナはその使用を行わないようになります。デフォルトはonです。

enable_sort (boolean)

問い合わせプランナが明示的並び替え手順を選択することを有効もしくは無効にします。明示的並び替えを完全に禁止することは不可能ですが、この変数をオフにすると、もし他の方法が利用できるのであれば、プランナはその使用を行わないようになります。デフォルトはonです。

enable_tidscan (boolean)

問い合わせプランナがTID走査計画型を選択することを有効もしくは無効にします。デフォルトはonです。

19.7.2. プランナコスト定数

本節で扱うコスト変数は、任意の尺度で測られます。これらは相対的な値のみが意味を持つため、それらの値をすべて同じ係数で大きく、あるいは小さくしても、プランナの選択は結果として変わりません。デフォルトではこれらのコスト変数はシーケンシャルなページ取り込みに基づいています。つまり、seq_page_costを慣習的に1.0とし、他のコスト変数はそれを参考にして設定されています。しかし望むなら、特定のマシンにおけるミリ秒単位の実行時間など、異なる尺度を使用することができます。

> **注記**
>
> 残念ながら、コスト変数に対する理想的な値を決定する、上手く定義された方法がありません。特定のインストレーションが受け取る問い合わせ全体を混在させたものの平均として扱うのが最善でしょう。数回の実験のみを根拠にこの値を変更することは危険であるといえます。

seq_page_cost (floating point)

シーケンシャルな一連の取り出しの一部となる、ディスクページ取り出しに関する、プランナの推定コストを設定します。デフォルトは1.0です。この値は同じ名前のテーブル空間パラメータを設定することで、特定のテーブル空間の中にあるテーブルとインデックスに対して上書きできます（ALTER TABLESPACEを参照してください）。

random_page_cost (floating point)

非シーケンシャル的に取り出されるディスクページのコストに対するプランナの推測を設定します。デフォルトは4です。この値は同じ名前のテーブル空間パラメータを設定することで、特定のテーブル空間の中にあるテーブルとインデックスに対して上書きできます（ALTER TABLESPACEを参照してください）。

この値をseq_page_costと比較して小さくすると、システムはなるべくインデックススキャンを使用するようになります。大きくすると、インデックススキャンが相対的に高価になります。両方の値を増減させるこ

とで、CPUコストに対するディスクI/Oコストの重要性を変更させることができます。これについては、後述のパラメータで説明します。

機械的ディスク記憶装置に対するランダムアクセスは通常はシーケンシャルアクセスの4倍よりもかなり高価です。しかし、より低いデフォルト（4.0）が使用されます。というのはインデックスのついた読み取りのようなディスクに対するランダムアクセスのほとんどはキャッシュにあると想定されるからです。このデフォルト値は、ランダムアクセスがシーケンシャルアクセスより40倍遅い一方で、ランダム読み込みの90%はキャッシュされていることが期待されるというモデルとして考えることができます。

自環境の作業負荷において、90%のキャッシュ率は誤った仮定と考えられるのであれば、ランダム記憶装置読み込みのコストをより良く反映するため random_page_cost を大きくすることができます。反対に、データが完全にキャッシュされていると思われるのであれば、random_page_cost を小さくすることが適切です。例えば、データベースの容量がサーバのメモリより小さい場合などです。例えばSSDのような、シーケンシャルアクセスに比べてランダム読み込みコストがあまり大きくない記憶装置の場合も、random_page_cost に対し1.1のようにより低い値のモデル化の方が良いでしょう。

> **ヒント**
>
> システムはrandom_page_costをseq_page_costよりも小さな値に設定することを許しますが、そのようにすることは物理的にはおかしなことです。しかし、データベースが完全にRAMにキャッシュされる場合、同じ値に設定することは意味を持ちます。この場合、順序通りではないページアクセスに対するペナルティが存在しないからです。また、多くがキャッシュされるデータベースでは、CPUパラメータに対して両値を小さく設定すべきです。RAM内に存在するページの取り出しコストは通常よりもかなり小さくなるためです。

cpu_tuple_cost (floating point)

問い合わせ時のそれぞれの行の処理コストに対するプランナの推測を設定します。デフォルトは0.01です。

cpu_index_tuple_cost (floating point)

インデックス走査時のそれぞれのインデックス行の処理コストに対するプランナの推測を設定します。デフォルトは0.005です。

cpu_operator_cost (floating point)

問い合わせ時に実行される各演算子や関数の処理コストに対するプランナの推測を設定します。デフォルトは0.0025です。

parallel_setup_cost (floating point)

パラレルワーカープロセスを起動するためのコストに対するプランナの推測値を設定します。デフォルトは1000です。

parallel_tuple_cost (floating point)

あるパラレルワーカープロセスから、1行を他のプロセスに転送するためのコストに対するプランナの推測値を設定します。デフォルトは0.1です。

min_parallel_table_scan_size (integer)

パラレルスキャンを考慮する最小のテーブルデータのサイズを指定します。パラレル順スキャンでは、スキャンされるテーブルのデータ量は、常にテーブルのサイズと同じです。しかし、インデックスが使われる場合は、スキャンされるテーブルの量は通常少なくなるでしょう。この値が単位なしで指定された場合は、ブロック単位であるとみなします。すなわち、BLCKSZバイト、一般的には8kBです。デフォルトは8メガバイト(8MB)です。

min_parallel_index_scan_size (integer)

パラレルスキャンが考慮されるために、スキャンされなければならないインデックスデータの最小量を設定します。通常パラレルインデックススキャンは、典型的にはインデックス全体をアクセスしないことに注意してください。これは、関連するスキャンにより、プランナが実際にアクセスされると信じるページ数です。またこのパラメータは、あるインデックスがパラレルVACUUMで利用できるかどうかを決定するのにも使われます。VACUUMをご覧ください。この値が単位なしで指定された場合は、ブロック単位であるとみなします。すなわち、BLCKSZバイト、一般的には8kBです。デフォルトは512キロバイト(512kB)です。

effective_cache_size (integer)

単一の問い合わせで利用できるディスクキャッシュの実効容量に関するプランナの条件を設定します。これは、インデックスを使用するコスト推定値の要素となります。より高い値にすれば、よりインデックススキャンが使用されるようになり、より小さく設定すれば、シーケンシャルスキャンがより使用されるようになります。このパラメータを設定する時には、PostgreSQLの共有バッファとPostgreSQLデータファイルに使用されるカーネルのディスクキャッシュの量の両方を考慮しなければなりませんが、データは両方に存在することもあります。また、利用可能な領域を共有しますので、異なるテーブルに対して同時に実行される問い合わせの想定数も考慮してください。このパラメータは、PostgreSQLで割り当てられる共有メモリの大きさには影響を与えません。また、カーネルのディスクキャッシュを予約したりもしません。これは推定目的のみで使用されます。同時に、システムは問い合わせの間のディスクキャッシュ内のデータの残滓を想定していません。この値が単位なしで指定された場合は、ブロック単位であるとみなします。すなわち、BLCKSZバイト、一般的には8kBです。デフォルトは4ギガバイト(4GB)です。(BLCKSZが8kbでなければ、この設定のデフォルト値と最大値がBLCKSZに比例して変更されます。)

jit_above_cost (floating point)

JITが有効な場合(第31章参照)、それ以上ならJITコンパイルが起動する問い合わせコストを設定します。JITを実行するとプラン時間がかかりますが、問い合わせの実行を高速化することができます。これを-1にすると、JITコンパイルは無効になります。デフォルトは100000です。

jit_inline_above_cost (floating point)

それ以上ならJITコンパイルが関数と演算子のインライン化を試みる問い合わせコストを設定します。インライン化するとプラン時間がかかりますが、問い合わせの実行速度を改善できます。これをjit_above_costよりも小さくするのは意味がありません。これを-1にすると、インライン化は無効になります。デフォルトは500000です。

jit_optimize_above_cost (floating point)

それ以上ならJITコンパイルが高価な最適化を実行する問い合わせコストを設定します。そうした最適化にはプラン時間がかかりますが、問い合わせの実行速度を改善できます。これをjit_above_costより

も小さくするのは意味がなく、またjit_inline_above_costよりも大きくしても利益はないでしょう。これを-1にすると、高価な最適化は無効になります デフォルトは500000です。

19.7.3. 遺伝的問い合わせオプティマイザ

遺伝的問い合わせオプティマイザ（GEQO）はヒューリスティック（発見的）検索法を用いた問い合わせ計画を行なう演算手法です。通常のしらみつぶしの検索演算手法で見いだされる計画よりも時として劣った計画を作成するという代償を払いますが、この手法は（多くのリレーションを結合するような）複雑な問い合わせに対し計画時間を軽減します。より詳細は第59章を参照してください。

geqo (boolean)

遺伝的問い合わせ最適化を有効もしくは無効にします。デフォルトは有効です。運用時には無効にしないことが通常最善です。geqo_threshold変数は、GEQOを制御するためよりきめ細かな方法を提供します。

geqo_threshold (integer)

少なくともこれだけの数のFROM項目数があるときに、問い合わせを計画するのに遺伝的問い合わせ最適化を使用します。（FULL OUTER JOINの生成子は、FROM項目が1つだけとして計算することに注意してください。）デフォルトは12です。もっと単純な問い合わせでは、通常の、そしてしらみつぶしの検索プランナを使用するのが最善ですが、多くのテーブルを持つ問い合わせでは、しらみつぶしの検索は非常に時間がかかり、しばしば次善の計画を実行する代償より長くなります。従って、問い合わせの大きさに対する閾値はGEQOの使用を管理するのに便利な方法です。

geqo_effort (integer)

GEQOにおける計画時間と問い合わせ計画の品質間のトレードオフを制御します。この変数は1から10までの範囲の整数でなければなりません。デフォルトの値は5です。値を大きくすると、問い合わせ計画作成により多くの時間を費すことになりますが、より効率的な問い合わせ計画が選択される可能性が増加します。

実際geqo_effortは直接何も行いません。それはGEQOの動作に影響を与える他の変数に対し、デフォルトの値を計算するためにのみ使用されます（以下で説明します）。もしよければ、代わりに手作業で他のパラメータを設定できます。

geqo_pool_size (integer)

GEQOで使用されるプール容量を管理します。それは遺伝的個体群内の個体数です。最低でも2つはなければならず、よく100から1000までの値が使用されます。もし（デフォルトの設定である）零に設定されると、geqo_effortおよび問い合わせの中のテーブル数に基づいて、適切な値が選択されます。

geqo_generations (integer)

GEQOで使用される世代の数を管理します。それはアルゴリズムの反復数です。最低でも1はなければならず、よくプールサイズと同じ範囲の値が使用されます。これを0に設定（デフォルトの設定）すると、適切な値がgeqo_pool_sizeに基づいて選択されます。

geqo_selection_bias (floating point)

GEQOで使用される淘汰の偏りを管理します。淘汰の偏りは個体群内の（遺伝的な）自然淘汰です。値は1.50から2.00で、2.00がデフォルトです。

geqo_seed (floating point)

結合順序検索空間にわたって、GEQOが無作為のパスを選択するために使用される乱数発生器の初期値を制御します。値は0（デフォルト）から1までの範囲です。値を変動させると探査される結合パスの集合が変化するため、見つかる最善のパスが良くなる場合も悪くなる場合もあります。

19.7.4. その他のプランナオプション

default_statistics_target (integer)

ALTER TABLE SET STATISTICSで列特定の目的セットの無いテーブル列に対し、デフォルトの統計対象を設定します。より大きい値はANALYZEに必要な時間を増加させますが、プランナの予測の品質を向上させます。デフォルトは100です。PostgreSQLの問い合わせプランナによる統計情報の使用方法に関するより詳細な情報は、14.2を参照してください。

constraint_exclusion (enum)

問い合わせプランナが問い合わせを最適化する際のテーブル制約の使用を制御します。constraint_exclusionに許容される値は、on（全てのテーブルに対し制約を検査する）、off（決して制約を検査しない）、およびpartition（継承された子テーブルおよびUNION ALL副問い合わせのみ制約を検査する）です。partitionがデフォルトの設定です。伝統的な継承ツリーでしばしば性能向上のために使用されます。

このパラメータが特定のテーブルに対して許される時、プランナはそのテーブルのCHECK制約で問い合わせ条件を比較し、制約と矛盾する条件のテーブルの走査を省きます。例えば以下のようになります。

```
CREATE TABLE parent(key integer, ...);
CREATE TABLE child1000(check (key between 1000 and 1999)) INHERITS(parent);
CREATE TABLE child2000(check (key between 2000 and 2999)) INHERITS(parent);
...
SELECT * FROM parent WHERE key = 2400;
```

制約排除が有効であると、このSELECTは全くchild1000を走査せず、性能を向上させます。

現在、継承ツリーを使ってテーブルパーティショニングを実装するために使用されるよくある場合のことだけを考慮して、制約排除はデフォルトで有効です。すべてのテーブルに対して有効にするのは、単純な問い合わせにおいて特にはっきりわかる計画作成の余計なオーバーヘッドをもたらし、単純な問い合わせにはメリットがありません。伝統的な継承を使うパーティショニングされたテーブルがない場合、完全に無効にする方が良いでしょう。（パーティショニングされたテーブル用の同等の機能は別のパラメータenable_partition_pruningで制御します。）

パーティショニングを実装するための制約排除の利用についてのより進んだ情報は5.11.5を参照ください。

cursor_tuple_fraction (floating point)

検索されるカーソル行の割合のプランナの見積もりを設定します。デフォルトは0.1です。この設定をより小さくすると、プランナはカーソルに対し「起動を高速にする」計画を使用するようになりがちになります。

この場合先頭の数行の取り出しは高速になりますが、行全体を取り出す場合に時間がかかるようになる可能性があります。この値をより大きくすると、推定時間全体がより強調されるようになります。最大の設定である1.0の場合、カーソルは通常の問い合わせとまったく同様に計画されます。つまり、推定時間全体のみが考慮され、先頭の行の取り出しにかかる時間は考慮されなくなります。

from_collapse_limit (integer)

プランナは、FROMリストがこの数の項目より少ない結果の場合、副問い合わせを上位の問い合わせに併合します。より小さい値は計画時間を縮小させますが、劣った問い合わせ計画をもたらす可能性があります。デフォルトは8です。詳細は14.3を参照してください。

この値をgeqo_thresholdか、それ以上に設定するとGEQOプランナ使用の誘引となり、最適ではない計画をもたらします。19.7.3を参照してください。

jit (boolean)

PostgreSQLが、可能ならばJITコンパイルを使うかどうかを決定します(第31章を参照)。デフォルトはonです。

join_collapse_limit (integer)

最終的にリストがこの項目数以下になる時、プランナは、明示的なJOIN構文(FULL JOINを除く)をFROM項目のリストに直します。この値を小さくすれば計画作成時間は減少しますが、劣った問い合わせ計画が作成される可能性があります。

デフォルトでは、この値はfrom_collapse_limitと同じ値に設定されており、殆どの場合に適切です。これを1に設定すると明示的なJOINの再順序付けは行われなくなります。したがって、問い合わせで指定された明示的結合順序は、関係(リレーション)が結合される実際の順序となります。問い合わせプランナは常に最適な結合順序を選択するとは限らないので、上級ユーザなら暫定的にこの変数を1に設定し、明示的に希望とする結合順序を指定してもよいでしょう。詳細は14.3を参照してください。

この値をgeqo_thresholdか、それ以上に設定するとGEQOプランナ使用の誘引となり、最適ではない計画をもたらします。19.7.3を参照してください。

parallel_leader_participation (boolean)

ワーカープロセスを待つ代わりに、GatherノードとGather Mergeノード配下の問い合わせプランをリーダープロセスが実行できるようにします。デフォルトはonです。この値をoffにすると、リーダーがタプルを十分早く読まないためにワーカーがブロックされる可能性を減らすことができますが、リーダープロセスは最初のタプルが生成される前にワーカープロセスが起動するのを待つ必要があります。これがリーダーの性能を助けるのか、阻害要因になるかは計画型、ワーカーの数、問い合わせの長さによります。

force_parallel_mode (enum)

性能改善が期待できなくても、テスト目的のためにパラレルクエリを利用できるようにします。force_parallel_modeに設定できる値は、off(性能改善が期待できるときにだけパラレルクエリを使用する)、on(安全なクエリに対しては常にパラレルクエリを強制する)、regress(onと同様だが、下記のような振る舞いの変更を伴う)です。

正確に言えば、この値をonにすると、安全と見なされるすべての問い合わせ計画の上にGatherノードを追加し、クエリをパラレルワーカー上で実行するようにします。プランナがこれによってクエリが失敗する

と思わない限り、パラレルワーカーが利用できない、あるいは使用できないような場合でも、たとえばサブトランザクションの開始のように、パラレルクエリコンテキストでは許可されない操作は不許可となります。このオプションを設定することによって、エラーとなったり、あるいは期待していなかった結果がもたらされる場合には、クエリで使用されている関数はPARALLEL UNSAFE（もしくは、PARALLEL RESTRICTED）と印を付ける必要があるかもしません。

この設定値をregressとすると、onとするのに加え、自動リグレッションテストを助けるための付加的な効果が現れます。通常パラレルワーカーからのメッセージは、そのことを表すコンテキスト行を表示しますが、regressと設定すると、非パラレル実行と同じ出力になるように、これを抑止します。また、プランに追加されたGatherノードは、EXPLAIN出力から隠され、offに設定したときと同じ出力が得られるようにします。

plan_cache_mode (enum)

準備された文（明示的に準備されたものあるいはPL/pgSQLのように暗黙的に生成されたもののどちらにおいても）は、カスタムあるいは汎用(generic)プランで実行することができます。カスタムプランは実行ごとに指定されたパラメータ値の集合で新たに作られます。一方汎用プランはパラメータ値に依存せず、複数の実行にまたがって再利用できます。したがって、汎用プランはプランニングに要する時間を節約できますが、理想的なプランがパラメータ値に強く依存している場合は、汎用プランは非効率かも知れません。通常この両者の選択は自動的に行われますが、plan_cache_modeで上書きできます。可能な値はauto（デフォルト）、force_custom_plan、force_generic_planです。この設定は、準備するときではなく、キャッシュされたプランを実行する際に考慮されます。詳細はPREPAREをご覧ください。

19.8. エラー報告とログ取得

19.8.1. ログの出力先

log_destination (string)

PostgreSQLは、stderr、csvlogおよびsyslogを含めて、サーバメッセージのログ取得に対し数種類の方法を提供します。Windowsでは、eventlogも同時に提供します。このパラメータを設定するには、コンマ区切りでお好みのログ出力先を記載します。デフォルトでは、ログはstderrのみに出力されます。このパラメータはpostgresql.confファイル、または、サーバのコマンドラインでのみ設定されます。

csvlogがlog_destinationに含まれる場合、ログ項目はプログラムへの読み込みが簡便な「カンマ区切り値」書式（CSV）で出力されます。詳細は19.8.4を参照してください。CSV書式のログ出力を生成するためにはlogging_collectorを有効にする必要があります。

stderrあるいはcsvlogが含まれる場合、ログ収集機構が使用しているログファイルの場所と、その出力先を記録するcurrent_logfilesが作られます。これにより、現在のインスタンスが使っているログを簡単に見つけることができます。このファイルの内容の例を示します。

```
stderr log/postgresql.log
csvlog log/postgresql.csv
```

ログローテーションの効果で新しいログファイルが作られると、current_logfilesは再作成され、log_destinationは再ロードされます。このファイルは、stderrもcsvlogもlog_destinationに含まれなくなり、かつログ収集機構が無効になったときに削除されます。

> **注記**
>
> log_destinationでsyslogオプションを使用できるようにするために、ほとんどのUnixシステムではシステムのsyslogデーモンの設定を変更しなければならないでしょう。PostgreSQLではログをLOCAL0からLOCAL7までのsyslogファシリティで記録することができます（syslog_facilityを参照してください）。しかし、ほとんどのプラットフォームのデフォルトのsyslog設定ではこれらのメッセージはすべて破棄されます。うまく動作させるためにsyslogデーモンの設定に以下のようなものを追加しなければならないでしょう。
>
> ```
> local0.*    /var/log/postgresql
> ```
>
> Windowsでlog_destinationに対しeventlogオプションを使用する場合、Windows Event Viewer がイベントログメッセージを手際良く表示できるよう、オペレーティングシステムでイベントソースとそのライブラリを登録しなければなりません。詳細は18.12を参照ください。

logging_collector (boolean)

このパラメータはログ収集機構を有効にします。それはstderrに送られたログメッセージを捕捉し、ログファイルにリダイレクトするバックグラウンドプロセスです。この手法はsyslogへのログよりもしばしば有用です。メッセージの一部の種類がsyslogでは出力されない可能性があるためです。（一般的な例として、ダイナミックリンカのエラーメッセージがあり、その他の例としてarchive_commandのようなスクリプトにより生成されたエラーメッセージが挙げられます）。このパラメータはサーバ起動時のみ設定可能です。

> **注記**
>
> ログ収集機構を使用せずにstderrのログを取ることは可能です。ログメッセージはサーバのstderrが指し示すいかなる場所にも向かうだけです。しかし、その方法はログファイルを巡回させる都合のよい方法を提供しないので、ログ容量が小さい場合のみに適しています。同時に、ログ収集機構を使用しないいくつかのプラットフォームにおいては、ログ出力が失われたり、文字化けします。なぜなら、同一のログファイルに同時に書き込みを行うマルチプロセッサはそれぞれの出力を上書きできるからです。

> **注記**
>
> ログ収集機構はメッセージを決して失わないために設計されています。これは、極端に高い負荷の場合、サーバプロセスはコレクタが遅れをとった場合、追加のログメッセージを送信しようと試みる時に阻止される可能性があります。それとは対象的にsyslogは、もし書き込みができなかったときメッセージの廃棄を選びます。これらの場合にはいくつかのログメッセージを失うことになりますが、残ったシステムを阻止しません。

log_directory (string)

logging_collectorを有効と設定した場合、このパラメータはログファイルが作成されるディレクトリを確定します。ディレクトリは、絶対パス、もしくはデータベースクラスタのディレクトリに対する相対パスで指定することができます。このパラメータはpostgresql.confファイル、またはサーバコマンドラインからのみ設定可能です。デフォルトはlogです。

log_filename (string)

logging_collectorが有効な場合、このパラメータは作成されたログファイルのファイル名を設定します。値はstrftimeパターンとして扱われるため、%エスケープを使用して、時刻によって変動するファイル名を指定することができます。(時間帯に依存した%エスケープが存在する場合、log_timezoneで指定された時間帯で計算が行われます。) サポートされている%-エスケープはstrftime[1] 仕様によく似ています。システムのstrftimeは直接使用されないので、プラットフォーム固有の(非標準)の拡張は動作しません。デフォルトはpostgresql-%Y-%m-%d_%H%M%S.logです。

エスケープすることなくファイル名を指定する場合、ディスク全体を使い切ってしまうことを防止するためにログローテーションを行うユーティリティを使用することを計画しなければなりません。8.4より前のリリースのPostgreSQLでは、%エスケープがなければ、新しいログファイルの生成時のエポック時刻を付与しますが、これはもはや当てはまりません。

CSV書式の出力がlog_destinationで有効な場合、タイムスタンプ付きのログファイル名に.csvを付与し、最終的なCSV書式出力用のファイル名が作成されます。(log_filenameが.logで終わる場合は後置詞が置き換えられます。)

このパラメータはpostgresql.confファイルか、サーバコマンドラインのみ設定可能です。

log_file_mode (integer)

Unixシステムにおいては、logging_collectorが有効になっている場合、このパラメータはログファイルのパーミッションを設定します。(Microsoft Windowsではこのパラメータは無視されます。) パラメータの値はchmod および umaskシステムコールで許容されるフォーマットで指定される数値モードであると期待されます。(慣例的な8進数フォーマットを使用する場合、番号は0(ゼロ)で始まらなければなりません。

デフォルトのパーミッションは0600で、意味するところはサーバの所有者のみログファイルの読み書きが可能です。そのほか一般的に実用的な設定は0640で、所有者のグループはファイルを読み込めます。しかし、これらの設定を活用するにはlog_directoryがクラスタデータディレクトリの外部のどこかにあるファイルを格納できるように変更する必要があります。

このパラメータはpostgresql.confファイル、またはサーバコマンドラインからのみ設定可能です。

log_rotation_age (integer)

logging_collectorが有効な場合、このパラメータは個々のログファイルの最大寿命を決定します。ここで指定した時間経過すると、新しいログファイルが生成されます。この値が単位なしで指定された場合は、分単位であるとみなします。デフォルトは24時間です。ゼロに設定することで、時間に基づいた新しいログファイルの生成は無効になります。このパラメータは、postgresql.confファイル、または、サーバのコマンドラインでのみで設定されます。

[1] https://pubs.opengroup.org/onlinepubs/009695399/functions/strftime.html

log_rotation_size (integer)

logging_collectorが有効な場合、このパラメータは個々のログファイルの最大容量を決定します。ここで指定したデータ量がログファイルに出力された後、新しいログファイルが生成されます。この値が単位なしで指定された場合は、キロバイト単位であるとみなします。デフォルトは10メガバイトです。ゼロに設定することで、サイズに基づいた新しいログファイルの生成は無効になります。このパラメータはpostgresql.confファイル、または、サーバのコマンドラインでのみで設定されます。

log_truncate_on_rotation (boolean)

logging_collectorが有効な場合、このパラメータにより、PostgreSQLは既存の同名のファイルに追加するのではなく、そのファイルを切り詰める(上書きする)ようになります。しかし、切り詰めは時間を基にしたローテーションのために新規にファイルが開かれた時にのみ発生し、サーバ起動時やサイズを基にしたローテーションでは発生しません。偽の場合、全ての場合において既存のファイルは追記されます。例えば、この設定をpostgresql-%H.logのようなlog_filenameと組み合わせて使用すると、24個の時別のログファイルが生成され、それらは周期的に上書きされることになります。このパラメータはpostgresql.confファイル、または、サーバのコマンドラインで設定されます。

例:7日間のログを保存し、毎日のログを server_log.Mon、server_log.Tue、等とし、そして自動的に前週のログを今週のログで上書きするには以下のように設定します。log_filename を server_log.%aとし、log_truncate_on_rotation を onにし、そしてlog_rotation_age を 1440に設定します。

例:24時間のログを保持、1時間おきに1つのログファイルを作成、ただし、ログファイルのサイズが1ギガバイトを超えた場合それより早く切り替えさせるには、log_filenameをserver_log.%H%Mにし、log_truncate_on_rotationをonにし、log_rotation_ageを60にし、そしてlog_rotation_sizeを1000000に設定します。log_filenameに%Mを含めると、サイズを元にしたローテーションが時間毎の始めのファイル名とは異なる名前のファイルを選択するようにできます。

syslog_facility (enum)

syslogへのログ取得が有効な場合、このパラメータはsyslogの「facility」が使われるように確定します。LOCAL0、LOCAL1、LOCAL2、LOCAL3、LOCAL4、LOCAL5、LOCAL6、LOCAL7の中から選んでください。デフォルトはLOCAL0です。使用しているシステムのsyslogデーモンの文書を同時に参照してください。このパラメータはpostgresql.confファイル、または、サーバのコマンドラインでのみで設定されます。

syslog_ident (string)

syslogにログ取得が有効な場合、このパラメータはsyslogログ内のPostgreSQLメッセージを特定するのに使用するプログラム名を確定します。デフォルトはpostgresです。このパラメータは、postgresql.confファイル、または、サーバのコマンドラインでのみで設定されます。

syslog_sequence_numbers (boolean)

syslogにログを出力している場合で、これがオン(デフォルト)であると、各メッセージには([2]のような)増加する順序数が頭に追加されます。これにより、多くのsyslogの実装がデフォルトで行う「--- last message repeated N times ---」による出力の抑止が回避されます。より近代的なsyslogの実装では、繰り返されるメッセージの抑止は設定変更できるので(たとえば、rsyslog)における$RepeatedMsgReduction)、この機能は必要ないかもしれません。繰り返されるメッセージを抑止したい場合には、これをオフにできます。

このパラメータは、postgresql.confファイル、または、サーバのコマンドラインでのみで設定されます。

syslog_split_messages (boolean)

syslogへのログ出力が有効な場合、このパラメータはメッセージがどのようにsyslogに送られるかを規定します。オンなら(デフォルト)、メッセージは行に分割され、長い行は、伝統的なsyslog実装のサイズ上限である1024バイト以内に分割されます。オフならば、PostgreSQLサーバログメッセージは、そのままsyslogサービスに送られます。大きなサイズになるかもしれないメッセージにどう対応するかは、syslogサービス次第となります。

もしsyslogが最終的にテキストファイルにログを出力するのであれば、どちらに設定しても効果は同じです。設定値をオンにしておくのが最善です。多くのsyslogの実装では、長いメッセージを扱えないか、長いメッセージを扱うための特別な設定が必要だからです。しかし、syslogが最終的に他のメディアに書き込むのであれば、メッセージを論理的に一緒にしておくことが必要か、もしくは有用です。

このパラメータは、postgresql.confファイル、または、サーバのコマンドラインでのみで設定されます。

event_source (string)

event logへのログ取得が有効になっていると、このパラメータはログ中のPostgreSQLメッセージを特定するのに使用されるプログラム名を決定します。デフォルトはPostgreSQLです。このパラメータは、postgresql.confファイル、または、サーバのコマンドラインでのみで設定されます。

19.8.2. いつログを取得するか

log_min_messages (enum)

どのメッセージレベルをサーバログに書き込むかを管理します。有効な値はDEBUG5、DEBUG4、DEBUG3、DEBUG2、DEBUG1、INFO、NOTICE、WARNING、ERROR、LOG、FATAL、およびPANICです。それぞれの階層はその下の全ての階層を含みます。階層を低くする程、より少ないメッセージがログに送られます。デフォルトはWARNINGです。ここでのLOGの優先順位がclient_min_messagesの場合と異なることに注意してください。スーパーユーザのみこの設定を変更できます。

log_min_error_statement (enum)

エラー条件の原因となったどのSQL文をサーバログに記録するかを制御します。設定したレベル以上のメッセージについては現在のSQL文がログに記録されます。有効な値は、DEBUG5、DEBUG4、DEBUG3、DEBUG2、DEBUG1、INFO、NOTICE、WARNING、ERROR、LOG、FATAL、PANICです。デフォルトはERRORです。エラー、ログメッセージ、致命的エラー、パニックを引き起こした文がログに記録されることを意味します。失敗した文の記録を実質的に無効にするには、このパラメータをPANICに設定してください。スーパーユーザのみがこのオプションを変更することができます。

log_min_duration_statement (integer)

文の実行に少なくとも指定した時間かかった場合、それぞれの文の実行に要した時間をログに記録します。例えば、250msと設定した場合、250msもしくはそれ以上長くかかった全てのSQL文がログとして残ります。このパラメータを有効にすることにより、アプリケーションで最適化されていない問い合わせを追跡するのが便利になります。この値が単位なしで指定された場合は、ミリ秒単位であるとみなします。0

に設定すれば、すべての文の実行時間が出力されます。-1（デフォルト）は、文実行時間の記録を無効にします。スーパーユーザのみこの設定を変更できます。

これはlog_min_duration_sampleを上書きします。つまり、問い合わせがこの設定を越えると、実行時間のサンプリングの対象にならず、常に記録されます。

拡張問い合わせプロトコルを使用するクライアントでは、Parse、Bind、Executeそれぞれの段階で要した時間が独立して記録されます。

> **注記**
>
> このオプションとlog_statementを一緒に使用する時、log_statementによってログされるテキスト文は、実行時間のログには重複されません。syslogを使用していなければ、プロセスIDとセッションIDを使用して、文メッセージと後の実行時間メッセージを関連付けできるように、log_line_prefixを使用してPIDまたはセッションIDをログに記録することを勧めます。

log_min_duration_sample (integer)

指定した時間以上で実行完了した文の実行時間のサンプルを許可します。これによりlog_min_duration_statementと同様ですが、log_statement_sample_rateで制御するレートでサンプルされた実行時間のサブセットのログエントリを生成します。たとえば、100msに設定すると、100ミリ秒以上かかったSQL文がサンプルの対象となります。このパラメータを有効にすることで、トラフィックが多すぎてすべての問い合わせをログできない状況で助けになることもあります。この値を単位無しで指定すると、ミリ秒と見なされます。これをゼロに設定すると、すべての文の実行時間がサンプルされます。-1（デフォルトです）とすると、文の実行時間のサンプリングが無効になります。スーパーユーザのみがこの設定を変更できます。

この設定はlog_min_duration_statementよりも優先度が低いです。つまり、log_min_duration_statementを超えた実行時間の文はサンプリングの対象となり、常に記録されます。

log_min_duration_statementのその他の注意事項もこの設定に適用されます。

log_statement_sample_rate (floating point)

log_min_duration_sampleを越え、記録対象となる文の割合を決定します。サンプリングは確率論的で、たとえば0.5は2つの文のうちひとつが統計的に記録対象になることを意味します。デフォルトは1.0で、サンプルされた文はすべて記録対象となります。ゼロに設定すると、log_min_duration_sampleを-1にしたのと同じで、文の実行時間のサンプルを記録しません。スーパーユーザのみがこの設定を変更できます。

log_transaction_sample_rate (floating point)

他の理由に加え、トランザクションの文のうち、ログの対象となる割合を設定します。文の実行時間にかかわらず、新しいトランザクションに適用されます。サンプリングは確率論的で、たとえば0.1は10のトランザクションのうちひとつが統計的に記録対象になることを意味します。デフォルトは0で、これは追加のトランザクションのログを取らないことを意味します。1に設定すると、すべてのトランザクションのすべての文のログを取ります。log_transaction_sample_rateは、トランザクションのサンプルを調査するのに役立ちます。スーパーユーザのみこの設定を変更できます。

> **注記**
>
> 他の文のログを取るオプション同様、このオプションも大きなオーバヘッドを与える可能性があります。

表 19.2で、PostgreSQLで使用されるメッセージ深刻度レベルを説明します。ログ出力がsyslogまたはWindowsのeventlogに送られる場合、この深刻度レベルは表で示すように変換されます。

表19.2 メッセージ深刻度レベル

深刻度	使用方法	syslog	eventlog
DEBUG1 .. DEBUG5	開発者が使用する連続的かつより詳細な情報を提供します。	DEBUG	INFORMATION
INFO	VACUUM VERBOSEの出力などの、ユーザによって暗黙的に要求された情報を提供します。	INFO	INFORMATION
NOTICE	長い識別子の切り詰めに関する注意など、ユーザの補助になる情報を提供します。	NOTICE	INFORMATION
WARNING	トランザクションブロック外でのCOMMITの様な、ユーザへの警告を提供します。	NOTICE	WARNING
ERROR	現在のコマンドを中断させる原因となったエラーを報告します。	WARNING	ERROR
LOG	チェックポイントの活動の様な、管理者に関心のある情報を報告します。	INFO	INFORMATION
FATAL	現在のセッションを中断させる原因となったエラーを報告します。	ERR	ERROR
PANIC	全てのデータベースセッションを中断させる原因となったエラーを報告します。	CRIT	ERROR

19.8.3. 何をログに

application_name (string)

application_nameはNAMEDATALEN(標準構築では64)文字以下の任意の文字列を指定できます。通常はサーバへの接続時にアプリケーションによって設定されます。この名前は pg_stat_activityビューに表示され、CSVログに含まれます。またlog_line_prefixパラメータにより通常のログ項目に含めることができます。application_nameには表示可能なASCII文字のみ使用することができ、それ以外の文字は疑問符(?)に置換されます。

debug_print_parse (boolean)
debug_print_rewritten (boolean)
debug_print_plan (boolean)

これらのパラメータは生成される各種デバッグ出力を有効にします。設定すると実行された問い合わせそれぞれに対し、最終的な解析ツリー、問い合わせリライタの出力、実行計画を出力します。これらのメッセージはLOGメッセージレベルで出力されますので、デフォルトではサーバログに出力され、クライ

アントには渡されません。client_min_messages、log_min_messagesまたはその両方を調整することで変更することができます。デフォルトではこれらのパラメータは無効です。

debug_pretty_print (boolean)

設定された場合、debug_print_parse、debug_print_rewritten、または debug_print_planで生成されたメッセージを字下げします。設定されない場合の「コンパクト」形式よりもより見やすく、しかしより長いものとなります。デフォルトは有効です。

log_checkpoints (boolean)

チェックポイントおよびリスタートポイントをサーバログに記録するようにします。書き出されたバッファ数や書き出しに要した時間など、いくつかの統計情報がこのログメッセージに含まれます。このパラメータはpostgresql.confファイルまたはサーバのコマンドラインでのみ設定可能です。デフォルトはoffです。

log_connections (boolean)

これにより、クライアント認証の成功終了などのサーバへの接続試行がログに残ります。スーパーユーザだけがセッション開始時にこのパラメータを変更でき、セッションが開始された後は変更できません。デフォルトはoffです。

> **注記**
>
> psqlなどクライアントプログラムは、パスワードが要求されているかどうか確認するまで2回接続を試みるので、二重の「connection received」メッセージは必ずしも問題を示すものではありません。

log_disconnections (boolean)

セッションの終了をログします。ログ出力の情報はlog_connectionsと同様で、更にセッションの経過時間が追加されます。スーパーユーザだけがセッション開始時にこのパラメータを変更でき、セッションが開始された後は変更できません。デフォルトはoffです。

log_duration (boolean)

すべての完了した文について、その経過時間をログするようにします。デフォルトはoffです。スーパーユーザのみがこの設定を変更することができます。

拡張問い合わせプロトコルを使用するクライアントでは、Parse、Bind、Executeそれぞれの段階で要した時間が独立して記録されます。

> **注記**
>
> log_durationを有効にするのとlog_min_duration_statementを0に設定する方法との違いは、log_min_duration_statementを超えた場合、テキスト版の問い合わせが強制的に出力されるのに対して、このオプションでは出力されないという点です。したがって、log_durationがon、かつ、log_min_duration_statementが正の値を持つ場合、すべての経過時間がログに記録されますが、閾値を超えた文のみがテキスト版の問い合わせが含められるようになります。この動作は、高負荷なインストレーションで統計情報を収集する際に有用です。

log_error_verbosity (enum)

ログ取得されるそれぞれのメッセージに対し、サーバログに書き込まれる詳細の量を制御します。有効な値は、TERSE、DEFAULT、およびVERBOSEで、それぞれは表示されるメッセージにより多くのフィールドを追加します。TERSEはDETAIL、HINT、QUERY、およびCONTEXTエラー情報を除外します。VERBOSE出力は、SQLSTATEエラーコード(付録Aも参照)、および、ソースコードファイル名、関数名、そしてエラーを生成した行番号を含みます。スーパーユーザのみこの設定を変更できます。

log_hostname (boolean)

デフォルトでは、接続ログメッセージは接続元ホストのIPアドレスのみを表示します。このパラメータを有効にすると、ホスト名もログに残るようになります。ホスト名解決方法の設定に依存しますが、これが無視できないほどの性能劣化を起こす可能性があることに注意してください。このパラメータはpostgresql.confファイル内またはサーバのコマンドラインでのみ設定可能です。

log_line_prefix (string)

これは、各ログ行の先頭に出力するprintfの書式文字列です。%から始まる「エスケープシーケンス」は、後述の通りのステータス情報で置き換えられます。この他のエスケープは無視されます。他の文字はそのままログ行に出力されます。エスケープの中には、セッションプロセスによってのみ認識可能なものがあり、これらはメインサーバプロセスなどのバックグラウンドプロセスでは空文字として扱われます。状態情報はオプション名の%の後かつ前に数字を指定することにより、左寄せまた右寄せにすることができます。数字が負ならば状態情報を右側に空白を詰めて最小限の幅にし、正の値は左に空白を詰めます。ログファイルではパディングは人間の視認性を向上させるので有用です。

このパラメータは、postgresql.confファイル、または、サーバのコマンドラインでのみで設定することができます。デフォルトは、タイムスタンプとプロセスIDをログ出力する'%m [%p] 'です。

エスケープ	効果	セッションのみ
%a	アプリケーション名	○
%u	ユーザ名	○
%d	データベース名	○
%r	遠隔ホスト名、またはIPアドレス、およびポート番号	○
%h	遠隔ホスト名、またはIPアドレス	○
%b	バックエンドタイプ	×
%p	プロセス識別子	×
%t	ミリ秒無しのタイムスタンプ	×
%m	ミリ秒付きタイムスタンプ	×
%n	ミリ秒付きタイムスタンプ(Unixエポックとして)	×
%i	コマンドタグ。セッションの現在のコマンド種類	○
%e	SQLSTATE エラーコード	×
%c	セッションID。下記参照	×

エスケープ	効果	セッションのみ
%l	各セッションまたは各プロセスのログ行の番号。1から始まります。	×
%s	プロセスの開始タイムスタンプ	×
%v	仮想トランザクションID（backendID/localXID）	×
%x	トランザクションID（未割り当ての場合は0）	×
%q	何も出力しません。非セッションプロセスではこのエスケープ以降の出力を停止します。セッションプロセスでは無視されます。	×
%%	%文字そのもの	×

バックエンドタイプはpg_stat_activityビューのbackend_typeに関連します。しかし、ビューにない他のタイプがログに現れることがあります。

%cエスケープは、2つの4バイトの16進数（先頭のゼロは省略）をドットで区切った構成の、準一意なセッション識別子を表示します。この数値はプロセスの起動時間とそのプロセスIDです。したがって、%cを使用して、これらの項目を出力するための文字数を省略することができます。例として、pg_stat_activityからセッション識別子を生成するには以下の問い合わせを行ないます。

```
SELECT to_hex(trunc(EXTRACT(EPOCH FROM backend_start))::integer) || '.' ||
       to_hex(pid)
FROM pg_stat_activity;
```

ヒント

log_line_prefixに空白文字以外の値を設定する場合、通常、ログ行の残りとの区切りを明確にするために、その最後の文字を空白文字にすべきです。句読点用の文字も使用できます。

ヒント

Syslogは独自にタイムスタンプとプロセスID情報を生成します。ですのでおそらく、Syslogにログを保管する場合は、こうしたエスケープを含めるとは考えないでしょう。

ヒント

%qエスケープは、ユーザやデータベース名のように、セッション（バックエンド）コンテキストでのみ存在する情報を含める場合に有用です。%q

```
log_line_prefix = '%m [%p] %q%u@%d/%a '
```

log_lock_waits (boolean)

セッションがロックの獲得までの間にdeadlock_timeoutより長く待機する場合にログメッセージを生成するかどうかを制御します。これは、ロック待ちによって性能がでていないのかどうか確認する時に有用です。デフォルトはoffです。スーパーユーザのみこの設定を変更できます。

log_parameter_max_length (integer)

ゼロよりも大きければ、エラーではない時にバインドパラメータ値が、文とともにこの指定バイト数に短縮されて記録されます。ゼロなら、エラーではない時のバインドパラメータ値と文の記録は行われません。-1（デフォルトです）ならバインドパラメータはすべて記録されます。この値が単位なしに指定されると、バイト単位と見なされます。スーパーユーザのみがこの設定を変更できます。

この設定は、log_statement、log_durationおよび関連設定の結果を表示するログメッセージにのみ影響します。ゼロ以外の値の設定は、とりわけパラメータがバイナリ形式で送信される際に多少のオーバヘッドをもたらします。テキストへの変換が必要になるからです。

log_parameter_max_length_on_error (integer)

ゼロよりも大きければ、エラー時のバインドパラメータ値が、エラーメッセージ中にこの指定バイト数に短縮されて記録されます。ゼロ（デフォルトです）ならエラーメッセージ中のバインドパラメータは記録されません。-1ならバインドパラメータはすべて記録されます。この値が単位なしに指定されると、バイト単位と見なされます。

ゼロ以外の値の設定は多少のオーバヘッドをもたらします。エラーが起きるかどうかに関わらず、PostgreSQLは文を開始する際にテキスト形式のパラメータ値をメモリに保存する必要があるからです。パラメータがバイナリ形式で送信される場合にテキスト形式で送信するよりもオーバヘッドが大きくなります。前者はデータ変換が必要なのに対し、後者は単に文字列をコピーするだけだからです。

log_statement (enum)

どのSQL文をログに記録するかを制御します。有効な値は、none（off）、ddl、mod、およびall（全てのメッセージ）です。ddlは、CREATE、ALTER、およびDROP文といった、データ定義文を全てログに記録します。modは、全てのddl文に加え、INSERT、UPDATE、DELETE、TRUNCATE、およびCOPY FROMといった、データ変更文をログに記録します。PREPAREとEXPLAIN ANALYZEコマンドも、そこに含まれるコマンドが適切な種類であればログが録られます。拡張問い合わせプロトコルを使用するクライアントでは、Executeメッセージを受け取った時にBindパラメータの値が（すべての単一引用符が二重にされた状態で）含まれていた場合、ログに記録されます。

デフォルトはnoneです。スーパーユーザのみこの設定を変更できます。

> **注記**
>
> ログメッセージの発行は、基本解析により文の種類が決まった後に行われますので、log_statement = allという設定を行ったとしても、単純な構文エラーを持つ文は記録されません。拡張問い合わせプロトコルの場合も同様に、この設定ではExecute段階以前（つまり、解析や計画作成期間）に失敗した文は記録されません。こうした文のログを記録するには、log_min_error_statementをERROR（以下）に設定してください。

log_replication_commands (boolean)

サーバログにレプリケーションコマンドを記録します。レプリケーションコマンドの更なる情報は52.4をご覧ください。デフォルト値はoffです。スーパーユーザだけがこの設定を変更できます。

log_temp_files (integer)

一時ファイルのファイル名とサイズのログ出力を制御します。一時ファイルはソート処理やハッシュ処理、一時的な問い合わせの結果のために作成されます。有効にすると、ログの項目はすべての一時ファイルそれぞれについて削除されたときに生成されます。0を指定すると、すべての一時ファイル情報のログが残ります。正の数を指定すると、指定した以上の容量のファイルのみがログに残ります。この値が単位なしで指定された場合は、キロバイト単位であるとみなします。デフォルトの設定は-1で、このログ出力を無効にします。スーパーユーザのみがこの設定を変更できます。

log_timezone (string)

サーバログに書き出す際に使用される時間帯を設定します。TimeZoneと異なり、すべてのセッションで一貫性を持ってタイムスタンプが報告されるようにこの値はクラスタ全体に適用されます。組み込まれているデフォルトはGMTですが、postgresql.confにより通常は上書きされます。initdbによりこれらと関連した設定をシステム環境にインストールされます。詳細は8.5.3を参照してください。このパラメータはpostgresql.confファイル内またはサーバのコマンドラインでのみ設定することができます。

19.8.4. CSV書式のログ出力の利用

log_destinationリストにcsvlogを含めることは、ログファイルをデータベーステーブルにインポートする簡便な方法を提供します。このオプションはカンマ区切り値書式（CSV）で以下の列を含むログ行を生成します。ミリ秒単位のtimestamp、ユーザ名、データベース名、プロセス識別子、クライアントホスト：ポート番号、セッション識別子、セッション前行番号、コマンドタグ、セッション開始時間、仮想トランザクション識別子、通常トランザクション識別子、エラーの深刻度、SQL状態コード、エラーメッセージ、詳細エラーメッセージ、ヒント、エラーとなった内部的な問い合わせ（もしあれば）、内部問い合わせにおけるエラー位置の文字数、エラーの文脈、PostgreSQLソースコード上のエラー発生場所（log_error_verbosityがverboseに設定されているならば）アプリケーション名とバックエンドタイプ。以下にcsvlog出力を格納するためのテーブル定義のサンプルを示します。

```
CREATE TABLE postgres_log
(
  log_time timestamp(3) with time zone,
  user_name text,
  database_name text,
  process_id integer,
  connection_from text,
  session_id text,
  session_line_num bigint,
  command_tag text,
  session_start_time timestamp with time zone,
  virtual_transaction_id text,
  transaction_id bigint,
  error_severity text,
```

```
  sql_state_code text,
  message text,
  detail text,
  hint text,
  internal_query text,
  internal_query_pos integer,
  context text,
  query text,
  query_pos integer,
  location text,
  application_name text,
  backend_type text,
  PRIMARY KEY (session_id, session_line_num)
);
```

このテーブルにインポートするためには、COPY FROMコマンドを使用してください。

```
COPY postgres_log FROM '/full/path/to/logfile.csv' WITH csv;
```

提供されたfile_fdwモジュールを使って外部テーブルとしてファイルをアクセスすることも可能です。

CSVログファイルをインポートする作業を単純にするためにいくつか必要な作業があります。

1. 一貫性があり、予測可能なログファイル命名機構を提供するために、log_filenameおよびlog_rotation_ageを設定してください。これによりどのようなファイル名になると、個々のログファイルが完了しインポートする準備が整ったかが推測できるようになります。

2. ログファイル名の予測が困難になりますので、log_rotation_sizeを0にして容量を基にしたログの回転を無効にしてください。

3. 同じファイルに古いログデータと新しいログデータが混在しないようにするために、log_truncate_on_rotationをonに設定してください。

4. 上のテーブル定義には主キーの指定が含まれています。これにより、同じ情報が2回インポートされる事故を防止するために有用です。COPYコマンドは、一度にインポートするすべてのデータをコミットしますので、何か1つでもエラーがあればインポート全体が失敗します。ログファイルの一部をインポートし、そのファイルが完了した後に再度インポートしようとした場合、主キー違反によりインポートが失敗します。インポートする前に、ログファイルの完了を待ち、閉じるまで待機してください。この手順は、COPYが失敗する原因となる、完全に書き込まれなかった欠落した行をインポートするという事故も防止します。

19.8.5. プロセスの表題

これらの設定項目は、サーバプロセスの表題を変更します。プロセスの表題は、典型的にはps、あるいはWindowsにおいてはProcess Explorerで見ることができます。詳細については、27.1を参照してください。

cluster_name (string)

様々な目的のために、このデータベースクラスタ(インスタンス)を識別する名前を設定します。クラスタ名はこのクラスタのすべてのサーバプロセスのプロセス表題に表れます。更に、スタンバイ接続のデフォルトのアプリケーション名となります。(synchronous_standby_names参照。)

クラスタ名はNAMEDATALEN文字(標準のビルドでは64文字)より少ない文字列です。表示可能なASCII文字だけがcluster_nameの値として設定できます。他の文字は、疑問符(?)に置き換えられます。空文字''(これがデフォルト値です)が設定されると、クラスタ名は表示されません。このパラメータはサーバ起動時にのみ設定できます。

update_process_title (boolean)

サーバが新しいSQLコマンドを受け取る時に毎回、プロセスタイトルを更新できるようにします。この設定値はたいていのプラットフォームでonがデフォルトになっていますが、Windowsではプロセスタイトルを更新するオーバヘッドが大きいため、offがデフォルトになっています。スーパーユーザのみがこの設定を変更することができます。

19.9. 実行時統計情報

19.9.1. 問い合わせおよびインデックスに関する統計情報コレクタ

これらのパラメータは、サーバ全体の統計情報収集機能を制御します。統計情報収集が有効ならば、生成されるデータはpg_statとpg_statio系のシステムビュー経由でアクセス可能です。詳細は第27章を参照してください。

track_activities (boolean)

各セッションで実行中のコマンドに関する情報とそのコマンドの実行開始時刻の収集を有効にします。このパラメータはデフォルトで有効です。有効な場合であっても、すべてのユーザがこの情報を見ることができず、スーパーユーザと報告されたセッションの所有者のみから可視である点に注意してください。このためセキュリティ上の危険性はありません。スーパーユーザのみがこの設定を変更することができます。

track_activity_query_size (integer)

pg_stat_activity.queryフィールドに対し、それぞれの活動中のセッションで現在実行されているコマンドを追跡記録するため予約されるメモリ量を指定します。この値が単位なしで指定された場合は、バイト単位であるとみなします。デフォルトの値は1024バイトです。このパラメータはサーバ起動時のみ設定可能です。

track_counts (boolean)

データベースの活動についての統計情報の収集を有効にします。収集される情報を自動バキュームデーモンが必要とするため、このオプションはデフォルトで有効です。スーパーユーザのみがこの設定を変更することができます。

track_io_timing (boolean)

データベースによるI/O呼び出しの記録を有効にします。このパラメータはデフォルトで無効になっています。その理由は、現時点の時刻をオペレーティングシステムに繰り返し問い合わせるので、

プラットフォームによっては深刻な負荷の原因になるからです。使用しているシステムにおける負荷のタイミングを計測するためpg_test_timingツールが使用できます。I/O呼び出し情報は、pg_stat_databaseに表示されます。BUFFERSオプションが設定されている時はEXPLAIN の出力として、またpg_stat_statementsにより表示されます。スーパーユーザのみこの設定を変更できます。

track_functions (enum)

関数の呼び出し数と費やされた時間の追跡を有効にします。手続き言語関数のみを追跡するためにはplと指定してください。SQL関数、C言語関数も追跡するためにはallと指定してください。デフォルトは、統計情報追跡機能を無効にするnoneです。スーパーユーザのみがこの設定を変更できます。

> **注記**
>
> 呼び出す問い合わせ内に「インライン化」できる位単純なSQL言語関数は、この設定と関係なく、追跡されません。

stats_temp_directory (string)

統計情報データを一時的に格納するディレクトリを設定します。これをデータディレクトリからの相対パスとすることも絶対パスとすることもできます。デフォルトはpg_stat_tmpです。これをRAMベースのファイルシステムを指し示すようにすることで物理I/O要求が減り、性能を向上させることができます。このパラメータは、postgresql.confファイルまたはサーバのコマンドラインのみで設定可能です。

19.9.2. 統計情報の監視

log_statement_stats (boolean)
log_parser_stats (boolean)
log_planner_stats (boolean)
log_executor_stats (boolean)

各問い合わせに対し、対応するモジュールの性能に関する統計情報をサーバログに出力します。これは、Unixのgetrusage()オペレーティングシステム機能に類似した、雑なプロファイリング手段です。log_statement_statsは文に関する統計情報全体を、この他はモジュール毎の統計情報を報告します。log_statement_statsとモジュール毎のオプションを一緒に有効にすることはできません。デフォルトでこれらのオプションはすべて無効です。スーパーユーザのみがこの設定を変更することができます。

19.10. 自動Vacuum作業

以下に示す設定は自動バキューム機能の動作を制御します。詳細は24.1.6を参照してください。これらの設定の多くは、テーブル単位で変更できることに注意してください。Storage Parametersを参照してください。

autovacuum (boolean)

サーバがautovacuumランチャデーモンを実行すべきかどうかを管理します。デフォルトでは有効です。しかしautovacuumを作動させるためにはtrack_countsも有効でなければなりません。このパラメータはpostgresql.confファイル、または、サーバのコマンドラインでのみで設定されます。ただし、テー

ブルストレージパラメータを変更することにより、autovacuumは個々のテーブルに対して無効にできます。

このパラメータが無効であったとしてもシステムは、トランザクションIDの周回を防止する必要があれば、autovacuumプロセスを起動することに注意してください。詳細は24.1.5を参照してください。

log_autovacuum_min_duration (integer)

少なくとも指定時間実行した場合、autovacuumで実行される各活動がログに残るようになります。これをゼロに設定すると、すべてのautovacuumの活動がログに残ります。-1（デフォルト）はautovacuum活動のログを無効にします。この値が単位なしで指定された場合は、ミリ秒単位であるとみなします。例えば、これを250msに設定すると、250ms以上かかって実行されたautovacuumや解析はすべてログに残ります。さらに、-1以外の値にこのパラメータが設定された場合、競合するロックや並行して削除されたリレーションによりautovacuum動作が省略されるとメッセージはログに記録されます。このパラメータを有効にすることは、autovacuum活動の追跡に役に立ちます。このパラメータはpostgresql.confファイル、または、サーバのコマンドラインでのみで設定されます。ただし、この設定はテーブルストレージパラメータの変更により、それぞれのテーブルに対して上書きすることができます。

autovacuum_max_workers (integer)

同時に実行することができるautovacuumプロセス（autovacuumランチャ以外）の最大数を指定します。デフォルトは3です。サーバ起動時のみで設定可能です。

autovacuum_naptime (integer)

あるデータベースについて実行されるautovacuumデーモンの最小遅延を指定します。それぞれの周期で、デーモンはそのデータベースを試験し、そしてそのデータベース内のテーブルで必要性が認められると、VACUUMおよびANALYZEコマンドを発行します。この値が単位なしで指定された場合は、秒単位であるとみなします。デフォルトは1分（1min）です。このパラメータはpostgresql.confファイル、または、サーバのコマンドラインでのみで設定されます。

autovacuum_vacuum_threshold (integer)

どのテーブルに対してもVACUUMを起動するために必要な、更新もしくは削除されたタプルの最小数を指定します。デフォルトは50タプルです。このパラメータはpostgresql.confファイル、または、サーバのコマンドラインでのみで設定されます。ただし、この設定はテーブルストレージパラメータの変更により、それぞれのテーブルに対して上書きすることができます。

autovacuum_vacuum_insert_threshold (integer)

あるテーブルでVACUUMを起動するきっかけとなるのに必要な挿入タプル数を設定します。デフォルトは1000タプルです。-1が指定されると、自動VACUUMが挿入タプル数に基づいてVACUUM操作を引き起こすことはなくなります。このパラメータはpostgresql.confファイル、または、サーバのコマンドラインでのみで設定されます。ただし、この設定はテーブルストレージパラメータの変更により、それぞれのテーブルに対して上書きすることができます。

autovacuum_analyze_threshold (integer)

どのテーブルに対してもANALYZEを起動するのに必要な、挿入、更新、もしくは削除されたタプルの最小数を指定します。デフォルトは50タプルです。このパラメータはpostgresql.confファイル、または、

サーバのコマンドラインでのみで設定されます。この設定はテーブルストレージパラメータの変更により、それぞれのテーブルに対して上書きすることができます。

autovacuum_vacuum_scale_factor (floating point)

VACUUMを起動するか否かを決定するときに、autovacuum_vacuum_thresholdに足し算するテーブル容量の割合を指定します。デフォルトは0.2（テーブルサイズの20%）です。このパラメータはpostgresql.confファイル、または、サーバのコマンドラインでのみで設定されますが、テーブルストレージパラメータの変更により、それぞれのテーブルに対して上書きすることができます。

autovacuum_vacuum_insert_scale_factor (floating point)

VACUUMを起動するかどうか決める際のautovacuum_vacuum_insert_thresholdに追加するテーブルサイズの割合を指定します。デフォルトは0.2（テーブルサイズの20%）です。このパラメータはpostgresql.confファイル、または、サーバのコマンドラインでのみで設定されますが、テーブルストレージパラメータの変更により、それぞれのテーブルに対して上書きすることができます。

autovacuum_analyze_scale_factor (floating point)

ANALYZEを起動するか否かを決定するときに、autovacuum_analyze_thresholdに足し算するテーブル容量の割合を指定します。デフォルトは0.1（テーブルサイズの10%）です。このパラメータはpostgresql.confファイル、または、サーバのコマンドラインでのみで設定されます。この設定はテーブルストレージパラメータの変更により、それぞれのテーブルに対して上書きされます。

autovacuum_freeze_max_age (integer)

トランザクションID周回を防ぐためにVACUUM操作が強制される前までにテーブルのpg_class.relfrozenxid フィールドが到達できる最大（トランザクションにおける）年代を指定します。自動バキュームが無効であった時でも、システムは周回を防ぐために自動バキューム子プロセスを起動することに注意してください。

vacuumは同時にpg_xactサブディレクトリから古いファイルの削除を許可します。これが、比較的低い2億トランザクションがデフォルトである理由です。このパラメータはサーバ起動時にのみ設定可能です。しかし、この設定はテーブルストレージパラメータの変更により、それぞれのテーブルで減らすことができます。詳細は24.1.5を参照してください。

autovacuum_multixact_freeze_max_age (integer)

トランザクションID周回を防ぐためにVACUUM操作が強制される前までにテーブルのpg_class.relminmxid フィールドが到達できる最大（マルチトランザクションにおける）年代を指定します。自動バキュームが無効であった時でも、システムは周回を防ぐために自動バキューム子プロセスを起動することに注意してください。

またマルチトランザクションIDのvacuumはpg_multixactとpg_multixact/offsetsサブディレクトリから古いファイルの削除します。これがデフォルトが4億トランザクションをやや下回る理由です。このパラメータはサーバ起動時にのみ設定可能です。しかし、この設定はテーブルストレージパラメータの変更により、それぞれのテーブルで減らすことができます。詳細は24.1.5.1を参照してください。

autovacuum_vacuum_cost_delay (floating point)

自動VACUUM操作に使用されるコスト遅延値を指定します。-1に指定されると、一定のvacuum_cost_delayの値が使用されます。この値が単位なしで指定された場合は、ミリ秒単位である

とみなします。デフォルト値は2ミリ秒です。このパラメータはpostgresql.confファイル内、または、サーバのコマンドラインのみで設定可能です。この設定はテーブルストレージパラメータの変更により、それぞれのテーブルに対して上書きすることができます。

autovacuum_vacuum_cost_limit (integer)

自動VACUUM操作に使用されるコスト限界値を指定します。(デフォルトの)-1が指定されると、一定のvacuum_cost_limitの値が使用されます。この値は、実行中の自動バキュームワーカが複数存在する場合ワーカすべてに比例分配されることに注意してください。したがって各ワーカの制限を足し合わせてもこの変数による制限を超えることはありません。このパラメータはpostgresql.confファイル、または、サーバのコマンドラインのみで設定可能です。この設定はテーブルストレージパラメータの変更により、それぞれのテーブルに対して上書きすることができます。

19.11. クライアント接続デフォルト

19.11.1. 文の動作

client_min_messages (enum)

どのメッセージレベルをクライアントに送るかを管理します。有効な値は、DEBUG5、DEBUG4、DEBUG3、DEBUG2、DEBUG1、LOG、NOTICE、WARNINGおよびERRORです。それぞれのレベルはそれに続く全てのレベルを包含します。レベルが後の方になるにつれ、より少ないメッセージが送られます。デフォルトはNOTICEです。ここでのLOGの優先順位がlog_min_messagesの場合と異なることに注意してください。

INFOレベルのメッセージは常にクライアントに送られます。

search_path (string)

この変数は、オブジェクト(テーブル、データ型、関数など)がスキーマを指定されていない単純な名前で参照されている場合に、スキーマを検索する順番を指定します。異なるスキーマに同じ名前のオブジェクトがある場合、検索パスで最初に見つかったものが使用されます。検索パス内のどのスキーマにも存在しないオブジェクトを参照するには、修飾名(ドット付き)でそのオブジェクトが含まれるスキーマを指定する必要があります。

search_pathの値は、スキーマの名前をカンマで区切った一覧でなければなりません。存在していないスキーマ、またはユーザがUSAGE権限を所有していないスキーマは警告なしに無視されます。

もしそのようなスキーマが存在し、ユーザがそれにたいしてUSAGE権限を所有している場合、一覧内の項目の1つが特別な名前である$userの場合、CURRENT_USERと同じ名前を持つスキーマがあれば、そのスキーマが置換されます。(このような名前空間がない場合は$userは無視されます。)

システムカタログのスキーマであるpg_catalogは、パスでの指定の有無にかかわらず、常に検索されます。パスで指定されている場合は、指定された順序で検索されます。pg_catalogがパスに含まれていない場合、パスに含まれる項目を検索する前に検索が行われます

同様に、現在のセッションの一時テーブルスキーマpg_temp_nnnも、存在すれば常に検索されます。これはpg_tempという別名を使用してパスに明示的に列挙させることができます。パスに列挙されていない場合、最初に(pg_catalogよりも前であっても)検索されます。しかし、一時スキーマはリレーション(テー

ブル、ビュー、シーケンスなど)とデータ型名に対してのみ検索されます。関数や演算子名に対してはまったく検索されません。

対象となる特定のスキーマを指定せずにオブジェクトが作成された場合、それらのオブジェクトはsearch_pathで名前を付けられた最初に有効となっているスキーマに配置されます。検索パスが空の場合、エラーが報告されます。

このパラメータのデフォルト値は"$user", publicです。この設定はデータベースの共有(どのユーザも非公開のスキーマを持たず、全員がpublicを共有)、ユーザごとの非公開のスキーマ、およびこれらの組み合わせがサポートします。デフォルトの検索パスの設定を全体的またはユーザごとに変更することで、その他の効果を得ることもできます。

スキーマの扱いについての詳細は、5.9をご覧ください。とりわけ、デフォルトの設定はデータベースのユーザが、一人あるいはお互いに信頼できる少数のユーザだけである場合にのみ適切です。

SQL関数のcurrent_schemas によって、検索パスの現在の有効な値を調べることができます(9.26を参照してください)。これは、search_path の値を調べるのとは異なります。current_schemasは、search_pathに現れる項目がどのように解決されたかを表すからです。

row_security (boolean)

この設定値は、行セキュリティポリシーの適用によってエラーを生じさせるかどうかを制御します。onに設定すると、通常通りポリシーが適用されます。offにすると、少なくともひとつのポリシーが適用されたクエリは失敗します。デフォルトはonです。行の可視性が制限されている場合、offにすると不正な結果を招くことがあります。たとえば、pg_dumpはデフォルトでoffにしています。この設定値は、行セキュリティポリシーを迂回するロールには効果がありません。それはすなわち、BYPASSRLSアトリビュートを持つスーパーユーザです。

行セキュリティポリシーについての更なる情報はCREATE POLICYをご覧ください。

default_table_access_method (string)

このパラメータは、テーブルあるいはマテリアライズドビューの作成時にCREATEコマンドが明示的にアクセスメソッドを指定しないか、あるいはテーブルアクセスメソッドを指定できないSELECT ... INTOが利用された時に使われるデフォルトのテーブルアクセスメソッドを指定します。デフォルトはheapです。

default_tablespace (string)

この変数は、CREATEコマンドで明示的にテーブル空間を指定していない場合にオブジェクトの作成先となるデフォルトのテーブル空間を指定します。また、パーティション化されたリレーションが将来パーティションを格納するテーブルスペースを決定します。

値はテーブル空間名、もしくは現在のデータベースのデフォルトのテーブル空間を使用することを意味する空文字列です。この値が既存のテーブル空間名と一致しない場合、PostgreSQLは自動的に現在のデータベースのデフォルトのテーブル空間を使用します。デフォルト以外のテーブル空間が指定された場合、ユーザはそのテーブル空間でCREATE権限を持たなければなりません。さもなくば作成に失敗します。

この変数は一時テーブル向けには使用されません。一時テーブル向けには代わりにtemp_tablespacesが考慮されます。

同時に、この変数はデータベース作成時には使用されません。

テーブル空間に付いてより詳細な情報は22.6を参照してください。

temp_tablespaces (string)

この変数は、CREATEコマンドで明示的にテーブル空間が指定されない場合に、生成する一時オブジェクト(一時テーブルと一時テーブル上のインデックス)を格納するテーブル空間(複数可)を指定します。大規模データ集合のソートなどを目的とした一時ファイルもまた、このテーブル空間(複数可)に作成されます。

この値はテーブル空間名のリストです。リストに複数の名前が存在する場合、一時オブジェクトが作成される度にPostgreSQLは無作為にリストから要素を選択します。トランザクションの内側は例外で、連続して作成される一時オブジェクトはそのリストで連続するテーブル空間に格納されます。リスト内の選択された要素が空文字列だった場合、PostgreSQLは自動的に現在のデータベースのデフォルトのテーブル空間を代わりに使用します。

temp_tablespacesを対話式に設定する場合、存在しないテーブル空間を指定するとエラーになります。ユーザがCREATE権限を持たないテーブル空間を指定した場合も同様です。しかし事前に設定された値を使用する場合、存在しないテーブル空間は無視されます。ユーザがCREATE権限を持たないテーブル空間も同様です。具体的には、この規則はpostgresql.conf内で設定した値を使用する場合に適用されます。

デフォルト値は空文字列です。この結果、すべての一時オブジェクトは現在のデータベースのデフォルトのテーブル空間内に作成されます。

default_tablespaceも参照してください

check_function_bodies (boolean)

このパラメータは通常オンです。offに設定すると、CREATE FUNCTIONの間で関数本体文字列の妥当性検証を無効にします。妥当性検証を無効にするとその妥当性検証処理の副作用を避け、前方参照による問題から起こる偽陽性(false positive)を避けることができます。関数をロードする前にこのパラメータを他のユーザとしてoffにします。pg_dumpはこれを自動的に行います。

default_transaction_isolation (enum)

SQLトランザクションはそれぞれ、「read uncommitted」、「read committed」、「repeatable read」、または「serializable」のいずれかの分離レベルを持ちます。このパラメータは各新規トランザクションのデフォルトの分離レベルを制御します。デフォルトは「read committed」です。

より詳細は 第13章 および SET TRANSACTION を調べてください。

default_transaction_read_only (boolean)

読み取り専用のSQLトランザクションでは、非一時的テーブルを変更することができません。このパラメータは、各新規トランザクションのデフォルトの読み取りのみ状況を制御します。デフォルトoff(読み書き)です。

より詳細な情報はSET TRANSACTIONを調べてください。

default_transaction_deferrable (boolean)

シリアライザブル分離レベルで運用されている場合、繰り延べ読み取り専用SQLトランザクションは、その処理の許可の前に遅延されることがあります。しかし、ひとたび処理が開始されるとシリアライザブ

ル可能性を保障するために必要ないかなるオーバヘッドも発生させません。従って、シリアル化（直列化）のコードは、このオプションを長期間にわたる読み取り専用トランザクションに対して適切な処置と位置づけ、同時実行の更新の観点から中断を強制する理由はありません。

このパラメータはそれぞれの新規トランザクションのデフォルトでの繰り延べ状態を制御します。現時点では、読み取り専用トランザクション、またはシリアライザブルより低位の分離レベルの運用に対して効果はありません。

より詳細はSET TRANSACTIONを参照してください。

session_replication_role (enum)

現在のセッションでのレプリケーションに関連したトリガおよびルールの発行を制御します。この変数を設定するにはスーパーユーザ権限が必要で、かつ、これまでにキャッシュされた問い合わせ計画が破棄されることになります。取り得る値は、origin（デフォルト）、replica、localです。

この設定の使い方の趣旨としては、レプリケーションされた更新を適用する際にロジカルレプリケーションシステムがreplicaに設定するということです。このことによる効果としては、（デフォルトの設定から変更されていない）トリガとルールはレプリカ上では起動されない、ということです。更なる情報は、ALTER TABLE節のENABLE TRIGGERとENABLE RULEをご覧ください。

PostgreSQLはoriginとlocalの設定を内部的に同じものとして扱います。サードパーティのレプリケーションシステムは内部的な目的、たとえばlocalを使ってレプリケーションされるべきでないセッションを指定するためにこれら2つの値を使って構いません。

外部キーはトリガとして実装されているため、このパラメータをreplicaとすることによって同時にすべての外部キー検査が無効になります。このことにより、正しく使用しないと、データを不整合状態にしてしまう可能性があります。

statement_timeout (integer)

コマンドがクライアントからサーバに届いた時から数えて、実行時間が指定された時間を越えた文を停止します。log_min_error_statementがERRORもしくはそれ以下に設定されている場合は、タイムアウトした文はログに書き込まれます。この値が単位なしで指定された場合は、ミリ秒単位であるとみなします。値がゼロ（デフォルト）の場合、これを無効にします。

タイムアウトは、コマンドがサーバに到着したときから、サーバがそのコマンドの実行を完了するまでを基準にします。複数のSQL文が一つの単純問い合わせメッセージに含まれる場合、タイムアウトは個々のSQL文に別々に適用されます。（13よりも前のPostgreSQLバージョンでは、通常SQL文字列全体に対してタイムアウトが適用されていました。）拡張問い合わせでは、タイムアウトは問い合わせに関するメッセージ（Parse、Bind、Execute、Describe）が到着したときに開始し、ExecuteあるいはSyncメッセージが完了した時にキャンセルされます。

すべてのセッションに影響することがあるので、postgresql.conf内でstatement_timeoutを設定することは推奨されません。

lock_timeout (integer)

テーブル、インデックス、行、またはその他のデータベースオブジェクトに対してロック獲得を試みている最中、指定された時間を超えて待機するいかなる命令も停止されます。時間制限はそれぞれのロッ

ク取得の試みに対し個別に適用されます。制限は明示的ロック要求（例えばLOCK TABLE、またはSELECT FOR UPDATE without NOWAITなど）および暗黙的に取得されるロックに適用されます。この値が単位なしで指定された場合は、ミリ秒単位であるとみなします。値ゼロ（デフォルト）はこの機能を無効にします。

statement_timeoutと異なり、このタイムアウトはロックを待機しているときのみ発生します。命令によるタイムアウトは常に第一に起動されるため、もしstatement_timeoutが非ゼロであればlock_timeoutを同一、もしくはより大きい値に設定するのは的を射ていません。log_min_error_statementがERRORまたはそれより低く設定されると、時間制限を超えた命令はログに記録されます。

lock_timeoutをpostgresql.confにて設定することは、すべてのセッションに影響を与える可能性があるため推奨されません。

idle_in_transaction_session_timeout (integer)

開いているトランザクションが、指定された時間を超えてアイドルだった場合に、セッションを終了します。これにより、そのセッションが獲得したロックを解放し、コネクションスロットを再利用できるようになります。また、このトランザクションからのみ見えるタプルがVACUUMできるようになります。更なる詳細は24.1を見てください。

この値が単位なしで指定された場合は、ミリ秒単位であるとみなします。値がゼロ（デフォルト）の場合、この機能は無効になります。

vacuum_freeze_table_age (integer)

テーブルのpg_class.relfrozenxidフィールドがこの設定で指定した時期に達すると、VACUUMは積極的なテーブル走査を行います。積極的な走査は、無効タプルを含む可能性のあるページだけではなく、凍結されていないXIDあるいはMXIDを含むすべてのページを読む点で通常のVACUUMとは異なります。デフォルトは1.5億トランザクションです。ユーザはこの値をゼロから20億までの任意の値に設定することができますが、VACUUMは警告することなく、周回問題対策のautovacuumがテーブルに対して起動する前に定期的な手動VACUUMが実行する機会を持つように、autovacuum_freeze_max_ageの95%に実効値を制限します。詳細は24.1.5を参照してください。

vacuum_freeze_min_age (integer)

VACUUMがテーブルスキャン時に行バージョンをフリーズするかどうかを決定する際に使用する、カットオフ（トランザクション）年代を指定します。デフォルトは5千万トランザクションです。ユーザはこの値を0から10億までの間で任意の値に設定することができますが、VACUUMは警告なくautovacuum_freeze_max_ageの半分までの値に値を制限します。このため、強制的なautovacuumの間隔が不合理に短くなることはありません。詳細は24.1.5を参照してください。

vacuum_multixact_freeze_table_age (integer)

pg_class.relminmxidフィールドがこの設定値で指定した年代に達するとVACUUMはテーブルの積極的なスキャンを行います。積極的なスキャンは、無効タプルを含む可能性のあるページだけではなく、凍結XIDあるいはMXIDを含むすべてのページを読む点で通常のVACUUMとは異なります。デフォルトは1億5千万トランザクションです。ユーザは0から20億まで任意の値を設定できますが、テーブルに対してラップアラウンド防止処理が起動される前に定期的な手動VACUUMが走ることができるように、VACUUMはautovacuum_multixact_freeze_max_ageの95%に暗黙的に制限します。詳細は24.1.5.1をご覧ください。

vacuum_multixact_freeze_min_age (integer)

VACUUMがテーブルをスキャンする際に、マルチトランザクションIDをより新しいトランザクションIDまたはマルチトランザクションIDに置き換えるかどうかを決める下限値をマルチトランザクション単位で指定します。デフォルトは500万マルチトランザクションです。ユーザは0から10億まで任意の値を設定できますが、強制的な自動バキュームの間隔が短くなり過ぎないように、VACUUMはautovacuum_multixact_freeze_max_ageの半分に暗黙的に実効的な値を制限します。詳細は24.1.5.1をご覧ください。

vacuum_cleanup_index_scale_factor (floating point)

VACUUMクリーンアップステージで、前回の収集統計情報中でカウントされ、インデックススキャンせずに統計情報を挿入できる合計ヒープタプル数の割合を指定します。この設定は今の所B-treeインデックスのみに適用されます。

ヒープから削除されたタプルがない場合、次の条件のうち一つでも当てはまればB-treeインデックスはスキャンされます。インデックスの統計情報が古い、あるいはインデックスに削除されたページが含まれていてクリーンアップ中に再利用できる場合です。新しく挿入されたタプル数の、以前の統計情報収集によって検出されたヒープタプルの合計に対する割合が、vacuum_cleanup_index_scale_factorで指定された値よりも多いときにインデックスの統計情報は古いと見なされます。ヒープタプルの合計数はインデックスのメタページに格納されています。VACUUMが不要タプルがないことを確認するまでメタデータページはこのデータを含まないことに留意してください。ですから、2番目あるいはそれ以降のVACUUMサイクルが不要タプルがないことを確認できるまで、クリーンアップステージでのB-treeインデックスのスキャンは単にスキップされるだけです。

この値は0から10000000000までの値を取ることができます。
vacuum_cleanup_index_scale_factorが0に設定されると、インデックススキャンは決してVACUUMクリーンアップ中にスキップされません。デフォルト値は0.1です。

bytea_output (enum)

bytea型の値の出力形式を設定します。有効な値はhex（デフォルト）、およびescape（PostgreSQLの伝統的な書式）です。より詳細は8.4を参照してください。bytea型は常にこの設定に係わらず、入力時に双方の書式を受け付けます。

xmlbinary (enum)

バイナリデータをXMLに符号化する方法を設定します。例えばこれは、xmlelementやxmlforest関数でbytea値をXMLに変換する際に適用されます。取り得る値はbase64とhexです。どちらもXMLスキーマ標準で定義されています。デフォルトはbase64です。XMLに関連した関数については9.15を参照してください。

実のところこの選択はほとんど趣味の問題で、クライアントアプリケーションで起こり得る制限のみに制約されます。どちらの方法もすべての値をサポートしますが、hex符号化方式はbase64符号化方式より少し大きくなります。

xmloption (enum)

XMLと文字列値との変換時にDOCUMENTとするかCONTENTとするかを設定します。この説明については8.13を参照してください。有効な値はDOCUMENTとCONTENTです。デフォルトはCONTENTです。

標準SQLに従うと、このオプションを設定するコマンドは以下のようになります。

```
SET XML OPTION { DOCUMENT | CONTENT };
```

この構文はPostgreSQLでも使用可能です。

gin_pending_list_limit (integer)

fastupdateが有効なときに使用されるGINインデックスのペンディングリストの最大サイズを設定します。リストがこの設定値よりも大きくなったら、エントリをインデックスのGINの主データ構造に一括転送してリストはクリアされます。この値が単位なしで指定された場合は、キロバイト単位であるとみなします。デフォルトは4メガバイト(4MB)です。この設定は、個々のGINインデックスに対してインデックスストレージパラメータを変更することにより、上書きできます。更なる情報については、66.4.1と66.5を参照してください。

19.11.2. ロケールと書式設定

DateStyle (string)

日付時刻値の表示書式を設定し、曖昧な日付入力の解釈規則を設定します。歴史的な理由により、この変数には2つの独立した要素が含まれています。出力書式指定(ISO、Postgres、SQL、German)と年/月/日の順序の入出力指定(DMY、MDY、YMD)です。これらは分けて設定することもまとめて設定することもできます。EuroおよびEuropeanキーワードはDMYの同義語であり、US、NonEuro、NonEuropeanはMDYの同義語です。詳細は8.5を参照してください。組み込みのデフォルトはISO, MDYですが、initdbにより、選択されたlc_timeロケールの動作に対応した設定で設定ファイルが初期化されます。

IntervalStyle (enum)

間隔の値の表示形式を設定します。sql_standard値は、SQL標準間隔リテラルに一致する出力を生成します。(デフォルトの)値postgresは、DateStyleパラメータがISOに設定されている場合、リリース8.4以前のPostgreSQLに一致する出力を生成します。値postgres_verboseは、DateStyleパラメータが非ISO出力に設定されている場合、リリース8.4以前のPostgreSQLに一致する出力を生成します。値iso_8601は、ISO 8601の4.4.3.2節で定義されている時間間隔「format with designators」に一致する出力を生成します。

またIntervalStyleパラメータはあいまいに入力された時間間隔の解釈に影響を与えます。詳細については8.5.4を参照してください。

TimeZone (string)

表示用およびタイムスタンプ解釈用の時間帯を設定します。組み込まれているデフォルトはGMTですが、通常はpostgresql.confにより上書きされます。initdbによりこれらと関連した設定をシステム環境にインストールされます。詳細は8.5.3を参照してください。

timezone_abbreviations (string)

サーバで日付時刻の入力として受付け可能となる時間帯省略形の集合を設定します。デフォルトは'Default'です。これはほぼ全世界で通じる集合です。また、Australia、India、その他特定のインストレーションで定義可能な集合が存在します。詳細はB.4を参照してください。

extra_float_digits (integer)

このパラメータは、float4、float8、幾何データ型などにおいて、浮動小数点数値のテキスト形式の出力で使用される桁数を調整します。

この値が1（デフォルト）あるいはそれ以上なら、浮動小数点数値の出力は最も短い精度の形式で出力されます。8.1.3を参照してください。生成される実際の桁数は出力される値にのみ依存します。float8値では最大でも17桁、float4値では最大9桁必要です。この形式は高速かつ高精度で、正しく読まれる際には元の2進数における浮動小数点値を正確に保存します。歴史的な互換性の理由により、3までの値が許容されています。

この値がゼロまたは負なら、出力は与えられた10進精度に丸められます。使用される精度は、各々の型の標準的の桁数（型に応じてFLT_DIGあるいはDBL_DIG）がこのパラメータの値により制限されたものになります。（たとえば、-1にするとfloat4値は5桁に、float8値では14桁に丸められます。）この形式は低速、かつ元の2進数における浮動小数点値のビットを保存しませんが、人間にとってより読みやすいかも知れません。

> **注記**
>
> このパラメータの意味とデフォルト値はPostgreSQL 12で変更されました。更なる議論については8.1.3をご覧ください。

client_encoding (string)

クライアント側符号化方式（文字セット）を設定します。デフォルトはデータベース符号化方式を使用します。PostgreSQLサーバでサポートされている文字セットは23.3.1に記載されています。

lc_messages (string)

メッセージが表示される言語を設定します。使用可能な値はシステムに依存します。詳細については23.1を参照してください。この変数が空に設定された場合（これがデフォルトです）、値はシステムに依存する方法でサーバの実行環境から継承されます。

システムによっては、このロケールのカテゴリが存在しません。この変数を設定することはできますが、実効性はありません。また、指定の言語に翻訳されたメッセージが存在しないこともあります。その場合は、引き続き英語のメッセージが表示されます。

サーバログやクライアントに送信されるメッセージに影響するため、および、全ての不適切な値がサーバログの信頼性を損ねる可能性があるため、スーパーユーザのみがこの設定を変更することができます。

lc_monetary (string)

通貨書式で使用するロケールを設定します。例えば、to_char()系の関数で使用します。使用可能な値はシステムに依存します。詳細については23.1を参照してください。この変数が空に設定された場合（これがデフォルトです）、値はシステムに依存する方法でサーバの実行環境から継承されます。

lc_numeric (string)

数字の書式で使用するロケールを設定します。例えば、to_char系の関数で使用します。使用可能な値はシステムに依存します。詳細については23.1を参照してください。この変数が空に設定された場合（これがデフォルトです）、値はシステムに依存する方法でサーバの実行環境から継承されます。

lc_time (string)

例えばto_char系関数における、日付と時間の書式で使用するロケールを設定します。使用可能な値はシステムに依存します。詳細については23.1を参照してください。この変数が空に設定された場合（これがデフォルトです）、値はシステムに依存する方法でサーバの実行環境から継承されます。

default_text_search_config (string)

明示的な設定指定引数を持たないテキスト検索関数の亜種で使用される、テキスト検索設定を選択します。詳細は第12章を参照してください。組み込みのデフォルトはpg_catalog.simpleですが、initdbは、ロケールに合う設定を認識することができれば、選択されたlc_ctypeロケールに対応した設定で設定ファイルを初期化します。

19.11.3. 共有ライブラリのプリロード

追加機能や性能改良の目的で共有ライブラリをプリロードするいくつかの設定があります。たとえば'$libdir/mylib'を設定するとmylib.so(あるいは他のプラットフォームではmylib.sl)を導入設定したの標準ディレクトリからプリロードします。各設定の違いは、設定変更を行うためにいつ、どのような権限が必要かにあります。

典型的には'$libdir/plXXX'のような構文を用いてPostgreSQL手続き言語ライブラリをこの方法でプリロードできます。XXXはpgsql、perl、tcl、pythonです。

PostgreSQLで使用することを意図したライブラリだけがこの方法でロードできます。すべてのPostgreSQL用のライブラリは「magic block」を持ち、互換性を保証するためにチェックされます。ですからPostgreSQL用ではないライブラリはこの方法ではロードできません。LD_PRELOADのようなOSの機能を使えばあるいは使用できるかもしれません。

一般的に言ってモジュールのドキュメントを参照し、推奨される方法でロードしてください。

local_preload_libraries (string)

この変数は、接続時に事前読み込みされる、1つまたは複数の共有ライブラリを指定します。ここにはカンマ区切りでライブラリ名のリストを格納し、各々の名前はLOADコマンドと同じように解釈されます。項目の間の空白は無視されます。名前の中に空白あるいはカンマを含める場合は、二重引用符で囲ってください。このパラメータは接続開始時にのみ有効です。以降の変更は効果がありません。もし指定したライブラリが見つからない場合は、接続は失敗します。

このオプションはすべてのユーザが設定できます。この理由で、読み込み可能なライブラリはインストレーションの共有ライブラリディレクトリのサブディレクトリplugins内にあるものに制限されています。（確実に「安全」なライブラリのみをここにインストールすることはデータベース管理者の責任です。）local_preload_libraries内の項目で、たとえば$libdir/plugins/mylibのようにこのディレクトリを明示的に指定することも、単にライブラリ名を指定することも可能です。mylib は$libdir/plugins/mylibと同じ効果です。

この機能の意図するところは、明示的なLOADコマンドを使わずに、特定のセッションにおいて非特権ユーザがデバッグ用あるいは性能計測用のライブラリをロードできるようにすることにあります。そのためにも、クライアント側でPGOPTIONS環境変数を使う、あるいはALTER ROLE SETを使うことが典型的になるでしょう。

しかし、モジュールが特にスーパーユーザ以外に使われることを意図しているのでない限り、通常この方法は正しい使い方ではありません。代わりにsession_preload_librariesを見てください。

session_preload_libraries (string)

この変数は接続開始時にプリロードされる一つ以上の共有ライブラリを指定します。ここにはカンマ区切りでライブラリ名のリストを格納し、各々の名前はLOADコマンドと同じように解釈されます。項目の間の空白は無視されます。名前の中に空白あるいはカンマを含める場合は、二重引用符で囲ってください。このパラメータは接続開始時にのみ有効です。以降の変更は効果がありません。もし指定したライブラリが見つからない場合は、接続は失敗します。スーパーユーザだけがこの設定を変更できます。

この機能は、デバッグや性能測定の目的でLOADコマンドを使わずに特定のセッションでライブラリをロードする目的で使われます。たとえばALTER ROLE SETで設定することにより、特定のユーザが開始するすべてのセッションでauto_explainが有効になります。また、このパラメータはサーバを再起動せずに変更できます(しかし変更は新しいセッションが開始するときにのみ有効となります)。すべてのセッションで有効にしたいのであれば、この方法で新しいモジュールを容易に追加できます。

shared_preload_librariesと違って、ライブラリがはじめて使われるときにロードする方法と比べてセッションが開始するときにライブラリをロードする方法には大きな性能的な優位性はありません。しかし、コネクションプーリングを使うのであれば、いくらか優位性があります。

shared_preload_libraries (string)

この変数はサーバ起動時にプリロードされる一つ以上の共有ライブラリを指定します。ここにはカンマ区切りでライブラリ名のリストを格納し、各々の名前はLOADコマンドと同じように解釈されます。項目の間の空白は無視されます。名前の中に空白あるいはカンマを含める場合は、二重引用符で囲ってください。このパラメータは接続開始時にのみ有効です。もし指定したライブラリが見つからない場合は、接続は失敗します。

ライブラリによってはpostmaster起動時にのみ可能な操作を実行する必要があるものがあります。たとえば、共有メモリの獲得、軽量ロックの予約、バックグラウンドワーカの起動などです。このようなライブラリはこのパラメータを使ってサーバ起動時にロードしなければなりません。詳細は各ライブラリのドキュメントを見てください。

これ以外のライブラリもプリロードできます。共有ライブラリをプリロードすることにより、最初にライブラリが使われる際にライブラリが起動する時間を避けることができます。しかし、そのライブラリが使われないとしても、サーバプロセスが起動する時間がわずかに長くなる可能性があります。したがって、この方法は、ほとんどのセッションで使われるライブラリにのみ使用することを推奨します。また、パラメータの変更にはサーバの再起動が必要になります。ですから、たとえば短期のデバッグ仕事にこの設定を使うのは適当とは言えません。session_preload_librariesを代わりに使ってください。

> **注記**
>
> Windowsのホストでは、ライブラリのプリロードは、新しいサーバプロセスの起動に要する時間を短縮しません。個々のサーバプロセスは、すべてのプリロードライブラリを再読み込みします。それでもpostmaster起動時に操作を実行しなければならないライブラリを使用するWindowsホストにとってはshared_preload_librariesは有用です。

jit_provider (string)

この変数は、使用するJITプロバイダライブラリ(31.4.2参照)の名前です。デフォルトはllvmjitです。このパラメータはサーバ起動時にのみ設定可能です。

存在しないライブラリが指定されると、JITは利用できませんが、エラーは起こりません。これにより、PostgreSQLパッケージとは別にJITサポートをインストールできるようになります。

19.11.4. その他のデフォルト

dynamic_library_path (string)

オープンする必要がある動的ロード可能なモジュールについて、そのCREATE FUNCTIONやLOADコマンドで指定されたファイル名にディレクトリ要素がなく(つまり、名前にスラッシュが含まれずに)指定された場合、システムは必要なファイルをこのパスから検索します。

dynamic_library_pathの値は、絶対パスのディレクトリ名をコロン(Windowsの場合はセミコロン)で区切った一覧です。この一覧の要素が特別な$libdirという値から始まる場合、コンパイルされたPostgreSQLパッケージのライブラリディレクトリで$libdirは置換されます。ここには、PostgreSQLの標準配布物により提供されるモジュールがインストールされます (このディレクトリ名を表示するには、pg_config --pkglibdir を使用してください)。例を以下に示します。

```
dynamic_library_path = '/usr/local/lib/postgresql:/home/my_project/lib:$libdir'
```

Windows環境の場合は以下です。

```
dynamic_library_path = 'C:\tools\postgresql;H:\my_project\lib;$libdir'
```

このパラメータのデフォルト値は'$libdir'です。この値が空に設定された場合、自動的なパス検索は無効になります。

このパラメータはスーパーユーザによって実行時に変更することができますが、この方法での設定は、そのクライアント接続が終わるまでしか有効になりません。ですので、この方法は開発目的でのみ使用すべきです。推奨方法はこのパラメータをpostgresql.conf設定ファイル内で設定することです。

gin_fuzzy_search_limit (integer)

GINインデックス走査により返されるセットのソフトな上限サイズです。詳細は66.5を参照してください。

19.12. ロック管理

deadlock_timeout (integer)

これは、デッドロック状態があるかどうかを調べる前にロックを待つ時間です。デッドロックの検査は比較的高価なので、サーバはロックを待つ度にこれを実行するわけではありません。楽天的ですがデッドロックは実用レベルのアプリケーションでは頻繁に発生しないと仮定し、デッドロックの検査の前にしばらくはロック待ちをします。この値を増やすことにより必要のないデッドロックの検査で無駄にされる時間

は減りますが、本当にデッドロックがあった場合の報告が遅れます。この値が単位なしで指定された場合は、ミリ秒単位であるとみなします。デフォルトは1秒（1s）で、おそらく実用の際にはこれ以上は必要でしょう。負荷の大きいサーバではもっと必要かもしれません。理想としてはこの設定は通常のトランザクションにかかる時間を超えているべきです。そうすればロック待ちトランザクションがデッドロックの検査をする前にロックが解除される可能性が改善されます。スーパーユーザのみこの設定を変更できます。

log_lock_waitsが設定された場合、このパラメータはロック待機に関するログメッセージを出力する前の待機時間を決定します。ロック遅延の調査を行う場合は、通常のdeadlock_timeoutよりも短い値を設定することを勧めます。

max_locks_per_transaction (integer)

共有ロックテーブルは、max_locks_per_transaction * (max_connections + max_prepared_transactions)オブジェクト（例えばテーブル）上のロック追跡します。したがって、ある時点でこの数以上の個々のオブジェクトをロックすることはできません。このパラメータは各トランザクションで割り当てられるオブジェクトロックの平均値を制御します。個々のトランザクションでは、このロックテーブルにすべてのトランザクションのロックが収まる限りオブジェクトのロックを獲得できます。これは、ロックできる行数ではありません。この値には制限がありません。デフォルトの64は、経験的に十分であると証明されていますが、単一のトランザクションで数多くの異なるテーブルをいじる問い合わせがいる場合、たとえば、数多くの子テーブルを持つ親テーブルの問い合わせなど、この値を大きくする必要があるかも知れません。このパラメータはサーバ起動時のみ設定されます。

スタンバイサーバを稼動するとき、このパラメータをマスターサーバと同じか、より高い値に設定しなければなりません。そうしないと、問い合わせはスタンバイサーバでは許可されません。

max_pred_locks_per_transaction (integer)

共有記述ロックテーブル（shared predicate lock table）は、max_pred_locks_per_transaction * (max_connections + max_prepared_transactions)オブジェクト（例えば諸テーブル）のロックを追跡します。従って、この数以上の明確なオブジェクトは同時にロックされません。このパラメータはそれぞれのトランザクションに対して割り当てられたオブジェクトのロックの平均数を管理します。個別のトランザクションはロックテーブル内の全てのトランザクションのロックが適合する限り、より多くのオブジェクトをロックできます。これはロック可能な行数ではありません。その値は無制限です。デフォルトは64で、テストでは一般的に充分ですが、単一のシリアライザブルトランザクションで数多くの異なるテーブルに触れるクライアントが存在する場合、この値を大きくする必要があることがあります。このパラメータはサーバ起動時のみ設定可能です。

max_pred_locks_per_relation (integer)

リレーション全体をカバーするロックに昇格する前に、一つリレーションの中で述語ロックできるページ数あるいはタプル数を指定します。0以上の値は、絶対的な制限を表し、負の数はmax_pred_locks_per_transactionをその絶対値で割ったものを表します。デフォルトは-2で、以前のバージョンのPostgreSQLの振る舞いを維持します。このパラメータはpostgresql.confファイル、または、サーバのコマンドラインのみで設定可能です。

max_pred_locks_per_page (integer)

ページ全体をカバーするロックに昇格する前に、一つページの中で述語ロックできる行数を指定します。デフォルトは2です。このパラメータはpostgresql.confファイル、または、サーバのコマンドラインのみで設定可能です。

19.13. バージョンとプラットフォーム互換性

19.13.1. 以前のPostgreSQLバージョン

array_nulls (boolean)

これは、配列入力パーサが引用符のないNULLをNULL配列要素として認識するかどうかを制御します。デフォルトでは、これはonで、NULL値を持つ配列値を入力することができます。しかし、8.2より前のバージョンのPostgreSQLでは、配列内のNULL値をサポートしておらず、NULLを「NULL」という値の文字列を持つ通常の配列要素として扱っていました。古い動作を必要とするアプリケーションの後方互換性のため、この変数をoffにすることができます。

この変数がoffであっても、NULL値を含む配列値を作成することができることに注意してください。

backslash_quote (enum)

文字列リテラルの中で引用符が\'で表現されるかどうかを管理します。引用符の表現としてSQL準拠の方式では二重化('')ですが、PostgreSQLは歴史的に\'も受け付けます。とは言っても、いくつかのクライアント文字集合符号化方式において、最終バイトが数値的にASCIIの\に等しいマルチバイト文字があり、\'を使用するとセキュリティ上問題を引き起こす可能性があります。クライアント側のコードが事実上エスケープを正しく扱わない場合、SQLインジェクション攻撃が可能になります。この危険性の回避は、サーバが逆スラッシュでエスケープされた引用符を含む問い合わせを拒絶するようにします。許可されるbackslash_quoteの値は、on（常に\'を許可）, off（常に拒否）、および safe_encoding（クライアント符号化方式がASCIIの\を許可しないときのみ、マルチバイト文字内で許可）。safe_encoding がデフォルトの設定。

標準に従った文字列リテラルでは、\は単に\を意味するものです。このパラメータのみが、エスケープ文字列構文（E'...'）を含む標準に従わないリテラルの取り扱いに影響します。

escape_string_warning (boolean)

有効の場合、通常の文字列リテラル（'...'構文）にバックスラッシュ（\）があり、standard_conforming_stringsが無効な場合、警告が発せられます。デフォルトはonです。

通常文字列のデフォルトの振る舞いは、SQL標準ではバックスラッシュを通常文字として取り扱うため、バックスラッシュをエスケープとして使用したいアプリケーションは、エスケープ文字列構文(E'...')を使用するように変更すべきです。この変数は変更すべきコードを突き止めるのに役立つよう、有効にすることができます。

lo_compat_privileges (boolean)

9.0以前のPostgreSQLリリースでは、ラージオブジェクトはアクセス権限が無く、従って全てのユーザが常に読み込み、書き込みが可能でした。この変数をonにすると、以前のリリースとの互換性のため、新規の権限チェックが無効になります。デフォルトはoffです。スーパーユーザのみこの設定を変更できます。

この変数を設定しても、ラージオブジェクトに関連した全ての安全性チェックを無効にする訳ではありません。PostgreSQL 9.0で変更されたデフォルトの動きに対してのみです。

operator_precedence_warning (boolean)

onの場合、演算子の優先順位の変更の結果、PostgreSQL 9.4以降で意味が変わる可能性のある構文に対して、パーサは警告を出力します。これは、優先順位の変更によりアプリケーションに何か不具合が起きないか監視するのに役立ちます。しかし、これはプロダクション環境でオンにしたままにしておくためのものではありません。なぜなら、完全に正しいSQL標準準拠コードに対しても警告を発するからです。デフォルトはoffです。

詳細は4.1.6を見てください。

quote_all_identifiers (boolean)

データベースがSQLを生成する時、たとえ(現在)キーワードになっていなくても、全ての識別子を引用符で囲むことを強制します。これは EXPLAINの出力に影響を与えるのみならず、pg_get_viewdefのような関数の結果にも影響します。pg_dump および pg_dumpallの--quote-all-identifiersオプションも参照してください。

standard_conforming_strings (boolean)

標準SQLで規定されたように、通常の文字列リテラル('...')がバックスラッシュをそのまま取り扱うか否かを制御します。PostgreSQL 9.1からデフォルトはonになっています(それ以前のリリースではoffがデフォルトでした)。どのように文字列リテラルが処理されるかを決めるこのパラメータを、アプリケーションで検査することができます。このパラメータの存在は、エスケープ文字列構文(E'...')がサポートされているかどうかを示すものとも考えられます。エスケープ文字列構文 (4.1.2.2)は、アプリケーションでバックスラッシュをエスケープ文字として扱いたい場合に使用すべきです。

synchronize_seqscans (boolean)

これにより、同時実行スキャンがほぼ同じ時間に同じブロックを読み取り、I/Oへの負荷を分散できるように、互いに同期して、大規模テーブルをシーケンシャルスキャンすることができます。これが有効な場合、スキャンはテーブルの途中から始まり、進行中のスキャンの活動と同期するように、行全体を覆うように終端を「巻き上げる」可能性があります。これにより、ORDER BY句を持たない問い合わせが返す行の順序は予想できない程変わってしまいます。このパラメータをoffにすることで、シーケンシャルスキャンが常にテーブルの先頭から始まるという、8.3より前の動作を保証します。デフォルトはonです。

19.13.2. プラットフォームとクライアント互換性

transform_null_equals (boolean)

有効の場合、expr = NULL(もしくはNULL = expr)形式の式はexpr IS NULLとして取り扱われ、それは、もしexprがNULL値と評価すれば真を返し、そうでなければ偽を返します。expr = NULLの正しいSQL仕様準拠の動作は常にNULL(判らない)を返すことです。従って、このパラメータのデフォルトはoffになっています。

しかし、Microsoft Accessのフィルタ形式はNULL値を検査するためにexpr = NULLを使用する問い合わせを生成しますので、そのインタフェースを使用してデータベースにアクセスする場合は、このオプションを有効にする方が良いでしょう。expr = NULLという形の式は(SQL標準解釈を使用した結果)常にNULL値を返しますので、通常のアプリケーションでは意味がほとんどなく、滅多に使用されません。ですので、このオプションは実際は害はありません。しかし、慣れていないユーザはしばしばNULL値に関する式の意味に戸惑いますので、デフォルトでこのオプションはoffです。

このオプションは= NULLという形式にのみ影響することに注意してください。他の比較演算子や等価演算子を呼び出す他の(INのような)式と計算する上で等価となる式には影響を与えません。したがって、このオプションは間違ったプログラミングの汎用的な問題解決を行いません。

関連する情報は9.2を参照してください。

19.14. エラー処理

exit_on_error (boolean)

onなら、全てのエラーは現在のセッションを中止させます。デフォルトではこれはoffに設定されているので、FATALエラーのみがセッションを中止させます。

restart_after_crash (boolean)

デフォルトであるonの場合、PostgreSQLはバックエンドのクラッシュの後、自動的に再初期化を行います。この値を真のままにしておくことが、通常データベースの可用性を最大化する最適の方法です。しかし、PostgreSQLがクラスタウェアにより起動された時のような状況では、クラスタウェアが制御を獲得して、適切とみなすいかなる振る舞いをも行えるように再起動を無効にすることが有益かもしれません。

data_sync_retry (boolean)

デフォルトであるoffに設定すると、PostgreSQLは変更されたデータファイルのファイルシステムへの吐き出しの失敗に対してPANICレベルのエラーを発生させます。これによりデータベースサーバのクラッシュが引き起こされます。このパラメータはサーバ起動時のみ設定可能です。

オペレーティングシステムによっては、カーネルキャッシュのページ内のデータの状態は、書き戻しの失敗の後は不明です。このような状況では、データロスを避ける唯一の方法は、失敗が報告された後、可能ならば失敗の根本原因を調査して故障したハードウェアを交換したのち、WALからの回復することだけです。

onに設定すると、代わりにPostgreSQLはエラーを報告して実行を継続し、後のチェックポイントでデータの吐き出しをリトライします。書き戻しの失敗が起きたときのオペレーティングシステムのバッファデータの扱いを調査した後でのみonに設定してください。

19.15. 設定済みのオプション

以下の「パラメータ」は読み取り専用で、PostgreSQLのコンパイル時、もしくはインストール時に決定されます。そのため、これらはpostgresql.confのサンプルから除かれています。このオプションは、特定のアプリケーション、特に管理用フロントエンドによって注目される可能性があるPostgreSQLの様々な部分の振舞いを報告します。

block_size (integer)

ディスクブロックの容量を報告します。サーバ構築の際にBLCKSZの値で決定されます。デフォルトの値は8192バイトです。(shared_buffersの様な)いくつかの構成変数の意味はblock_sizeによって影響されます。これに関しての情報は19.4を参照してください。

data_checksums (boolean)

このクラスタでデータチェックサムが有効になっているかどうかを報告します。詳細はdata checksumsを見てください。

data_directory_mode (integer)

このパラメータは、Unixシステムでは(data_directory)により定義されたデータディレクトリのパーミッションを起動時に報告します。(Microsoft Windowsではこのパラメータは常に0700を表示します。) さらなる情報はgroup accessをご覧ください。

debug_assertions (boolean)

PostgreSQLがアサーションを有効にしてビルドされているかどうかを報告します。これは、USE_ASSERT_CHECKINGマクロがPostgreSQLをビルドされた際に定義されている場合(つまり、configureオプションの--enable-cassertが適用されている)に該当します。デフォルトではPostgreSQLは、アサーションなしにビルドされます。

integer_datetimes (boolean)

PostgreSQLが64ビット整数による日付と時刻のサポート付きで構築されたかどうかを報告します。PostgreSQL 10では、これは常にonです。

lc_collate (string)

テキストデータの並び替えが行なわれるロケールを報告します。詳細は23.1 を参照してください。この値はデータベースが作成されたときに決定されます。

lc_ctype (string)

文字分類を決定するロケールを報告します。詳細は23.1を参照してください。この値はデータベースが作成されたときに決定されます。通常、これはlc_collateと同一ですが、特殊なアプリケーションでは異なって設定されることがあります。

max_function_args (integer)

関数の引数の最大数を報告します。サーバを構築する時、FUNC_MAX_ARGSの値で決定されます。デフォルトの値は100引数です。

max_identifier_length (integer)

最長の識別子の長さを報告します。サーバ構築時のNAMEDATALENの値より一つ少なく設定されます。デフォルトのNAMEDATALENの値は64ですので、デフォルトのmax_identifier_lengthは63バイトで、マルチバイト符号化方式を使用している場合、63文字以下になることがあります。

max_index_keys (integer)

インデックスキーの最大数を報告します。サーバをビルドする際にINDEX_MAX_KEYSの値で決定されます。デフォルトの値は32キーです。

segment_size (integer)

あるファイルセグメントの中に格納できるブロック数(ページ数)を報告します。サーバ構築時にRELSEG_SIZEの値で決定されます。バイト単位の一セグメントファイルの最大容量は、block_size倍のsegment_sizeと等しくなります。デフォルトでは1GBです。

server_encoding (string)

データベース符号化方式（文字セット）を報告します。データベースが作成された時に決定されます。通常クライアントはclient_encodingの値にのみ注意する必要があります。

server_version (string)

サーバのバージョン番号を報告します。サーバ構築の際のPG_VERSIONの値によって決定されます。

server_version_num (integer)

サーバのバージョン番号を整数として返します。この値は、サーバ構築時のPG_VERSION_NUMの値により決まります。

ssl_library (string)

（このインスタンスでSSLが設定あるいは使用されていなくても）このPostgreSQLサーバの構築時に使用されたSSLライブラリの名前、たとえばOpenSSLあるいは空文字列、を報告します。

wal_block_size (integer)

WALディスクブロックの容量を報告します。サーバ構築時にXLOG_BLCKSZの値で決定されます。デフォルトの値は8192バイトです。

wal_segment_size (integer)

ログ先行書き込みのセグメントの大きさを報告します。デフォルト値は16MBです。さらなる詳細については29.4をご覧ください。

19.16. 独自のオプション

この機能は追加モジュール（手続き言語など）によって追加されるPostgreSQLが識別できないパラメータを使えるように設計されたものです。これにより拡張モジュールは標準の方法で構成されます。

カスタムオプションには2つに分かれた名称があります。拡張名につづいてドット、そして特定のパラメータ名です。SQLの修飾名に良く似ています。例としてplpgsql.variable_conflictが挙げられます。

カスタムオプションは読み込まれていない関連性のある拡張モジュールのプロセスに設定される必要がある場合があるので、PostgreSQLはどんな2つの部分のパラメータ名による設定を受け付けます。これらの変数は代替物として取り扱われ、それらを定義したモジュールが読み込まれるまで機能しません。拡張モジュールが読み込まれた時、その変数定義が追加され、それら定義に基づいた代替値が変換され、そしてその拡張名の確認されない代替物に対して警告が発せられます。

19.17. 開発者向けのオプション

以下のパラメータは、PostgreSQLのソースコードに対する作業用のものです。中には深刻な損傷を負ったデータベースの復旧に役立つものもあります。実運用のデータベースでこれらを設定する理由はないはずで

す。したがって、これらはサンプルのpostgresql.confからは除外されています。これらのパラメータの多くは、それを動作させるために特殊なソースコンパイルを必要としていることに注意してください。

allow_system_table_mods (boolean)

システムテーブルの構造変更とその他の危険性の高いシステムテーブルに対するアクションを許可します。これは通常スーパーユーザにさえ許可されません。この設定の無分別な使用は回復不能なデータ喪失やデータベースシステムの重大な破損を招きます。スーパーユーザだけがこの設定を変更できます。

backtrace_functions (string)

このパラメータはカンマ区切りのC関数名を含みます。エラーが発生し、エラー発生箇所の内部C関数がこのリストの値と一致すると、エラーメッセージとともにバックトレースが一緒にサーバログに書かれます。これはソースコードの特定箇所をデバッグするのに役立ちます。

バックトレースのサポートはすべてのプラットフォームで提供されているわけではありませんし、バックトレースの品質はコンパイルオプションに依存します。

スーパーユーザだけがこのパラメータを設定できます。

ignore_system_indexes (boolean)

システムテーブルの読み込み時にシステムインデックスを無視します(しかしテーブルが更新された時はインデックスを更新します)。障害があるシステムインデックスを復旧する時、これは有用です。セッションが始まった後に、このパラメータを変更することはできません。

post_auth_delay (integer)

サーバプロセスが始まり認証手続きが終わった後の遅延時間です。これは、デバッガを使用してサーバプロセスに接続する機会を開発者に提供することを目的としています。この値が単位なしで指定された場合は、秒単位であるとみなします。値がゼロ(デフォルト)の場合、この遅延は無効になります。セッションが始まった後に、このパラメータを変更することはできません。

pre_auth_delay (integer)

新しくサーバプロセスがforkした後、認証手続きに入る前の遅延時間です。これは、認証における誤動作を追跡するために、デバッガを使用してサーバプロセスに接続する機会を開発者に提供することを目的としたものです。この値が単位なしで指定された場合は、秒単位であるとみなします。値がゼロ(デフォルト)の場合、この機能は無効になります。このパラメータはpostgresql.confファイル内、または、サーバのコマンドラインでのみ設定可能です。

trace_notify (boolean)

LISTENとNOTIFYコマンドのための大量なデバッグ出力を生成します。この出力をクライアントもしくはサーバログに送信するためには、それぞれ、client_min_messagesもしくはlog_min_messagesはDEBUG1以下でなければなりません。

trace_recovery_messages (enum)

復旧関連のデバッグ出力のログ取得を有効にします。さもないとログは取られません。このパラメータはユーザに対し、log_min_messagesの通常設定を上書きすることを許可します。しかし、特定のメッセー

ジに対してのみです。これはホットスタンバイのデバッグを意図したものです。有効な値は、DEBUG5、DEBUG4、DEBUG3、DEBUG2、DEBUG1、および LOGです。デフォルトのLOGは、ログ取得の決定に全く影響しません。その他の値は、あたかもLOG優先度を所有しているごとく、それ、またはより高い優先度でログ取得される復旧関連デバッグメッセージの要因となります。log_min_messagesの通常設定に対し、これは無条件にそれらをサーバログに送り込みます。このパラメータはpostgresql.confファイル内、または、サーバコマンドラインでのみ設定可能です。

trace_sort (boolean)

もしも有効であれば、並び替え操作の間のリソース使用についての情報を放出します。このパラメータは PostgreSQLがコンパイルされた時、TRACE_SORTマクロが定義されている場合にのみ有効です。(とは言っても、現在TRACE_SORTはデフォルトで定義されています。)

trace_locks (boolean)

有効な場合、ロックの使用状況に関する情報を出力します。出力される情報には、ロック操作の種類、ロックの種類、ロックまたはロック解除されているオブジェクトの一意な識別子が含まれます。また、このオブジェクトに既に与えられているロック種類やこのオブジェクトで待機しているロック種類を表すビットマスクも含まれます。ロック種類それぞれについて、与えられているロック数、待機中のロック数がその総数と共に出力されます。ログファイル出力例を以下に示します。

```
LOG:  LockAcquire: new: lock(0xb7acd844) id(24688,24696,0,0,0,1)
      grantMask(0) req(0,0,0,0,0,0,0)=0 grant(0,0,0,0,0,0,0)=0
      wait(0) type(AccessShareLock)
LOG:  GrantLock: lock(0xb7acd844) id(24688,24696,0,0,0,1)
      grantMask(2) req(1,0,0,0,0,0,0)=1 grant(1,0,0,0,0,0,0)=1
      wait(0) type(AccessShareLock)
LOG:  UnGrantLock: updated: lock(0xb7acd844) id(24688,24696,0,0,0,1)
      grantMask(0) req(0,0,0,0,0,0,0)=0 grant(0,0,0,0,0,0,0)=0
      wait(0) type(AccessShareLock)
LOG:  CleanUpLock: deleting: lock(0xb7acd844) id(24688,24696,0,0,0,1)
      grantMask(0) req(0,0,0,0,0,0,0)=0 grant(0,0,0,0,0,0,0)=0
      wait(0) type(INVALID)
```

ダンプされる構造の詳細は、src/include/storage/lock.h にあります。

このパラメータはPostgreSQLがコンパイル時にLOCK_DEBUGマクロが定義された場合のみ有効です。

trace_lwlocks (boolean)

有効な場合、軽量ロックの使用状況に関する情報を出力します。軽量ロックは主に、共有メモリ上のデータ構造へのアクセスに関する排他制御機能を提供することを意図したものです。

このパラメータはPostgreSQLがコンパイル時にLOCK_DEBUGマクロが定義された場合のみ有効です。

trace_userlocks (boolean)

有効な場合、ユーザロックの使用状況に関する情報を出力します。出力はtrace_locksと同じですが、勧告的ロックに関するもののみを出力します。

このパラメータはPostgreSQLがコンパイル時にLOCK_DEBUGマクロが定義された場合のみ有効です。

trace_lock_oidmin (integer)

設定すると、このOID未満のテーブルに関するロックの追跡を行いません。(システムテーブルに関する出力を抑えるために使用します。)

このパラメータはPostgreSQLがコンパイル時にLOCK_DEBUGマクロが定義された場合のみ有効です。

trace_lock_table (integer)

このテーブル(OID)に対し無条件でロックを追跡します。

このパラメータはPostgreSQLがコンパイル時にLOCK_DEBUGマクロが定義された場合のみ有効です。

debug_deadlocks (boolean)

設定すると、デッドロックタイムアウトが発生した時全ての進行中のロックについての情報がダンプされます。

このパラメータはPostgreSQLがコンパイル時にLOCK_DEBUGマクロが定義された場合のみ有効です。

log_btree_build_stats (boolean)

設定すると、各種B-tree操作に関するシステムリソース(メモリとCPU)の使用についての統計情報をログに出力します。

このパラメータはPostgreSQLがコンパイル時にBTREE_BUILD_STATSマクロが定義された場合のみ有効です。

wal_consistency_checking (string)

このパラメータは、WALのREDOルーチンのバグをチェックするために使うことを意図しています。有効にすると、WALレコードと一緒に変更されたバッファのフルページイメージをレコードに追加します。後でそのレコードがリプレイされるときは、システムはまず各々のレコードを適用し、次にレコードによって変更されたバッファが、格納したイメージと一致するかどうかをテストします。ある種のケース(たとえばヒントビット)では、些細な変化は許容され、無視されます。予期しない差異は、致命的エラーを引き起こし、リカバリが中断されます。

デフォルト値は空文字で、この機能を無効にします。すべてのレコードをチェックするために、allにすることができます。カンマ区切りのリストにすると、対応するリソースマネージャに由来するレコードのみをチェックします。今のところ、サポートされているリソースマネージャは、heap、heap2、btree、hash、gin、gist、sequence、spgist、brin、genericです。スーパーユーザだけがこの設定を変更できます。

wal_debug (boolean)

もしonであれば、WALに関連したデバッグ出力が有効になります。このパラメータはWAL_DEBUGマクロがPostgreSQLのコンパイルの時に定義された場合にのみ有効です。

ignore_checksum_failure (boolean)

data checksumsが有効の時のみ効果があります。

読み込み過程でチェックサム障害が検出されると、通常PostgreSQLはエラーを報告し、現時点のトランザクションを停止します。ignore_checksum_failureを有効(on)に設定するとシステムはその障害を無視し(しかし警告は報告をします)、処理を継続します。この振る舞いはたぶんクラッシュの原因、破損の伝播や隠ぺい、もしくはその他の深刻な問題の原因になることがあります。とは言っても、エラーを切り抜け、ブロックヘッダが健全に存在するテーブルにある障害を受けていないタプルの回収は行えます。もしヘッダーが破損されたら、オプションが有効になっていたとしても報告はなされます。デフォルトの設定はoffで、スーパーユーザのみが変更可能です。

zero_damaged_pages (boolean)

ページヘッダの障害がわかると、通常PostgreSQLはエラーの報告を行い、現在のトランザクションを中断させます。zero_damaged_pagesをonに設定することにより、システムは代わりに警告を報告し、障害のあるメモリ内のページをゼロで埋め、処理を継続します。この動作により、障害のあったページ上にある全ての行のデータが破壊されます。しかし、これによりエラーを確実に無視し、正常なページに存在するテーブル内の行を取り出すことができます。ハードウェアまたはソフトウェアのエラーによって破損が発生した場合のデータの復旧時に有用です。障害のあるページからのテーブルのデータの復旧をあきらめた場合を除き、通常はこれをonにしてはいけません。ゼロで埋められたページはディスクに書き込みを強要されないため、このパラメータを再び無効にする以前にテーブル、またはインデックスを再作成することを勧めます。デフォルトはoffであり、スーパーユーザのみ変更可能です。

ignore_invalid_pages (boolean)

off(デフォルトです)に設定すると、無効なページを参照しているWALレコードはPostgreSQLに対してPANICレベルのエラーを引き起こし、リカバリをアボートします。ignore_invalid_pagesをonに設定すると、WALレコードの無効なページへの参照を無視し(しかしワーニングは報告されます)、リカバリを継続します。この振る舞いにより、クラッシュ、データロス、破壊を増長したり見えなくするなどの深刻な問題を被るかもしれません。しかし、これによってPANICレベルのエラーを回避してリカバリを完了し、サーバを起動できるかもしれません。このパラメータはサーバ起動時にのみ設定できます。リカバリ中、あるいはスタンバイモードでのみ効果があります。

jit_debugging_support (boolean)

LLVMに要求された機能がある場合は、生成した関数をGDB用に登録します。これにより、デバッグが容易になります。デフォルト設定はoffです。このパラメータはサーバ起動時のみ設定可能です。

jit_dump_bitcode (boolean)

生成されたLLVM IRをdata_directory内のファイルシステムに出力します。これはJITコンパイルのインターナルについて作業するときだけ有用です。デフォルト設定はoffです。このパラメータはスーパーユーザだけが変更可能です。

jit_expressions (boolean)

JITコンパイルが有効な時に式がJITコンパイルされるかどうかを決定します。(31.2参照。) デフォルトはonです。

jit_profiling_support (boolean)

LLVMに要求された機能がある場合は、JITが生成した関数をperfでプロファイルすることができるデータを出力します。これにより$HOME/.debug/jit/にファイルが書き出されます。ユーザは自分の責任で必要なときに後始末を行わなければなりません。デフォルト設定はoffです。

jit_tuple_deforming (boolean)

JITコンパイルが有効な時にタプルデフォーミングがJITコンパイルされるかどうかを決定します。（31.2参照。）デフォルトはonです。

19.18. 短いオプション

簡便性のために、一文字のコマンドラインオプションスイッチも、幾つかのパラメータのために用意されています。それらは表 19.3に解説されています。一部のオプションは歴史的な理由のために存在します。また、この一文字オプションが存在することが、このオプションを多く使用することを支持することを示しているわけではありません。

表19.3 短いオプションキー

短いオプション	同義
-B x	shared_buffers = x
-d x	log_min_messages = DEBUGx
-e	datestyle = euro
-fb, -fh, -fi, -fm, -fn, -fo, -fs, -ft	enable_bitmapscan = off, enable_hashjoin = off, enable_indexscan = off, enable_mergejoin = off, enable_nestloop = off, enable_indexonlyscan = off, enable_seqscan = off, enable_tidscan = off
-F	fsync = off
-h x	listen_addresses = x
-i	listen_addresses = '*'
-k x	unix_socket_directories = x
-l	ssl = on
-N x	max_connections = x
-O	allow_system_table_mods = on
-p x	port = x
-P	ignore_system_indexes = on
-s	log_statement_stats = on
-S x	work_mem = x
-tpa, -tpl, -te	log_parser_stats = on, log_planner_stats = on, log_executor_stats = on
-W x	post_auth_delay = x

第20章 クライアント認証

クライアントアプリケーションがデータベースサーバに接続する時、Unixコンピュータに特定のユーザとしてログインする時と同じように、どのPostgreSQLユーザ名で接続するかを指定します。SQL 環境の中では存在するユーザ名でデータベースオブジェクトへのアクセス権限が決まります。詳しい情報は第21章を参照してください。ですから、どのデータベースユーザがデータベースに接続できるかを制限することが基本となります。

> **注記**
>
> 第21章で説明されていますが、実はPostgreSQLは「ロール」によって権限の管理を行っています。この章では、「LOGIN権限を持つロール」を、一貫してデータベースユーザという呼び方で使用します。

認証はデータベースサーバがクライアントの身元を識別し、その延長としてクライアントアプリケーション（もしくはクライアントアプリケーションを実行するユーザ）が要求されたデータベースユーザ名で接続することができるかどうかを決定する手順です。

PostgreSQLは異なったクライアント認証方法を複数提供します。特定のクライアント接続の認証に使用する方法は、（クライアントの）ホストアドレス、データベース、およびユーザに従って選択できます。

PostgreSQLデータベースユーザ名は稼働しているサーバのオペレーティングシステムのユーザ名とは論理的に分かれています。もし特定のサーバの全てのユーザがサーバマシン上にもアカウントを持っている場合、そのオペレーティングシステムのユーザ名に一致するデータベースユーザ名を割り当てることは理にかなっています。しかし、リモート接続を受け付けるサーバは、ローカルなオペレーティングシステムのアカウントを持たないデータベースユーザを多く持っている場合もあります。そのような時にはデータベースユーザ名とOSのユーザ名との間の関連性は必要ありません。

20.1. pg_hba.confファイル

クライアント認証はデータベースクラスタのデータディレクトリ内の、伝統的にpg_hba.confという名前の設定ファイルで管理されています（HBAとは、host-based authentication: ホストベース認証の略です）。デフォルトのpg_hba.confファイルは、データディレクトリがinitdbで初期化される時にインストールされます。しかし、この認証設定ファイルを他の場所に設置することができます。hba_file設定パラメータを参照してください。

pg_hba.confファイルの一般的な書式は、1行につき1つのレコードというレコードの集合です。空行はコメント用の#文字以降の文字と同じく無視されます。レコードは行をまたいで続けることはできません。レコードはスペースもしくはタブ、もしくはその両方で区切られた、複数のフィールドで構成されています。フィールドには、フィールド値が二重引用符付きの場合空白文字を含むことができます。データベース、ユーザもしくはアドレスフィールド内のキーワード(例：allまたはreplication)の一つを引用するとその特別な意味が失われ、その名称のデータベース、ユーザもしくはホストと一致するようになります。

それぞれのレコードは接続形式、（接続形式に対して意味を持つのであれば）クライアントのIPアドレス範囲、データベースの名前、ユーザ名およびこれらのパラメータに一致する接続で使用される認証方法を指定します。接続形式、クライアントアドレス、要求されたデータベース、およびユーザ名に一致する最初のレコー

ドが認証処理に使用されます。「失敗時の継続」や、あるいは「バックアップ」はありません。これは、もしあるレコードが選択されて認証に失敗した場合、後続のレコードは考慮されないということです。どのレコードも一致しない時はアクセスが拒否されます。

レコードはいくつかの形式があります。

```
local         database  user  auth-method [auth-options]
host          database  user  address     auth-method  [auth-options]
hostssl       database  user  address     auth-method  [auth-options]
hostnossl     database  user  address     auth-method  [auth-options]
hostgssenc    database  user  address     auth-method  [auth-options]
hostnogssenc  database  user  address     auth-method  [auth-options]
host          database  user  IP-address  IP-mask      auth-method  [auth-options]
hostssl       database  user  IP-address  IP-mask      auth-method  [auth-options]
hostnossl     database  user  IP-address  IP-mask      auth-method  [auth-options]
hostgssenc    database  user  IP-address  IP-mask      auth-method  [auth-options]
hostnogssenc  database  user  IP-address  IP-mask      auth-method  [auth-options]
```

フィールドの意味は以下のようになっています。

`local`

このレコードはUnixドメインソケットを使用する接続に対応します。この種類のレコードを使用しないと、Unixドメインソケット経由の接続は拒否されます。

`host`

このレコードは、TCP/IPを使用した接続に対応します。hostレコードは、SSLまたは非SSL接続、GSSAPI暗号化、非GSSAPI暗号化のいずれかに対応します。

> **注記**
>
> サーバのデフォルトの動作は、ローカルループバックアドレスであるlocalhostのみTCP/IP接続を監視しています。よってサーバにおいてlisten_addressesパラメータが適切な値に設定された状態で起動されていない限り、リモートのTCP/IP接続はできません。

`hostssl`

このレコードは、接続がSSLで暗号化されている場合にのみTCP/IPネットワークを使用する接続に対応します。

このオプションを使用するためには、サーバはSSLサポートができるように構築されていなければいけません。また、SSLはsslパラメータを設定することによりサーバの起動時に有効になっていなくてはなりません（詳細は18.9を参照してください）。そうでなければ、どのような接続にも対応していないという警告が表示されることを除き、hostsslレコードは無視されます。

`hostnossl`

このレコードは、hostsslと反対の動作で、SSLを使用していないTCP/IPの接続のみに対応します。

hostgssenc

このレコードは、TCP/IPを使用した接続に対応しますが、GSSAPI暗号化を使用して接続が行われた場合に限ります。

このオプションを使用するためには、サーバはGSSAPIサポートができるように構築されていなければいけません。そうでなければ、どのような接続にも対応していないという警告が表示されることを除き、hostgssencレコードは無視されます。

GSSAPI暗号化で使用できる認証方式は、gss、reject、およびtrustのみであることに注意してください。

hostnogssenc

このレコードは、hostgssencとは反対の動作で、GSSAPI暗号化を使用していないTCP/IPの接続のみに対応します。

database

このレコードで対応するデータベース名を指定します。allという値は、全てのデータベースと対応することを指定します。sameuserという値は、要求されたデータベースが要求ユーザと同じ名前を持つ場合にレコードが対応することを指定します。sameroleという値は、要求ユーザが要求されたデータベースと同じ名前のロールのメンバでなければならないことを指定します。(以前はsamegroupと書いていましたが、sameroleと記述してください) スーパーユーザは、直接的であれ間接的であれ、明示的にsameroleのメンバでない限りsameroleのメンバとはみなされません。また、スーパーユーザであるからといってsameroleのメンバとはみなされません。replicationという値は、もし物理レプリケーション接続が要求された場合(レプリケーション接続は特定のデータベースを指定しないことに注意して下さい)にレコードが一致することを指定します。それ以外の場合には、特定のPostgreSQLデータベースの名前になります。データベースの名前はカンマで区切ることで複数指定できます。データベース名を含む別のファイルを、そのファイル名の前に@を付けることで指定できます。

user

このレコードで対応するデータベースユーザを指定します。allという値は、全てのユーザが対応することを指定します。それ以外の場合には特定のデータベースユーザの名前もしくは+で始まるグループ名のどちらかになります。(PostgreSQLではユーザとグループの明確な区別がないことを思い出してください。+のマークは、「このロールの直接的もしくは間接的なメンバのどちらかに一致していること」を意味しています。一方、+のマークのない名前は特定のロールにのみ一致します) このため、スーパーユーザは、直接的であれ間接的であれ明示的にロールのメンバである場合にのみ、ロールのメンバとみなされます。スーパーユーザであるからといってロールのメンバとはみなされません。ユーザ名は、カンマで区切ることで複数指定できます。ユーザ名を含む別のファイルを、そのファイル名の前に@を付けることで指定できます。

address

このレコードに対応しているクライアントマシンのアドレス。このフィールドはホスト名、IPアドレスの範囲、もしくは下記の特別なキーワードの1つを含んでいます。

IPアドレスの範囲は、範囲の開始アドレス、続いてスラッシュ(/)とCIDRマスクの長さという標準の数値表記で指定されます。CIDRマスク長とは、クライアントIPアドレスが一致しなければならない、高位のビット数を表すものです。指定するIPアドレスのこれより右側のビットには、0を指定しなければなりません。IPアドレスと/、およびCIDRマスク長の間には空白を入れてはいけません。

典型的なIPv4アドレス範囲の例は、単一のホストでは172.20.143.89/32、小規模ネットワークでは172.20.143.0/24、大規模ネットワークでは10.6.0.0/16のようなものです。IPv6アドレスの範囲は、単一のホストでは::1/128(この場合はIPv6ループバックアドレス)、小規模ネットワークではfe80::7a31:c1ff:0000:0000/96のようなものです。0.0.0.0/0は全てのIPv4アドレスを意味します。また、::0/0は全てのIPv6アドレスを意味しています。単一ホストを指定するには、IPv4では32、IPv6では128というマスク長を使用してください。ネットワークアドレスでは末尾の0を省略できません。

IPv4書式で与えられたエントリは、IPv4接続のみに対応し、IPv6書式で与えられた項目は、たとえそのアドレスがIPv6内のIPv4の範囲内であったとしてもIPv6接続のみに対応します。IPv6書式の項目は、システムのCライブラリがIPv6アドレスをサポートしていない場合拒絶されることに注意してください。

どのIPアドレスにも一致するようにallと書くこともできますし、サーバ自身のIPアドレスのいずれかにも一致するようにsamehostと書くこともできます。もしくは、サーバが直接接続されているサブネット内のアドレスのいずれかにも一致するようにsamenetと書くことができます。

もし、ホスト名(IPアドレスの範囲ではない場合の全て、もしくはホスト名として処理される特別なキーワード)が指定されている場合は、その名前は、クライアントのIPアドレスの逆引き名前解決の結果と比較されます(例えば、もしDNSが使用されている場合は逆引きDNS検索により解決されます)。ホスト名の比較は、大文字小文字が区別されません。もし一致するものがあった場合は、解決された、どのアドレスもクライアントのIPアドレスと等しいか否かをチェックするために(例えば、正引きDNS検索のような)ホスト名の正引き名前解決が実行されます。もし正引き、逆引きの両方で一致した場合は、エントリは一致するものとみなされます。(pg_hba.conf内で使用されているホスト名は、クライアントのIPアドレスのアドレス-名前解決が返すホスト名の1つでなければいけません。もしそうでなければこの行は一致しません。1つのIPアドレスを複数のホスト名に関連付けるホスト名データベースもありますが、IPアドレスの解決を要求された場合にオペレーティングシステムは1つのホスト名のみを返します。)

ドット(.)で始まるホスト名の特定は実際のホスト名のサフィックスに一致します。よって、.example.comは、foo.example.comに一致します (example.comだけでは一致しません)。

ホスト名がpg_hba.conf内で指定されている場合、名前解決が適度に早いことを 確かめてください。nscdのようなローカル名前解決のキャッシュを設定すると便利です。また、クライアントのIPアドレスの代わりにホスト名がログで見られるように、log_hostnameの 設定パラメータを有効化することもできます。

これらのフィールドはlocalレコードには適用されません。

注記

時折、ユーザは、クライアントのIPアドレスの逆引きを含む2つの名前解決が必要になる、というような一見複雑に見える方法でなぜホスト名が扱われるのか不思議に思うことがあります。このため、クライアントの逆引きDNSエントリが設定されていなかったり、いくつかの望ましくないホスト名を生成する場合にこの機能の使用が複雑になります。これは主に効率のために行なわれます。このように、接続要求では最大2つのリゾルバの検索、1つは逆引き、1つは正引き、 が必要になります。もしリゾルバにおいて、アドレスに問題があった場合、クライアントのみの問題となります。正引き検索のみを行うような実装を仮に行っていると、全ての接続要求においてpg_hba.conf内に記載された全てのホスト名を解決しなくてはいけなくなります。これは、多くの名前が列挙されていた場合にかなり遅くなります。また、リゾルバにおいて1つのホスト名に問題があった場合、全員の問題となってしまいます。

> さらに、逆引き検索はサフィックス一致の機能を実装するために必要です。というのも実際のクライアントのホスト名は ホスト名がパターンに対して一致するために、知られる必要があるためです。
>
> このふるまいは、Apache HTTPサーバやTCPラッパーのような他のよくあるホスト名ベースのアクセス制御の実装と 一致していることに注意して下さい。

`IP-address`
`IP-mask`

この2つのフィールドは`IP-address/mask-length`表記の代替として使用可能です。マスク長を指定する代わりに、実際のマスクを分離した列で指定します。例えば`255.0.0.0`はIPv4のCIDRマスク長8を意味し、`255.255.255.255`はCIDRマスク長32を意味しています。

これらのフィールドは`local`レコードには適用されません。

`auth-method`

接続がこのレコードに一致する場合に使用する認証方式を指定します。使用できる選択肢は以下にまとめていますが、詳しくは20.3を参照してください。

`trust`

接続を無条件で許可します。この方式は、PostgreSQLデータベースサーバに接続できる全てのユーザが、任意のPostgreSQLユーザとしてパスワードや他の認証なしでログインすることを許可します。詳細は20.4を参照してください。

`reject`

接続を無条件に拒否します。特定のホストをあるグループから「除外」するために便利です。例えば、1行の`reject`は特定のホストが接続することを拒否します。一方、後ろの行では特定のネットワーク内の残りのホストが接続することを許可します。

`scram-sha-256`

ユーザのパスワードを検証するためにSCRAM-SHA-256認証を実行します。詳細は20.5をご覧ください。

`md5`

ユーザのパスワードを検証するために、SCRAM-SHA-256あるいはMD5認証を実行します。詳細は20.5を参照してください。

`password`

クライアントに対して認証時に平文のパスワードを要求します。パスワードはネットワークを通じて普通のテキスト形式で送られますので、信頼されていないネットワークでは使用しないでください。詳細は20.5を参照してください。

`gss`

ユーザの認証にGSSAPIを使用します。これはTCP/IP接続を使用するときのみ使用可能です。詳細は20.6を参照してください。GSSAPI暗号化と組み合わせて使用できます。

`sspi`

ユーザの認証にSSPIを使用します。これはWindowsを使用するときのみ使用可能です。詳細は20.7を参照してください。

`ident`

クライアントのオペレーティングシステムにおけるユーザ名をクライアント上のidentサーバに尋ねてユーザ名が要求されたデータベースユーザ名と一致するか検査します。ident認証は、TCP/IP接続でのみ使用可能です。ローカル接続が指定されている場合は、peer認証が代わりに使用されます。詳細は20.8を参照してください。

`peer`

クライアントのオペレーティングシステムにおけるユーザ名をオペレーティングシステムから取得し、ユーザ名が要求されたデータベースユーザ名と一致するか検査します。これはローカル接続の時にのみ使用可能です。詳細は20.9を参照してください。

`ldap`

LDAPサーバを使用して認証します。詳細は20.10を参照してください。

`radius`

RADIUSサーバを使用して認証します。詳細は20.11を参照してください。

`cert`

SSLクライアント証明書を使用して認証します。詳細は20.12を参照してください。

`pam`

オペレーティングシステムによって提供されるPAM（Pluggable Authentication Modules）サービスを使用した認証です。詳細は20.13を参照してください。

`bsd`

オペレーティングシステムによって提供されたBSD認証サービスを使用して認証します。詳細は20.14を参照してください。

`auth-options`

`auth-method`フィールドの後ろに、認証方式のオプションを指定する、`name=value`の形式のフィールドが存在する可能性があります。どのオプションがどの認証方式に使用できるのか、についての詳細は以下で説明します。

以下に示された方式特定のオプションに加えて、方式に依存しないのひとつの認証オプションclientcertがあり、hostssslレコードで指定することができます。このオプションは、`verify-ca`または`verify-full`に設定できます。どちらのオプションも、クライアントに有効な（信頼された）SSL証明書の提出を要求し、`verify full`は、証明書の`cn`（Common Name）がユーザー名または適用可能なマッピングと一致することをさらに強制します。この動作は`cert`認証方式（詳細は20.12を参照してください）に似ていますが、クライアント証明書の検証を`hostssl`エントリをサポートする任意の認証方式と組み合わせることができます。

@式により含められるファイルは、空白文字あるいはカンマのどちらかで区切られた名前の列挙として読み込まれます。コメントは、pg_hba.confと同様に#から始まります。また、@式を入れ子にすることもできます。@の後のファイル名が絶対パスでない限り、参照元ファイルが存在するディレクトリから見た相対パスであるとみなされます。

pg_hba.confレコードは接続が試みられる度に順番に検査されますので、レコードの順序はとても大切です。典型的には、始めの方のレコードには厳しい接続照合パラメータと緩い認証方式があるのに対し、終わりの方のレコードにはより緩い照合パラメータとより厳しい認証方式があります。例えば、ローカルTCP接続ではtrust認証方式、リモートTCP接続に対してはパスワードを要求したいとします。この場合、広範囲にわたって許可されるクライアントのIPアドレスに対するパスワード認証を指定するレコードの前に127.0.0.1からの接続に対するtrust認証指定のレコードが置かれなければなりません。

pg_hba.confファイルは起動時と、主サーバプロセスがSIGHUPシグナルを受け取った時に読み込まれます。稼働中のシステムでファイルを編集した場合は、(pg_ctl reloadの使用、SQL関数のpg_reload_conf()の呼び出し、またはkill -HUPを使用して)postmasterにファイルをもう一度読み込むようにシグナルを出さなければなりません。

> **注記**
>
> 上記はマイクロソフトWindowsに対しては当てはまりません。つまり、pg_hba.confに対する変更は、ただちにそれ以降の新しい接続に反映されます。

pg_hba.confに対する変更をテストする際、あるいはそのファイルをロードしても期待していた結果が得られなかった場合には、システムビューpg_hba_file_rulesが役に立ちます。そのビューのerrorフィールドがNULLでない行は、そのファイルの該当行に問題があることを示しています。

> **ヒント**
>
> 特定のデータベースに接続するためには、ユーザはpg_hba.confによる検査を通過しなければならない他、そのデータベースに対するCONNECT権限を持たなければなりません。どのユーザがどのデータベースに接続できるかを制限したければ、通常、pg_hba.conf項目に規則を追加するよりも、CONNECT権限の付与・削除を行う方が簡単です。

pg_hba.confファイルの例をいくつか例 20.1に示します。各種認証方式の詳細についてはその後で説明します。

例20.1 pg_hba.confの項目の例

```
# ローカルシステム上の全てのユーザが、
      任意のデータベースに
# 任意のデータベースユーザ名でUnixドメインソケットを使用して接続することを許可
# （ローカル接続ではデフォルト）。
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
local   all             all                                     trust
```

```
# 上記と同じことをローカルループバックのTCP/IP接続を使って行なう。
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    all             all             127.0.0.1/32            trust


# 上記と同じだが、
      独立したネットマスク列を使用する
#
# TYPE  DATABASE        USER            IP-ADDRESS      IP-MASK             METHOD
host    all             all             127.0.0.1       255.255.255.255     trust


# IPv6で上記と同じことを行なう。
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    all             all             ::1/128                 trust


# ホスト名を使用して上記と同じことを行なう（通常はIPv4とIPv6の両方をカバーします）。
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    all             all             localhost               trust


# IPアドレス192.168.93.xを持つ全てのホストの全てのユーザが、

# identがその接続について報告するのと同じユーザ名（典型的にはオペレーティングシステムのユーザ名）
で
# データベース「postgres」へ接続することを許可。
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    postgres        all             192.168.93.0/24         ident


# ユーザのパスワードが正しく入力された場合、

# ホスト192.168.12.10からのどのようなユーザでもデータベース「postgres」へ接続することを許可。
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    postgres        all             192.168.12.10/32        scram-sha-256


# ユーザのパスワードが正しく指定された場合は、
```

```
# example.comドメイン内のホストからの、
      どのユーザからのデータベース接続も許可する。
#
# Require SCRAM authentication for most users, but make an exception
# for user 'mike', who uses an older client that doesn't support SCRAM
# authentication.
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    all             mike            .example.com            md5
host    all             all             .example.com            scram-sha-256


# 先行する「host」行がなければ、
      これら3行によって、

# 192.168.54.1からの接続の試みを全て拒否（この項目が最初に照合されるため）、

# ただし、
      インターネット上の他の全ての場所からのGSSAPI接続は許可。
# ゼロマスクは、
      ホストIPアドレスのビットが考慮されずに
# どのホストでも照合できることになる。
# 暗号化されていないGSSAPI接続（「hostgssenc」は暗号化されたGSSAPI接続
# にのみに一致するので、
      3行目までは「通過」）は許可されるが、
      192.168.12.10からのみ許可。
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    all             all             192.168.54.1/32         reject
hostgssenc all          all             0.0.0.0/0               gss
host    all             all             192.168.12.10/32        gss


# 192.168.x.xホストからのユーザが、
      ident検査に通る場合、

# どのデータベースにでも接続を許可。もし、
      例えば、
      identが「bryanh」と認定し
# 「bryanh」がPostgreSQLのユーザ「guest1」として
# 接続要求を出す場合、
      「bryanh」は「guest1」として接続が許可されるという
# マップ「omicron」に対する記載事項がpg_ident.confにあれば接続を許可。
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    all             all             192.168.0.0/16          ident map=omicron
```

```
# ローカル接続に対して、
      以下のたった3行しか記載がない場合、
      ローカルユーザは
# 自分のデータベース（データベースユーザ名と同じ名前のデータベース）にのみ接続許可。
# ただし管理者とロール「support」のメンバは全てのデータベースに接続可能。
# $PGDATA/adminsファイルは管理者のリストを含む。
# 全ての場合にパスワードが必要。
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
local   sameuser        all                                     md5
local   all             @admins                                 md5
local   all             +support                                md5


# 上記の最後の2行は1つの行にまとめることが可能。
local   all             @admins,+support                        md5


# データベースの列にはリストやファイル名も使用できる。
local   db1,db2,@demodbs  all                                   md5
```

20.2. ユーザ名マップ

identやGSSAPIといった外部の認証システムを使用する場合は、接続を開始したオペレーティングシステムのユーザ名が接続先のデータベースユーザ（ロール）名と同じであるとは限りません。ユーザ名マップを使用するには、pg_hba.conf内でmap=map-nameオプションを指定してください。このオプションは、外部ユーザ名を受け取るすべての認証方式をサポートしています。異なる接続に対して、異なるマップが必要となる可能性があります。そのため、それぞれの接続に対して使用されるマップを指定するために、使用するマップの名称はpg_hba.conf内のmap-nameパラメータで指定されます。

ユーザ名マップはidentマップファイルに定義されています。デフォルトではファイル名はpg_ident.confでクラスタのデータディレクトリに保存されています。（他の場所にも保存できますが、詳細はident_file設定パラメータを参照してください。）identマップファイルは一般的な形式の行を含んでいます。

```
map-name system-username database-username
```

コメントと空白はpg_hba.confと同様に扱われます。map-nameは pg_hba.conf内で参照される任意の名称です。他の2つのフィールドは、どのオペレーティングシステムユーザが、どのデータベースユーザに接続することを許可されているかを指定しています。同じmap-nameは、1つのマップ内でユーザをマップするために繰り返し使用されます。

どれだけのデータベースユーザがオペレーティングシステムのユーザに対して一致しているか、またその逆に対しても制限はありません。よってマップ内のエントリは、それらが等しいというよりもむしろ「このオペレー

ティングシステムのユーザはこのデータベースユーザとして接続する」という意味になります。もし外部の認証システムから得られたユーザ名と接続要求を行ったデータベースユーザ名が対となるエントリがマップ内にある場合は、接続は許可されます。

もしsystem-usernameフィールドがスラッシュ(/)で始まっている場合は、このフィールドの残りは正規表現として扱われます。(PostgreSQLの正規表現構文の詳細については9.7.3.1を参照してください。) 正規表現は単一検索や括弧を使用した表現、database-usernameフィールドで\1(バックスラッシュ-1)で参照されるような表現を含みます。これにより、1行で複数のユーザ名のマップが可能となり、簡単な構文で特に使いやすくなります。例を以下に示します。

```
mymap   /^(.*)@mydomain\.com$      \1
mymap   /^(.*)@otherdomain\.com$   guest
```

上記のエントリでは、@mydomain.comで終わるシステムユーザ名のドメイン部分を削除して、@otherdomain.comで終わるシステムユーザ名がguestとしてログインすることを許可します。

> **ヒント**
>
> デフォルトでは正規表現は、文字列の一部を一致させることに注意してください。上記の例で示したように、システムユーザ名全体を強制的に一致させるために^や$を使用すると有用です。

pg_ident.confファイルは起動時と、メインサーバのプロセスが SIGHUPを受信したときに読み込まれます。起動しているシステムで編集した場合は、ファイルを再読み込みするために(pg_ctl reloadの使用、SQL関数pg_reload_conf()の呼び出し、またはkill -HUPを使用して)postmasterにシグナルを送信する必要があります。

pg_ident.confファイルは、pg_hba.confファイルと結合して使用されます。例 20.2に例 20.1の例があります。この例では、192.168のネットワーク内のマシンにログインしている、オペレーティングシステムのユーザ名でbryanh、ann、robert以外の誰もが、アクセスを許可されていません。Unixユーザrobertはは PostgreSQLユーザであるbobとして接続しようとした時のみ アクセス可能で、robertや他の名前ではアクセスできません。annはannとして接続した時のみ許可され、bryanhはbryanh自身もしくはguest1として アクセスが可能となります。

例20.2 pg_ident.confファイルの例

```
# MAPNAME       SYSTEM-USERNAME         PG-USERNAME

omicron         bryanh                  bryanh
omicron         ann                     ann

# bobはこれらのマシン内でrobertというユーザ名を持っています。
omicron         robert                  bob

# bryanhはguest1としても接続可能です。
omicron         bryanh                  guest1
```

20.3. 認証方式

PostgreSQLでは、様々なユーザ認証方式を提供しています。

- trust認証は、ユーザーが本人であることを単純に信頼します。
- パスワード認証は、ユーザにパスワードが必要であることを示します。
- GSSAPI認証は、GSSAPI互換のセキュリティライブラリに依存します。通常、これはKerberosまたはMicrosoft Active Directoryサーバなどの認証サーバにアクセスするために使用されます。
- SSPI認証は、GSSAPIに似たWindows固有のプロトコルを使用します。
- ident認証は、クライアントのマシン上の「Identification Protocol」(RFC 1413)サービスに依存します。(ローカルのUnixドメインソケット接続では、これはpeer認証として扱われます)。
- peer認証は、オペレーティングシステムの機能に依存して、ローカル接続の相手側のプロセスを識別します。これはリモート接続ではサポートされません。
- LDAP認証は、LDAP認証サーバに依存します。
- RADIUS認証は、RADIUS認証サーバに依存します。
- 証明書認証は、SSL接続を必要とし、送信されるSSL証明書をチェックしてユーザを認証します。
- PAM認証は、PAM(Pluggable Authentication Modules)ライブラリに依存します。
- BSD認証は、BSD認証フレームワーク(現在はOpenBSDでのみ利用可能)に依存します。

peer認証は、通常ローカル接続に推奨されますが、trust認証で十分な場合もあります。パスワード認証は、リモート接続の最も簡単な選択肢です。その他のオプションはすべて、何らかの外部セキュリティ基盤(通常は、認証サーバやSSL証明書を発行するための認証局)を必要とするか、またはプラットフォーム固有のものです。

以下の節では、これらの認証方式についてそれぞれ詳しく説明します。

20.4. Trust認証

trust認証が指定されるとPostgreSQLは、サーバに接続できる全ての人に対して(データベーススーパーユーザさえも)その人が指定する任意のデータベースユーザ名としてのアクセス権限が付与されていると想定します。当然ながらdatabaseとuser列にある制限は適用されます。この方式はサーバに接続する際に適切なオペレーティングシステムレベルの保護が掛けられている場合にのみ使用すべきです。

trust認証はユーザが1人のみのワークステーション上でローカル接続を行う場合は適切であると同時に非常に便利です。複数ユーザが存在するマシン上では一般的に適切ではありません。とは言っても、ファイルシステムの許可属性を使ってサーバのUnixドメインソケットファイルへのアクセスを制限すればtrust認証を複数ユーザのマシン上で使用することも可能です。その方法は、19.3に記載されているよう

にunix_socket_permissions（およびunix_socket_groupパラメータの可能性もあります）パラメータを設定します。もしくは、unix_socket_directories設定パラメータでソケットファイルをそれに相応しく制限されているディレクトリにします。

ファイルシステムの許可を設定することが役に立つのはUnixソケット接続だけです。ローカルのTCP/IP接続は、ファイルシステムにより制限はされていません。よってローカルでファイルシステムの許可を使用したい場合はpg_hba.confから host ... 127.0.0.1 ...の行を削除するか、trust認証とは異なる方法に変更する必要があります。

TCP/IP接続におけるtrust認証は、trustを指定するpg_hba.confの行によってサーバに接続を許可される全てのマシン上の全てのユーザを信用（trust）できる場合にのみ相応しいものです。localhost（127.0.0.1）以外からのTCP/IP接続にtrust認証を用いる理由はほとんど見当たりません。

20.5. パスワード認証

複数のパスワードに基づく認証方式があります。これらは似たような方法で使用されますが、ユーザのパスワードをサーバに格納する方法と、クライアントによって提供されたパスワードが接続を通じて送信される方法が異なります。

scram-sha-256

scram-sha-256方式は、RFC 7677[1]に記述された方法でSCRAM-SHA-256認証を実行します。これはチャレンジ／レスポンス方式のひとつであり、信頼できない接続におけるパスワードの漏洩を防ぎ、安全だと見なされる暗号学的ハッシュ形式でパスワードをサーバに格納するのを支援します。

これは、現在実装されている認証方式の中では最も安全ですが、古いクライアントライブラリではサポートされていません。

md5

md5方式は、独自のより安全性の低いチャレンジ／レスポンス機構を使います。パスワードの漏洩を防ぎ、平文でパスワードをサーバに格納するのを避けることができますが、攻撃者がサーバからパスワードハッシュを盗むことを防ぐことはできません。また、MD5ハッシュアルゴリズムは、昨今では強い意志をもった攻撃に対しては安全ではないと考えられています。

md5は、db_user_namespace機能と一緒に使用することはできません。

md5方式からより新しいSCRAM方式への移行を容易にするため、pg_hba.confでmd5が指定されているにもかかわらず、パスワードがSCRAM（下記参照）で暗号化されている場合には、自動的にSCRAMに基づく認証が代わりに使用されます。

password

password方式は、パスワードを平文で送信するので、パスワード「盗聴」攻撃に対して脆弱です。可能ならば、常に避けるようにしてください。しかしながら、接続がSSL暗号で保護されていれば、passwordは

[1] https://tools.ietf.org/html/rfc7677

安全に使用できます。(もっとも、SSLの利用に依存するのであれば、SSL証明書認証がより良い選択かもしれません。)

PostgreSQLデータベースパスワードはオペレーティングシステムのユーザパスワードとも別のものです。各データベースユーザのパスワードはpg_authidシステムカタログテーブルの中に格納されます。**CREATE ROLE foo WITH LOGIN PASSWORD 'secret'**のように、パスワードはSQLコマンドCREATE ROLEとALTER ROLEを使って管理できます。あるいは、psqlの\passwordコマンドでも管理できます。もしユーザに対してパスワードが設定されない場合、格納されるパスワードはNULLとなり、そのユーザのパスワード認証は常に失敗します。

パスワードにもとづく異なる認証方式が利用可能かどうかは、サーバ上でユーザのパスワードがどのように暗号化(正確には、ハッシュ化)されるのかに依存します。これは、パスワードが設定されたときに、設定パラメータのpassword_encryptionによって制御されます。パスワードがscram-sha-256によって暗号化されていれば、認証方式のscram-sha-256とpasswordで利用できます。(ただし、後者の場合にはパスワードの転送は平文になります。) 前述のように、ここで認証方式のmd5を指定すると、scram-sha-256方式に自動的に切り替わります。パスワードがmd5で暗号化されていると、md5とpasswordでのみ使用されます。(ここでも、後者の場合にはパスワードは平文で転送されます。) (以前のPostgreSQLのリリースでは、パスワードを平文で格納することをサポートしていました。これはもはや不可能です。) 現在格納されているパスワードのハッシュを確認するには、システムカタログpg_authidを参照してください。

既存のインストールにおいて、md5からscram-sha-256にアップグレードするには、すべてのクライアントライブラリが十分新しく、SCRAMをサポートをできることを確認してから、postgresql.confでpassword_encryption = 'scram-sha-256'を設定し、すべてのユーザに新しいパスワードを設定してもらい、pg_hba.confの認証方式をscram-sha-256に変更してください。

20.6. GSSAPI認証

GSSAPIは、RFC 2743で定義されている安全な認証のための業界標準のプロトコルです。PostgreSQLは、GSSAPIをサポートしています。このGSSAPIは、暗号化された認証済みレイヤとして使用するか、認証のみに使用します。GSSAPIは、GSSAPIをサポートするシステムに自動認証(シングルサインオン)を提供します。認証自体は安全です。GSSAPI暗号化(詳細はhostgssencを参照してください)、またはSSL暗号化を使用すると、データベース接続に沿って送信されるデータは暗号化されますが、それ以外の場合は暗号化されません。

GSSAPIサポートは、PostgreSQLを構築する時に有効にしなければなりません。詳細は、第16章を参照してください。

GSSAPIがKerberosを使用しているとき、GSSAPIは、servicename/hostname@realmという書式の標準のプリンシパルを使用します。[訳注:プリンシパルとは大雑把に2つのものを指します。1つはサービスを受けるクライアントで、もう1つはサービスを提供するサーバアプリケーションです。どちらも、認証に関してはKerberosのKDCから見るとクライアントになります] PostgreSQLサーバはサーバにより使われるkeytabに含まれるいかなるプリンシパルも受け付けますが、krbsrvname接続パラメータを使ってクライアントから接続する場合には、プリンシパルの詳細を正確に指定することに注意を払う必要があります。(33.1.2も参照してください。) ビルド時に./configure --with-krb-srvnam=whateverを使用することで、インストール時のデフォルトはデフォルトのpostgresから変更が可能です。多くの環境では、このパラメータは変更する必要はないでしょう。いくつかのKerberosの実装では、異なるサービス名が必要になります。Microsoftアクティブディレクトリではサービス名は(POSTGRES)のように大文字にする必要があります。

hostnameはサーバマシンの完全修飾されたホスト名です。サービスプリンシパルのrealmはサーバマシンが提起したrealmです。

クライアントのプリンシパルはpg_ident.confで異なるPostgreSQLのデータベースユーザ名にマップできます。例えば、pgusername@realmを単なるpgusernameにマップできます。もう1つの方法として、プリンシパル名全体username@realmをPostgreSQLのロール名としてマッピングなしに使うこともできます。

PostgreSQLはプリンシパルからrealmを外すパラメータもサポートしています。この方法は後方互換のためにサポートされているものであり、異なるrealmから来た同じユーザ名の異なるユーザを区別することができませんので、使用しないことを強く薦めます。この方法を有効にするにはinclude_realmを0に設定してください。単純な単一realmの設定では、(プリンシパルのrealmがkrb_realmパラメータ内のものと正確に一致するか確認する)krb_realmパラメータと組み合わせることが安全です。しかし、これはpg_ident.confで明示的なマッピングを指定するのに比べてあまり適切でない選択でしょう。

サーバ鍵ファイルがPostgreSQLサーバアカウントによって読み込み可能(そしてできれば読み込み専用で書き込み不可)であることを確認してください。(18.1を参照してください。) 鍵ファイルの保存場所はkrb_server_keyfile設定パラメータで指定されます デフォルトは、/usr/local/pgsql/etc/krb5.keytab(もしくはビルド時にsysconfdirで指定されたディレクトリ)です。セキュリティ上の理由から、システムkeytabファイルで許可するよりも、PostgreSQLサーバ用に別のkeytabファイルを使うことをお薦めします。

keytabファイルはKerberosのソフトウェアによって作成されます。詳細はKerberosのドキュメントを参照してください。MIT互換のKerberos5実装の例を以下に示します。

```
kadmin% ank -randkey postgres/server.my.domain.org
kadmin% ktadd -k krb5.keytab postgres/server.my.domain.org
```

データベースに接続しようとしている時要求されるデータベースユーザ名に一致するプリンシパルのチケットを所有しているか確認してください。例えば、データベースユーザ名fredに対し、fred@EXAMPLE.COMのプリンシパルは接続できるでしょう。fred/users.example.com@EXAMPLE.COMのプリンシパルも許可するためには20.2内に記述されているユーザ名マップを使用して下さい。

次の設定オプションはGSSAPIのためにサポートされています。

include_realm

0に設定されている場合は、認証されたユーザプリンシパルからのrealm名が、ユーザ名マッピング(20.2)で渡されるシステムユーザ名から外されています。krb_realmも一緒に使われていない限り、これは複数realm環境で安全ではありませんので、非推奨であり、主に後方互換性のために利用できます。include_realmをデフォルト(1)にしたまま、プリンシパル名をPostgreSQLユーザ名に変換するためにpg_ident.confで明示的なマッピングを指定することをお薦めします。

map

システムとデータベースの間のマッピングを許可します。詳細は20.2を参照してください。GSSAPI/Kerberosプリンシパルusername@EXAMPLE.COM(もしくは、あまり一般的ではありませんがusername/hostbased@EXAMPLE.COM)に対しては、もしinclude_realmが0に設定されていない限り、マッピングに使われるユーザ名はusername@EXAMPLE.COM(もしくはusername/hostbased@EXAMPLE.COM)です。0に設定されている場合には、username(もしくはusername/hostbased)がマッピング時のシステムユーザ名です。

krb_realm

realmをユーザプリンシパル名に一致するように設定します。もしこのパラメータが設定されている場合はそのrealmのユーザのみが受け付けられます。もしこれが設定されていない場合は、どのようなrealmのユーザも接続可能で、ユーザ名マッピングが設定されていれば、どれでも影響を受けます。

20.7. SSPI認証

SSPIは、シングルサインオンで安全な認証を行うためのWindowsの技術です。PostgreSQLは、negotiateモードにおいてSSPIを使用します。これは、可能な場合はKerberosを使用し、他の場合については自動的にNTLMを使用することを意味しています。SSPI認証は、サーバ、クライアントが共にWindows上もしくはGSSAPIが利用可能な場合はWindowsではないプラットフォームで稼動しているときにのみ動作します。

Kerberos認証を使用しているとき、SSPIは、GSSAPIと同じように動作します。詳細は20.6を参照してください。

次の設定オプションはSSPIのためにサポートされています。

include_realm

0に設定されている場合は、認証されたユーザプリンシパルからのrealm名が、ユーザ名マッピング(20.2)で渡されるシステムユーザ名から外されています。krb_realmも一緒に使われていない限り、これは複数realm環境で安全ではありませんので、非推奨であり、主に後方互換性のために利用できます。include_realmをデフォルト(1)にしたまま、プリンシパル名をPostgreSQLユーザ名に変換するためにpg_ident.confで明示的なマッピングを指定することをお薦めします。

compat_realm

1に設定されている場合は、(NetBIOS名としても知られている)ドメインのSAM互換名がinclude_realmオプションのために使用されます。これはデフォルトの動作です。0に設定されている場合は、ケルベロスユーザプリンシパル名からの真のrealm名が使用されます。

ドメインアカウント(これはドメインメンバーシステムの仮想サービスアカウントを含みます)にて実行されているサーバで、SSPIで認証されているすべてのクライアントがドメインアカウントを使用してる場合を除き、このオプションを無効にしないでください。さもなくば認証は失敗します。

upn_username

compat_realmと共にこのオプションが有効の場合、認証にはケルベロスUPNからユーザ名が使用されます。無効(デフォルト)である場合は、SAM互換ユーザ名が使用されます。デフォルトでは、これらの2つのユーザ名は新しいユーザアカウントでは同じものとなります。

明示的なユーザ名が指定されない場合、libpqはSAM互換名を使用することに注意してください。libpqもしくはlibpqを基礎としたドライバを使用する場合は、このオプションを無効のままにするか、明示的なユーザ名を接続文字列にて指定してください。

map

システムとデータベースユーザ名の間のマッピングを許可します。詳細は20.2を参照してください。SSAPI/Kerberosプリンシパルusername@EXAMPLE.COM(もしくは、あまり一般的ではありません

がusername/hostbased@EXAMPLE.COM)に対しては、もしinclude_realmが0に設定されていない限り、マッピングに使われるユーザ名はusername@EXAMPLE.COM(もしくはusername/hostbased@EXAMPLE.COM)です。0に設定されている場合には、username(もしくはusername/hostbased)がマッピング時のシステムユーザ名です。

krb_realm

realmをユーザプリンシパル名に一致するように設定します。もしこのパラメータが設定されている場合は realmのユーザのみが受け付けられます。もしこれが設定されていない場合は、どのようなrealmのユーザも接続可能で、ユーザ名マッピングが設定されていれば、どれでも影響を受けます。

20.8. Ident認証

ident認証方式は、クライアントのオペレーティングシステムのユーザ名をidentサーバから入手し、それを(オプションのユーザ名マップとともに)許可されているデータベースのユーザ名として使用します。これはTCP/IP接続のみサポートされます。

> **注記**
>
> identが(TCP/IPではない)ローカル接続で指定されている場合、peer認証(20.9を参照してください)が代わりに使用されます。

次の設定オプションはidentのためにサポートされています。

map

システムとデータベースユーザ名の間のマッピングを許可します。詳細は20.2を参照してください。

「身元特定(Identification)プロトコル」についてはRFC 1413で説明されています。事実上全てのUnix系のオペレーティングシステムの配布には、デフォルトでTCPポート113を監視するidentサーバが付属しています。identサーバの基本的な機能は「どのユーザがポートXからの接続を開始し、自分のポートYへの接続を初期化したのか?」というような質問に答えることです。PostgreSQLは物理的な接続が確立された時にXとYの両方を認識するので、接続するクライアントのホスト上のidentサーバに応答指令信号を送ることができ、理論的には与えられたどの接続にもオペレーティングシステムユーザを決定できます。

この手続きの欠点は、クライアントの正直さに頼るところが大きいということです。もしクライアントマシンが信用されない、もしくは危険に晒されている場合、攻撃者はポート113上でほぼどんなプログラムでも実行することができ、どのユーザ名でも好きに選んで返すことができます。したがってこの認証方式は、各々のクライアントマシンが厳格な管理下にあり、データベースとシステム管理者が密接に連絡を取り合って動作している、外界から閉ざされたネットワークにのみ適していると言えます。言い換えると、identサーバが稼働しているマシンを信用しなければなりません。次の警告に注意してください。

> 身元特定プロトコルは、認証、あるいはアクセス管理プロトコルには意図されていません。
>
> —RFC 1413

いくつかの身元特定サーバは、ユーザ名を(マシンの管理者のみが知っているキーで)暗号化して返すような非標準のオプションを持っています。このオプションは、身元特定サーバとPostgreSQLとを一緒に使用す

る場合には、使用してはいけません。理由はPostgreSQLは、返された文字列を復号化して本当のユーザを決定するための手段を持っていないためです。

20.9. Peer認証

peer認証方式はカーネルからクライアント上のオペレーティングシステムのユーザ名を取得し、それをデータベースユーザ名（オプションのユーザ名マップとともに）として使用することにより動作します。この方法はローカル接続でのみ使用可能です。

次の設定オプションはpeerのためにサポートされています。

`map`

システムとデータベースのユーザ名のマッピングを許可します。詳細は20.2を参照してください。

peer認証はオペレーティングシステムが、`getpeereid()`関数、`SO_PEERCRED`のソケットパラメータ、もしくは同じような仕組みを提供しているときにのみ使用可能です。現状では、Linux、OS Xを含むBSD系、そしてSolarisに含まれています。

20.10. LDAP認証

この認証方式はpasswordと似ていますが、パスワード確認にLDAPを使用する点が異なります。LDAPはユーザの名前とパスワードの組み合わせの検証のみに使用されます。そのため、LDAPを使用して認証を行うようにする前に、ユーザはデータベースに存在しなければなりません。

LDAP認証は2つのモードで動作します。1つ目のモードでは、それは単なるバインド・モードを呼び出すものですが、サーバは`prefix username suffix`として区別された名前にバインドします。一般的に、`prefix`パラメータはActive Directory環境での`cn=`や`DOMAIN\`を特定するために使用されます。`suffix`は、Active Directory環境ではない場合でのDNの残りの部分を特定するために使用されます。

2つ目のモードでは、それはsearch/bindモードを呼び出すもので、サーバは最初に`ldapbinddn`と`ldapbindpasswd`で指定された、固定されたユーザ名とパスワードを使用してLDAPディレクトリにバインドします。それからデータベースにログインしようとしているユーザを検索します。もしユーザとパスワードが指定されていなかった場合は、ディレクトリに対して匿名でバインドします。検索は`ldapbasedn`のサブツリーまで行われ、`ldapsearchattribute`で指定された属性に正確に一致するかどうかまで行われます。この検索において、一度ユーザが見つかるとサーバは切断して、クライアントで指定されたパスワードを使用してこのユーザとして再度ディレクトリにバインドします。これはそのログインが正しいかどうかを検証するためです。このモードはApache `mod_authnz_ldap`および`pam_ldap`などの他のソフトウェアと同じように、LDAP認証の仕組みで使用されるものと同じです。この方法は、ユーザオブジェクトがディレクトリに配置されている場合に、かなりの柔軟性があります。しかし、LDAPサーバへの2つの分離した接続が作成されます。

次の設定オプションは両方のモードで使用されます。

`ldapserver`

接続するLDAPサーバの名称もしくはIPアドレス。空白で区切ることで複数のサーバを指定できます。

ldapport

LDAPサーバに接続するためのポート番号。もしポートが指定されていない場合は LDAPライブラリ内のデフォルトポート設定が使用されます。

ldapscheme

ldapsに設定するとLDAPSを使用します。これはいくつかのLDAPサーバーの実装でサポートされている、SSL経由のLDAPを使用する非標準の方法です。代替方法については、ldaptlsオプションを参照してください。

ldaptls

1に設定すると、PostgreSQLとLDAPサーバ間の接続にTLSによる暗号化を使用します。これはRFC 4513のStartTLS操作を使用します。代替方法については、ldapschemeオプションを参照して下さい。

ldapschemeやldaptlsを使うときにはPostgreSQLサーバとLDAPサーバ間のトラフィックのみが暗号化されることに注意して下さい。SSLがそこでも使用されていない限り、PostgreSQLサーバとPostgreSQLクライアントとの接続は、暗号化されません。

以下のオプションは単純バインド・モードのみで使用されます。

ldapprefix

単純なバインド認証を行う場合のDNを生成する際にユーザ名の前に追加する文字列

ldapsuffix

単純なバインド認証を行う場合のDNを生成する際にユーザ名の後に追加する文字列

以下のオプションはsearch/bindモードのみで使用されます。

ldapbasedn

検索とバインドの認証を行う場合のユーザ名がログインするための検索を始めるためのルートDN

ldapbinddn

検索とバインドの認証を行う場合のディレクトリと検索をバインドするためのユーザのDN

ldapbindpasswd

検索とバインドの認証を行う場合のディレクトリと検索をバインドするためのユーザのパスワード

ldapsearchattribute

検索とバインドの認証を行う場合の検索時のユーザ名に対して一致させる属性。属性が指定されない場合、属性uidが使用されます。

ldapsearchfilter

search/bind認証を行うときに使用する検索フィルタです。$usernameの出現はユーザ名に置き換えられます。これによりldapsearchattributeよりも柔軟な検索フィルタが可能になります。

ldapurl

RFC 4516 LDAP URL。これはその他いくつかのLDAPオプションをより簡潔、かつ一般的な形式で記述する別の方法です。フォーマットは以下のようになっています。

```
ldap[s]://host[:port]/basedn[?[attribute][?[scope][?[filter]]]]
```

scopeはbase、one、subのいずれかでなくてはならず、一般的には最後のものです。(デフォルトはbaseです。これは通常このアプリケーションでは役に立ちません。) attributeは単一の属性を指定できます。その場合、それはldapsearchattributeの値として使用されます。もしattributeが空の場合は、ldapsearchfilterの値としてfilterを使用することができます。

URLスキームldapsは、ldapscheme=ldapsを使用するのと同じ、SSL上のLDAP接続をするLDAPS方式を選択します。StartTLS操作による暗号化されたLDAP接続を使用するには、通常のURLスキームldapを使用し、ldapurlに加えldaptlsオプションを使用しなければなりません。

非匿名バインド(non-anonymous bind)に対し、ldapbinddnおよびldapbindpasswdは個別のオプションとして指定されなければなりません。

LDAP URLは現在、OpenLDAPのみでサポートされており、Windowsではサポートされていません。

seartch/bindオプションと単純バインドに対するオプションの設定を混在させるのはエラーです。

search/bindモードを使用するときは、ldapsearchattributeで指定される単一の属性を使って、あるいはldapsearchfilterで指定されるカスタム検索フィルターを使って、検索を実行できます。ldapsearchattribute=fooの指定は、ldapsearchfilter="(foo=$username)"と同等です。どちらのオプションもない場合は、ldapsearchattribute=uidがデフォルトです。

PostgreSQLが、LDAPクライアントライブラリとしてOpenLDAPを使用するようにコンパイルされていた場合、ldapserverの設定は省略出来ます。その場合、ホスト名とポート番号のリストは、RFC 2782 DNS SRVレコードを使用して検索されます。_ldap._tcp.DOMAINという名前が検索され、ldapbasednからDOMAINが抽出されます。

以下に単純バインドLDAP設定の例を示します。

```
host ... ldap ldapserver=ldap.example.net ldapprefix="cn=" ldapsuffix=", dc=example, dc=net"
```

データベースのユーザ、someuserからデータベースサーバに接続を要求された場合、PostgreSQLはDN cn=someuser, dc=example, dc=netおよびクライアントから提供されたパスワードを用いてLDAPサーバにバインドを試みます。その接続が成功すればデータベースへのアクセスが認められます。

以下はsearch/bind設定の例です。

```
host ... ldap ldapserver=ldap.example.net ldapbasedn="dc=example, dc=net" ldapsearchattribute=uid
```

データベースユーザsomeuserとしてデータベースに接続するとき、PostgreSQLは(ldapbinddnが指定されていないので)匿名的にバインドを試み、指定されたベースDNの基で(uid=someuser)の検索を行います。あるエントリが見つかると、見つかった情報とクライアントから与えられたパスワードを用いて、その結果バインドを試みます。その二番目の接続が成功するとデータベースアクセスが認められます。

URLとして記述した同じsearch/bind設定の例です。

```
host ... ldap ldapurl="ldap://ldap.example.net/dc=example,dc=net?uid?sub"
```

LDAPに対し認証をサポートする幾つかの他のソフトウェアは同じURLフォーマットを使用します。従って、設定をより簡易に共有することができます。

ldapsearchattributeの代わりにldapsearchfilterを使用してユーザーIDまたは電子メールアドレスによる認証を可能にするsearch/bind設定の例です。

```
host ... ldap ldapserver=ldap.example.net ldapbasedn="dc=example, dc=net" ldapsearchfilter="(|
(uid=$username)(mail=$username))"
```

DNS SRV検出を使用してドメイン名example.netのLDAPサービスのホスト名とポート番号を検索する、search/bind設定の例です。

```
host ... ldap ldapbasedn="dc=example,dc=net"
```

ヒント

LDAPはDNの異なる構成要素を区切るために往々にしてコンマとスペースを使用します。例で示されたように、LDAPオプションを設定する場合、二重引用符で括られたパラメータ値を使用することがしばしば必須となります。

20.11. RADIUS認証

この認証方法は、RADIUSをパスワード検証として使用するという点を除いてpasswordと似た動作をします。RADIUSはユーザ名/パスワードの組のみを検証するために使用されます。よってユーザはRADIUSが認証に使用される以前にデータベースにすでに存在していなければいけません。

RADIUS認証を使用する場合に、設定されたRADIUSサーバにアクセスリクエストメッセージが送信されます。このリクエストはAuthenticate Onlyの形式になり、ユーザ名,（暗号化された）パスワード、NAS Identifierを含んでいます。リクエストはサーバと共有している秘密を用いて暗号化されます。RADIUSサーバは、このリクエストに対してAccess AcceptもしくはAccess Rejectを返します。RADIUSアカウントのサポートはありません。

複数のRADIUSサーバを指定することができ、その場合には各々が順に試行されます。サーバから負の応答があると、認証は失敗します。サーバから応答がない場合は、リスト内の次のサーバが試されます。複数のサーバを指定するには、サーバ名をカンマで区切り、リストを二重引用符で囲みます。複数のサーバを指定した場合は、別のRADIUSオプションをカンマ区切りのリストとして指定して、各サーバの値を個別に指定することもできます。オプションは単一の値としても指定でき、その場合にはこの値がすべてのサーバに対して適用されます。

RADIUSのために次の設定オプションがサポートされています。

radiusservers

接続するRADIUSサーバのDNS名称もしくはIPアドレス。このパラメータは必須です。

radiussecrets

RADIUSサーバと安全なやり取りに使用される共有の秘密データ。これはPostgreSQLとRADIUSサーバにおいて厳密に同じ値にする必要があります。少なくとも16文字以上の文字列が推奨されます。このパラメータは必須です。

> **注記**
>
> 使用されている暗号化ベクターはPostgreSQLがOpenSSLをサポートするよう構築している場合にのみ暗号論的に強力です。他の場合にはRADIUSサーバへの伝送は難読化されているだけで安全ではなく、必要ならば外部のセキュリティ方法を適用すべきです。

radiusports

接続するRADIUSサーバのポート番号。もしポート番号が指定されていない場合は、デフォルトのRADIUSポートである1812が使用されます。

radiusidentifiers

RADIUSリクエスト内でNAS Identifierとして使用される文字列。このパラメータは、例えばユーザがどのデータベースクラスタに接続しようとしているかを識別するために使用できます。これはRADIUSサーバにおいてポリシーを一致させるのに便利です。もし識別子が指定されていない場合は、デフォルトのpostgresqlが使用されます。

RADIUSパラメータ値にカンマまたは空白を含める必要がある場合は、値を二重引用符で囲むことで実行できますが、二重引用符の2つのレイヤーが必要になるため面倒なことになります。RADIUSシークレット文字列に空白を入れる例を次に示します。

```
host ... radius radiusservers="server1,server2" radiussecrets="""secret one"","secret two"""
```

20.12. 証明書認証

この認証方法は、認証のためにSSLクライアント証明書を使用します。よってこの方法は、SSL接続を使用します。この認証方法を使用する際は、サーバはクライアントが有効かつ信頼された証明書を提供することを要求します。パスワードのプロンプトはクライアントに送信されません。証明書のcn（Common Name）属性は、要求されたデータベースユーザ名と比較されます。もしそれらが一致した場合はログインが許可されます。ユーザ名マッピングは、cnがデータベースユーザ名と異なるものであることを許可するために使用されます。

次の設定オプションはSSL証明書認証のためにサポートされています。

map

システムとデータベースユーザ名の間のマッピングを許可します。詳細は20.2を参照してください。

cert認証でclientcertオプションを使うことは冗長です。cert認証は実質的にclientcert=verify-fullを持つtrust認証であるためです。

20.13. PAM認証

この認証方式は認証機構としてPAM（Pluggable Authentication Modules）を使用することを除いてpasswordのように動作します。デフォルトのPAMサービス名はpostgresqlです。PAMはユーザ名/パスワードの組の確認と接続されたリモートホスト名またはIPアドレスを任意に確認するためだけに使用されます。PAMについての詳細はLinux-PAMページ[2]を読んでください。

次の設定オプションはPAMのためにサポートされています。

`pamservice`

PAMサービス名。

`pam_use_hostname`

PAM_RHOSTアイテムを通じてPAMモジュールに提供されるものがリモートのIPアドレスかホスト名かを決定します。デフォルトではIPアドレスが使用されます。ホスト名にて使用するためにはこのオプションを1にセットしてください。ホスト名の解決はログインの遅延をもたらします。（ほとんどのPAM設定はこの情報を利用せず、PAM設定がホスト名を使用するために明確に作成された場合のみ、この設定値を考慮する必要があります。）

> **注記**
>
> PAMが/etc/shadowを読み取るように設定されている場合は、PostgreSQLがルートユーザで起動されていないため、認証は失敗するでしょう。しかしPAMがLDAPや他の認証方法を使用するように設定されている場合は、これは問題ではありません。

20.14. BSD認証

この認証方式は、パスワードを照合するためにBSD認証を使用すること以外はpasswordと同じように動作します。BSD認証は、ユーザ名/パスワードの組の確認のみに使用されます。それゆえ、ユーザのロールはBSD認証が認証に使用可能となる前にデータベースに存在していなければいけません。BSD認証フレームワークは現在OpenBSDでのみ利用可能です。

PostgreSQLでのBSD認証は、auth-postgresqlログイン型を使用し、postgresqlログインクラスがlogin.confにて定義されている場合はそれを使った認証を使用します。デフォルトでは、そのログインクラスは存在せず、PostgreSQLはデフォルトログインクラスを使用します。

> **注記**
>
> BSD認証を使用するために、PostgreSQLユーザアカウント（サーバを起動しているオペレーティングシステムユーザ）が、まずはauthグループに追加されていなければいけません。authグループはOpenBSDシステムではデフォルトで存在しています。

[2] https://www.kernel.org/pub/linux/libs/pam/

20.15. 認証における問題点

本来の認証失敗とそれに関連した問題は、一般的に以下のようなエラーメッセージを通して明示されます。

```
FATAL:  no pg_hba.conf entry for host "123.123.123.123", user "andym", database "testdb"
```

たいがい、サーバとの接触に成功はしたものの、サーバが通信を拒否した場合です。メッセージが指摘するようにサーバは接続要求を拒否しました。なぜならpg_hba.conf設定ファイルに一致する項目を見つけることができなかったからです。

```
FATAL:  password authentication failed for user "andym"
```

この種のメッセージは、サーバと接触し、サーバも通信することを許可したが、pg_hba.confファイルの中で指定された認証方式に合格していないことを表します。入力したパスワードを確認するか、もしエラーがKerberos、ident認証型のいずれかを指摘している場合はKerberosあるいはidentソフトウェアを確認してください。

```
FATAL:  user "andym" does not exist
```

与えられたデータベースユーザ名は見つかりませんでした。

```
FATAL:  database "testdb" does not exist
```

接続しようとしているデータベースは存在しません。データベース名を指定しなければ、それが望むと望まざるとにかかわらず、データベースユーザ名がデフォルトとなることに注意してください。

ヒント

クライアントに報告される以上により多くの情報がサーバログに残ります。失敗した原因についてよくわからなければサーバのログを見てください。

第21章 データベースロール

PostgreSQLは、ロールという概念を使用してデータベースへの接続承認を管理します。ロールは、その設定方法に応じて、データベースユーザ、またはデータベースユーザのグループとみなすことができます。ロールはデータベースオブジェクト（例えばテーブルや関数）を所有することができます。またロールは、どのオブジェクトに誰がアクセスできるかを制御するために、それらのオブジェクトに対しての権限を他のロールに割り当てることができます。更に、ロールのメンバ資格を他のロールに与えることもできます。そのため、メンバとなったロールは別のロールに割り当てられた権限を使用することができます。

ロールの概念には、「ユーザ」という概念と「グループ」という概念が含まれます。PostgreSQLバージョン8.1より前まででは、ユーザとグループは異なる種類の実体として扱われていました。しかし、現在ではロールしか存在しません。すべてのロールは、ユーザとして、グループとして、またはその両方として動作することができます。

本章では、ロールの作成と管理の方法について説明します。様々なデータベースオブジェクト上の権限の効果について、さらに詳細な情報は5.7に記載されています。

21.1. データベースロール

データベースロールは概念的に、オペレーティングシステムユーザとは完全に分離されています。実行する上でユーザ名を一致させておくと便利ですが、必須ではありません。データベースロール名はデータベースクラスタインストレーション全体で共通です（個別のデータベースごとではありません）。ユーザを作成するためにはCREATE ROLE SQLコマンドを使います。

```
CREATE ROLE name;
```

nameはSQL識別子の規則に従います。特殊な文字を持たない無装飾のものか、二重引用符に囲まれたもののどちらかです。（現実的には、通常他のオプション、例えばLOGINなどをこのコマンドに付与することになるでしょう。詳細は後で説明します。）既存のユーザを削除するためには類似のコマンドDROP ROLEを使用してください。

```
DROP ROLE name;
```

利便性のために、これらのSQLコマンドのラッパである、シェルのコマンドラインから呼び出し可能なcreateuserプログラムとdropuserプログラムが提供されています。

```
createuser name
dropuser name
```

既存のロール群を求めるためには、以下のようにpg_rolesシステムカタログを確認してください。

```
SELECT rolname FROM pg_roles;
```

また、psqlプログラムの\duメタコマンドも既存のロールを列挙する際に役に立ちます。

データベースシステム自身を起動するために、初期化されたばかりのシステムは常に定義済みのロールを1つ持ちます。このロールは必ず「スーパーユーザ」であり、デフォルトでは（initdb実行時に変更しない限り）そのデータベースクラスタを初期化したオペレーティングシステムユーザと同じ名前となります。習慣的にこのロールはpostgresと名付けられます。ロールを追加する場合はまずこの初期ロールで接続しなければいけません。

すべてのデータベースサーバへの接続は、特定のロールの名前を使用して確立し、そのロールによりその接続で発行されるコマンドの初期のアクセス権限が決まります。特定のデータベース接続に使うロールは、アプリケーション固有の方式で接続要求を開始するクライアントによって指示されます。例えば、psqlプログラムでは、-Uコマンドラインオプションを使って接続するロールを指示します。多くのアプリケーション（createuserおよびpsqlを含む）では、オペレーティングシステムの現在のユーザ名をデフォルトと想定します。したがって、ロールとオペレーティングシステムのユーザの組み合わせ間で名前を一致させておくと便利です。

第20章で説明されているように、あるクライアント接続で与えられたデータベースロールの集合は、クライアント認証設定で決定された内容で接続できます。（したがって、ユーザのログイン名が本名と一致していなくても構わないのと同様に、クライアントはオペレーティングシステムのユーザ名と同じロール名で接続しなくても構いません）。接続したクライアントに付与される権限の内容はロールIDによって決定されるため、マルチユーザ環境を設定する際には権限を注意深く設定することが重要です。

21.2. ロールの属性

データベースロールは、権限を定義し、クライアント認証システムと相互作用する数多くの属性を持つことができます。

ログイン権限

LOGIN属性を持つロールのみがデータベース接続の初期ロール名として使用できます。LOGIN 属性を持つロールは「データベースユーザ」と同じであるとみなすことができます。ログイン権限を持つロールの作成方法は、以下のいずれかです。

```
CREATE ROLE name LOGIN;
CREATE USER name;
```

（CREATE USERはデフォルトで LOGINを持ち、CREATE ROLEは持たないという点を除き、CREATE USERはCREATE ROLEと同じです。）

スーパーユーザ状態

ログイン権限を除き、データベーススーパーユーザに対する権限検査は全て行われません。これは危険な権限ですので、安易に使用してはいけません。作業のほとんどを非スーパーユーザのロールで行うことが最善です。新しいデータベーススーパーユーザを作成するには、CREATE ROLE name SUPERUSERを使用してください。これはスーパーユーザのロールで実行しなければなりません。

データベース作成

（全ての権限検査が行われないスーパーユーザを除き）ロールに明示的にデータベースを作成するための権限を指定しておかねばいけません。そのようなロールを作るためにはCREATE ROLE name CREATEDBを使用してください。

ロールの作成

あるロールがロールを作成するには、明示的な権限が付与されていなければなりません。(スーパーユーザは、すべての権限検査を迂回しますので、例外です。) こうしたロールを作成するには、CREATE ROLE name CREATEROLEを使用してください。CREATEROLE権限を持つロールは他のロールを変更したり削除したりすることもできます。また、他のロールのメンバ資格を付与したり取り上げたりすることもできます。しかし、スーパーユーザロールの変更、削除、メンバ資格の変更を行うにはスーパーユーザ状態が必要です。CREATEROLEだけでは不足しています。

レプリケーションの新規接続

あるロールがストリーミングレプリケーションの新規接続を実施するには、明示的な権限が付与されていなければなりません。(スーパーユーザは、すべての権限検査を迂回しますので、例外です。) ストリーミングレプリケーションを行うロールは、LOGIN権限も持っていることが必要です。こうしたロールを作成するには、CREATE ROLE name REPLICATION LOGINを使用してください。

パスワード

パスワードは、クライアント認証方法においてデータベースに接続する際にユーザにパスワードを要求する場合にのみ重要になります。passwordとmd5認証方式でパスワードが使用されます。データベースパスワードはオペレーティングシステムのパスワードとは異なります。ロール作成時にCREATE ROLE name PASSWORD 'string'のようにパスワードを指定します。

ロール属性は、ロールを作成した後でもALTER ROLEコマンドで変更できます。詳細はCREATE ROLEとALTER ROLEのマニュアルページを参照してください。

ヒント

スーパーユーザ以外にCREATEDB権限とCREATEROLE権限を持つロールを作成することを勧めます。そして、このロールを使用して、データベースとロールを管理するためのすべての処理を行ってください。この方法によって、実際には不要な処理をスーパーユーザとして行う危険性を避けることができます。

ロールは、第19章で説明されている実行時の設定の多くをロールごとのデフォルトに設定することもできます。例えば何らかの理由で、自分が接続する時は常にインデックススキャンを無効にしたい場合(注:お勧めしません)、次のようにします。

```
ALTER ROLE myname SET enable_indexscan TO off;
```

このようにして設定を保存します(ただし、すぐに反映はされません)。以降のこのロールによる接続においては、セッションの開始の直前にSET enable_indexscan TO off;が呼び出されたのと同様になります。これはデフォルトとする設定をするだけなので、設定をセッション途中いつでも変更できます。ロール固有のデフォルト設定を削除するには、ALTER ROLE rolename RESET varnameを使用してください。呼び出されることがありませんので、LOGIN権限を持たないロールにロール固有のデフォルトを持たせることに意味がないことに注意してください。

21.3. ロールのメンバ資格

権限の管理を簡単にするために、ユーザをグループにまとめることはしばしば便利です。グループ全体に対して権限を与えることも、取り消すこともできます。PostgreSQLでは、グループを表すロールを作成することで行われます。そして、そのグループロールに個々のユーザロールの*メンバ資格*を与えます。

グループロールを設定するには、まずロールを作成します。

```
CREATE ROLE name;
```

通常、グループとして使用されるロールにはLOGIN属性を持たせません。しかし、そうしたければ持たせることもできます。

グループロールをいったん作成すれば、GRANTおよびREVOKEコマンドを使用してメンバの追加と削除を行うことができます。

```
GRANT group_role TO role1, ... ;
REVOKE group_role FROM role1, ... ;
```

他のグループロールへのメンバ資格を与えることもできます。(グループロールと非グループロールとの間には実際には区別がないからです。) データベースはグループのメンバ資格がループし、循環するような設定はさせません。また、ロール内のメンバ資格をPUBLICに付与することはできません。

グループロールのメンバは、2つの方法でロールの権限を使用することができます。1つ目は、グループ内のすべてのメンバは明示的に、一時的にそのグループロールに「なる」ためにSET ROLEを行うことができます。この状態では、データベースセッションは元々のログインロールの権限ではなくグループの権限でアクセスされます。そして、作成されたデータベースオブジェクトの所有者はログインロールではなくグループロールであるとみなされます。2つ目は、INHERIT属性を持つメンバロールは、それらがメンバとなるロールの権限を自動的に使用します。これには、ロールによって継承されるいかなる権限も含んでいます。例えば、以下の状態を想定します。

```
CREATE ROLE joe LOGIN INHERIT;
CREATE ROLE admin NOINHERIT;
CREATE ROLE wheel NOINHERIT;
GRANT admin TO joe;
GRANT wheel TO admin;
```

joeロールで接続するとすぐに、joeはadmin権限を「継承」しますので、そのデータベースセッションではjoeに直接与えられた権限に加えて、adminに与えられた権限を使用することができます。しかし、wheelに与えられた権限は利用できません。joeは間接的にwheelのメンバですが、admin経由のメンバ資格はNOINHERIT属性を持っているためです。

```
SET ROLE admin;
```

を行った後、セッションはadminに与えられた権限のみを使用できるようになります。joeに与えられた権限は使用できなくなります。

```
SET ROLE wheel;
```

を行った後、セッションはwheelに与えられた権限のみを使用できるようになり、joeやadminに与えられた権限は使用できなくなります。元の状態の権限に戻すには、以下のいずれかを行います。

```
SET ROLE joe;
SET ROLE NONE;
RESET ROLE;
```

注記

SET ROLEコマンドによりいつでも、元のログインロールが直接あるいは間接的にメンバ資格を持つすべてのロールを選ぶことができます。従って、上の例において、wheelになる前にadminになることは必要ありません。

注記

標準SQLでは、ユーザとロールとの間に明確な違いがあり、ユーザはロールのように自動的に権限を継承することができません。PostgreSQLでこの振舞いを実現させるには、SQLロールとして使用するロールにはINHERIT属性を付与し、SQLユーザとして使用するロールにはNOINHERIT属性を付与します。しかし、8.1リリースより前との互換性を保持するために、PostgreSQLはデフォルトで、すべてのロールにINHERIT属性を付与します。以前は、ユーザは常にメンバとして属するグループに付与された権限を常に使用できました。

LOGIN、SUPERUSER、CREATEDB、およびCREATEROLEロール属性は、特別な権限とみなすことができますが、データベースオブジェクトに対する通常の権限のように継承されません。こうした属性の1つを使用できるようにするためには、その属性を特定のロールに設定するように実際にSET ROLEを行う必要があります。上の例を続けると、adminロールにCREATEDB権限とCREATEROLE権限を付与することを選ぶことができます。こうすると、joeロールとして接続するセッションでは、すぐさまこれらの権限を持ちません。SET ROLE adminを行った後で、この権限を持ちます。

グループロールを削除するには、DROP ROLEを使用してください。

```
DROP ROLE name;
```

グループロール内のメンバ資格も自動的に取り上げられます。(しかし、メンバロールには何も影響ありません。)

21.4. ロールの削除

ロールはデータベースオブジェクトを所有したり、他のオブジェクトにアクセスする権限を保持したりできるので、ロールを削除するのは、単にDROP ROLEを実行すれば良いというだけのものでないことがよくあります。そのロールが所有するすべてのオブジェクトについて、まずそれを削除するか、あるいは他の所有者に移すかしなければなりません。また、そのロールに付与されたすべての権限を取り上げる必要があります。

オブジェクトの所有権はALTERコマンドを使って、1度に1つずつ移管することができます。以下に例を示します。

```
ALTER TABLE bobs_table OWNER TO alice;
```

その代わりにREASSIGN OWNEDコマンドを使って、削除予定のロールが所有するすべてのオブジェクトの所有権を、単一の他のロールに移管することもできます。REASSIGN OWNEDは他のデータベースのオブジェクトにはアクセスできないので、そのロールが所有するオブジェクトを含むそれぞれのデータベース内で実行する必要があります。(最初にそのようなREASSIGN OWNEDを実行した時に、データベース間で共有されるオブジェクト、つまりデータベースとテーブル空間については、すべて削除予定のロールから所有権が変更されることに注意してください。)

重要なオブジェクトがすべて新しい所有者に移管された後は、削除予定のロールが所有する残りのオブジェクトはすべてDROP OWNEDコマンドで削除することができます。ここでも、このコマンドは他のデータベースのオブジェクトにはアクセスできないので、そのロールが所有するオブジェクトを含むそれぞれのデータベース内で実行する必要があります。また、DROP OWNEDはデータベース全体、あるいはテーブル空間全体を削除することはありませんので、ロールが所有するデータベースあるいはテーブル空間で新しい所有者に移管されていないものがあれば、手作業でそれを削除する必要があります。

DROP OWNEDは対象のロールが所有しないオブジェクトについて、そのロールに付与されたすべての権限を削除することも行います。REASSIGN OWNEDはそのようなオブジェクトに関与しないので、削除されるロールによる依存関係を完全に取り除くには、多くの場合、REASSIGN OWNEDとDROP OWNEDの両方を(この順序で!)実行する必要があります。

まとめると、オブジェクトを所有するために使用されたロールを削除する最も一般的な手順は以下のようになります。

```
REASSIGN OWNED BY doomed_role TO successor_role;
DROP OWNED BY doomed_role;

-- 上記のコマンドをクラスタ内の各データベースについて繰り返す
DROP ROLE doomed_role;
```

すべての所有オブジェクトを同一の後継所有者に移管するのでない場合は、例外部分を手作業で処理した後で、上記の手順を実行して残りを処理するのが最善でしょう。

依存するオブジェクトがまだ残っている状態でDROP ROLEを実行すると、どのオブジェクトが所有者変更または削除の必要があるかを特定するメッセージが発行されます。

21.5. デフォルトロール

PostgreSQLでは、ある種の共通に必要で、特権のある機能や情報にアクセスできるよう、いくつかのデフォルトロールを提供しています。管理者は自分の環境のユーザあるいはロールに対し、これらのロールを付与(GRANT)することで、それらのユーザに、その機能や情報を提供することができます。

デフォルトロールについては表 21.1で説明します。それぞれのデフォルトロールの個別の権限については、将来、さらなる機能が追加されるに連れて変更されるかもしれません。管理者は、変更がないかリリースノートを確認するようにしてください。

表21.1 デフォルトロール

ロール	可能なアクセス
pg_read_all_settings	通常スーパーユーザのみが読み取れる、全ての設定変数を読み取る

ロール	可能なアクセス
pg_read_all_stats	通常スーパーユーザのみが読み取れる、すべてのpg_stat_*ビューを読み取り、各種の統計関連のエクステンションを使用する
pg_stat_scan_tables	潜在的に長時間、テーブルのACCESS SHAREロックを取得する可能性がある監視機能を実行する
pg_monitor	各種の監視ビューや機能を読み取り/実行する。このロールは、pg_read_all_settings、pg_read_all_statsおよびpg_stat_scan_tablesのメンバです。
pg_signal_backend	他のバックエンドに問い合わせのキャンセルやセッションの終了のシグナルを送信する
pg_read_server_files	COPYやその他のファイルアクセス関数で、データベースがサーバ上でアクセスできる任意の場所からファイルを読み取ることを許可する
pg_write_server_files	COPYやその他のファイルアクセス関数で、データベースがサーバ上でアクセスできる任意の場所にファイルを書き込むことを許可する
pg_execute_server_program	COPYやサーバ側のプログラムを実行できるその他の関数で、データベースを実行しているユーザとしてデータベースサーバ上でのプログラムの実行を許可する

pg_monitor、pg_read_all_settings、pg_read_all_statsおよびpg_stat_scan_tablesロールは、データベースサーバを監視するためのロールを、管理者が簡単に設定できるようにする目的があります。これらのロールは通常スーパーユーザに限定されている各種の有用な構成設定、統計情報およびその他のシステム情報を読むことができる一般的な権限のセットを与えることができます。

pg_signal_backendロールは、信頼はできるが非スーパーユーザであるロールが他のバックエンドにシグナルを送信することを、管理者が許可できるようにすることを意図しています。今のところ、このロールは他のバックエンドでの問い合わせをキャンセルしたり、セッションを終了するシグナルを送信できます。しかしながら、このロールの権限を与えられたユーザは、スーパーユーザが所有するバックエンドにシグナルを送信できません。9.27.2を参照してください。

pg_read_server_files、pg_write_server_files、pg_execute_server_programロールは、信頼はできるが非スーパーユーザであるロールがデータベースを実行しているユーザとしてデータベースサーバ上でファイルにアクセスしたりプログラムを実行したりすることを、管理者が許可できるようにすること意図しています。これらのロールはサーバファイルシステムの任意のファイルにアクセスできますので、ファイルに直接アクセスする時にはデータベースレベルの権限検査はすべて行なわれず、スーパーユーザレベルのアクセスを得るのに使えます。ですので、これらのロールをユーザに許可する時には注意すべきです。

これらのロールを許可する場合は、必要な場合にのみ、これらのロールは機密情報へのアクセス権を与えることを理解して、使用されるように注意する必要があります。

管理者はGRANTコマンドを使って、ユーザにこれらのロールへのアクセスを許可できます。例えば、

```
GRANT pg_signal_backend TO admin_user;
```

21.6. 関数のセキュリティ

関数やトリガや行セキュリティポリシーによって、ユーザは他のユーザが意識しないで実行できるようなコードを、バックエンドサーバに挿入することができます。したがって、これらの機能によってユーザは比較的簡

単に他のユーザにとって「トロイの木馬」となるものを実行することができます。最も強力な保護は、誰がオブジェクトを定義できるかを厳格に管理することです。それが実行不可能な場合は、信頼できる所有者を持つオブジェクトのみを参照するクエリを記述します。search_pathから、パブリックスキーマや信頼できないユーザがオブジェクトを作成できるスキーマを削除します。

関数は、データベースサーバデーモンのオペレーティングシステム権限で、バックエンドサーバプロセスの中で実行されます。プログラミング言語で関数に未検査のメモリアクセスを許可している場合、サーバの内部データ構造を変更することが可能です。したがって、その他の多数のことの中でも、そのような関数はどのようなシステムアクセスコントロールも回避することができます。このようなアクセスを許可する関数言語は「信頼されない」ものとみなされ、PostgreSQLはこれらの言語による関数の作成をスーパーユーザのみに限定して許可しています。

第22章 データベース管理

稼動しているPostgreSQLサーバのすべてのインスタンスは、1つ以上のデータベースを管理します。したがって、データベースはSQLオブジェクト(「データベースオブジェクト」)を組織化する場合に最上位の階層レベルとなります。本章では、データベースの特性、作成方法、管理方法、および削除方法について説明します。

22.1. 概要

ロール、データベース、テーブル空間名のような少数のオブジェクトはクラスタレベルで定義されており、pg_globalテーブル空間に格納されています。クラスタの中には複数のデータベースがあり、互いに分離されているもののクラスタレベルのオブジェクトにはアクセスできます。各データベースの中には複数のスキーマがあり、スキーマはテーブルや関数などのオブジェクトを含みます。したがって階層の全体像は、クラスタ、データベース、スキーマ、テーブル(や関数などの何らかのオブジェクト)となります。

データベースサーバに接続する時、クライアントはその接続要求の中で接続するデータベース名を指定しなければなりません。1つの接続で複数のデータベースにアクセスすることはできません。しかし、クライアントは同じデータベースに対して複数の接続を開いたり、異なるデータベースに対して複数の接続を開いたりできます。データベースレベルでのセキュリティには2つの構成要素があります。接続レベルで管理されるアクセス制御(20.1参照)と、権限付与システムで管理される認証制御(5.7参照)です。外部データラッパ(postgres_fdw参照)により、1つのデータベース内のオブジェクトが他のデータベースやクラスタ内にあるオブジェクトに対するプロキシとして動作できるようなります。古いdblinkモジュール(dblink参照)は同様の機能を提供します。デフォルトでは、すべてユーザはすべてのデータベースにすべての接続方法で接続できます。

1つのPostgreSQLサーバクラスタに、たいていの場合お互いのことを意識しない、関係のないプロジェクトやユーザを含めるつもりなら、これらを別々のデータベースに含め、それに従って認証制御とアクセス制御を調整することが推奨されます。複数のデータベースは物理的に分離されていて、アクセス制御は接続レベルで管理されています。したがって、分離して、ほとんどの場面で互いに見えないようにする必要のある複数のプロジェクトやユーザを単一のPostgreSQLサーバインスタンスに収容する場合、これらを別々のデータベースに含めることが推奨されます。もし、複数のプロジェクトやユーザが相互に関連していて互いのリソースを使用できる必要がある場合、これらは同じデータベースに含めるべきですが、スキーマを別にすることは可能です。これは名前空間での分離と認証制御によるモジュラー構造を提供します。スキーマの管理についての詳細は5.9に記載されています。

1つのクラスタ内に複数のデータベースを作成できますが、その利点がその危険性と制限に勝るかどうか慎重に検討することを勧めます。特に、共有のWAL(第29章参照)を持つことの影響がバックアップとリカバリのオプションにあります。クラスタ内の個々のデータベースは、ユーザの視点から考えれば分離していても、データベース管理者の観点からは密接に結びついています。

データベースはCREATE DATABASEコマンド(22.2を参照)で作成され、DROP DATABASEコマンド(22.5を参照)で破棄されます。既存のデータベース群を求めるには、以下の例のようにpg_databaseシステムカタログを確認してください。

```
SELECT datname FROM pg_database;
```

また、psqlプログラムの\lメタコマンドや-lコマンドラインオプションも既存のデータベースを列挙する際に役に立ちます。

> **注記**
>
> 標準SQLでは、データベースを「カタログ」(catalog)と呼ぶこともありますが、実際のところ違いはありません。

22.2. データベースの作成

データベースを作成する場合、PostgreSQLサーバが起動している必要があります(18.3を参照してください)。

CREATE DATABASESQLコマンドでデータベースを作成することができます。

```
CREATE DATABASE name;
```

ここで、nameはSQL識別子の通常の規則に従います。現在のロールが自動的に新しいデータベースの所有者になります。作成後、データベースを削除する権限はこの所有者にあります(この作業では、そのデータベースに属している、所有者のものではないオブジェクトでも、すべて削除されます)。

データベースの作成は制限された作業です。権限の付与に関する詳細は21.2を参照してください。

CREATE DATABASEコマンドを実行するためには、データベースサーバに接続している必要があります。そうすると、あるサイトの最初のデータベースはどのようにして作成するのかという疑問が出てきます。最初のデータベースはinitdbコマンドでデータ格納領域が初期化される時、必ず作成されます。(18.2を参照してください。) このデータベースはpostgresと呼ばれます。したがって、最初の「通常の」データベースを作成するにはpostgresに接続してください。

また、template1という2つ目のデータベースもデータベースクラスタの初期化時に作成されます。クラスタ内に新しいデータベースが作成されたら、実際にtemplate1が複製されます。つまりtemplate1に変更を与えると、その後に作成されるデータベースすべてにその変更が反映されることを意味します。このため新しく作成するデータベースすべてに反映させたい場合でない限りtemplate1内にオブジェクトを作成することは避けてください。詳細については22.3を参照してください。

利便性のために、シェルからcreatedbを実行して、新しいデータベースを作成することができます。

```
createdb dbname
```

createdbは魔法ではありません。これはpostgresデータベースに接続し、先に解説した通りにCREATE DATABASEコマンドを実行します。createdbのマニュアルページに実行方法の詳細が説明されています。引数のないcreatedbは現在のユーザ名のデータベースを作成しますので、注意してください。

> **注記**
>
> 特定のデータベースに誰が接続できるかを制限する方法については第20章に記載されています。

他のユーザのためにデータベースを作成し、そのユーザ自身が設定、管理できるように新しいデータベースの所有者にさせたい場合も考えられます。そのためには、次のコマンドのいずれかを使用します。SQL環境からは

```
CREATE DATABASE dbname OWNER rolename;
```

シェルからは

```
createdb -O rolename dbname
```

他のユーザのために（つまり、自身がメンバではないロールのために）データベースを作成することができるのはスーパーユーザだけです。

22.3. テンプレートデータベース

実際のCREATE DATABASEの動作は、既存のデータベースをコピーすることです。デフォルトでは、template1という名前の標準のシステムデータベースをコピーします。したがって、このデータベースは新しく作成するデータベースの元になる「テンプレート」となります。template1にオブジェクトを追加した場合、追加したオブジェクトはその後に作成されるユーザデータベースにコピーされます。この振舞いによって、データベース標準オブジェクト群にサイト独自の変更を行うことができます。例えば、PL/Perl手続き言語をtemplate1にインストールした場合、データベースを作成する時に追加作業を行うことなく、自動的にこの言語をユーザデータベースで使用することができます。

template0という名前の二次的な標準システムデータベースがあります。このデータベースにはtemplate1の初期内容と同じデータが含まれています。つまり、使用しているバージョンのPostgreSQLで定義済みの標準オブジェクトのみから構成されています。template0をデータベースクラスタを初期化した後に変更してはいけません。CREATE DATABASEをtemplate1ではなくtemplate0をコピーするように実行することで、template1に追加されたサイト独自のものを含まない、(そこではユーザ定義オブジェクトは存在せず、システムオブジェクトは変更されていない)「汚れがない」ユーザデータベースを作成することができます。これは特に、pg_dumpダンプからリストアする時に便利です。このダンプスクリプトは、後でtemplate1に追加される可能性のあるオブジェクトと衝突しないように、ダンプしたデータベースの内容を正しく再作成するために、汚れのないデータベースにリストアされなければなりません。

template1の代わりにtemplate0をコピーするその他の一般的な理由は、template0をコピーするときに新規の符号化方式とロケールを設定できることです。一方、template1のコピーはそれが行ったと同一の設定を使用しなければなりません。これはtemplate0が理解しない符号化方式特有の、またはロケール特有のデータを含んでいる可能性があることに依ります。

template0をコピーしてデータベースを作成するには、

```
CREATE DATABASE dbname TEMPLATE template0;
```

をSQL環境から実行するか、または

```
createdb -T template0 dbname
```

をシェルから実行します。

さらにテンプレートデータベースを作成することができます。また、実際のところCREATE DATABASEのテンプレートとして名前を指定することで、クラスタ内の任意のデータベースをコピーすることができます。しかし、この機能は、(まだ)汎用目的の「COPY DATABASE」能力を意図したものではないことは理解しておいてください。コピー操作の間、他のセッションから元のデータベースに接続することができないという点が大きな制限です。CREATE DATABASEは、その起動時に他の接続が存在する場合失敗します。コピー操作中は元のデータベースへの新しい接続を許しません。

datistemplate列とdatallowconn列という、データベースそれぞれに有用なフラグがpg_databaseに存在します。datistemplateは、そのデータベースがCREATE DATABASEのテンプレートとして使用されることを目的としているものであることを意味するために設定することができます。このフラグが設定された場合、CREATEDB権限を持つすべてのユーザはそのデータベースを複製することができます。設定されていない場合は、スーパーユーザとそのデータベース所有者のみがそれを複製することができます。datallowconnが偽の場合、そのデータベースへの新規接続はできません(しかし、このフラグを偽にするだけでは既存のセッションは閉ざされません)。template0データベースは、変更を防ぐために、通常datallowconn = falseとされています。template0とtemplate1の両方は、常にdatistemplate = trueとされていなければなりません。

> **注記**
>
> template1とtemplate0には、template1という名前がCREATE DATABASEのデフォルトのソースデータベースの名前であること以上の特別な地位はありません。例えば、template1を削除し、template0から再作成しても何も問題ありません。この操作は、不注意にごみをtemplate1に追加してしまった場合にお勧めします。(template1を削除するには、pg_database.datistemplate = falseとしなければなりません。)
>
> データベースクラスタが初期化される時、postgresデータベースも作成されます。このデータベースは、ユーザとアプリケーションのデフォルトの接続先を意図したものです。これはtemplate1の単純なコピーで、必要に応じて削除したり再作成したりすることができます。

22.4. データベースの設定

第19章で説明したように、PostgreSQLサーバには多数の実行時の設定変数が存在します。これらの設定の多くに対して、データベース固有のデフォルト値を設定できます。

例えば、何らかの理由で特定のデータベースについてGEQOオプティマイザを無効にしたい場合、通常はすべてのデータベースでこれを無効にするか、またはすべての接続しているクライアントで間違いなくSET geqo TO off;が実行されていることを確認する必要があります。特定のデータベースでこの設定をデフォルトにするには、次のコマンドを実行します。

```
ALTER DATABASE mydb SET geqo TO off;
```

これにより設定が保存されます(ただし、すぐに反映はされません)。以降のこのデータベースに対する接続においては、セッションの開始の直前にSET geqo TO off;が呼び出されたのと同様になります。これはデフォルトでしかありませんので、ユーザはセッションの途中であってもこの設定を変更することができます。このような設定を解除するには、ALTER DATABASE dbname RESET varnameを使用します。

22.5. データベースの削除

データベースの削除には、以下のDROP DATABASEコマンドを使用します。

```
DROP DATABASE name;
```

データベースの所有者とスーパーユーザのみがデータベースを削除することができます。データベースの削除はそのデータベースに含まれるすべてのオブジェクトを削除します。削除されたデータベースは復元できません。

削除しようとしているデータベースに接続している時にはDROP DATABASEを実行することはできません。しかし、template1などのその他のデータベースに接続すれば削除することができます。また、そのクラスタの最後のユーザデータベースを削除する時には、template1データベースに接続するしかありません。

利便性のため、データベースを削除するdropdbシェルプログラムもあります。

```
dropdb dbname
```

（createdbと異なり、デフォルトで現在のユーザ名のデータベースを削除するようにはなっていません。）

22.6. テーブル空間

PostgreSQLのテーブル空間により、データベース管理者はデータベースオブジェクトを表すファイルを格納できるファイルシステム上の場所を定義することができます。テーブル空間を一度作成すると、データベースオブジェクトを作成する時に名前により参照することができます。

テーブル空間を使用することで、管理者はPostgreSQLインストレーションのディスクレイアウトを制御することができます。これは、少なくとも2つの点で有用です。1つは、クラスタを初期化したパーティションもしくはボリュームの容量が不足し、拡張を行うことができない場合、システムを再構成するまで、別のパーティションにテーブル空間を作成して、このテーブル空間を使用することができます。

もう1つは、テーブル空間により、管理者はデータベースオブジェクトの使用パターンに基づいてデータ格納場所を調整することができることです。例えば、非常によく使用されるインデックスを、例えば、高価なソリッドステートディスクなどの高速・高可用性ディスクに格納することができます。同時に、めったに使用されない保存用のデータや性能が求められていない保存用のデータを格納するテーブルを低価格・低速ディスクシステムに格納することもできます。

警告

主PostgreSQLデータディレクトリの外に位置していたとしても、テーブル空間はデータベースクラスタの不可欠な部分であり、データファイルの自律的な集合として扱うことはできません。それらは主データディレクトリに含まれるメタデータに依存しますので、異なるデータベースクラスタに所属させたり、個別にバックアップしたりすることはできません。同様に、テーブル空間を失えば(ファイル削除、ディスクの破損など)、データベースクラスタは読み取れなくなったり、開始できなくなったりするかもしれません。テーブル空間をRAMディスクのような一時ファイルシステムに置くことは、クラスタ全体の信頼性を危険にさらします。

テーブル空間を定義するにはCREATE TABLESPACEコマンドを使用してください。以下に例を示します。

```
CREATE TABLESPACE fastspace LOCATION '/ssd1/postgresql/data';
```

この場所は、既存かつ空のディレクトリでなければならず、また、PostgreSQLオペレーティングシステムユーザが所有していなければなりません。その後に、テーブル空間内に作成されるオブジェクトはすべてこのディレクトリ以下のファイルに格納されます。テーブル空間が見つからなかったり失われたりするとクラスタは機能しませんので、この場所は取り外し可能な記憶装置や一時的な記憶装置上にあってはいけません。

> **注記**
>
> 通常、論理ファイルシステム内の個々のファイルの場所を制御することができませんので、1つの論理ファイルシステムに複数のテーブル空間を作成することは意味がありません。しかし、PostgreSQLにはこうした制限はありません。実際、システムのファイルシステムの境を直接意識しません。単に使用することを宣言したディレクトリにファイルを格納します。

テーブル空間自体の生成はデータベーススーパーユーザが行う必要があります。しかし、その後に、データベース一般ユーザにそのテーブル空間を使用させることができます。これを行うには、ユーザにテーブル空間に対するCREATE権限を与えてください。

テーブル、インデックス、およびデータベース全体は特定のテーブル空間に割り当て可能です。これを行うには、指定テーブル空間にCREATE権限を持つユーザが関連するコマンドにテーブル空間をパラメータとして渡す必要があります。例えば、以下はspace1テーブル空間にあるテーブルを作成しています。

```
CREATE TABLE foo(i int) TABLESPACE space1;
```

他の方法として、以下のようにdefault_tablespaceパラメータの使用があります。

```
SET default_tablespace = space1;
CREATE TABLE foo(i int);
```

default_tablespaceが空文字以外の何かに設定された場合、この値が、明示的なテーブル空間の指定がないCREATE TABLEコマンドやCREATE INDEXコマンドの暗黙的な TABLESPACE句として適用されます。

temp_tablespacesというパラメータも存在します。これは、一時テーブルとそのインデックス、大規模データ集合のソートなどを目的に使用される一時ファイルの場所を決定するものです。これは、テーブル空間名を1つだけ指定するものではなく、テーブル空間名のリストを取ることができます。このため、一時的なオブジェクトに関連する負荷を、複数のテーブル空間にわたって分散することができます。一時的なオブジェクトを作成する度に、このリストから無作為に要素が選択されます。

データベースに関連付けされたテーブル空間は、そのデータベースのシステムカタログを格納するために使用されます。さらに、TABLESPACE句を付けずに、または、(適切な)default_tablespaceやtemp_tablespacesで指定された選択肢がなければ、データベース内に作成されたテーブルやインデックスのデフォルトのテーブル空間としても使用されます。テーブル空間の指定なしで作成されたデータベースは、コピー元のテンプレートデータベースのテーブル空間と同じものを使用します。

データベースクラスタが初期化される時に2つのテーブル空間が自動的に作成されます。pg_globalテーブル空間は共有システムカタログ用に使用されます。pg_defaultテーブル空間はtemplate1とtemplate0データベースのデフォルトテーブル空間です（したがって、CREATE DATABASEのTABLESPACE句で変更されない限り、このテーブル空間が同様に他のデータベースに対するデフォルトのテーブル空間になります）。

テーブル空間は、一度作成すると、要求しているユーザが十分な権限を持っていればすべてのデータベースから使用することができます。これは、テーブル空間を使用するすべてのデータベースのすべてのオブジェクトが削除されるまで、そのテーブル空間を削除できないことを意味します。

空のテーブル空間を削除するには、DROP TABLESPACEコマンドを使用してください。

既存のテーブル空間群を求めるには、以下の例のようにpg_tablespaceを確認してください。

```
SELECT spcname FROM pg_tablespace;
```

また、psqlプログラムの\dbメタコマンドも既存のテーブル空間を列挙する際に役に立ちます。

テーブル空間の実装を単純化するために、PostgreSQLはシンボリックリンクを使用します。これは、テーブル空間はシンボリックリンクをサポートするシステムでのみ使用することができるということを意味します。

$PGDATA/pg_tblspcディレクトリには、そのクラスタで定義された非組み込みテーブル空間1つひとつを指し示すシンボリックリンクがあります。推奨はしませんが、こうしたリンクを手作業で再定義してテーブル空間のレイアウトを調整することができます。2点警告します。これをサーバが実行している時に行わないでください。PostgreSQL 9.1およびそれ以前の場合、pg_tablespaceカタログを新規の場所に更新してください。（これを行わないと、pg_dumpは古いテーブル空間の場所に出力し続けます。）

第23章 多言語対応

本章では、管理者の立場から見た、利用可能な多言語対応機能について説明します。PostgreSQLでは、2つの手法で多言語対応をサポートします。

- ロケール固有の照合順序、数字の書式、翻訳されたメッセージなどを提供するためオペレーティングシステムのロケールの機能を使います。これは23.1と23.2内で解説されています。
- 全ての種類の言語によるテキストの格納のサポート、およびクライアントサーバ間の文字セット翻訳の提供を行うため、多くの文字セットを提供します。これは23.3内で解説されています。

23.1. ロケールのサポート

ロケールのサポートはアルファベット、並び換え、数字の書式など文化的嗜好を配慮したアプリケーションを対象にします。PostgreSQLは、サーバのオペレーティングシステムが提供する、標準ISO CとPOSIXのロケール機能を使用します。これ以上の情報についてはお使いのシステムのドキュメントを参照ください。

23.1.1. 概要

ロケールのサポートは、initdbを使用してデータベースクラスタを作成する時自動的に初期化されます。initdbは、デフォルトでその実行環境のロケール設定に従ってデータベースクラスタを初期化します。そのため、システムがデータベースクラスタで使用したいロケールを使用するように既に設定してある場合は何も行う必要はありません。違うロケールを使用したい場合（またはシステムのロケール設定が不明な場合）は、initdbの--localeオプションで希望のロケールを指定することができます。以下に例を示します。

```
initdb --locale=sv_SE
```

Unixシステム用のこの例の設定はロケールをスウェーデン（SE）で使用されているスウェーデン語（sv）に合わせています。他にもen_US（米国英語）やfr_CA（カナダのフランス語）などの設定もできます。ロケールに複数の文字セットが使用可能であれば、language_territory.codesetのように記述することができます。例えば、fr_BE.UTF-8はベルギー（BE）で使用されているフランス語（fr）でUTF-8の文字セットを表します。

お使いのシステムでどのロケールがどういう名前で使えるかはオペレーティングシステムのベンダがどのようなものを提供しているかと、何がインストールされているかに依存します。ほとんどのUnixシステムでは、locale -aというコマンドで利用可能なロケールの一覧を入手することができます。Windowsは、German_GermanyやSwedish_Sweden.1252のようなもっと冗長なロケール名を使用しますが、原理は同じです。

英語の照合順序規則でスペイン語のメッセージを使用する時など、時として複数のロケールの規則を併用すると便利です。これをサポートするために、ロケールには以下のような多言語対応規則の特定の箇所だけを管理する一連のサブカテゴリがあります。

LC_COLLATE	文字列の並び換え順
LC_CTYPE	文字の分類（文字とはどんなもの?大文字小文字を区別しない?）

LC_MESSAGES	メッセージの言語
LC_MONETARY	通貨書式
LC_NUMERIC	数字の書式
LC_TIME	日付と時刻の書式

これらのカテゴリの名前は、特定のカテゴリについてのロケールの選択を上書きするためのinitdbオプションの名前としてそのまま使用できます。例えば、ロケールをカナダのフランス語に設定しながら通貨書式については米国の規則を使用するには、initdb --locale=fr_CA --lc-monetary=en_USとします。

システムがロケールをサポートしていないように動作させたい場合は、特別なロケールのC、もしくは同等なPOSIXを使用してください。

一部のロケールカテゴリでは、その値がデータベース生成時に固定されていなければならないものがあります。他のデータベースで他の設定を使用することができますが、一度データベースが生成されると、そのデータベースでは変更することができません。LC_COLLATEとLC_CTYPEがこれらのカテゴリにあてはまります。これらはインデックスのソート順に影響を及ぼすため、固定されていなければなりません。さもないと、テキスト型の列上のインデックスは破壊されるでしょう。(しかし23.2内で述べられているように、照合順序を使用することで、この制限を緩和することができます) initdbが実行された時に、これらのカテゴリのデフォルト値は決定され、CREATE DATABASEコマンドで他を指定しない限り、新しいデータベースが作成されるときにこの値が使用されます。

その他のロケールカテゴリは、いつでも、ロケールカテゴリと同じ名前の実行時パラメータを設定することで、希望値に変更することができます（詳細は19.11.2を参照してください）。initdbで選択された値は、実際のところ、サーバの起動時にデフォルトとして動作するようにpostgresql.conf設定ファイルに書き込まれるだけです。この代入文をpostgresql.confから削除すると、サーバは実行環境の設定をそのまま使用します。

サーバのロケールの動作はどのクライアントの環境にも依存せず、サーバが参照できる環境変数で決まります。ですからサーバを稼動させる前に正しいロケール設定を行うように注意してください。結果としてサーバとクライアントで異なるロケールが設定されていると、メッセージはそれらがどこから生じたかによって、異なる言語で表示されます。

> **注記**
>
> 実行環境のロケールをそのまま使用するということは、ほとんどのオペレーティングシステムでは次のような意味を持ちます。指定されたロケールカテゴリ(例えば照合順序)について、設定するものが見つかるまで、以下の環境変数がこの順番で調べられます。LC_ALL、LC_COLLATE(またはそれぞれのカテゴリに対応する変数)、LANG。これらのいずれの環境変数も設定されない場合に、ロケールはデフォルトでCに設定されます。
>
> メッセージの言語を設定する目的で、メッセージ多言語化ライブラリの中には全てのロケール設定を上書きする環境変数LANGUAGEを検索するものがあります。お使いのシステムでの挙動が不明ならばより詳細な情報を得るためお使いのオペレーティングシステムの文書、特にgettextの文書を参照してください。

ユーザの選択した言語にメッセージを翻訳できるようにするためにはNLSを構築時に有効にする(configure --enable-nls)必要があります。他のロケールサポートはすべて自動的に構築されます。

23.1.2. 動作

ロケールの設定は以下のSQL機能に影響を与えます。

- 文字列データに対するORDER BYまたは標準の比較演算子を使用した問い合わせにおける並び替え順
- upper、lower、initcap関数
- (LIKE、SIMILAR TOやPOSIX形式の正規表現といった)パターンマッチング演算子では ロケールは大文字、小文字を区別せず正規表現の文字クラスによる文字の区別に影響を及ぼします。
- 一群のto_char関数
- LIKE節が付いたインデックスを使用する性能

CやPOSIX以外で、PostgreSQLでロケールを使用する際の欠点は実行速度です。ロケールは文字の扱いを遅くし、さらにLIKEで通常のインデックスが使用されなくなります。この理由から、本当に必要な時のみロケールを使用してください。

C以外のロケールにおいて、PostgreSQLがLIKE句を持つインデックスを使用できるようにする回避方法として、いくつかのカスタム演算子クラスがあります。これらを用いると、文字と文字を厳密に比較するようなインデックスや、ロケールの比較規則を無視するようなインデックスを作成できます。詳細は11.10を参照してください。もうひとつの方法は、23.2内で解説されているようなC照合順序を使用してインデックスを作成することです。

23.1.3. 問題点

上記の説明に従ってロケールのサポートが正常に動作しない場合、オペレーティングシステムのロケールサポートが正確に設定されているか確認してください。指定されたロケールがインストールされているかどうか確認するために、オペレーティングシステムが提供していれば、locale -aコマンドを使用することができます。

PostgreSQLが想定しているロケールを実際に使用しているかどうかを確認してください。LC_COLLATEとLC_CTYPEの設定はデータベース作成時に決定され、新しいデータベースを作成する方法以外に変更することはできません。LC_MESSAGESやLC_MONETARYなど他のロケール設定はサーバ起動時の環境変数によって初めに決定されますが、その場で変更することができます。SHOWコマンドを使用して、使用中のロケール設定を確認することができます。

ソース配布物のsrc/test/localeディレクトリには、PostgreSQLのロケールサポート用の試験一式があります。

エラーメッセージ内のテキストを解析してサーバ側のエラーを扱っているクライアントアプリケーションでは、サーバのメッセージが異なる言語で記載されると、明らかに問題になります。こうしたアプリケーションの作者には、エラーコードスキームで代替させることを推奨します。

メッセージ翻訳のカタログのメンテナンスにはPostgreSQLに選択した言語を話させてみたいという数多くのボランティアのたゆみのない努力を必要としています。もしあなたの言語が現在使えなかったり完全に翻訳されてない場合、助力をよろしくお願いします。もし助力頂けるのであれば、第54章を参照するか開発グループのメーリングリストに投稿してください。

23.2. 照合順序サポート

照合順序機能は、ソート順番と列ごともしくは操作ごとのデータの文字区別の振る舞いを指定することを可能にします。これにより、作成後のデータベースのLC_COLLATEとLC_CTYPEの設定が変更できない制限が緩和されます。

23.2.1. 概念

概念的に照合可能なデータ型のそれぞれの式は、照合順序を保持しています (組み込みの照合可能なデータ型はtext、varchar、charです。ユーザ定義の基本型は照合可能とマーキングできます。もちろん照合可能なデータ型上のドメインは照合可能となります)。もし、式が列参照である場合は、式の照合順序は列の定義された照合順序となります。もし、式が定数である場合は、照合順序は定数のデータ型のデフォルトの照合順序となります。より複雑な式の照合順序は、下記に示すように、その入力の照合順序から引き出されます。

式の照合順序は、「default」照合順序となります。これはデータベースに対して定義されたロケール設定を意味しています。式の照合順序は非決定となることもあります。そのような場合に、照合順序が必要となるような順序操作や他の操作は失敗するでしょう。

データベースシステムが並び変えや文字区別を行う場合、データベースは入力の照合順序を使用します。これは、たとえばORDER BY句や<演算子や関数を使用する際に発生します。ORDER BY句に適用する照合順序は、単純にソートキーの照合順序です。関数や演算子の呼び出しに対して適用される照合順序は、以下に述べるように引数により決まります。比較演算子に加えて、照合順序はlower、upper、initcapといった小文字と大文字を変換する関数やパターンマッチングの演算子、to_char関連の関数で考慮されています。

関数や演算子の呼び出しに対して、引数の照合順序検査により得られた照合順序は実行時に特定の操作を行うために使用されます。もし関数や演算子の呼び出しの結果が照合順序可能なデータ型であった場合、照合順序は関数もしくは演算子式の定義済みの照合順序として 解析時にも試用されます。このとき照合順序の知識が必要となるような囲み式があります。

式の照合順序の導出は暗黙でも明示的にでも可能です。この区別は、複数の異なる照合順序が式中に現れるときに照合順序がどのように組み合わされるか、に影響を与えます。明示的な照合順序の導出は、COLLATE句が使用されたときに発生します。他の全ての照合順序は暗黙となります。例えば関数呼び出しの中では、次の規則が用いられます。

1. 入力式に明示的な照合順序の導出がある場合、入力式の中の明示的に導出された全ての照合順序は同一でなくてはなりません。そうでない場合はエラーが発生します。もし明示的に導出された照合順序がある場合は、それは照合順序の組み合わせの結果となります。

2. そうでない場合は、全ての入力式は同一の暗黙の照合順序の導出またはデフォルトの照合順序を持たなくてはなりません。もしデフォルトではない照合順序がある場合は、それは照合順序の組み合わせの結果となります。もしそうでない場合は、結果はデフォルトの照合順序となります。

3. 入力式内でデフォルトではない暗黙の照合順序が衝突している場合、決定不能な照合順序であるとみなされます。これは、もし呼び出された特定の関数が適用するべき照合順序を知っておく必要がないかぎりエラーの条件ではありません。もし知っておく必要がある場合は、実行時にエラーとなります。

例えば、このテーブル定義を考えてみます。

```
CREATE TABLE test1 (
    a text COLLATE "de_DE",
    b text COLLATE "es_ES",
    ...
);
```

このとき

```
SELECT a < 'foo' FROM test1;
```

<の比較はde_DEの規則により実行されます。というのも式は暗黙的に導出されたデフォルトの照合順序と組み合わせます。しかし、

```
SELECT a < ('foo' COLLATE "fr_FR") FROM test1;
```

このとき比較は、明示的な照合順序の導出は暗黙の照合順序をオーバライドするためfr_FR規則が用いられます。さらに、次の例では

```
SELECT a < b FROM test1;
```

パーサはどの照合順序を適用するか決定できません。というのもaとb列は暗黙の衝突する照合順序を持つためです。<演算子がどちらの照合順序を使用するか知る必要があるため、これはエラーとなります。

```
SELECT a < b COLLATE "de_DE" FROM test1;
```

もしくは同じく

```
SELECT a COLLATE "de_DE" < b FROM test1;
```

一方で、以下のように構造的に似たケースとして

```
SELECT a || b FROM test1;
```

これはエラーとなりません。というのも||演算子は、照合順序には関係がないためです。この結果は照合順序とは関係なく同じになります。

もし関数や演算子が照合順序可能なデータ型の結果を出力する場合は、関数に割り当てられた照合順序、 もしくは演算子の組み合わされた入力式は、関数もしくは演算子の結果に対しても 適用されると考えられます。よって、以下の例では

```
SELECT * FROM test1 ORDER BY a || 'foo';
```

順序はde_DE規則に基づき実行されますが、以下のクエリでは

```
SELECT * FROM test1 ORDER BY a || b;
```

エラーとなります。というのも||演算子が照合順序を知る必要がなかったとしてもORDER BY句が照合順序を知る必要があるためです。以前と同様、この衝突は明示的に照合順序を指定することにより解決できます。

```
SELECT * FROM test1 ORDER BY a || b COLLATE "fr_FR";
```

23.2.2. 照合順序の管理

照合順序は、SQL名称を、オペレーティングシステム中にインストールされたライブラリによって提供されるロケールにマッピングするSQLスキーマオブジェクトです。照合順序の定義には、ロケールデータを提供するライブラリを指定するプロバイダ(provider)が含まれます。標準プロバイダの一つはlibcで、オペレーティングシステムのCライブラリが提供するロケールを使用します。オペレーティングシステムが提供するほとんどのツールが、このロケールを使用します。他のプロバイダとしてはicuがあり、外部のICUライブラリを使います。ICUロケールは、PostgreSQLがビルドされた際にICUサポートが設定されていた場合にのみ利用可能です。

libcが提供する照合順序は、setlocale()システムライブラリの呼び出しが許可するLC_COLLATEとLC_CTYPEの組み合わせ設定にマッピングします。(名称から推測されるように、照合順序の主な目的はソート順序を制御するLC_COLLATEを設定することです。しかし実際にはLC_CTYPEの設定をLC_COLLATEと異なるようにする必要はほとんどありません。そのため、式ごとにLC_CTYPEを設定するような別の機構を作成するより、これらの設定を収集する方が、より便利です。) また、libcの照合順序は文字エンコーディングと結びついています(23.3を参照下さい)。同一の照合順序名称が異なるエンコーディングに対して存在しています。

icuが提供する照合順序オブジェクトは、ICUライブラリが提供する照合順序機能(collator)にマップします。ICUは「collate」と「ctype」を別々に設定する機能を提供しないので、それら常に同じものになります。また、ICUの照合順序はエンコーディングからは独立しています。ですから、データベース中のある名前のICU照合順序は、常にただひとつだけです。

23.2.2.1. 標準の照合順序

すべてのプラットフォーム上でdefault、CそしてPOSIXという名称の照合順序が 利用できます。オペレーティングシステムによっては追加の照合順序が利用可能な場合もあります。default照合順序は、データベース作成時にLC_COLLATE値とLC_CTYPE値を 選択します。CとPOSIX照合順序は共に「traditional C」の動作を指定します。これはASCII文字の「A」から「Z」を文字として扱い、ソート順は厳密な文字コードのバイト値によります。

加えて、エンコーディングUTF8では、SQL標準の照合順序名ucs_basicが利用できます。ucs_basicはCと同等のもので、ソート順はUnicodeのコードポイントです。

23.2.2.2. 定義済みの照合順序

オペレーティングシステムが単一のプログラム内(newlocaleや関連する関数)で複数のロケールを使用することをサポートしているか、ICUサポートが組み込み済みの場合、データベースクラスタが初期化される

とinitdbは、オペレーティングシステム上で見つけた全てのロケールに基づく照合順序をシステムカタログのpg_collationに書き込みます。

現在利用可能なロケールを調べるには、SELECT * FROM pg_collationという問合せを使うか、psql内で\dOS+コマンドを使用します。

23.2.2.2.1. libc照合順序

例えば、オペレーティングシステムがde_DE.utf8という名称のロケールを提供した場合、initdbは、de_DE.utf8に設定されたLC_COLLATEとLC_CTYPEの両方を持つUTF8エンコーディングのde_DE.utf8という名称の照合順序を作成します。同時に照合順序の名称から.utf8タグを削除した照合順序も作成します。これは手間を省き、名称がエンコーディングに依存しにくいようになります。それにもかかわらず、照合順序名称の初期値はプラットフォーム依存となることに気をつけてください。

libcが提供するデフォルトの照合順序の集合は、直接オペレーティングシステム内にインストールされたロケールにマップされ、コマンドlocale -aで参照できます。LC_COLLATEとLC_CTYPEで違う値を持つlibc照合順序が必要な場合、あるいはデータベースシステムが初期化された後に新しいロケールがインストールされた場合は、新しい照合順序をCREATE COLLATIONコマンドで作成できます。また、新しいオペレーティングシステムロケールは、pg_import_system_collations()関数でインポートできます。

どのようなデータベース内でも、データベースのエンコーディングを使用している照合順序のみが興味の対象となります。pg_collation内の他のエントリは無視されます。よってde_DEといったようなエンコーディング名が省かれた照合順序は、一般的には一意でなくてもデータベース内では一意であるとみなされます。エンコーディング名が省かれた照合順序を使用することを推奨します。というのも、データベースのエンコーディングを変更するときに、変えなければならないものを1つ減らせるからです。しかし、default、CそしてPOSIX照合順序は、データベースのエンコーディングに関係なく使用可能であることに注意してください。

PostgreSQLは、異なる照合順序オブジェクトは、それらが同じプロパティを持っていても互換性がないものとみなします。例えば、

```
SELECT a COLLATE "C" < b COLLATE "POSIX" FROM test1;
```

は、CとPOSIX照合順序が同じ動作であってもエラーとなります。よってエンコーディング名が省かれた照合順序を混ぜて使用することは推奨されません。

23.2.2.2.2. ICU照合順序

ICUにおいては、すべての可能なロケール名を列挙するのは賢明ではありません。ICUはロケールの固有の名前付けシステムを使っています。しかし、実際の個別のロケール名以上の名前を付ける多くの方法があります。initdbはICUのAPIを使い、照合順序の初期集合を入力するための 個別のロケールの集合を取り出します。ICUが提供する照合順序は、libcロケールと区別するために、SQL環境において、「私的利用」拡張-x-icuを追加したBCP 47言語タグ形式の名前で作成されます。

以下は作成されるかもしれない照合順序の例です。

de-x-icu

ドイツ語照合順序、デフォルトの異型

de-AT-x-icu

オーストリアのドイツ語照合順序、デフォルトの異型

（他に、de-DE-x-icuあるいはde-CH-x-icuというのがあります。しかし、本稿執筆時点では、それらはde-x-icuと同じです。）

und-x-icu（「undefined」用）

ICU「root」照合順序。言語に依存しない適当なソート順を得るために使用してください。

ある種の（利用頻度が低い）エンコーディングをICUはサポートしません。データベースエンコーディングがこのようなものであった場合、pg_collation中のICU照合順序は無視されます。このようなものを使おうとすると、「collation "de-x-icu" for encoding "WIN874" does not exist」というメッセージを伴ったエラーが発生します。

23.2.2.3. 新しい照合順序オブジェクトの作成

標準の定義済み照合順序が十分でない場合は、ユーザはSQLコマンドCREATE COLLATIONで照合順序オブジェクトを作成できます。

すべての定義済みオブジェクト同様、標準の定義済み照合順序はpg_catalogスキーマにあります。これはまた、pg_dumpの保存対象になることを確実にします。

23.2.2.3.1. libc照合順序

以下のようにして新しいlibc照合順序を作成できます。

```
CREATE COLLATION german (provider = libc, locale = 'de_DE');
```

コマンド中のlocale句に使用できる正確な値は、オペレーティングシステムに依存します。Unix系のシステムでは、locale -aコマンドでこのリストを表示できるでしょう。

定義済みのlibc照合順序は、データベースインスタンスが初期化された際に、オペレーティングシステムで定義されたすべての照合順序をすでに含んでいるので、新しいものを手動で作る必要はあまりないでしょう。そうしたことをする理由があるとすれば、異なる命名規則が必要である（この場合は、23.2.2.3.3も参照してください）、あるいはオペレーティングシステムが更新されて、新しい照合順序の定義が提供されるようになった場合です。（この場合はpg_import_system_collations()も参照してください。）

23.2.2.3.2. ICU照合順序

ICUでは、initdbであらかじめロードされた基本的な言語＋国の集合を超えて照合順序をカスタマイズできます。ユーザが、これらの機能を利用して、ソート処理の挙動が自分の要件に適合する照合順序オブジェク

トを定義することを推奨します。ICUロケールの命名規則に関する情報に関しては、http://userguide.icu-project.org/localeとhttp://userguide.icu-project.org/collation/apiを参照してください。利用可能な名前と属性の集合は、ICUのバージョンに依存します。

例を示します。

```
CREATE COLLATION "de-u-co-phonebk-x-icu" (provider = icu, locale = 'de-u-co-phonebk');
CREATE COLLATION "de-u-co-phonebk-x-icu" (provider = icu, locale =
'de@collation=phonebook');
```

電話帳照合順形式を伴うドイツ語の照合順序

最初の例では、BCP 47による「language tag」を使ったICU照合順序を選択しています。次の例は伝統的なICU固有のロケール構文を使用しています。最初の形式を以後使用しますが、これは古いICUのバージョンではサポートされません。

SQL環境における照合順序オブジェクトにはどのような名前でも付けられることに注意してください。この例では、定義済みの照合順序が使っている名前付け形式に従っているので、結果としてBCP 47に従っています。しかし、これはユーザ定義照合順序では必須ではありません。

```
CREATE COLLATION "und-u-co-emoji-x-icu" (provider = icu, locale = 'und-u-co-emoji');
CREATE COLLATION "und-u-co-emoji-x-icu" (provider = icu, locale = '@collation=emoji');
```

絵文字照合順序型を伴うRoot照合順序。Unicode標準51番による。

伝統的なICUロケール命名規則では、rootロケールは空文字列によって選択されることに注目してください。

```
CREATE COLLATION latinlast (provider = icu, locale = 'en-u-kr-grek-latn');
CREATE COLLATION latinlast (provider = icu, locale = 'en@colReorder=grek-latn');
```

ラテン文字の前にギリシャ文字が来るように並べます。(デフォルトではギリシャ文字の前にラテン文字が来ます。)

```
CREATE COLLATION upperfirst (provider = icu, locale = 'en-u-kf-upper');
CREATE COLLATION upperfirst (provider = icu, locale = 'en@colCaseFirst=upper');
```

小文字の前に大文字が来るように並べます。(デフォルトでは最初に小文字が来ます。)

```
CREATE COLLATION special (provider = icu, locale = 'en-u-kf-upper-kr-grek-latn');
CREATE COLLATION special (provider = icu, locale = 'en@colCaseFirst=upper;colReorder=grek-
latn');
```

上記のオプションを組み合わせます。

```
CREATE COLLATION numeric (provider = icu, locale = 'en-u-kn-true');
CREATE COLLATION numeric (provider = icu, locale = 'en@colNumeric=yes');
```

数値の値により、数字の列を並べる数値の順序付けです。例: A-21 < A-123 (自然順ソート(natural sort)としても知られています。)

詳細は、Unicode Technical Standard #35[1]と、BCP 47[2]をご覧ください。可能な照合順序型(co下位タグ)のリストは、CLDR repository[3]で参照できます。ICU Locale Explorer[4]で、特定のロケール定義の詳細を確認できます。k*下位タグを使う例では、は少なくともICUのバージョン54が必要です。

このシステムでは、「大文字小文字の違いを無視」、あるいは「アクセントを無視」、あるいはその類似(ksキーを使用)の照合順序を作成できますが、照合順序が真の大文字小文字あるいはアクセントを無視するように動作するためには、CREATE COLLATION内で決定論的でないとして宣言されている必要があることに注意してください。23.2.2.4を参照してください。宣言されていない場合、照合順序としては同一に扱われるものの、バイトごとの比較では異なる文字列は、バイト値によってソートされます。

> **注記**
>
> 設計上、ICUはロケール名としてどのような文字列も受け入れ、ドキュメントで説明されているフォールバック手続きを使って、ICUが提供するもっとも近いロケールにマッチさせます。したがって、インストールされたICUが実際にはサポートしていない機能を使って照合順序の指定が構成されていたとしても、直接にはフィードバックはありません。そういうわけで、照合順序の定義が要件を満たしているかどうかを確認するためのアプリケーションレベルのテストケースを作成することをお勧めします。

23.2.2.3.3. 照合順序の複製

コマンドCREATE COLLATIONは、既存の照合順序から新しい照合順序を作る際にも利用できます。これは、オペレーティングシステムから独立した照合順序名をアプリケーションで使用可能にしたり、互換性のある名称を作成したり、ICUが提供する照合順序を、よりわかりやすい名称で利用するのに役立ちます。例を示します。

```
CREATE COLLATION german FROM "de_DE";
CREATE COLLATION french FROM "fr-x-icu";
```

23.2.2.4. 非決定論的な照合順序

照合順序は決定論的もしくは非決定論的のどちらかです。決定論的な照合順序は決定論的な比較を使用します。つまり、同じバイト列で構成される場合に限り等しい文字列とみなします。非決定論的な比較は、異なるバイト値で構成される文字列の場合でさえ文字列が等しいと判定するかもしれません。一般的な状況では、大文字小文字を区別しない比較、アクセントを区別しない比較および異なるUnicode正規化形式による文字列の比較が含まれます。このような区別しない比較を実際に実装するかは照合順序のプロパイダ次第です。deterministicフラグはバイト単位の比較を用いて分解されるかどうかのみを決定します。用語の詳細については、Unicode Technical Standard 10[5]を参照してください。

[1] https://www.unicode.org/reports/tr35/tr35-collation.html

[2] https://tools.ietf.org/html/bcp47

[3] https://github.com/unicode-org/cldr/blob/master/common/bcp47/collation.xml

[4] https://ssl.icu-project.org/icu-bin/locexp

[5] https://www.unicode.org/reports/tr10

非決定論的な照合順序を作るためにはCREATE COLLATIONにdeterministic = falseプロパティを指定します。以下に例を示します。

```
CREATE COLLATION ndcoll (provider = icu, locale = 'und', deterministic = false);
```

この例では非決定論的な方法で標準のUnicode照合順序を使えます。具体的には、これは異なる正規形の文字列を正しく比較できるでしょう。より興味深い例は上述したICUカスタマイズ機能を用いた場合です。以下に例を示します。

```
CREATE COLLATION case_insensitive (provider = icu, locale = 'und-u-ks-level2', deterministic =
 false);
CREATE COLLATION ignore_accents (provider = icu, locale = 'und-u-ks-level1-kc-true', deterministic
 = false);
```

すべての標準および事前に定義された照合順序は決定論的であり、すべてのユーザ定義の照合順序はデフォルトで決定論的です。特にUnicodeの全機能およびその特別な場合を考えた際、非決定論的な照合順序はより多くの「正しい」振る舞いを与えると同時に、いくつかの欠点もあります。第一にそれらを使用するとパフォーマンスが低下します。B-treeは非決定的照合順序を使用したインデックスでは重複排除には使用できないことに特に注意してください。また、パターンマッチング操作などで非決定論的な照合順序による操作ができないことも避けられません。したがって、これらは明確に必要とされる場合のみに使用されるべきです。

> **ヒント**
>
> 異なるUnicode正規化形式のテキストを処理する場合、非決定論的な照合を使用する代わりにnormalizeおよびis normalized関数もしくは式を使用して文字列の前処理もしくはチェックをするオプションもあります。それぞれのアプローチで異なるトレードオフがあります。

23.3. 文字セットサポート

PostgreSQLの文字セット（エンコーディングとも呼ばれます）サポートにより、ISO 8859シリーズなどのシングルバイト文字やEUC（拡張Unixコード）、UTF-8、Mule内部コードなどのマルチバイト文字を含む、各種文字セットでテキストを保存することができます。全ての文字セットはクライアントにより透過的に使用することができますが、いくつかは、サーバ内での（つまりサーバサイドエンコーディングとして）使用はサポートされていません。デフォルトの文字セットは、initdbを使用したPostgreSQLデータベースクラスタの初期化時に決定されます。これは、データベースを作成する時に上書きすることができるので、異なる文字セットを使用した複数のデータベースを持つことができます。

しかし重要な制限として、それぞれのデータベースの文字セットがサーバのLC_CTYPE（文字分類）およびLC_COLLATE（文字列並び替え順序）ロケール設定と互換性がなくてはいけないことがあげられます。Cもしくは POSIXロケール設定の場合、どのような文字セットも許可されています。しかし、libcが提供する他のロケール設定の場合、正しく動作する文字セットはひとつだけとなります。（しかしWindowsではUTF-8符号化方式をどのロケールでも使用できます。）ICUサポートが組み込まれている場合は、サーバサイドのすべてではないにしても、ほとんどのエンコーディングで、ICUが提供する照合順序が利用できます。

23.3.1. サポートされる文字セット

PostgreSQLで使用できる文字セットを表 23.1に示します。

表23.1 PostgreSQL文字セット

名前	説明	言語	サーバ?	ICU?	バイト数/文字	別名
BIG5	Big Five	繁体字	いいえ	いいえ	1-2	WIN950、Windows950
EUC_CN	Extended UNIX Code-CN	簡体字	はい	はい	1-3	
EUC_JP	Extended UNIX Code-JP	日本語	はい	はい	1-3	
EUC_JIS_2004	Extended UNIX Code-JP, JIS X 0213	日本語	はい	いいえ	1-3	
EUC_KR	Extended UNIX Code-KR	韓国語	はい	はい	1-3	
EUC_TW	Extended UNIX Code-TW	繁体字、台湾語	はい	はい	1-3	
GB18030	National Standard	中国語	いいえ	いいえ	1-4	
GBK	Extended National Standard	簡体字	いいえ	いいえ	1-2	WIN936、Windows936
ISO_8859_5	ISO 8859-5、ECMA 113	ラテン/キリル	はい	はい	1	
ISO_8859_6	ISO 8859-6、ECMA 114	ラテン/アラビア語	はい	はい	1	
ISO_8859_7	ISO 8859-7、ECMA 118	ラテン/ギリシャ語	はい	はい	1	
ISO_8859_8	ISO 8859-8、ECMA 121	ラテン/ヘブライ語	はい	はい	1	
JOHAB	JOHAB	韓国語（ハングル）	いいえ	いいえ	1-3	
KOI8R	KOI8-R	キリル文字（ロシア）	はい	はい	1	KOI8
KOI8U	KOI8-U	キリル文字（ウクライナ）	はい	はい	1	
LATIN1	ISO 8859-1、ECMA 94	西ヨーロッパ	はい	はい	1	ISO88591
LATIN2	ISO 8859-2、ECMA 94	中央ヨーロッパ	はい	はい	1	ISO88592
LATIN3	ISO 8859-3、ECMA 94	南ヨーロッパ	はい	はい	1	ISO88593

名前	説明	言語	サーバ?	ICU?	バイト数/文字	別名
LATIN4	ISO 8859-4、ECMA 94	北ヨーロッパ	はい	はい	1	ISO88594
LATIN5	ISO 8859-9、ECMA 128	トルコ	はい	はい	1	ISO88599
LATIN6	ISO 8859-10、ECMA 144	北欧	はい	はい	1	ISO885910
LATIN7	ISO 8859-13	バルト語派	はい	はい	1	ISO885913
LATIN8	ISO 8859-14	ケルト	はい	はい	1	ISO885914
LATIN9	ISO 8859-15	LATIN1でヨーロッパと訛りを含む	はい	はい	1	ISO885915
LATIN10	ISO 8859-16、ASRO SR 14111	ルーマニア	はい	いいえ	1	ISO885916
MULE_INTERNAL	Mule内部コード	多言語Emacs	はい	いいえ	1-4	
SJIS	Shift JIS	日本語	いいえ	いいえ	1-2	Mskanji、ShiftJIS、WIN932、Windows932
SHIFT_JIS_2004	Shift JIS, JIS X 0213	日本語	いいえ	いいえ	1-2	
SQL_ASCII	未指定（テキストを参照）	何でも	はい	いいえ	1	
UHC	統合ハングルコード	韓国語	いいえ	いいえ	1-2	WIN949、Windows949
UTF8	Unicode、8ビット	すべて	はい	はい	1-4	Unicode
WIN866	Windows CP866	キリル文字	はい	はい	1	ALT
WIN874	Windows CP874	タイ語	はい	いいえ	1	
WIN1250	Windows CP1250	中央ヨーロッパ	はい	はい	1	
WIN1251	Windows CP1251	キリル文字	はい	はい	1	WIN
WIN1252	Windows CP1252	西ヨーロッパ	はい	はい	1	
WIN1253	Windows CP1253	ギリシャ	はい	はい	1	
WIN1254	Windows CP1254	トルコ	はい	はい	1	
WIN1255	Windows CP1255	ヘブライ	はい	はい	1	
WIN1256	Windows CP1256	アラビア語	はい	はい	1	
WIN1257	Windows CP1257	バルト語派	はい	はい	1	
WIN1258	Windows CP1258	ベトナム語	はい	はい	1	ABC、TCVN、TCVN5712、VSCII

全てのクライアントのAPIが上の一覧表に示した文字セットをサポートしているわけではありません。例えばPostgreSQL JDBCドライバはMULE_INTERNAL、LATIN6、LATIN8、そしてLATIN10をサポートしません。

SQL_ASCIIの設定は、他の設定とかなり異なります。サーバのキャラクタセットがSQL_ASCIIのとき、サーバは0から127のバイト値をASCIIに変換します。一方、128から255までは変換されません。設定がSQL_ASCIIの場合は、符号化は実行されません。よって、この設定は特定の符号化を使用している場合には、その符号化を無視するようになってしまいます。多くの場合、ASCIIではない環境で作業する場合はSQL_ASCIIの設定を使用するのは、賢いことではありません。なぜならPostgreSQLはASCIIではない文字を変換したり検査したりすることは出来ないからです。

23.3.2. 文字セットの設定

initdbでPostgreSQLクラスタのデフォルト文字セット(エンコーディング)を定義します。以下に例を示します。

```
initdb -E EUC_JP
```

これはデフォルトの文字セットをEUC_JP(日本語拡張Unixコード)に設定します。より長いオプションの文字列がお好みなら-Eの代わりに--encodingと書くこともできます。-Eオプションも--encodingオプションも与えられない場合、initdbは、指定もしくはデフォルトのロケールに基づいて適当な符号化方式を決定しようとします。

データベース作成時に選択したロケールと互換性を持つ符号化方式を提供することで、デフォルト以外の符号化方式を指定することができます。

```
createdb -E EUC_KR -T template0 --lc-collate=ko_KR.euckr --lc-ctype=ko_KR.euckr korean
```

これはEUC_KR文字セットとko_KRロケールを使用するkoreanという名前のデータベースを作成します。SQLコマンドで同じことを行うには次のようにします。

```
CREATE DATABASE korean WITH ENCODING 'EUC_KR' LC_COLLATE='ko_KR.euckr' LC_CTYPE='ko_KR.euckr'
 TEMPLATE=template0;
```

上のコマンドにてtemplate0データベースのコピーが指定されていることに注目してください。他のデータベースからコピーする場合、データが破損する結果となる可能性がありますので、符号化方式とロケール設定を元のデータベースの設定から変更することはできません。詳細については22.3を参照してください。

データベースの符号化方式はpg_databaseシステムカタログに格納されます。psqlの-lオプションか\lコマンドで符号化方式を確認することができます。

```
$ psql -l
                                  List of databases
   Name    |  Owner   | Encoding |  Collation  |    Ctype    |          Access Privileges

-----------+----------+-----------+-------------+-------------
+-----------------------------------
 clocaledb | hlinnaka | SQL_ASCII | C           | C           |
 englishdb | hlinnaka | UTF8      | en_GB.UTF8  | en_GB.UTF8  |
 japanese  | hlinnaka | UTF8      | ja_JP.UTF8  | ja_JP.UTF8  |
 korean    | hlinnaka | EUC_KR    | ko_KR.euckr | ko_KR.euckr |
 postgres  | hlinnaka | UTF8      | fi_FI.UTF8  | fi_FI.UTF8  |
```

```
 template0 | hlinnaka | UTF8      | fi_FI.UTF8  | fi_FI.UTF8  | {=c/hlinnaka,hlinnaka=CTc/
hlinnaka}
 template1 | hlinnaka | UTF8      | fi_FI.UTF8  | fi_FI.UTF8  | {=c/hlinnaka,hlinnaka=CTc/
hlinnaka}
(7 rows)
```

重要

最近のオペレーティングシステムでは、PostgreSQLは、LC_CTYPEの設定によりどの文字セットが指定されているか決定できます。そして、一致するデータベース符号化方式のみを強制的に使用します。古いオペレーティングシステムでは、自分で選択したロケールが想定している符号化方式を確実に使用することは各自の責任になります。ここでの間違いは、ソート処理などのロケールに依存する操作が、奇妙な動作するといったことにつながります。

PostgreSQLは、LC_CTYPEがCもしくはPOSIXでもない場合にも、スーパーユーザがSQL_ASCIIエンコーディングでデータベースを作成することを許可します。上記のように、SQL_ASCIIは、データベースに保存されているデータが特定のエンコーディングを持つことを強制しません。さらに、この選択はロケールに依存したおかしな動作を引き起こすリスクを高めます。この設定の組み合わせを使用することは、お勧めできませんし、いつの日か完全に禁止されるかもしれません。

23.3.3. サーバ・クライアント間の自動文字セット変換

PostgreSQLは、多数の文字セットの組み合わせ (23.3.4 のいずれか)に対してサーバとクライアントの間で自動的に文字セットを変換する機能を提供しています。

自動文字セット変換を有効にするためには、クライアントでどのような文字セット(符号化方式)を使用させたいかをPostgreSQLに伝えなければなりません。これを行うにはいくつかの方法があります。

- psqlで\encodingコマンドを使います。\encodingは実行中であってもクライアントの符号化方式を変更させることができます。例えば符号化方式をSJISに変えたい場合は次のように入力します。

  ```
  \encoding SJIS
  ```

- libpq (33.10)はクライアントの符号化方式を制御する関数を保持しています。

- SET client_encoding TOを使います。次のSQLコマンドでクライアントの符号化方式を設定できます。

  ```
  SET CLIENT_ENCODING TO 'value';
  ```

  標準SQLの構文SET NAMESを同じ目的で使うこともできます。

  ```
  SET NAMES 'value';
  ```

  現在のクライアントの符号化方式を問い合わせるには次のようにします。

  ```
  SHOW client_encoding;
  ```

デフォルトの符号化方式に戻すのには次のようにします。

```
RESET client_encoding;
```

- PGCLIENTENCODINGを使います。クライアントの環境でPGCLIENTENCODING環境変数が定義されていると、サーバと接続が確立した時点で自動的にクライアントの符号化方式が選択されます（上で説明したその他のどんな方法でもその後書き換えできます）。
- client_encoding変数を使います。client_encoding変数が設定されていると、サーバとの接続が確立した時点で自動的にクライアントの符号化方式が選択されます（上で説明したその他のどんな方法でもその後書き換えできます）。

EUC_JPをサーバに、そしてLATIN1をクライアントに選んだ場合のように、特定の文字の変換ができない時、日本語文字はLATIN1に入っていないという旨の日本語が返され、エラーが報告されます。

クライアント側のキャラクタセットがSQL_ASCIIに定義されている場合は、符号化変換はサーバ側のキャラクタセットに関係無く無効化されます。サーバ側と同じように、SQL_ASCIIを使用することは、すべてASCIIのデータを扱っている場合を除き、賢い方法ではありません。

23.3.4. 利用可能な文字セットの変換

PostgreSQLは、pg_conversionシステムカタログ内にリストされた変換関数によって2つの文字セット間を変換することが可能です。PostgreSQLでは表 23.2で要約され表 23.3に詳細が示されているように、いくつかの変換があらかじめ組み込まれています。CREATE CONVERSIONSQLコマンドを用いることで新しい変換を作成することができます。（クライアントもしくはサーバの自動変換を使用するためには、変換がその文字セットの組み合わせのための「デフォルト」として設定されている必要があります。）

表23.2 組み込みクライアントもしくはサーバ文字セット変換

サーバ文字セット	利用可能なクライアント文字セット
BIG5	サーバの符号化方式としてサポートされていません
EUC_CN	*EUC_CN*, MULE_INTERNAL, UTF8
EUC_JP	*EUC_JP*, MULE_INTERNAL, SJIS, UTF8
EUC_JIS_2004	*EUC_JIS_2004*, SHIFT_JIS_2004, UTF8
EUC_KR	*EUC_KR*, MULE_INTERNAL, UTF8
EUC_TW	*EUC_TW*, BIG5, MULE_INTERNAL, UTF8
GB18030	サーバの符号化方式としてサポートされていません
GBK	サーバの符号化方式としてサポートされていません
ISO_8859_5	*ISO_8859_5*, KOI8R, MULE_INTERNAL, UTF8, WIN866, WIN1251
ISO_8859_6	*ISO_8859_6*, UTF8
ISO_8859_7	*ISO_8859_7*, UTF8
ISO_8859_8	*ISO_8859_8*, UTF8
JOHAB	サーバの符号化方式としてサポートされていません

サーバ文字セット	利用可能なクライアント文字セット
KOI8R	*KOI8R*, ISO_8859_5, MULE_INTERNAL, UTF8, WIN866, WIN1251
KOI8U	*KOI8U*, UTF8
LATIN1	*LATIN1*, MULE_INTERNAL, UTF8
LATIN2	*LATIN2*, MULE_INTERNAL, UTF8, WIN1250
LATIN3	*LATIN3*, MULE_INTERNAL, UTF8
LATIN4	*LATIN4*, MULE_INTERNAL, UTF8
LATIN5	*LATIN5*, UTF8
LATIN6	*LATIN6*, UTF8
LATIN7	*LATIN7*, UTF8
LATIN8	*LATIN8*, UTF8
LATIN9	*LATIN9*, UTF8
LATIN10	*LATIN10*, UTF8
MULE_INTERNAL	*MULE_INTERNAL*, BIG5, EUC_CN, EUC_JP, EUC_KR, EUC_TW, ISO_8859_5, KOI8R, LATIN1 から LATIN4, SJIS, WIN866, WIN1250, WIN1251
SJIS	サーバの符号化方式としてサポートされていません
SHIFT_JIS_2004	サーバの符号化方式としてサポートされていません
SQL_ASCII	任意（変換は実行されません）
UHC	サーバの符号化方式としてサポートされていません
UTF8	すべての符号化方式がサポートされています
WIN866	*WIN866*, ISO_8859_5, KOI8R, MULE_INTERNAL, UTF8, WIN1251
WIN874	*WIN874*, UTF8
WIN1250	*WIN1250*, LATIN2, MULE_INTERNAL, UTF8
WIN1251	*WIN1251*, ISO_8859_5, KOI8R, MULE_INTERNAL, UTF8, WIN866
WIN1252	*WIN1252*, UTF8
WIN1253	*WIN1253*, UTF8
WIN1254	*WIN1254*, UTF8
WIN1255	*WIN1255*, UTF8
WIN1256	*WIN1256*, UTF8
WIN1257	*WIN1257*, UTF8
WIN1258	*WIN1258*, UTF8

表23.3 すべての組み込み文字セット変換

変換名 [a]	変換元符号化方式	変換先符号化方式
big5_to_euc_tw	BIG5	EUC_TW

変換名 [a]	変換元符号化方式	変換先符号化方式
big5_to_mic	BIG5	MULE_INTERNAL
big5_to_utf8	BIG5	UTF8
euc_cn_to_mic	EUC_CN	MULE_INTERNAL
euc_cn_to_utf8	EUC_CN	UTF8
euc_jp_to_mic	EUC_JP	MULE_INTERNAL
euc_jp_to_sjis	EUC_JP	SJIS
euc_jp_to_utf8	EUC_JP	UTF8
euc_kr_to_mic	EUC_KR	MULE_INTERNAL
euc_kr_to_utf8	EUC_KR	UTF8
euc_tw_to_big5	EUC_TW	BIG5
euc_tw_to_mic	EUC_TW	MULE_INTERNAL
euc_tw_to_utf8	EUC_TW	UTF8
gb18030_to_utf8	GB18030	UTF8
gbk_to_utf8	GBK	UTF8
iso_8859_10_to_utf8	LATIN6	UTF8
iso_8859_13_to_utf8	LATIN7	UTF8
iso_8859_14_to_utf8	LATIN8	UTF8
iso_8859_15_to_utf8	LATIN9	UTF8
iso_8859_16_to_utf8	LATIN10	UTF8
iso_8859_1_to_mic	LATIN1	MULE_INTERNAL
iso_8859_1_to_utf8	LATIN1	UTF8
iso_8859_2_to_mic	LATIN2	MULE_INTERNAL
iso_8859_2_to_utf8	LATIN2	UTF8
iso_8859_2_to_windows_1250	LATIN2	WIN1250
iso_8859_3_to_mic	LATIN3	MULE_INTERNAL
iso_8859_3_to_utf8	LATIN3	UTF8
iso_8859_4_to_mic	LATIN4	MULE_INTERNAL
iso_8859_4_to_utf8	LATIN4	UTF8
iso_8859_5_to_koi8_r	ISO_8859_5	KOI8R
iso_8859_5_to_mic	ISO_8859_5	MULE_INTERNAL
iso_8859_5_to_utf8	ISO_8859_5	UTF8
iso_8859_5_to_windows_1251	ISO_8859_5	WIN1251
iso_8859_5_to_windows_866	ISO_8859_5	WIN866
iso_8859_6_to_utf8	ISO_8859_6	UTF8

変換名 [a]	変換元符号化方式	変換先符号化方式
iso_8859_7_to_utf8	ISO_8859_7	UTF8
iso_8859_8_to_utf8	ISO_8859_8	UTF8
iso_8859_9_to_utf8	LATIN5	UTF8
johab_to_utf8	JOHAB	UTF8
koi8_r_to_iso_8859_5	KOI8R	ISO_8859_5
koi8_r_to_mic	KOI8R	MULE_INTERNAL
koi8_r_to_utf8	KOI8R	UTF8
koi8_r_to_windows_1251	KOI8R	WIN1251
koi8_r_to_windows_866	KOI8R	WIN866
koi8_u_to_utf8	KOI8U	UTF8
mic_to_big5	MULE_INTERNAL	BIG5
mic_to_euc_cn	MULE_INTERNAL	EUC_CN
mic_to_euc_jp	MULE_INTERNAL	EUC_JP
mic_to_euc_kr	MULE_INTERNAL	EUC_KR
mic_to_euc_tw	MULE_INTERNAL	EUC_TW
mic_to_iso_8859_1	MULE_INTERNAL	LATIN1
mic_to_iso_8859_2	MULE_INTERNAL	LATIN2
mic_to_iso_8859_3	MULE_INTERNAL	LATIN3
mic_to_iso_8859_4	MULE_INTERNAL	LATIN4
mic_to_iso_8859_5	MULE_INTERNAL	ISO_8859_5
mic_to_koi8_r	MULE_INTERNAL	KOI8R
mic_to_sjis	MULE_INTERNAL	SJIS
mic_to_windows_1250	MULE_INTERNAL	WIN1250
mic_to_windows_1251	MULE_INTERNAL	WIN1251
mic_to_windows_866	MULE_INTERNAL	WIN866
sjis_to_euc_jp	SJIS	EUC_JP
sjis_to_mic	SJIS	MULE_INTERNAL
sjis_to_utf8	SJIS	UTF8
windows_1258_to_utf8	WIN1258	UTF8
uhc_to_utf8	UHC	UTF8
utf8_to_big5	UTF8	BIG5
utf8_to_euc_cn	UTF8	EUC_CN
utf8_to_euc_jp	UTF8	EUC_JP
utf8_to_euc_kr	UTF8	EUC_KR

変換名 [a]	変換元符号化方式	変換先符号化方式
utf8_to_euc_tw	UTF8	EUC_TW
utf8_to_gb18030	UTF8	GB18030
utf8_to_gbk	UTF8	GBK
utf8_to_iso_8859_1	UTF8	LATIN1
utf8_to_iso_8859_10	UTF8	LATIN6
utf8_to_iso_8859_13	UTF8	LATIN7
utf8_to_iso_8859_14	UTF8	LATIN8
utf8_to_iso_8859_15	UTF8	LATIN9
utf8_to_iso_8859_16	UTF8	LATIN10
utf8_to_iso_8859_2	UTF8	LATIN2
utf8_to_iso_8859_3	UTF8	LATIN3
utf8_to_iso_8859_4	UTF8	LATIN4
utf8_to_iso_8859_5	UTF8	ISO_8859_5
utf8_to_iso_8859_6	UTF8	ISO_8859_6
utf8_to_iso_8859_7	UTF8	ISO_8859_7
utf8_to_iso_8859_8	UTF8	ISO_8859_8
utf8_to_iso_8859_9	UTF8	LATIN5
utf8_to_johab	UTF8	JOHAB
utf8_to_koi8_r	UTF8	KOI8R
utf8_to_koi8_u	UTF8	KOI8U
utf8_to_sjis	UTF8	SJIS
utf8_to_windows_1258	UTF8	WIN1258
utf8_to_uhc	UTF8	UHC
utf8_to_windows_1250	UTF8	WIN1250
utf8_to_windows_1251	UTF8	WIN1251
utf8_to_windows_1252	UTF8	WIN1252
utf8_to_windows_1253	UTF8	WIN1253
utf8_to_windows_1254	UTF8	WIN1254
utf8_to_windows_1255	UTF8	WIN1255
utf8_to_windows_1256	UTF8	WIN1256
utf8_to_windows_1257	UTF8	WIN1257
utf8_to_windows_866	UTF8	WIN866
utf8_to_windows_874	UTF8	WIN874
windows_1250_to_iso_8859_2	WIN1250	LATIN2

変換名[a]	変換元符号化方式	変換先符号化方式
windows_1250_to_mic	WIN1250	MULE_INTERNAL
windows_1250_to_utf8	WIN1250	UTF8
windows_1251_to_iso_8859_5	WIN1251	ISO_8859_5
windows_1251_to_koi8_r	WIN1251	KOI8R
windows_1251_to_mic	WIN1251	MULE_INTERNAL
windows_1251_to_utf8	WIN1251	UTF8
windows_1251_to_windows_866	WIN1251	WIN866
windows_1252_to_utf8	WIN1252	UTF8
windows_1256_to_utf8	WIN1256	UTF8
windows_866_to_iso_8859_5	WIN866	ISO_8859_5
windows_866_to_koi8_r	WIN866	KOI8R
windows_866_to_mic	WIN866	MULE_INTERNAL
windows_866_to_utf8	WIN866	UTF8
windows_866_to_windows_1251	WIN866	WIN
windows_874_to_utf8	WIN874	UTF8
euc_jis_2004_to_utf8	EUC_JIS_2004	UTF8
utf8_to_euc_jis_2004	UTF8	EUC_JIS_2004
shift_jis_2004_to_utf8	SHIFT_JIS_2004	UTF8
utf8_to_shift_jis_2004	UTF8	SHIFT_JIS_2004
euc_jis_2004_to_shift_jis_2004	EUC_JIS_2004	SHIFT_JIS_2004
shift_jis_2004_to_euc_jis_2004	SHIFT_JIS_2004	EUC_JIS_2004

[a] 変換名は、標準の命名規定に従います。英数字以外のすべての文字がアンダースコアに置き換えられた変換元符号化方式の正式名に_to_が続き、同様に処理された変換先符号化方式名が続きます。したがって、これらの名前は、表 23.1に示されている通常の符号化方式名と異なる場合があります。

23.3.5. 推奨文書

ここに記したものは様々な符号化方式システムを学習するのに良い資料です。

*CJKV*日中韓越情報処理: 中国語、日本語、韓国語 & ベトナム語処理

EUC_JP、EUC_CN、EUC_KR、EUC_TWの詳しい説明があります。

https://www.unicode.org/

Unicode協会のWebサイトです。

RFC 3629

ここでUTF-8（8ビットUCS/Unicode変換書式）が定義されています。

第24章 定常的なデータベース保守作業

他のデータベースソフトウェア同様、PostgreSQLも、最適な性能を得るために定常的に実施しなければならない作業があります。ここで説明する作業は*必要な*ものであり、その性質上繰り返し行うべきものです。しかし、cronスクリプトなどの標準ツールや、Windowsのタスクスケジューラを使用して簡単に自動化することができます。適切なスクリプトを設定し、その実行がうまく行くかどうかを点検することは、データベース管理者の責任です。

明らかに必要な保守作業の1つに、定期的なデータのバックアップコピーの作成があります。最近のバックアップがなければ、(ディスクの破損、火災、重要なテーブルの間違った削除などの)破滅の後、復旧することができません。PostgreSQLで可能なバックアップとリカバリ機構については、第25章にて詳細に説明します。

他の保守作業の主なカテゴリには、定期的なデータベースの「バキューム」があります。この作業については24.1で説明します。問い合わせプランナで使用される統計情報の更新も密接に関連しますが、こちらに関しては24.1.3で説明します。

この他、定期的に行わなければならない作業にログファイルの管理があります。これについては24.3で説明します。

check_postgres[1]が、データベースの健全性を監視し、異常な状態を報告するために用意されています。check_postgresはNagiosおよびMRTGに組み込まれたものですが、独立して実行させることができます。

PostgreSQLは他のデータベース管理システムに比べ、保守作業は少ないと言えます。それでもなお、これらの作業に適切に注意することは、システムに対する快適かつ充実した経験を確実に得るのに効果があります。

24.1. 定常的なバキューム作業

PostgreSQLデータベースは*バキューム処理*として知られている定期的な保守を必要とします。多くのインストレーションでは、24.1.6で説明されている*自動バキュームデーモン*でのバキューム処理を行わせることで充分です。それぞれの状況に合った最善の結果を得るため、そこで説明する自動バキューム用パラメータの調整が必要かもしれません。データベース管理者によっては、cronもしくはタスクスケジューラスクリプトに従って典型的に実行される、手作業管理のVACUUMコマンドによりデーモンの活動を補足したり、置き換えたりすることを意図するかもしれません。手作業管理のバキューム処理を適切に設定するためには、以下のいくつかの小節で説明する問題点を理解することが必須です。自動バキューム処理に信頼をおいている管理者にとっても、この資料に目を通すことはそれらの理解と自動バキューム処理の調整に役に立つことでしょう。

24.1.1. バキューム作業の基本

PostgreSQLのVACUUMコマンドは以下の理由により定期的にそれぞれのテーブルを処理しなければなりません。

1. 更新、あるいは削除された行によって占められたディスク領域の復旧または再利用。

[1] https://bucardo.org/check_postgres/

2. PostgreSQL問い合わせプランナによって使用されるデータ統計情報の更新。

3. 可視性マップの更新。これによりインデックスオンリースキャンが高速化される。

4. トランザクション*ID*の周回またはマルチトランザクション*ID*の周回による非常に古いデータの損失を防止。

以降の小節で説明するように、これらの理由の1つ1つはVACUUM操作の実行について、その頻度の変動や対象領域の変動に影響します。

VACUUMには、標準VACUUMとVACUUM FULLという2つの種類があります。VACUUM FULLはより多くのディスク容量を回収することができますが、実行にとても時間がかかります。また、VACUUMの標準形式は実運用のデータベースに対する操作と同時に実行させることができます。(SELECT、INSERT、UPDATE、DELETEなどのコマンドは通常通りに動作し続けます。しかし、バキューム処理中はALTER TABLEなどのコマンドを使用してテーブル定義を変更することはできません。) VACUUM FULLはそれが作用する全てのテーブルに対し排他ロックを必要とするので、それらテーブルのその他の用途と並行して行うことはできません。一般的に、管理者は標準VACUUMの使用に努め、VACUUM FULLの使用を避けるべきです。

VACUUMは、かなりの量のI/Oトラフィックを発生させます。このため、他の実行中のセッションの性能を劣化させる可能性があります。バックグラウンドで実行されるバキューム処理による性能への影響を軽減させることを調整できるような設定パラメータがあります。19.4.4を参照してください。

24.1.2. ディスク容量の復旧

PostgreSQLでは、行のUPDATEもしくはDELETEは古い行を即座に削除しません。この方法は、多版同時性制御(MVCC。第13章を参照してください)の恩恵を受けるために必要なものです。あるバージョンの行は他のトランザクションから参照される可能性がある場合は削除されてはなりません。しかし最終的には、更新された前の行や削除された行を参照するトランザクションはなくなります。必要なディスク容量が無制限に増加しないように、これらが占める領域は、新しい行で再利用できるように回収されなければなりません。これはVACUUMを実行することで行われます。

標準形式のVACUUMは、テーブルとインデックス内の不要な行を削除し、その領域を将来の再利用が可能であるものとして記録します。しかし、その領域をオペレーティングシステムに返却することはありません。例外として、テーブルの末尾に完全に空のページが存在し、かつそのテーブルの排他ロックが容易に獲得できるような特殊な場合には、その領域を返却します。対照的にVACUUM FULLは、不要な領域のない全く新しいバージョンのテーブルファイルを書き出すことで、積極的にテーブルを圧縮します。テーブルの容量を最小化しますが、長い時間がかかる可能性があります。また操作が終わるまで、テーブルの新しいコピー用に余計なディスク領域を必要とします。

定常的なバキューム作業の通例の目安はVACUUM FULLの必要性を避けるに充分な頻度で標準VACUUMを行うことです。自動バキュームデーモンはこのようにして作動を試みます。そして実際VACUUM FULLを行いません。この手法において、その発想はテーブルを最小サイズに保つのではなく、ディスク領域使用の安定状態を保持することです。それぞれのテーブルは、その最小サイズにバキューム作業とバキューム作業の間で使用されることになる容量を加えたのに等しい空間を占有します。VACUUM FULLは、テーブルをその最小サイズまで縮小し、ディスク空間をオペレーティングシステムに返却するために使用することができますが、もし将来そのテーブルが再び肥大化するのであれば、大した意味がありません。従って、程よい頻度の標準VACUUMを実行するほうが、不定期のVACUUM FULLを実行するより大量の更新テーブルを保守するにはより良い取り組みとなります。

例えば負荷が少ない夜間に全ての作業を行うように、一部の管理者は自身で計画したバキューム作業の方を選びます。固定したスケジュールに従ってバキューム作業を行うことについての問題は、もし更新作業によりテーブルが予期せぬ急増に遭遇した場合、空き領域を回収するためにVACUUM FULLが本当に必要となるところまで肥大化することです。自動バキュームデーモンを使用することにより、この問題は緩和されます。なぜなら、このデーモンは更新作業に反応して動的にバキューム作業を計画するからです。完全に作業量を予測することができない限り、デーモンを完全に無効化するのは勧められません。取り得る妥協案の1つは、いつになく激しい更新作業にのみ反応するよう、デーモンのパラメータを設定することです。これにより、抑制可能な範囲を維持しつつ、負荷が標準的な場合に計画化されたVACUUMがまとめて作業を行うことを想定することができます。

自動バキュームを使用しない場合の典型的な方式は、データベース全体のVACUUMを1日1回使用頻度が低い時間帯にスケジュールすることです。必要に応じて、更新頻度の激しいテーブルのバキューム処理をより頻繁に行うよう追加してください。(非常に高い頻度でデータの更新を行うインストレーションの中では、分間隔位という頻度で高負荷なテーブルのVACUUMを行うこともあります。) 1つのクラスタで複数のデータベースがある場合、それぞれをバキュームすることを忘れないでください。vacuumdbプログラムが役に立つかもしれません。

ヒント

大規模な更新や削除作業の結果としてテーブルが不要な行バージョンを大量に含む場合、通常のVACUUMでは満足のゆくものではないかもしれません。もしそのようなテーブルを所有し、それが占有する余分なディスク空間の回収が必要であれば、VACUUM FULL、またはその代わりにCLUSTERやテーブルを書き換えるALTER TABLE構文の1つを使用しなければなりません。これらのコマンドはテーブル全体を新しいコピーに書き換え、それに対する新規インデックスを作成します。これらの選択肢はすべて排他ロックを必要とします。新しいものが完成するまで、テーブルの旧コピーとインデックスは解放されませんので、元のテーブルと同程度の容量の余計なディスク領域も一時的に使用することに注意してください。

ヒント

テーブルの内容が定期的に完全に削除される場合、DELETEの後にVACUUMを使用するよりも、TRUNCATEを使用する方が良いでしょう。TRUNCATEはテーブルの全ての内容を即座に削除します。また、その後に不要となったディスク容量を回収するためにVACUUMやVACUUM FULLを行う必要がありません。不利な点は厳格なMVCC動作が違反となることです。

24.1.3. プランナ用の統計情報の更新

PostgreSQL問い合わせプランナは、優れた問い合わせ計画を作成するのに、テーブルの内容に関する統計情報に依存しています。この統計情報はANALYZEによって収集されます。このコマンドはそのものを呼び出す以外にも、VACUUMのオプション処理としても呼び出すことができます。合理的な精度の統計情報を持つことは重要です。さもないと非効率的な計画を選択してしまい、データベースの性能を悪化させてしまいます。

自動バキュームデーモンが有効になっている場合は、テーブルの内容が大きく変更されたときはいつでも自動的にANALYZEコマンドを発行します。しかし、特にテーブルの更新作業が「興味のある」列の統計情報に影

響を与えないことが判っている時、手作業により計画されたANALYZE操作を当てにする方が好ましいと管理者は思うかもしれません。デーモンは、挿入または更新された行数の関数としてANALYZEを厳密に計画します。しかし、意味のある統計情報の変更につながるかどうかは判りません。

領域復旧のためのバキューム処理と同様、頻繁な統計情報の更新は、滅多に更新されないテーブルよりも更新の激しいテーブルにとってより有益です。しかし、頻繁に更新されるテーブルであっても、データの統計的な分布が大きく変更されなければ、統計情報を更新する必要はありません。単純な鉄則は、テーブル内の列の最小値、最大値にどのくらいの変化があったかを考えることです。例えば、行の更新時刻を保持するtimestamp列の場合、最大値は行が追加、更新されるにつれて、単純に増加します。こういった列は、おそらく、例えば、あるWebサイト上のアクセスされたページのURLを保持する列よりも頻繁に統計情報を更新する必要があるでしょう。このURL列の更新頻度も高いものかもしれませんが、その値の統計的な分布の変更は相対的に見ておそらく低いものです。

特定のテーブルに対してANALYZEを実行することができます。また、テーブルの特定の列のみに対してさえも実行することができます。ですので、アプリケーションの要求に応じて、他よりも頻繁に一部の統計情報を更新できるような柔軟性があります。しかし、実際には、操作が高速であるため、単にデータベース全体を解析することが最善です。ANALYZEは、すべての行を読むのではなく、テーブルから統計的にランダムな行を抽出して使用します。

> **ヒント**
>
> 列単位でのANALYZE実行頻度の調整はあまり実用的とは言えるものではありませんが、ANALYZEで集計される統計情報の詳細レベルの調整を列単位で行うことは価値がある場合があります。WHERE句でよく使用され、データ分布の規則性がほとんどない列は、他の列よりもより細かいデータの度数分布が必要になるでしょう。ALTER TABLE SET STATISTICSを参照するか、default_statistics_target設定パラメータでデータベース全体のデフォルトを変更してください。
>
> またデフォルトで、関数の選択性に関して利用可能な制限付きの情報があります。しかし、関数呼び出しを使用する式インデックスを作成する場合、有用な統計情報が関数に関して収集されます。これにより式インデックスを使用する問い合わせ計画を大きく改良することができます。

> **ヒント**
>
> 自動バキュームデーモンは、有益になる頻度を決定する手段がありませんので、外部テーブルに対してANALYZEコマンドを発行しません。問い合わせが適切な計画作成のために外部テーブルの統計情報が必要であれば、適当なスケジュールでこれらのテーブルに対して手作業で管理するANALYZEコマンドを実行することを勧めます。

24.1.4. 可視性マップの更新

バキュームは、どのページにすべての有効トランザクション（およびページが再度更新されるまでの将来のトランザクション）で可視であることが分かっているタプルのみが含まれるかを追跡するために、各テーブルの可視性マップの保守を行います。2つの目的があります。1つ目はバキューム自身が、整理するものがありませんので、こうしたページを次回飛ばすことができます。

2つ目は、PostgreSQLが、背後にあるテーブルを参照することなく、インデックスのみを使用して一部の問い合わせに応えることができるようになります。PostgreSQLのインデックスにはタプルの可視性に関する情報を持ちませんので、通常のインデックススキャンは合致したインデックス項目のヒープタプルを取り込み、現在のトランザクションから可視であるべきかどうか検査します。一方でインデックスオンリースキャンはまず可視性マップを検査します。そのページのタプルがすべて可視であることが分かれば、ヒープの取り出しを省くことができます。可視性マップによりディスクアクセスを防ぐことができる大規模なデータ群に対して、特に有効です。可視性マップはヒープより非常に小さいため、ヒープが非常に大きい場合であっても簡単にキャッシュすることができます。

24.1.5. トランザクションIDの周回エラーの防止

PostgreSQLのMVCCトランザクションのセマンティクスは、トランザクションID（XID）番号の比較が可能であることに依存しています。現在のトランザクションのXIDよりも新しい挿入時のXIDを持ったバージョンの行は、「未来のもの」であり、現在のトランザクションから可視であってはなりません。しかし、トランザクションIDのサイズには制限（32ビット）があり、長時間（40億トランザクション）稼働しているクラスタはトランザクションの周回を経験します。XIDのカウンタが一周して0に戻り、そして、突然に、過去になされたトランザクションが将来のものと見えるように、つまり、その出力が不可視になります。端的に言うと、破滅的なデータの損失です。（実際はデータは保持されていますが、それを入手することができなければ、慰めにならないでしょう。）これを防ぐためには、すべてのデータベースにあるすべてのテーブルを少なくとも20億トランザクションごとにバキュームする必要があります。

定期的なバキューム処理によりこの問題が解決する理由は、VACUUMが行に凍結状態という印をつけて、挿入トランザクションの効果が確実に可視になるような十分遠い過去にコミットされたトランザクションによりそれらが挿入されたことを表すからです。PostgreSQLは特別なXID、FrozenTransactionIdを確保します。このXIDは通常のXIDの比較規則には従わず、常に全ての通常のXIDよりも古いものとみなされます。通常のXID（2以上の値）はmodulo-2^{32}という数式を使用して比較されます。これは、全ての通常のXIDでは、20億の「より古い」XIDと20億の「より新しい」XIDが存在することを意味します。言い換えると、通常のXID空間は終わることなく循環されているということです。そのため、ある特定のXIDであるバージョンの行を作成すると、そのバージョンの行は、以降の20億トランザクションからはどの通常のXIDについて比較しているのかには関係なく、「過去のもの」と認識されます。そのバージョンの行が20億トランザクション以上後にも存在していた場合、それは突然に未来のものとして認識されます。これを防ぐために、凍結された行バージョンは挿入XIDがFrozenTransactionIdであるかのように扱われ、それで、周回問題に関係なく、すべての通常のトランザクションから「過去のもの」として認識され、また、そのバージョンの行はどれだけ古いものであろうと、削除されるまで有効状態となります。

> **注記**
>
> 9.4より前のバージョンのPostgreSQLでは、行の挿入XIDを実際にFrozenTransactionIdで置換することで凍結が実装されており、これは行のxminシステム列として見えていました。それより新しいバージョンでは単にフラグのビットをセットするだけで、行の元のxminは後の検証での利用に備えて保存します。しかし、9.4以前のバージョンからpg_upgradeでアップグレードしたデータベースでは、xminが FrozenTransactionId (2)に等しい行がまだあるかもしれません。
>
> また、システムカタログにはxminがBootstrapTransactionId (1)に等しい行が含まれる場合があり、これはその行がinitdbの最初の段階で挿入されたことを意味します。FrozenTransactionIdと同様、この特別なXIDはすべての通常のXIDよりも古いものとして扱われます。

vacuum_freeze_min_ageは、その行バージョンが凍結される前に、XID値がどのくらい経過しているのかを制御します。この設定値を大きくすることで、そうでなければ凍結状態になる行がすぐに再び修正されるのであれば、不必要な作業を避けられるかもしれませんが、この設定値を小さくすることでテーブルを次にバキュームする必要が起こるまで継続できるトランザクション数が増加します。

VACUUMは可視性マップを使用して、テーブルのどのページを走査する必要があるかを決定します。通常は、不要な行バージョンを持っていないページを読み飛ばします。このとき、そのページに古いXID値の行バージョンがまだある可能性があったとしても読み飛ばします。したがって、通常のVACUUMでは必ずしもテーブル内のすべての古い行バージョンを凍結するわけではありません。定期的にVACUUMは積極的なバキュームを実行し、そのときは不要な行も凍結されていないXID値やMXID値もないページのみを読み飛ばします。vacuum_freeze_table_ageはVACUUMがいつこれを行うかを制御します。つまり、最後にそのような走査が行われた後に実行されたトランザクションの数がvacuum_freeze_table_ageからvacuum_freeze_min_ageを引いた数より大きいとき、全可視ではあるが全凍結ではないページも走査されます。vacuum_freeze_table_ageを0に設定するとVACUUMはすべての走査についてこのより積極的な戦略を使うようになります。

テーブルをバキュームすることなく処理できる最大の時間は、20億トランザクションから最後に積極的なバキュームを実行した時点のvacuum_freeze_min_ageの値を差し引いたものです。この時間よりも長期間バキュームを行わないと、データ損失が発生するかもしれません。これを確実に防止するために、自動バキュームがautovacuum_freeze_max_age設定パラメータで指定された時代より古いXIDを持つ、凍結状態でない行を含む可能性がある任意のテーブルに対して呼び出されます。(これは自動バキュームが無効であっても起こります。)

これは、あるテーブルがバキュームされていなかったとしても、自動バキュームがおよそautovacuum_freeze_max_age - vacuum_freeze_min_ageトランザクション毎に呼び出されることを意味します。領域確保のために定常的にバキューム処理を行うテーブルでは、これは重要ではありません。しかし、(挿入のみで更新や削除が行われないテーブルを含む)静的なテーブルでは、領域確保のためのバキューム処理を行う必要がなくなりますので、非常に長期間静的なテーブルでは、強制的な自動バキューム間の間隔を最大まで延ばすことができます。記載するまでもありませんが、autovacuum_freeze_max_ageを増やすことでもvacuum_freeze_min_ageを減らすことでも、これを行うことができます。

vacuum_freeze_table_ageに対する有効な最大値は0.95 * autovacuum_freeze_max_ageです。これより値が高いと値は最大値までに制限されます。autovacuum_freeze_max_ageより高い値は、周回防止用の自動バキュームがその時点でいずれにせよ誘発され、0.95という乗算係数がそれが起こる前に手動によるVACUUM実行の余地を残すため、意味を持ちません。経験則に従うと、定期的に計画されたVACUUMもしくは通常の削除・更新作業により誘発された自動バキュームがその期間で実行されるように十分な間隔を残しておくように、vacuum_freeze_table_ageはautovacuum_freeze_max_ageより多少低い値に設定されるべきです。これを余りにも近い値に設定すると、たとえ領域を回収するために最近テーブルがバキュームされたとしても、周回防止用の自動バキュームに帰着します。一方より低い値はより頻繁な積極的バキュームを引き起こします。

autovacuum_freeze_max_age(およびそれに付随するvacuum_freeze_table_age)を増やす唯一の欠点は、データベースクラスタのサブディレクトリpg_xactとpg_commit_tsがより大きな容量となることです。autovacuum_freeze_max_ageの範囲まですべてのトランザクションのコミット状況と(track_commit_timestampが指定されていれば)タイムスタンプを格納しなければならないためです。コミット状況は1トランザクション当たり2ビット使用しますので、もしautovacuum_freeze_max_ageをその最大許容値である20億に設定している場合、pg_xactはおよそ0.5ギガバイトまで、pg_commit_tsは約20GBまで膨らむものと考えられます。これがデータベースサイズ全体に対してとるに足らないも

のであれば、autovacuum_freeze_max_ageを最大許容値に設定することを勧めます。さもなければ、pg_xactとpg_commit_tsの容量として許容できる値に応じてそれらを設定してください。（デフォルトは2億トランザクションです。換算するとpg_xactはおよそ50MB、pg_commit_tsはおよそ2GBの容量となります。）

vacuum_freeze_min_age を減らすことにも1つ欠点があります。これによりVACUUMが大して役に立たなくなるかもしれません。テーブル行がすぐに変更される場合（新しいXIDを獲得することになります）、行バージョンを凍結することは時間の無駄です。そのため、この設定は、行の変更が起こらなくなるまで凍結されない程度に大きくすべきです。

データベース内のもっとも古い凍結されていないXIDの年代を追跡するために、VACUUMはシステムテーブルpg_classとpg_databaseにXID統計情報を保持します。特に、テーブルに対応するpg_class行のrelfrozenxid列には、テーブルに対する最後の積極的なVACUUMで使用された凍結切捨てXIDが含まれます。この切り捨てXIDよりも古いXIDを持つトランザクションにより挿入されたすべての行は凍結状態であることが保証されています。同様に、データベースに対応するpg_database行のdatfrozenxid列は、データベース内で現れる凍結されていないXIDの下限値です。これは、そのデータベース内のテーブル当たりのrelfrozenxid値の最小値です。この情報を検査する簡便な方法は、以下の問い合わせを実行することです。

```
SELECT c.oid::regclass as table_name,
       greatest(age(c.relfrozenxid),age(t.relfrozenxid)) as age
FROM pg_class c
LEFT JOIN pg_class t ON c.reltoastrelid = t.oid
WHERE c.relkind IN ('r', 'm');

SELECT datname, age(datfrozenxid) FROM pg_database;
```

age列は切り捨てXIDから現在のトランザクションXIDまでのトランザクション数を測ります。

VACUUMは通常は最後のバキュームの後で変更されたページのみ走査しますが、relfrozenxidはテーブルの凍結されていないXIDを含むかもしれないすべてのページを走査したときのみ繰り上がります。これは、relfrozenxidがvacuum_freeze_table_ageトランザクション年齢より大きい時、VACUUMのFREEZEオプションが使用された時、もしくは使用されない行バージョンを削除するため全てのページをバキュームしなければならなくなった時に発生します。VACUUMがテーブルの全凍結になっていないすべてのページを走査したとき、age(relfrozenxid)は、使用されたvacuum_freeze_min_age設定より若干大きくなるはずです（VACUUMを起動してから始まったトランザクションの数分大きくなります）。relfrozenxidを繰り上げるVACUUMがautovacuum_freeze_max_ageに達するまでにテーブルに対して発行されない場合、そのテーブルに対して自動バキュームが早急に強制されます。

何らかの理由により自動バキュームがテーブルの古いXIDの整理に失敗した場合、システムはデータベースの最古のXIDが周回ポイントから1100万トランザクションに達した場合と似たような警告メッセージを発行し始めます。

```
WARNING:  database "mydb" must be vacuumed within 10985967 transactions
HINT:  To avoid a database shutdown, execute a database-wide VACUUM in that database.
```

（ヒントで示唆されたように手動VACUUMはこの問題を解決します。しかし、VACUUMはスーパーユーザで実行されなければなりません。さもないとシステムカタログの処理に失敗し、このためデータベース

のdatfrozenxidを桁上げすることができません。) こうした警告も無視し続け、周回するまでのトランザクションが100万より少なくなると、システムは停止し、新しいトランザクションの起動を拒絶します。

```
ERROR:  database is not accepting commands to avoid wraparound data loss in database "mydb"
HINT:  Stop the postmaster and vacuum that database in single-user mode.
```

この100万トランザクションという安全マージンは、管理者が必要なVACUUMコマンドを手作業で実行することで、データを失うことなくリカバリすることができるようにするために存在します。しかし、システムがこの安全のための停止モードになると、コマンドを実行しませんので、実行するためには、サーバを停止し、シングルユーザモードでサーバを起動してVACUUMを行う他ありません。この停止モードはシングルユーザモードでは強制されません。シングルユーザモードの使用に関する詳細はpostgresマニュアルページを参照してください。

24.1.5.1. マルチトランザクションと周回

マルチトランザクション*ID*は複数のトランザクションによる行ロックをサポートするのに使われます。タプルヘッダにはロック情報を格納するために限られた容量しかありませんので、二つ以上のトランザクションが同時に行をロックする時には必ず、その情報は「マルチプル(訳注:複数の)トランザクションID」、略してマルチトランザクションID、にエンコードされます。あるマルチトランザクションIDにどのトランザクションIDが含まれているかという情報はpg_multixactサブディレクトリに別に格納されており、マルチトランザクションIDのみがタプルヘッダのxmaxフィールドに現れます。トランザクションIDと同様に、マルチトランザクションIDは32ビットカウンタと対応する記憶領域として実装されており、どちらも注意深い年代管理や記憶領域の整理、周回の取り扱いが必要です。各マルチトランザクションにはメンバの一覧を保持する独立した記憶領域があり、そこでも32ビットカウンタを使っているので同じように管理しなければなりません。

テーブルの何らかの部分に対しVACUUM走査されるときはいつでも、そのときに見つかったvacuum_multixact_freeze_min_ageよりも古いマルチトランザクションIDはすべて異なる値で置き換えられます。異なる値とは、0かもしれませんし、単一のトランザクションIDかもしれませんし、より新しいマルチトランザクションIDかもしれません。各テーブルでは、pg_class.relminmxidがそのテーブルのタプルにまだ現れるマルチトランザクションIDのうちできるだけ古いものを保持しています。この値がvacuum_multixact_freeze_table_ageよりも古ければ、積極的バキュームが強制されます。前節で説明したように、積極的なバキュームでは全凍結であるとわかっているページのみがスキップされます。pg_class.relminmxidに対してその年代を調べるのにmxid_age()を使えます。

積極的なVACUUM走査は、その原因が何かに関わらず、そのテーブルのその値を進めることができます。結局、データベースすべてのテーブルすべてが走査され、最も古いマルチトランザクション値が進められますので、ディスク上でより古いマルチトランザクションを保持している領域は削除できます。

安全装置として、autovacuum_multixact_freeze_max_ageよりもそのマルチトランザクション年代が大きいどのテーブルに対しても、積極的なバキューム走査が起こります。メンバ記憶領域の使用量がアドレス指定可能な記憶領域の50%を越えた場合にも、積極的なバキューム走査は、マルチトランザクション年代の一番古いものから始めて、すべてのテーブルに対して段階的に起こります。この種の積極的走査はどちらも、自動バキュームが名目上は無効にされていても発生します。

24.1.6. 自動バキュームデーモン

PostgreSQLには、省略可能ですが強く推奨される自動バキュームという機能があります。これはVACUUMとANALYZEコマンドの実行を自動化することを目的としたものです。有効にすると、自動バキュームは大量のタプルの挿入、更新、削除があったテーブルを検査します。この検査は統計情報収集機能を使用します。したがって、track_countsがtrueに設定されていないと、自動バキュームを使用することができません。デフォルトの設定では、自動バキュームは有効で、関連するパラメータも適切に設定されています。

実際のところ「自動バキュームデーモン」は複数のプロセスから構成されます。自動バキュームランチャという永続的デーモンプロセスが存在し、自動バキュームワーカプロセスがすべてのデータベースを処理します。ランチャは、1つのワーカを各データベースに対しautovacuum_naptime秒ごとに開始するよう試みることにより、時間に対して作業を分散化します。(したがってインストレーションにN個のデータベースがある場合、新規ワーカがautovacuum_naptime/N秒毎に起動されます。) 同時に最大autovacuum_max_workers個のプロセスが実行可能です。処理対象のデータベースがautovacuum_max_workersより多くある場合、次のデータベースは最初のワーカが終了するとすぐに処理されます。それぞれのワーカプロセスはデータベース内の各テーブルを検査し、必要に応じてVACUUMまたはANALYZEコマンドを発行します。log_autovacuum_min_durationも自動バキュームワーカの活動を監視するために設定できます。

短期間にいくつかの大規模なテーブルがすべてバキューム対象として適切な状態になったとすると、すべての自動バキュームワーカはこうしたテーブルに対するバキューム処理に長い期間占領される可能性があります。これにより、ワーカが利用できるようになるまで、他のテーブルやデータベースに対するバキュームが行われなくなります。また、単一データベースに対するワーカ数には制限はありませんが、ワーカはすでに他のワーカによって実行された作業を繰り返さないように試みます。ワーカの実行数はmax_connections制限にもsuperuser_reserved_connections制限にも計上されないことに注意してください。

テーブルのrelfrozenxid値がautovacuum_freeze_max_ageトランザクション年齢よりも古い場合、そのテーブルは常にバキュームされます (これはfreeze max ageがストレージパラメータにより変更されたテーブルに対しても適用されます。以下を参照)。さもなければ、直前のVACUUMの後に不要となったタプル数が「バキューム閾値」を超えると、テーブルはバキュームされます。このバキューム閾値は以下のように定義されます。

```
バキューム閾値 = バキューム基礎閾値 + バキューム規模係数 * タプル数
```

ここで、バキューム基礎閾値はautovacuum_vacuum_threshold、バキューム規模係数はautovacuum_vacuum_scale_factor、タプル数はpg_class.reltuplesです。

直前のバキュームの後に挿入されたタプル数が定義された挿入閾値を超えた場合も、テーブルはバキュームされます。ここで挿入閾値は以下のように定義されます。

```
バキューム挿入閾値 = バキューム基礎挿入閾値 + バキューム挿入規模係数 * タプル数
```

ここで、バキューム挿入基礎閾値はautovacuum_vacuum_insert_threshold、バキューム挿入規模係数はautovacuum_vacuum_insert_scale_factorです。そのようなバキュームは、テーブルの一部を全可視と印づけたり、タプルを凍結したりもできますので、後続のバキュームで必要となる作業を減らせます。より早いバキュームによりタプルを凍結できますので、INSERT操作を受けたもののUPDATE/DELETE操作を全くもしくはほとんど受けていないテーブルに対しては、テーブルのautovacuum_freeze_min_ageを下げる助けになるでしょう。不要となったタプル数と挿入されたタプル数は、統計情報コレクタから

取り出されます。これは、各UPDATE、DELETE操作で更新される、ほぼ正確な数です。(負荷が高いと一部の情報が失われる可能性があることから、これはほぼ正確な数でしかありません。) テーブルのrelfrozenxid値がvacuum_freeze_table_ageトランザクション年齢より大きい場合、古いタプルを凍結して、relfrozenxidを繰り上げるため、積極的なバキュームが実行されます。そうでなければ最後のバキュームの後に変更されたページのみ走査されます。

解析でも似たような条件が使用されます。以下で定義される閾値が、前回のANALYZEの後に挿入、更新、削除されたタプル数と比較されます。

```
解析閾値 = 解析基礎閾値 + 解析規模係数 * タプル数
```

一時テーブルには自動バキュームでアクセスすることはできません。したがってセッションのSQLコマンドを用いて適切なバキュームおよび解析操作を行わなければなりません。

デフォルトの閾値と規模係数は、postgresql.confから取られますが、(他の多くの自動バキューム制御パラメータと合わせて)テーブル毎に上書きすることができます。より詳細な情報はStorage Parametersを参照してください。テーブルのストレージパラメータで設定が変更されると、そのテーブルを処理する時にその値が使用されます。そうでなければ、全体設定が使われます。全体設定についての詳細な情報は19.10を参照してください。

複数のワーカプロセスが実行している場合、自動バキュームコスト遅延パラメータ(19.4.4を参照してください)は実行中のワーカ全体に「振り分け」られます。このため、ワーカの実稼働数に関らず、システムに与えるI/Oの総影響は変わりありません。しかし、テーブル毎のautovacuum_vacuum_cost_delayまたはautovacuum_vacuum_cost_limitストレージパラメータが設定されたテーブルを処理するワーカは振り分けアルゴリズムでは考慮されません。

自動バキュームワーカは通常は他のコマンドをブロックしません。自動バキュームが保持するSHARE UPDATE EXCLUSIVEロックと衝突するロックを、プロセスが獲得しようとした場合には、ロックの獲得により自動バキュームが中断されます。衝突するロックモードに関しては表 13.2を参照してください。しかしながら、自動バキュームがトランザクションIDの周回を防ぐために動作している(すなわち、pg_stat_activityビューの自動バキューム問い合わせ名が(to prevent wraparound)で終わっている)場合には、自動バキュームは自動的には中断されません。

> **警告**
>
> SHARE UPDATE EXCLUSIVEロックと衝突するロックを獲得する、定期的に動作するコマンド(例えばANALYZE)により、自動バキュームが実質的に終わらなくなることがあります。

24.2. 定常的なインデックスの再作成

REINDEXコマンドまたは一連の個々の再構築処理を使用して定期的にインデックスを再構築することが価値がある状況があります。

完全に空になったB-treeインデックスページは再利用のために回収されます。しかしまだ非効率的な領域使用の可能性があります。ページからわずかを残しほとんどすべてのインデックスキーが削除されたとしても、

ページは割り当てられたまま残ります。各範囲において、わずかを残しほとんどすべてのキーが削除されるようなパターンで使用されると、領域が無駄に使用されることが分かります。こうした使用状況では、定期的なインデックス再構築を推奨します。

B-tree以外のインデックスが膨張する可能性はまだよく調査されていません。B-tree以外の任意の種類のインデックスを使用する際には、インデックスの物理容量を定期的に監視することを勧めます。

また、B-treeインデックスでは、新規に構築したインデックスの方が何度も更新されたインデックスよりもアクセスが多少高速です。新しく構築されたインデックスでは論理的に近接するページが通常物理的にも近接するからです。(これはB-tree以外のインデックスではあてはまりません。) アクセス速度を向上させるためだけに周期的にインデックスを再構築することは価値があるかもしれません。

REINDEXはすべての状況で安全に簡単に使うことができます。このコマンドはデフォルトでACCESS EXCLUSIVEロックを要求しますので、CONCURRENTLYオプションを付けて実行する方が好ましい場合がしばしばあります。その場合にはSHARE UPDATE EXCLUSIVEロックしか要求しません。

24.3. ログファイルの保守

データベースサーバのログ出力を/dev/nullに渡して単に破棄するのではなく、どこかに保存しておくことを推奨します。問題の原因を究明する時にログ出力は貴重です。しかし、ログ出力は(特により高いデバッグレベルの時に)巨大になりがちですので、際限なく保存したくはないでしょう。新しいログファイルを開始させ、適切な期間を経過した古いログファイルを捨てるために、ログファイルを回転させる必要があります。

単にpostgresのstderrをファイルに渡している場合、ログ出力を保持できますが、そのログファイルを切り詰めるためにはサーバを停止させ、再度起動させるしか方法がありません。開発環境でPostgreSQLを使用しているのであればこれで構いませんが、実運用サーバでこの振舞いが適切となることはほぼありません。

サーバのstderrを何らかのログ回転プログラムに送信する方が良いでしょう。組み込みのログ回転機能があり、postgresql.confのlogging_collector設定パラメータをtrueに設定することでこれを使用することができます。このプログラムを制御するパラメータについては19.8.1で説明します。また、この方法を使用して、機械読み取りしやすいCSV(カンマ区分値)書式でログデータを取り込むことができます。

また、既に他のサーバソフトウェアで使用している外部のログ回転プログラムがあるのであれば、それを使用したいと考えるでしょう。例えば、Apache配布物に含まれるrotatelogsをPostgreSQLで使用することができます。これを行う一つの方法は、単にサーバのstderrを目的のプログラムにパイプで渡すことです。pg_ctlを使用してサーバを起動している場合はstderrは既にstdoutにリダイレクトされていますので、以下の例のようにコマンドをパイプする必要があるだけです。

```
pg_ctl start | rotatelogs /var/log/pgsql_log 86400
```

PostgreSQL組み込みのログ収集機構により生成されるログファイルを集めるのにlogrotateを設定することで、上の方法を組み合わせることができます。この場合、ログ収集機構はログファイルの名前と位置を定義する一方、logrotateは周期的にそのファイルをアーカイブします。ログ回転を開始する時に、logrotateはアプリケーションが以降の出力を新しいファイルに送ることを確実にしなければなりません。通常これは、アプリケーションにSIGHUPシグナルを送るpostrotateスクリプトにより行なわれ、アプリケーションはその後ログファイルを再度開きます。PostgreSQLでは、その代わりにpg_ctlにlogrotateオプションを付けて実行できます。サーバはこのコマンドを受け取ると、ログ収集の設定に応じて新しいログファイルに切り替えるか、既存のファイルを再度開くかします(19.8.1を参照してください)。

> **注記**
>
> 静的なログファイル名を使う場合には、開けるファイルの最大数に達したりファイルテーブルのオーバーフローが起きた場合に、サーバがログファイルを再度開くのに失敗するかもしれません。この場合には、ログ回転が成功するまでログメッセージは古いログファイルに送られます。logrotateがログファイルを圧縮して削除するよう設定されていれば、サーバはこの期間にログに残そうとしたメッセージを失うかもしれません。この問題を避けるため、ログファイル名を動的に割り当てて、開いているログファイルを無視するprerotateスクリプトを使うようにログ収集機構を設定できます。

この他の実運用レベルのログ出力の管理方法は、syslogに送信し、syslogにファイルの回転を行わせることです。このためには、postgresql.confのlog_destination設定パラメータをsyslog(syslogのみにログを出力)に設定してください。そして、新しいログファイルへの書き込みを始めたい時に、syslogデーモンにSIGHUPシグナルを送信してください。ログ回転を自動化させたい場合は、logrotateプログラムを設定することで、syslogからのログファイルを扱うことができます。

しかし、多くのシステムではsyslogは特に巨大なログメッセージに関してあまり信頼できません。必要なメッセージを切り詰めてしまったり、破棄してしまったりする可能性があります。また、Linuxでは、syslogはメッセージごとにディスクに書き出すため、性能が良くありません。(同期化を無効にするため、syslog設定ファイル内のファイル名の先頭に「-」を使うことができます。)

上述の手法は全て、新しいログファイルを開始する周期を設定することができますが、古い、既に役に立たなくなったログファイルの削除は扱わないことに注意してください。おそらく定期的に古いログファイルを削除するバッチジョブを設定することになるでしょう。他に、回転用プログラムを設定して古いログファイルを周期的に上書きさせるという方法もあります。

pgBadger[2]という外部プロジェクトは洗練されたログファイルの解析を行います。check_postgres[3]は、通常ではない多くの状態の検出を行うのと同時にログファイルに重要なメッセージが現れた時にNagiosで警告する機構を提供します。

[2] https://pgbadger.darold.net/

[3] https://bucardo.org/check_postgres/

第25章 バックアップとリストア

貴重なデータを保持しているあらゆるもの同様、PostgreSQLデータベースも定期的にバックアップされなければなりません。バックアップの手順は基本的に簡単ですが、使用されている諸技術といくつかの前提条件を明確に理解しておくことが重要です。

PostgreSQLのデータをバックアップする場合、3つの異なる手法があります。

- SQLによるダンプ
- ファイルシステムレベルのバックアップ
- 継続的アーカイブ

それぞれ長所と短所があります。ひとつひとつ順を追って以下の節で説明します。

25.1. SQLによるダンプ

このダンプ方法の背景にはSQLコマンドでファイルを生成し、そのファイルをサーバが再度読み込みを行った時に、ダンプした時点と同じ状態が再構築されるという意図があります。この目的のため、PostgreSQLはpg_dumpユーティリティプログラムを提供しています。このコマンドの基本となる使い方は以下の通りです。

```
pg_dump dbname > dumpfile
```

見てわかる通り、pg_dumpは結果を標準出力に書き出します。これがどのように活用できるかをこれから説明します。上記のコマンドはテキストファイルを作成しますが、pg_dumpは並列処理を可能にしたり、オブジェクトのリストアをより細かく制御できる他のフォーマットでファイルを作れます。

pg_dumpは、PostgreSQLの通常のクライアントアプリケーションです（その中でも特に優れた機能を発揮するものですが）。ということは、データベースに接続可能なあらゆるリモートホストからこのバックアップ手順を実行できます。しかし、pg_dumpは特別な権限で実行される訳ではないことを忘れないでください。特に、バックアップを行う全てのテーブルに対して読み取り権限が必要ですので、データベース全体のバックアップを実行する場合、ほとんど常にデータベースのスーパーユーザとして実行しなければなりません。(もしデータベース全体のバックアップを取るのに十分な権限を持っていない場合には、`-n schema`もしくは、`-t table`のようなオプションを使って、データベースのアクセス権のある部分をバックアップできます。)

pg_dumpを行うデータベースサーバを特定するにはコマンドラインの`-h host`オプションと`-p port`オプションを使用します。デフォルトのホストはローカルホスト、またはPGHOST環境変数で指定したものです。同様に、デフォルトのポートはPGPORT環境変数で指定されているか、うまく行かない場合にはコンパイル時の設定がデフォルトとなります（そこはうまくできていて、サーバは通常コンパイル時の設定をデフォルトとします）。

他のPostgreSQLのクライアントアプリケーションのように、pg_dumpはデフォルトでオペレーティングシステムの現在のユーザ名と同じデータベースユーザ名で接続します。これを書き換えるには`-U`オプションを付けるかPGUSER環境変数を設定します。pg_dumpの接続は（第20章で説明されている）通常のクライアント認証方法によることを思い出してください。

後で述べる他のバックアップ手法に対するpg_dumpの重要な利点は、pg_dumpの出力は一般に新しいバージョンのPostgreSQLに再ロードできるということです。一方、ファイルレベルのバックアップと継続的

アーカイブは両方とも非常にサーバ、バージョン依存です。pg_dumpは、32ビットから64ビットのサーバに移行するなどの異なるマシンアーキテクチャにデータベースを移す場合に上手くいく唯一の方法でもあります。

pg_dumpで作成されたダンプは、内部的に整合性があります。つまり、ダンプはpg_dumpが開始された際のデータベースのスナップショットを示しています。pg_dumpの操作はデータベースに対する他の作業を妨げません（ALTER TABLEのほとんどの形態であるような排他的ロックが必要な作業は例外です）。

25.1.1. ダンプのリストア

pg_dumpで作成されたテキストファイルはpsqlプログラムで読み込まれることを意図しています。以下に、ダンプをリストアする一般的なコマンドを示します。

```
psql dbname < dumpfile
```

ここでdumpfileはpg_dumpコマンドにより出力されたファイルです。dbnameデータベースはこのコマンドでは作成されません。（例えばcreatedb -T template0 dbname のようにして）psqlを実行する前に自分でtemplate0から作成してください。psqlはpg_dumpと似たような、接続データベースサーバと使用するユーザ名を指定するオプションに対応しています。詳細については、psqlのリファレンスページを参照してください。テキスト形式ではないダンプファイルはpg_restore ユーティリティを使いリストアします。

SQLダンプのリストアを実行する前に、ダンプされたデータベース内のオブジェクトを所有するユーザやそのオブジェクト上に権限を与えられたユーザも存在しなければなりません。存在していない場合、リストアはそのオブジェクトの元々の所有権や付与された権限を再作成することができません（このようにしたい場合もあるでしょうが、通常そうではありません）。

デフォルトでpsqlスクリプトは、SQLエラーが起きた後も実行を継続します。ON_ERROR_STOP変数を設定してpsqlを実行することで、その動作を変更し、SQLエラーが起きた場合にpsqlが、終了ステータス3で終了するようにしたいと思うかもしれません。

```
psql --set ON_ERROR_STOP=on dbname < dumpfile
```

どちらにしても、部分的にリストアされたデータベースにしかなりません。他に、ダンプ全体を1つのトランザクションとしてリストアするように指定することができます。こうすれば、リストアが完全に終わるか、完全にロールバックされるかのどちらかになります。このモードは、psqlのコマンドラインオプションに-1または--single-transactionを渡すことで指定できます。このモードを使用する場合、数時間かけて実行していたリストアが軽微なエラーでロールバックしてしまうことに注意してください。しかし、部分的にリストアされたダンプから手作業で複雑なデータベースを整理するよりまだましかもしれません。

pg_dumpとpsqlではパイプから読み書きができるので、あるサーバから別のサーバへデータベースを直接ダンプできます。以下に例を示します。

```
pg_dump -h host1 dbname | psql -h host2 dbname
```

重要

pg_dumpで作成されるダンプはtemplate0と相対関係にあります。つまりtemplate1を経由して追加されたあらゆる言語、プロシージャなどもpg_dumpによりダンプされます。その結果としてリストアす

> る際に、カスタマイズされたtemplate1を使用している場合は、上記の例のように、template0から空のデータベースを作成する必要があります。

バックアップをリストアした後、問い合わせオプティマイザが有用な統計情報を使用できるように、各データベースに対してANALYZEを実行することを勧めます。より詳しくは、24.1.3と24.1.6を参照してください。効率的に大規模なデータをPostgreSQLにロードする方法に関するより多くの勧告については、14.4を参照してください。

25.1.2. pg_dumpallの使用

pg_dumpは一度に単一のデータベースのみをダンプします。また、ロールやテーブル空間についての情報はダンプしません。(これらはテーブル毎ではなくクラスタ全体のものだからです。) データベースクラスタの全内容の簡便なダンプをサポートするために、pg_dumpallプログラムが提供されています。pg_dumpallは指定されたクラスタの各データベースのバックアップを行い、そして、ロールやテーブル空間定義などのクラスタ全体にわたるデータを保存します。このコマンドの基本的な使用方法は

```
pg_dumpall > dumpfile
```

です。ダンプの結果はpsqlでリストアできます。

```
psql -f dumpfile postgres
```

(実際、開始時に任意の既存のデータベース名を指定することができますが、空のクラスタ内にロードする場合は、通常 postgres を使用すべきです。) ロールやテーブル空間の情報をリストアしなければならないので、pg_dumpallのダンプをリストアする時には、データベーススーパーユーザのアクセス権限を確実に必要とします。テーブル空間を使用している場合、ダンプ内のテーブル空間のパスが新しいインストレーションで適切であることを確認してください。

pg_dumpallはコマンドを発令することによりロール、テーブル空間、およびデータベースを再作成し、それぞれのデータベースに対してpg_dumpを起動します。このことは、それぞれのデータベースには内部的に矛盾がない一方、異なるデータベースのスナップショットは完全に同期しないことを示しています。

クラスタレベルでのデータはpg_dumpall の--globals-only オプションを使用して出力することができます。このコマンドは個々のデータベースにpg_dump コマンドを実行しつつ、フルバックアップを取得する際に必要です。

25.1.3. 大規模データベースの扱い

オペレーティングシステムの中には最大ファイルサイズに制限があるものがあり、大きなpg_dump出力ファイルを作成しているときに問題を引き起こします。幸運なことに、pg_dumpは標準出力に書き出すことができますので、Unix標準のツールを使ってこの潜在的な問題を解決できます。取りうる方法がいくつか存在します。

圧縮ダンプの使用. たとえば、自分が愛用しているgzipのような圧縮プログラムが使えます。

```
pg_dump dbname | gzip > filename.gz
```

元に戻すには次のようにします。

```
gunzip -c filename.gz | psql dbname
```

あるいは次のようにもできます。

```
cat filename.gz | gunzip | psql dbname
```

splitの使用. splitコマンドで結果を使用しているファイルシステムが受け付けられる大きさに分割することができます。例えば1メガバイトずつに分割するには次のようにします。

```
pg_dump dbname | split -b 1m - filename
```

元に戻すには次のようにします。

```
cat filename* | psql dbname
```

pg_dumpのカスタムダンプ書式の使用. もしPostgreSQLがzlib圧縮ライブラリインストール済みのシステム上で構築されたのなら、カスタムダンプ書式では出力ファイルに書き出す時にデータを圧縮します。gzipを使用した時と似通ったダンプサイズとなりますが、テーブルの復元を部分的に行えるという点で優れていると言えます。以下のコマンドは、カスタムダンプ書式でのデータベースのダンプを行います。

```
pg_dump -Fc dbname > filename
```

カスタム書式のダンプはpsql用のスクリプトではありませんので、代わりにpg_restoreでリストアしなければなりません。例えば以下のようにします。

```
pg_restore -d dbname filename
```

詳細はpg_dumpとpg_restoreのリファレンスページを参照してください。

巨大なデータベースに対しては、そのほかの2つの手法のうちの1つと一緒にsplitを組み合わせる必要があるかもしれません。

pg_dumpの並列実行. pg_dumpを並列実行することで、大きなデータベースのダンプを高速に実行することができます。これは同時に複数テーブルのダンプを実行します。並列度は-jパラメータを指定することで制御できます。並列ダンプはディレクトリダンプ書式のみサポートします。

```
pg_dump -j num -F d -f out.dir dbname
```

pg_restore -jコマンドでダンプファイルを並列でリストアすることができます。これはpg_dump -jでダンプファイルが作成されたか、否かにかかわらず、カスタムもしくはディレクトリダンプ書式で作成されたダンプファイルに使用できます。

25.2. ファイルシステムレベルのバックアップ

バックアップ戦略の代替案としてPostgreSQLがデータベース内のデータを保存するために使用しているファイルを直接コピーする方法があります。18.2にこれらのファイルがどこにあるか解説されています。下記のような通常のファイルシステムのバックアップを行うどんな方法でも問題ありません。

```
tar -cf backup.tar /usr/local/pgsql/data
```

しかしこの方法には2つの制約があり、そのためにあまり実用的ではなく、少なくともpg_dumpより劣ると言わざるを得ません。

1. 有効なバックアップを行うにはデータベースサーバを必ず停止しなければなりません。全ての接続を無効とするような中途半端な対策では作用しません（tarやその類似ツールはある時点におけるファイルシステムの原子的なスナップショットを取らないことと同時に、サーバ内の内部バッファリングの理由によるからです）。サーバの停止に関しては18.5を参照してください。言うまでもありませんが、データをリストアする前にもサーバを停止させる必要があります。

2. データベースのファイルシステムレイアウトの詳細を熟知している場合、ある個別のテーブルやデータベースをそれぞれのファイルやディレクトリからバックアップしたり復元したりすることを試みたいと思うかもしれません。しかし、それらのファイル内の情報はすべてのトランザクションのコミット状態を保持するコミットログファイルpg_xact/*なしでは使えないため、この方法では正常なバックアップは行えません。テーブルファイルはこの情報があって初めて意味をなします。もちろんテーブルとそれに付帯するpg_xactデータだけで復元することも、データベースクラスタにある他のテーブルを無効としてしまうのでできません。ですので、ファイルシステムバックアップは、データベースクラスタ全体の完全なバックアップとリストア処理にのみ動作します。

その他のファイルシステムバックアップ方法として、ファイルシステムが「整合性を維持したスナップショット」機能をサポートしている場合（かつ、正しく実装されていると信用する場合）、データディレクトリのスナップショットを作成する方法があります。典型的な手順では、データベースを含むボリュームの「凍結スナップショット」を作成し、データディレクトリ全体（上述のように、一部だけではいけません）をスナップショットからバックアップデバイスにコピーし、そして、凍結スナップショットを解放します。これはデータベースサーバが稼動中であっても動作します。しかし、こうして作成されたバックアップは、データベースサーバが適切に停止されなかった状態のデータベースファイルを保存します。そのため、このバックアップデータでデータベースサーバを起動する時、直前のサーバインスタンスがクラッシュしたものとみなされ、WALログが取り直されます。これは問題ではありません。単に注意してください（そして、確実にバックアップにWALファイルを含めてください）。CHECKPOINTコマンドをスナップショット取得前に発行することで復旧時間を減らすこともできます。

対象のデータベースが複数のファイルシステムにまたがって分散している場合、全てのボリュームに対して完全に同期した凍結スナップショットを得る方法が存在しない可能性があります。例えば、データファイルとWALログが異なったディスク上にあったり、テーブル空間が異なるファイルシステム上にある場合、スナップショットは同時でなければなりませんので、スナップショットのバックアップを使用できない可能性があります。こうした状況では、整合性を維持したスナップショット技術を信用する前に使用するファイルシステムの文書を熟読してください。

同時実行のスナップショットができない場合、選択肢の1つとして、全ての機能の停止したスナップショットを確定させるのに充分な時間、データベースサーバをシャットダウンさせることが挙げられます。他の選択肢は、継続的なベースバックアップの保管（25.3.2）を行うことです。こうしたバックアップには、バックアップ中のファイルシステムの変更を心配する必要がないためです。これにはバックアップ処理期間のみに継続的な保管を行う必要があり、継続的なアーカイブリカバリ（25.3.4）を使用してリストアを行います。

ファイルシステムをバックアップするその他の選択肢としてrsyncの使用が挙げられます。これを行うには、先ずデータベースサーバが稼働中にrsyncを実行し、そしてrsync --checksumを実行するのに充分な間だけデータベースサーバを停止します。(rsyncはファイルの更新時刻に関して1秒の粒度しかありませんので、

--checksumが必要です。) 次のrsyncは、比較的転送するデータ量が少なく、サーバが稼働していないため最終結果に矛盾がない事から、最初のrsyncよりも迅速です。この方法で最小の稼働停止時間でファイルシステムのバックアップを行う事ができます。

ファイルシステムバックアップは、概してSQLによるダンプより大きくなることに注意してください。(pg_dumpでは、例えばインデックスの内容をダンプする必要はありません。単にコマンドで再作成します。)しかし、ファイルシステムのバックアップを取るほうがより高速でしょう。

25.3. 継続的アーカイブとポイントインタイムリカバリ(PITR)

PostgreSQLは常に、クラスタのデータディレクトリ以下のpg_wal/ディレクトリ内で先行書き込みログ(WAL)を管理しています。このログはデータベースのデータファイルに行われた全ての変更を記録します。このログは主にクラッシュ時の安全性を目的としています。システムがクラッシュしたとしても、最後のチェックポイント以降に作成されたログ項目を「やり直し」することで、データベースを整合性を維持した状態にリストアすることができます。しかし、この存在するログファイルを使用して、データベースのバックアップ用の第3の戦略が可能になりました。ファイルシステムレベルのバックアップとWALファイルのバックアップを組み合わせるという戦略です。復旧が必要ならば、ファイルシステムバックアップをリストアし、その後にバックアップされたWALファイルを再生することで、システムを最新の状態にできます。管理者にとって、この方法はこれまで説明した方法よりかなり複雑になりますが、以下のような大きな利点が複数あります。

- 開始時点のファイルシステムバックアップは完全な整合状態である必要はありません。そのバックアップ内の内部的な不整合はログのやり直しによって修正されます (これは、クラッシュからの復旧時に行われることと大きな違いはありません)。ですので、ファイルシステムのスナップショット機能を必要としません。単にtarなどのアーカイブツールが必要です。

- 再生の際にWALファイルの並びを数に制限なく連ねて組み合わせられますので、単にWALファイルのアーカイブを続けることで連続したバックアップを達成できます。これは、頻繁に完全なバックアップを行うことが困難な、大規模なデータベースでは特に価値があります。

- WAL項目の再生を最後まで行わなければならないということはありません。やり直しを任意の時点までで停止することができ、それにより、その時点までのデータベースの整合性を持ったスナップショットを得ることができます。このような技術がポイントインタイムリカバリを補助するものであり、元となるベースバックアップの取得時点以降の任意の時点の状態にデータベースをリストアすることが可能になります。

- 連続的に一連のWALファイルを、同一のベースバックアップをロードしている別のマシンに配送することで、ウォームスタンバイシステムを保有することができます。つまり、任意の時点でその2番目のマシンを、ほぼ現時点のデータベースの複製を持った状態で有効にすることができます。

> **注記**
>
> pg_dumpとpg_dumpallはファイルシステムレベルのバックアップを生成しませんので、継続的アーカイブ方式の一部として使うことはできません。そのダンプは論理的なものであり、WALのやり直しで使うのに十分な情報を含んでいません。

通常のファイルシステムバックアップ技術の場合と同様、この方法は、一部ではなく、データベースクラスタ全体のリストア処理のみをサポートできます。また、アーカイブ用に大量の格納領域を必要とします。ベースバックアップはかさばる場合があり、また、高負荷なシステムではアーカイブしなければならないWALの流量をメガバイト単位で生成します。しかし、これは、高信頼性が必要な、多くの状況でむしろ好まれるバックアップ手法です。

継続的アーカイブ（多くのデータベースベンダで「オンラインバックアップ」とも呼ばれます）を使用して復旧を成功させるためには、少なくともバックアップの開始時点まで遡る、連続した一連のアーカイブ済みWALファイルが必要です。ですので、運用するためには、最初のベースバックアップを取得する前にWALファイルをアーカイブする手順を設定し試験しなければなりません。したがって、まずWALファイルのアーカイブ機構について説明します。

25.3.1. WALアーカイブの設定

抽象的な意味では、実行中のPostgreSQLシステムは無限に長い一連のWALレコードを生成します。システムは物理的にこの並びを、通常1つ16メガバイト（このセグメントサイズはinitdbの実行時に変更可能です）の、WALセグメントファイルに分割します。このセグメントファイルには、概念的なWALの並び内の位置を反映した、数字の名前が付与されます。WALアーカイブを行わない場合、システムは通常数個のセグメントファイルを生成し、不要となったセグメントファイルの名前をより大きなセグメント番号に変更することでそれを「リサイクル」します。最後のチェックポイントより前の内容を持つセグメントファイルはもはや重要でなく、リサイクルできると見なされます。

WALデータをアーカイブする場合、完成したセグメントファイルのそれぞれの内容を取り出し、再利用のために回収される前にそのデータをどこかに保存することが必要です。アプリケーションと利用できるハードウェアに依存しますが、数多くの「データをどこかに保存する」方法があります。例えば、NFSでマウントした他のマシンのディレクトリにセグメントファイルをコピーすること、あるいは、テープ装置に書き出すこと（元々のファイル名を識別する手段があることを確認してください）、それらを一度にまとめてCDに焼くこと、そのほか全く異なったなんらかの方法などです。柔軟性をデータベース管理者に提供するために、PostgreSQLは、どのようにアーカイブがなされたかについて一切想定しないようになっています。その代わりにPostgreSQLは、管理者に完全なセグメントファイルをどこか必要な場所にコピーするシェルコマンドを指定させます。このコマンドは単純なcpでも構いませんし、また、複雑なシェルスクリプトを呼び出しても構いません。全て管理者に任されています。

WALアーカイブを有効にするにはwal_level設定パラメータをreplica（またはreplicaより高いパラメータ）に、archive_modeをonに設定し、archive_command設定パラメータで使用するシェルコマンドを指定します。実行するには、これらの設定を postgresql.conf ファイルに常に置きます。archive_command では、%pはアーカイブするファイルのパス名に置換され、%fはファイル名部分のみに置換されます。（パス名は、サーバの現在の作業用ディレクトリ、つまり、クラスタのデータディレクトリから見て相対的なものです。）コマンド内に%文字自体を埋め込む必要があれば%%を使ってください。最も簡単でよく使用されるコマンドは以下のようなものになります。

```
archive_command = 'test ! -f /mnt/server/archivedir/%f && cp %p /mnt/server/archivedir/%f'  #
 Unix
archive_command = 'copy "%p" "C:\\server\\archivedir\\%f"'  # Windows
```

これは、アーカイブ可能なWALセグメントを/mnt/server/archivedirディレクトリにコピーします（これは一例です。推奨するものではなく、また、全てのプラットフォームで動作しない可能性があります）。%pおよび%fパラメータが置き換えられたあと、実行された実コマンドは以下のようになります。

```
test ! -f /mnt/server/archivedir/000000010000000A900000065 && cp pg_wal/000000010000000A900000065 /
mnt/server/archivedir/000000010000000A900000065
```

類似したコマンドがアーカイブされるそれぞれの新規ファイルに生成されます。

このアーカイブ用コマンドはPostgreSQLサーバを稼動させるユーザと同じ所有権で実行されます。アーカイブされる一連のWALファイルには、実質、データベース内の全てが含まれていますので、アーカイブしたデータをのぞき見から確実に保護しなければならないでしょう。例えば、グループや全員に読み込み権限を付与していないディレクトリにデータをアーカイブしてください。

アーカイブ用コマンドが成功した場合のみにゼロという終了ステータスを返すことが重要です。PostgreSQLは、ゼロという結果に基づいて、そのファイルのアーカイブが成功したことを想定し、そのファイルを削除したり回収するかもしれません。しかし、非ゼロのステータスは、PostgreSQLに対してファイルがアーカイブされなかったことを通知し、成功するまで定期的に再試行させます。

通常アーカイブ用コマンドは既存のアーカイブ済みファイルの上書きを行わないように設計されなければなりません。これは、管理者のミス（例えば2つの異なるサーバの出力を同一のアーカイブ用ディレクトリに送信してしまうなど）といった場合からアーカイブ状況の整合性を保護するための安全策として重要です。

実際に既存のファイルを上書きしないこと、かつ、その場合に非ゼロのステータスを返すことを確認するために使用するアーカイブ用コマンドを試験することを勧めます。上のUnix用のコマンド例では、別途testという段階を含めることでこれを確認しています。いくつかのUnixプラットフォームではcpコマンドには-i 引数を使うことで煩雑な出力を少なくし使うことができますが、正しい終了コードが返ることを確認せずに使用するべきではありません。(具体的にはGNUのcpコマンドは-i オプションを使い、ターゲットファイルがすでに存在している場合、ゼロのステータスを返します。これは期待していない動作です。)

アーカイブ設定を設計する時には、操作者の介入が必要であったり、アーカイブ場所の容量不足の理由でアーカイブ用コマンドが繰り返し失敗した時にどうなるかを考慮してください。例えば、これはオートチェンジャ機能のないテープに書き出している場合に発生する可能性があります。テープが一杯になった場合、テープを交換するまでアーカイブを行うことができなくなります。こうした状況を相応の早さで解消できるよう、適切に操作者に対しエラーや要求を確実に連絡できるようにしなければなりません。この状況が解消するまで、WALセグメントファイルはpg_wal/ディレクトリ内に格納され続けます。（pg_wal/を含むファイルシステムがいっぱいになると、PostgreSQLはパニック停止します。コミットされたトランザクションは失われませんが、データベースはいくらかの容量を解放するまでオフラインのままです。）

サーバのWALデータの生成に要する平均速度に追いついている限り、アーカイブ用コマンドの処理速度は重要ではありません。アーカイブプロセスが多少遅れたとしても通常の操作は続けられます。アーカイブ処理がかなり遅れると、災害時に損失するデータの量が増加することになります。また、これはpg_wal/ディレクトリ内に多くのアーカイブ処理待ちのセグメントファイルが格納され、ディスク容量が不足する状況になる可能性があることを意味します。アーカイブ処理が確実に意図通りに動作しているかを監視することを推奨します。

アーカイブ用コマンドを作成する時、アーカイブされるファイル名は最長64文字までで、ASCII文字と数字とドットのどんな組合せを使用しても構いません。元の相対パス（%p）を保存する必要はありませんが、ファイル名（%f）を保存する必要はあります。

WALアーカイブによってPostgreSQLデータベースでなされた変更は全てリストアすることができますが、設定ファイルはSQL操作ではなく手作業で変更されますので、設定ファイル（postgresql.conf、pg_hba.conf、およびpg_ident.conf）になされた変更までリストアしないことに注意してください。通常のファ

イルシステムバックアップ手続きでバックアップされる場所に設定ファイルを保持したい場合があります。設定ファイルの設置場所を変更するには19.2を参照してください。

アーカイブコマンドは完全なWALセグメントに対してのみ呼び出されます。このため、サーバが少ししかWAL流量がない(処理を行わないなぎの期間がある)場合、トランザクションの完了とアーカイブ格納領域への安全な記録との間に長期にわたる遅延があることになります。古い未アーカイブのデータをどうするかについて制限を付けるために、archive_timeoutを設定して、強制的にサーバを新しいWALセグメントにある程度の間隔で切り替えるようにすることができます。強制切り替えにより早期にアーカイブされたアーカイブ済みファイルは完全に完了したファイルと同じ大きさを持つことに注意してください。そのため、非常に小さなarchive_timeoutを使用することはお勧めしません。格納領域を膨張させてしまいます。通常ならば分単位のarchive_timeout設定が合理的です。

終わったばかりのトランザクションをできるだけ早くアーカイブさせたい場合、pg_switch_walを使用して手作業でセグメント切り替えを強制することができます。この他のWAL管理に関連した関数を表 9.85に列挙します。

wal_levelがminimalの場合、14.4.7に書かれているように、いくつかのSQLコマンドはWALロギングを回避するため最適化されます。アーカイビングもしくはストリーミングレプリケーションがこれら構文の1つを実行中に作動させられると、アーカイブ復旧のための十分な情報をWALが含まなくなります。(クラッシュ復旧は影響を受けません。) このことにより、wal_levelはサーバの起動時のみ変更可能です。とは言っても、archive_commandは構成ファイルを再読み込みすることで変更できます。一時的にアーカイビングを停止したい場合、1つの方法はarchive_commandを空文字列('')に設定することです。このようにすると、動作するarchive_commandが再構築されるまでWALファイルはpg_wal/に蓄積します。

25.3.2. ベースバックアップの作成

ベースバックアップを取得する最も簡単な方法はpg_basebackup を実行する方法です。通常のファイルやTAR形式のファイルとしてベースバックアップを取得することができます。もし、pg_basebackupより柔軟性が求められる場合は、低レベルなAPIを使ってバックアップを作成することもできます(詳細は 25.3.3を参照)。

ベースバックアップを取得するための時間を考慮する必要はありません。しかし、普段、full_page_writesを無効にして運用している場合、バックアップ取得中は強制的にfull_page_writesが有効になるため、パフォーマンスが落ちていると感じる可能性があります。

バックアップを使用するためには、ファイルシステムのバックアップ取得中、および、その後に生成されるWALセグメントファイル全てが保存されている必要があります。この目的のために、ベースバックアップの過程で即座にWALアーカイブ領域に*バックアップ履歴ファイル*が作成されます。このファイルにはファイルシステムのバックアップに最初に必要とされるWALセグメントの名前が付けられます。例えば、最初のWALファイルが 000000010000123400000055CDである場合、バックアップ履歴ファイルは000000010000123400000055CD.007C9330.backupというように名付けられます。(ファイル名の2番目のパートはWALファイルの厳密な位置が記載されます。通常は無視することができます。) 一旦、安全にファイルシステムのバックアップとそのバックアップ中に使用されたWALセグメントファイル(バックアップ履歴ファイルから特定できます)を取得すると、それより数値の小さな全てのWALアーカイブセグメントはファイルシステムの復旧には必要が無く、削除することができます。しかし、データを確実に復旧させるためには数世代のバックアップセットを保持することを考慮すべきです。

バックアップ履歴ファイルは、ほんの小さなテキストファイルです。これにはpg_basebackupで与えたラベル文字列の他、バックアップの開始、終了時間およびバックアップのWALセグメントが含まれます。このラ

ベルをバックアップを構成するために使うことで、アーカイブ履歴ファイルはどのバックアップをリストアするべきか間違いなく判断することができます。

最後のベースバックアップ以降のWALアーカイブを保持し続ける必要があるため、通常、ベースバックアップを取得すべき期間は、WALアーカイブを保持するためにどのくらいのストレージを拡張できるかによって決定されます。また、復旧が必要になった場合に、どのくらいの時間を復旧に使うと覚悟するのかも考慮すべきです。— システムは全てのWALセグメントを適用する必要があるため、もし、最後のベースバックアップを取得してから長い時間が経過している場合、適用に時間を要する可能性があります。

25.3.3. 低レベルAPIを使用したベースバックアップの作成

低レベルのAPIを使ったベースバックアップを取得するにはpg_basebackup を使う方法に加えて数ステップが必要ですが、比較的簡単です。これらのステップは順番に実行することが重要で、次のステップに進む前にこれらのステップが成功していることを確認する必要があります。

低レベルのベースバックアップは非排他的または排他的な手法で作成することができます。非排他的な手法が推奨され、排他的な手法は推奨されず、将来的に削除されます。

25.3.3.1. 非排他的な低レベルバックアップの作成

非排他的な低レベルバックアップは、実行中の（同じバックアップAPIを使用して起動したものと、pg_basebackupを使用して起動したものいずれも）他の同時バックアップを許可するものです。

1. WALアーカイブが有効であり、正常に動作することを確認してください。

2. pg_start_backupを実行する権限のあるユーザ（スーパーユーザまたは関数のEXECUTE権限を付与されたユーザ）としてサーバ（どのデータベースでも構いません）に接続し、以下のコマンドを実行してください。

   ```
   SELECT pg_start_backup('label', false, false);
   ```

   ここでlabelは、バックアップ操作を一意に識別するために使用する任意の文字列です。pg_start_backupを呼び出す接続は、バックアップの完了まで維持される必要があります。さもなくばバックアップは自動的に中止されます。

   デフォルトで、pg_start_backupは終了までに長い時間がかかる場合があります。その理由はあるチェックポイントを実行し、そのチェックポイントに必要なI/Oはかなりの時間にわたって広がるためです。そして、デフォルトでは設定したチェックポイント間隔の半分です（設定パラメータについてはcheckpoint_completion_targetを参照してください）。通常これは、問い合わせ処理における影響を極小化するので望ましいことです。すぐにバックアップを開始したい場合は、第2パラメータをtrueにすると、使えるだけのI/Oを使用して即座にチェックポイントを発行します。

   第3パラメータのfalseは、pg_start_backupが非排他ベースバックアップを開始すること指示します。

3. （pg_dumpやpg_dumpallではなく）tarやcpioなどの使い慣れた任意のファイルシステムバックアップツールを使用して、バックアップを実行してください。この作業時に、データベースの通常の操作を停止することは不要ですし、望ましい方法でもありません。このバックアップの実行中に考慮すべき点は25.3.3.3を参照してください。

4. 以前と同じ接続の中で、以下のコマンドを実行します。

```
SELECT * FROM pg_stop_backup(false, true);
```

これはバックアップモードを終了し、次のWALセグメントへの自動切換えを行います。この切換えの理由は、バックアップ期間中に書き出された最後のWALファイルがアーカイブできるよう準備することです。

pg_stop_backupは3つの値を含んだ1行を返します。2番目の値は、バックアップのルートディレクトリ内のbackup_labelという名称のファイルを作成の上、値を書き込む必要があります。3番目の値は、空でない限りはtablespace_mapという名称のファイルを作成の上、値を書き込む必要があります。これらのファイルは、バックアップの動作にきわめて重要であり、返り値の内容から変更なしに書き込む必要があります。

5. バックアップ中に使用されたWALセグメントファイルがアーカイブされれば完了です。pg_stop_backupの返り値の1番目の値で識別されるファイルは、バックアップファイル一式を完結させるのに必要となる最終セグメントです。プライマリでは、archive_modeが有効で、かつwait_for_archiveパラメータがtrueであれば、pg_stop_backup は最終セグメントがアーカイブされるまで戻りません。スタンバイでは、pg_stop_backupがアーカイブ完了を待つためには、 archive_modeはalwaysでなければなりません。すでにarchive_commandを設定していますので、これらのファイルのアーカイブ操作は自動的に発生します。ほとんどの場合、これは瞬時に行われます。しかし、バックアップの完了を確認できるよう、アーカイブシステムを監視し、遅延が無いことの確認をお勧めします。アーカイブコマンドの失敗によりアーカイブ処理が遅れてしまったとしても、アーカイブが成功し、そしてバックアップが完了するまで再試行を繰り返すようになっています。pg_stop_backup実行においての時間期限を設けたい場合、適切なstatement_timeoutの値を設定できますが、この設定値によってpg_stop_backupが中断したときにバックアップが正当ではない可能性があるということを肝に銘じてください。

バックアップに必要なすべてのWALセグメントファイルのアーカイブが成功したことを、バックアップ作業の中で監視して確認するのであれば、wait_for_archiveパラメータ(デフォルトでtrueです)をfalseに設定し、バックアップレコードがWALに書き込まれたら即座にpg_stop_backupが戻るようにすることができます。デフォルトでは、pg_stop_backupはすべてのWALがアーカイブされるのを待つので、少し時間がかかることがあります。このオプションは慎重に使わなければなりません。WALのアーカイブを適切に監視していない場合、バックアップにはすべてのWALファイルが含まれず、不完全かもしれません。そうなると、リストアできません。

25.3.3.2. 排他的低レベルバックアップの作成

> **注記**
>
> 排他的バックアップ方式は非推奨であり避けるべきです。これはPostgreSQL 9.6より前では唯一利用可能な低レベル手法でしたが、今日ではすべてのユーザが自身のスクリプトを非排他的バックアップを使用するようにアップグレードすることが推奨されています。

排他的バックアップの手順は、ほぼ非排他的バックアップのものと同様ですが、いくつかのキーとなる手順に違いがあります。このバックアップ方式は、プライマリからしか取得できず、バックアップの複数同時実行

を許可しません。さらに、後述するようにバックアップラベルファイルを作るため、マスターサーバのクラッシュ後の自動再起動をブロックする可能性があります。一方でバックアップラベルファイルのバックアップやスタンバイからの誤った削除はよくあるミスで、これは深刻なデータ破壊をもたらすおそれがあります。この方式を使う必要があるのであれば、以下の手順を用いてください。

1. WALアーカイブが有効であり、正常に動作することを確認してください。

2. pg_start_backupを実行する権限のあるユーザ(スーパーユーザまたは関数のEXECUTE権限を付与されたユーザ)としてサーバ(どのデータベースでも構いません)に接続し、以下のコマンドを実行してください。

```
SELECT pg_start_backup('label');
```

   ここでlabelは、バックアップ操作を一意に識別するために使用する任意の文字列です。pg_start_backupは、開始時刻やラベル文字列などのバックアップ情報を持つbackup_labelという名前のバックアップラベルファイルを、クラスタディレクトリ内に作成します。この関数はtablespace_mapという名前のテーブル空間マップファイルもクラスタディレクトリ内に作ります。テーブル空間マップファイルには、もしそのようなリンクが1つ以上存在すればpg_tblspc/内のテーブル空間シンボリックリンクに関する情報が入っています。どちらのファイルもバックアップの完全性保持のために重要な意味を持ちますので、バックアップから必ずリストアする必要があるでしょう。

   デフォルトで、pg_start_backupは終了までに長い時間がかかる場合があります。その理由はあるチェックポイントを実行し、そして、デフォルトでは設定したチェックポイント間隔の半分である、そのチェックポイントに必要なI/Oがかなりの時間にわたって広がるためです(設定パラメータについてはcheckpoint_completion_targetを参照してください)。通常これは、問い合わせ処理における影響を極小化するので望ましいことです。バックアップをなるべく早く行いたいのであれば、以下を使用します。

```
SELECT pg_start_backup('label', true);
```

   これはチェックポイントをできる限り早く行うよう強制します。

3. (pg_dumpやpg_dumpallではなく)tarやcpioなどの使い慣れた任意のファイルシステムバックアップツールを使用して、バックアップを実行してください。この作業時に、データベースの通常の操作を停止することは不要ですし、望ましい方法でもありません。このバックアップの実行中に考慮すべき点は25.3.3.3を参照してください。

   上記のように、バックアップ中にサーバがクラッシュした場合、backup_labelファイルがPGDATAディレクトリから手動で削除されない限り、再起動できないかもしれません。破損をもたらすため、バックアップをリストアするときにbackup_labelファイルを決して削除しないことは重要である点に注意してください。このファイルをいつ削除するのが適切かに関する混乱は、この方式を使ったときのデータ破損のよくある原因です。確実に、本ファイルは既存マスタでのみ削除して、スタンバイの構築やバックアップのリストアでは決して削除しないでください。たとえ次に新マスタに昇格させるつもりのスタンバイを構築している場合であってもです。

4. 再度、pg_stop_backupを実行する権限のあるユーザ(スーパーユーザまたは関数のEXECUTE権限を付与されたユーザ)としてデータベースに接続し、以下のコマンドを実行してください。

```
SELECT pg_stop_backup();
```

この関数はバックアップモードを終了し、次のWALセグメントへの自動切換えを行います。この切換えの理由は、バックアップ期間中に書き出された最後のWALファイルがアーカイブできるよう準備することです。

5. バックアップ中で使用されたWALセグメントファイルがアーカイブされれば完了です。pg_stop_backupの結果で識別されるファイルは、バックアップファイル一式を完結させるのに必要となる最終セグメントです。archive_modeが有効であればpg_stop_backup は最終セグメントがアーカイブされるまで戻りません。すでにarchive_commandを設定していますので、これらのファイルのアーカイブ操作は自動的に発生します。ほとんどの場合、これは瞬時に行われます。しかし、バックアップの完了を確認できるよう、アーカイブシステムを監視し、遅延が無いことの確認をお勧めします。アーカイブコマンドの失敗によりアーカイブ処理が遅れてしまったとしても、アーカイブが成功し、そしてバックアップが完了するまで再試行を繰り返すようになっています。

   排他的バックアップモードを使ったときは、バックアップ終了時に確実にpg_stop_backupを正常に完了することが絶対的に必要です。バックアップ自体が、例えばディスク領域不足で、失敗した場合であっても、pg_stop_backupを実行しそこなうと、サーバが無期限にバックアップモードのままになってしまい、将来のバックアップ失敗とbackup_labelがある間の再起動失敗のリスクの増大を引き起こします。

25.3.3.3. データディレクトリのバックアップ

ファイルシステムのバックアップツール中には複写している途中でファイルが変更されると警告もしくはエラーを報告するものがあります。稼働しているデータベースのベースバックアップを取っている場合には、この状況は正常でエラーではありません。しかし、この種の警告と本当のエラーとを区別できるか確認が必要です。例えば、rsyncのバージョンによっては「消滅したソースファイル」に対して別の終了コードを返し、そしてこの終了コードをエラーではないと受け付けるドライバスクリプトを記述することができます。同時にGNU tarのバージョンによっては、tarがそれを複写していた途中でファイルが切り詰められると、致命的エラーと識別できないエラーコードを返します。ありがたいことに、GNU tarのバージョン1.16もしくはそれ以降では、 バックアップ中にファイルが変更されると1で、それ以外のエラーの時は2でプログラムから抜けます。GNU の tarで1.23以降のバージョンを使用しているのであれば、--warning=no-file-changed --warning=no-file-removedオプションをつけることで関連する警告メッセージを隠すオプションを使用することができます。

バックアップに、データベースクラスタディレクトリ(例えば/usr/local/pgsql/data)以下にある全てのファイルが含まれていることを確認してください。このディレクトリ以下に存在しないテーブル空間を使用している場合、注意して、同様にそれらを含めてください (そして、バックアップがリンクとしてシンボリックリンクをアーカイブしていることを確認してください。さもないとリストアはテーブル空間を壊してしまいます)。

しかし、クラスタのpg_wal/サブディレクトリにあるファイルをバックアップから省いてください。このちょっとした調整は、リストア処理中の失敗の危険性を低減できますので、行う価値があります。pg_wal/がクラスタディレクトリ外のどこかを指し示すシンボリックリンクの場合は調整が簡単です。これは性能上の理由でよく使用される設定です。また、いずれこのバックアップを使うpostmasterではなく、今起動しているpostmasterの情報を記録しているpostmaster.pidとpostmaster.optsも除外できます。(これらのファイルはpg_ctlを誤作動させる可能性があります。)

マスター上に存在するレプリケーションスロットがバックアップに含まれないようにするために、クラスタの中のpg_replslot/ディレクトリをバックアップから除くのもしばしば良い考えです。もし、スタンバイを作成す

るためのバックアップを続けて使用すると、スタンバイのWALファイルの保持を無制限に保留する結果になり、ホットスタンバイからのフィードバックを有効にしている場合、マスターのWALを膨張させます。これは、これらのレプリケーションスロットを使っているクライアントはまだ、スタンバイではなく、マスターのスロットを接続し続け、更新しているからです。バックアップが新しいマスターを作成するためだけに作成されたとしても、レプリケーションスロットをコピーすることは特に有益であるとは考えられません。このようにバックアップにレプリケーションスロットを含むことは、新しいマスターがオンラインになったときにはスロットの内容が期限切れしており、有害である可能性があります。

ディレクトリpg_dynshmem/、pg_notify/、pg_serial/、pg_snapshots/、pg_stat_tmp/、pg_subtrans/の中身はバックアップから除外できます。(ただし、ディレクトリ自体は除外できません。) というのも、postmaster起動時に初期化されるからです。stats_temp_directoryが設定されていて、それがデータディレクトリの下にあるのであれば、そのディレクトリの中身も除外できます。

pgsql_tmpで始まるすべてのファイルとディレクトリはバックアップから除外できます。これらのファイルはpostmasterの起動時に削除されますし、ディレクトリも必要なら再作成されます。

pg_internal.initという名前のファイルが見つかった場合、それはバックアップから省くことができます。このファイルはリレーションキャッシュデータを含んでおり、常にリカバリの際に再構築されます。

バックアップラベルファイルには、pg_start_backupに付与したラベル文字列とpg_start_backupが実行された時刻、最初のWALファイルの名前が含まれます。したがって、当惑した時にバックアップファイルの中身を検索し、そのダンプファイルがどのバックアップセッションに由来したものかを確認することができます。テーブル空間マップファイルにはディレクトリpg_tblspc/に存在するシンボリックリンク名と各シンボリックリンクのフルパスが含まれています。このファイルはあなたのためだけの情報ではありません。その存在と内容はシステムのリカバリプロセスが適切に動作するために非常に重要です。

サーバが停止している時にバックアップを作成することも可能です。この場合、わかりきったことですが、pg_start_backupやpg_stop_backupを使用することができません。そのため、どのバックアップが、どのWALファイルと関連し、どこまで戻せばよいかを独自の方法で残さなければなりません。通常は、上述の継続的アーカイブ手順に従う方をお勧めします。

25.3.4. 継続的アーカイブによるバックアップを使用した復旧

さて、最悪の事態が発生し、バックアップから復旧する必要が出てきたものとします。以下にその手順を説明します。

1. もし稼動しているのであればサーバを停止してください。

2. もし容量があるのであれば、後で必要になる場合に備えてクラスタデータディレクトリ全体とテーブル空間を全て一時的な場所にコピーしてください。この予防措置は、既存のデータベースを2つ分保持できるだけの空き領域を必要とします。十分な領域がない場合でも、少なくともクラスタのpg_walサブディレクトリの内容は保存すべきです。ここには、システムが停止する前にアーカイブされなかったログファイルが含まれているかも知れないからです。

3. クラスタデータディレクトリ以下、および、使用中のテーブル空間の最上位ディレクトリ以下にある既存の全てのファイルとサブディレクトリを削除してください。

4. ファイルシステムバックアップからデータベースファイルをリストアします。ファイルが正しい所有権（rootではなくデータベースシステムユーザです！）でリストアされていることを確認してください。テーブル空間を使用している場合は、pg_tblspc/内のシンボリックリンクが正しくリストアされていることを検証する必要があります。

5. pg_wal/内にあるファイルをすべて削除してください。これらはファイルシステムバックアップから生成されたものであり、おそらく現在のものより古く使用できないものです。pg_wal/をまったくアーカイブしていなければ、適切な権限で再作成してください。以前シンボリックリンクとして設定していたのであれば、そのように確実に再構築するように注意してください。

6. 手順2で退避させた未アーカイブのWALセグメントファイルがあるのであれば、pg_wal/にコピーしてください。（問題が発生し、初めからやり直さなければならない場合に未変更のファイルが残るように、移動させるのではなくコピーすることが最善です。）

7. postgresql.conf（19.5.4を参照してください）に復旧の設定を記述し、クラスタデータディレクトリにrecovery.signalファイルを作成します。また、一時的にpg_hba.confを変更し、復旧の成功を確認できるまで一般ユーザが接続できないようにする必要があるかもしれません。

8. サーバを起動してください。サーバは復旧モードに入り、必要なアーカイブ済みWALファイル群の読み込みを行います。外部的なエラーにより復旧が中断したら、サーバを単に再起動させて、復旧処理を継続してください。復旧処理が完了したら、（誤って後で復旧モードに再度入らないように）サーバはrecovery.signalを削除します。その後通常のデータベース操作を開始します。

9. データベースの内容を検査し、希望する状態まで復旧できていることを確認してください。復旧できなかった場合は手順1に戻ってください。全て問題なければ、ユーザが接続できるようにpg_hba.confを正常状態に戻してください。

ここで重要となるのは、どのように復旧させたいのかやどこまで復旧させたいかを記述する復旧設定を設定することです。絶対に指定しなければならないことは、アーカイブ済みWALファイルセグメントをどのように戻すかをPostgreSQLに通知するrestore_commandです。archive_command同様、これはシェルコマンド文字列です。ここには、対象のログファイルの名前で置換される%fやログファイルのコピー先を示すパスで置換される%pを含めることができます。（パス名は現在の作業用ディレクトリ、つまり、クラスタのデータディレクトリから見た相対パスです。）コマンド内に%文字自体を埋め込む必要があれば%%と記載してください。最も簡単でよく使われるコマンドは以下のようなものです。

```
restore_command = 'cp /mnt/server/archivedir/%f %p'
```

これは事前にアーカイブされたWALセグメントを/mnt/server/archivedirディレクトリからコピーします。当然ながら、もっと複雑なものを使用することができます。例えば、操作者に適切なテープをマウントさせることを要求するようなシェルスクリプトでさえ可能です。

このコマンドが失敗した時に非ゼロの終了ステータスを返すことが重要です。このコマンドは、アーカイブに存在しないファイルを要求するかもしれませんが、その場合でも非ゼロを返さなければなりません。これはエラー状態ではありません。例外は、コマンドがシグナルによって中断された場合(データベースの停止に使用されるSIGTERM以外)か、シェルによるエラー(コマンドが見つかりませんなど)で復旧が中断され、サーバが起動しない場合です。

要求されるファイルはWALセグメントファイルだけではありません。.historyが付いているファイルが要求されることも想定しなければなりません。同時に、%pパスのファイル名部分は%fと異なることに注意してください。これらが相互に置き換え可能であるとは考えないでください。

アーカイブ場所で見つけられなかったWALセグメントはpg_wal/から検索されます。これにより、最近の未アーカイブのセグメントを使用することができます。しかし、アーカイブ場所から利用できるセグメントはpg_wal/内のファイルよりも優先的に使用されます。

通常は利用可能な全てのWALセグメントを使用して復旧処理が行われます。その結果、データベースを現時点まで(もしくは、利用可能なWALセグメントで得られる限り現在に近い時点まで)リストアします。従って、通常の復旧は「file not found」メッセージで終了します。エラーメッセージの正確な文言はrestore_commandの選択によります。また、復旧の開始時点で00000001.historyのようなファイル名のエラーメッセージが出ることがあります。これも単純な復旧作業では不具合を意味するものでなく正常です。論議については25.3.5を参照してください。

もし以前のある時点まで復旧させたい場合(例えば、経験不足のデータベース管理者が主トランザクションテーブルを消去した直前)、要求する停止時点を指定するだけです。停止時点は、「recovery target」として既知の停止時点で指定することも、日付と時刻で指定することも、リストアポイントか完了した特定のトランザクションIDで指定することもできます。本ドキュメントの執筆時点では使用するトランザクションIDの識別を補助するツールがありませんので、ほとんどの場合は日付と時刻による指定のみを使用することになるでしょう。

> **注記**
>
> 停止時点はバックアップの終了時刻、つまり、pg_stop_backupの最終時刻より後の時点でなければなりません。バックアップを行っている最中のある時点までベースバックアップを使用して復旧させることはできません(こうした時点まで復旧させるには、その前のベースバックアップまで戻って、そこからロールフォワードしてください)。

復旧時にWALデータの破損がわかると、復旧はその時点で止まり、サーバは起動しません。こうした場合、「復旧対象」に破損時点より前の時点を指定することで、復旧処理が正常に完了できるよう、復旧プロセスを初めからやり直すことができます。システムクラッシュなど外的理由により復旧処理が失敗した場合やWALアーカイブがアクセスできなくなった場合、復旧処理を単に再起動させることができます。この場合は失敗した時点とほぼ同じところから再開します。復旧処理の再起動は、次のような通常操作時のチェックポイント処理とほぼ同様に動作します。サーバは定期的にすべての状態をディスクに強制し、再度スキャンする必要がない処理済みのWALデータを示すpg_controlファイルを更新します。

25.3.5. タイムライン

過去のある時点までデータベースを復旧できる機能は、タイムトラベルやパラレルユニバースといったSFの物語に類似した、多少の複雑性があります。例えば、データベースの元の履歴で、火曜日の夕方5:15PMに重要なテーブルを削除し、水曜日のお昼まで手違いに気が付かなかったとします。慌てずに、バックアップを取り出して、火曜日の夕方5:14PMの時点にリストアし、データベースを起動させます。データベース世界のこの履歴では、そのテーブルを削除していません。しかし、後になって、これは大した問題ではなかったことが分かり、元の履歴における水曜日に朝の何時かにまで戻したいと考えたと仮定しましょう。データベースは既に起動していますので、元に戻したい時点に至るWALセグメントファイルの一部は上書きされていて、戻すことはできないかもしれません。ですので、このことを避けるために、ポイントインタイムで復旧させた後に生成された一連のWAL記録と元のデータベースの履歴において生成されたWAL記録とを区別する必要があります。

こうした問題を扱うためにPostgreSQLには*タイムライン*という概念があります。アーカイブ復旧が完了したときはいつでも、その復旧後に生成されたWAL記録を識別するための新しいタイムラインが生成されます。タイムラインID番号はWALセグメントファイル名の一部です。ですので、新しいタイムラインはこれまでのタイムラインで生成されたWALデータを上書きしません。実際、多くの異なるタイムラインをアーカイブすることができます。不要な機能と考えるかもしれませんが、命綱になることがしばしばあります。どの時点まで復旧すればよいか確実でないといった状況を考えてみてください。その時は、過去の履歴からの分岐点として最善の時点を見つけるために、試行錯誤して何度もポイントインタイムの復旧を行う必要があるでしょう。タイムラインがないと、この手続きはすぐに管理不能な混乱を招いてしまいます。タイムラインを使用して、以前捨てたタイムライン分岐における状態を含む、過去の*任意*の状態に復旧させることができます。

新しいタイムラインが生成される度に、PostgreSQLは、どのタイムラインがいつどこから分岐したかを示す「タイムライン履歴」ファイルを作成します。この履歴ファイルは、複数のタイムラインを含むアーカイブ場所から復旧する時にシステムが正しいWALセグメントファイルを選択できるようにするために必要です。したがって、履歴ファイルは、WALセグメントファイル同様にWALアーカイブ領域にアーカイブされます。履歴ファイルは（巨大になるセグメントファイルとは異なり）単なる小さなテキストファイルですので、安価かつ適切に無期限で保管できます。必要ならば、履歴ファイルにコメントを追加し、この特定のタイムラインがどのように、なぜ生成されたかについて独自の注釈を付与することができます。特にこうしたコメントは、実験の結果いくつものタイムラインのもつれがある場合に有用です。

復旧処理のデフォルトは、ベースバックアップが取得された時点のタイムラインと同一のタイムラインに沿った復旧です。別の子タイムラインに沿って復旧させたい（つまり、復旧試行以降に生成されたある状態に戻りたい）場合は、recovery_target_timelineで対象のタイムラインIDを指定しなければなりません。ベースバックアップより前に分岐したタイムラインに沿って復旧することはできません。

25.3.6. ヒントと例

継続的アーカイブを構成するいくつかのヒントを以下にあげます。

25.3.6.1. スタンドアローンホットバックアップ

スタンドアローンホットバックアップを形成するためPostgreSQLのバックアップ基盤を使用することができます。これらのバックアップはポイントインタイムリカバリに使用することはできないのですが、pg_dumpによるダンプよりバックアップとリストアが概してより速く行われます。（同時にpg_dumpのダンプより大きくなるので、場合によっては速度による利点が打ち消されるかもしれません。）

ベースバックアップと同様に、スタンドアローンホットバックアップを作成する最も簡単な方法はpg_basebackupツールを使用する方法です。実行時に-Xオプションをつけることでバックアップに必要な全ての先行書き込みログを自動的にバックアップに含めることができ、リストアするときには特に特別な作業を行う必要がありません。

バックアップファイルをコピーするのにより柔軟性が必要な場合、スタンドアローンホットバックアップのために低レベルのプロセスを使うこともできます。低レベルのスタンドアローンホットバックアップを準備するために、次のことを確実に行ってください。wal_levelをreplica以上にセットし、archive_modeをonにセットし、*switch* ファイルが存在する時のみに実行されるarchive_commandをセットします。例：

```
archive_command = 'test ! -f /var/lib/pgsql/backup_in_progress || (test ! -f /var/lib/pgsql/
archive/%f && cp %p /var/lib/pgsql/archive/%f)'
```

このコマンドは/var/lib/pgsql/backup_in_progress が存在する時のみ実行され、存在しない時は単に0の終了コードを返します(PostgreSQLに必要の無いWALファイルを再利用することを許可します)。

この準備によって、バックアップは以下のようなスクリプトを使用して取得されます。

```
touch /var/lib/pgsql/backup_in_progress
psql -c "select pg_start_backup('hot_backup');"
tar -cf /var/lib/pgsql/backup.tar /var/lib/pgsql/data/
psql -c "select pg_stop_backup();"
rm /var/lib/pgsql/backup_in_progress
tar -rf /var/lib/pgsql/backup.tar /var/lib/pgsql/archive/
```

完全なWALファイルのアーカイブが行われるように、スイッチファイル、/var/lib/pgsql/backup_in_progressが最初に作成されます。バックアップの後、スイッチファイルは削除されます。その後、 ベースバックアップとすべての必要なWALファイルが共に同じtarファイルの一部になるよう、アーカイブされたWALファイルはバックアップに追加されます。バックアップスクリプトにエラー処理を加えておくことを忘れないでください。

25.3.6.2. 圧縮アーカイブログ

もし、アーカイブのストレージ容量に懸念がある場合、アーカイブファイルを圧縮するためにgzipを使用することもできます。

```
archive_command = 'gzip < %p > /var/lib/pgsql/archive/%f'
```

復旧時は gunzipを使う必要があります。

```
restore_command = 'gunzip < /mnt/server/archivedir/%f > %p'
```

25.3.6.3. archive_commandスクリプト

postgresql.confの記入事項が以下のように簡素となるため、多くの人がarchive_commandの定義にスクリプトの使用を選択します。

```
archive_command = 'local_backup_script.sh "%p" "%f"'
```

アーカイブ処理手順において単一ではなくそれ以上の数のコマンドを使用したい場合はいつでも、別のスクリプトファイルの使用が推奨されます。そうするとスクリプト内で全ての複雑性が管理されます。スクリプトはbashまたはperlのようなよくあるスクリプト言語で記載できます。

スクリプト内で解決される要件の例として以下があります。

- セキュアなオフサイトデータストレージへのデータのコピー
- 一回に全てではなく3時間毎に転送されるようにWALファイルのバッチ
- その他のバックアップとリカバリのソフトウェアとのインタフェース

- エラー報告を行う監視ソフトとのインタフェース

> **ヒント**
>
> archive_commandスクリプトを使うときはlogging_collectorを使えるようにすることが望ましい方法です。そのスクリプトがstderrに書き出したメッセージはすべて、データベースのサーバーログとして書かれます。このため複雑な設定でエラーが発生した時に、簡単に原因を突き止められます。

25.3.7. 警告

本ドキュメント作成時点では、継続的アーカイブ技術にいくつかの制限があります。将来のリリースでは修正されるはずです。

- もしもベースバックアップが行われている時、CREATE DATABASEコマンドが実行され、ベースバックアップが処理を実行している期間にCREATE DATABASEがコピーしているtemplateデータベースが変更されると、復旧処理はこれらの変更を作成されたデータベースにも同時に伝播させることは確実です。もちろん、これは望まれる事ではありません。この危険を回避するには、ベースバックアップ期間中にはすべてのtemplateデータベースを変更しないことが一番です。

- CREATE TABLESPACEコマンドはリテラルの絶対パス付でWALにログが記録され、したがって、同じ絶対パスでのテーブル空間作成の時に再生されます。これは、もしログが異なったマシン上で再生される場合には好ましくありません。ログ再生がたとえ同一のマシンであっても、新規のデータディレクトリであれば危険です。なぜなら、再生は元のテーブル空間の内容を上書きし続けるからです。この種の潜在的な振舞いを防ぐためには、テーブル空間を作成もしくは削除後に新規ベースバックアップを行うのが最良の手段です。

また、デフォルトのWALフォーマットは数多くのディスクページのスナップショットを含んでいるため、かなりかさばるものになってしまっていることに触れておくべきでしょう。これらのページスナップショットは、クラッシュから回復のために設計されています。それというのも、回復処理の際には不完全に書き込まれているディスクページを修復しなければならないことがあるからです。システムのハードウェアやソフトウェアによっては、不完全なディスクページの書き込みが起きてしまう危険性は無視してもよい程微小です。この場合full_page_writesパラメータを設定してページスナップショットを無効にすることで、アーカイブされたログの総容量を大幅に縮小できます（実際に設定を行う前に、第29章の注意事項と警告を読んでください）。ページスナップショットを無効にしても PITR処理の際にログが使用できなくなることはありません。将来の課題は、full_page_writesがたとえオンになっている場合であっても不要なページを取り除き、アーカイブ済みWALデータの圧縮を行うことでしょう。差し当たり管理者は、可能な限りチェックポイント間隔パラメータを大きくすることによって、WALに含まれるページスナップショットの数を削減することができます。

第26章 高可用性、負荷分散およびレプリケーション

データベースサーバは共同して稼動できます。その目的は、最初のサーバが故障したとき次のサーバへ速やかに引き継ぎができること（高可用性）および複数のコンピュータが同一のデータを処理できること（負荷分散）です。データベースサーバがシームレスに共同稼動できれば理想的です。静的なウェブページを提供するウェブサーバは、ウェブからの要求で生ずる負荷を複数のマシンに分散するだけで、簡単に結合できます。実際、読み取り専用のデータベースサーバの結合は、同じようにかなり容易です。しかし、大部分のデータベースサーバは、読み書きの混在した要求を受け取り、読み書き両用のサーバの結合はとても困難です。なぜなら、読み取り要求だけの場合、全サーバへのデータの配布は1回で終わります。しかし、書き込み後の読み取り要求に対して一貫性のある結果を返すためには、書き込み要求を受けたサーバだけでなく、他の全サーバにおいてもデータに書き込まなければなりません。

この同時性を持たせるという問題は、共同して稼動するサーバにおいて根本的に困難なものです。すべての使用状況において、単一の解法を用いて同時性の問題の影響を軽減できないため、複数の解法が存在します。各々の解法はこの問題に異なったやり方を適用し、固有の作業負荷に対する影響を最小化します。

幾つかの解法では、1つのサーバだけにデータの更新を許可することにより、同時性を持たせています。データの更新ができるサーバを、読み書きサーバ、マスタサーバまたはプライマリサーバといいます。マスタの変更を追跡するサーバを、スタンバイサーバまたはセカンダリサーバといいます。マスタサーバに昇格するまで接続できないスタンバイサーバをウォームスタンバイサーバといいます。接続を受理できて読み取り専用の問い合わせを処理できるスタンバイサーバをホットスタンバイサーバといいます。

いくつかの同期の解法が提供されています。すなわち、データに書き込むトランザクションでは、全サーバがコミットするまでトランザクションはコミットされません。これによって、フェイルオーバーにおいてデータの消失がないことが保証されます。また、どのサーバが問い合わせを受理したかに関係なく、全ての負荷分散サーバが一貫した結果を返すことが保証されます。それに対して非同期の解法では、コミット時刻と他サーバへの伝達時刻に時間差がありうるため、バックアップサーバへ交代する時にトランザクションが消失する可能性があります。また、負荷分散サーバにおいては、最新でない結果を応答する可能性があります。サーバ間の非同期の通信は、同期が非常に低速な場合に使用されます。

解法は粒度によって分類することもできます。ある解法ではデータベースサーバ全体だけを範囲として処理しますが、他の解法では各テーブルまたは各データベースを範囲として管理できます。

すべての選択において、作業効率を考えなければなりません。通常、作業効率と機能性は相反する関係にあります。例えば、遅いネットワークの場合、完全同期の解法を使えば作業効率は半分以下となりますが、非同期の解法を使えば作業効率への影響が最小となります。

本節では、フェイルオーバーとレプリケーションと負荷分散における種々の解法を説明します。

26.1. 様々な解法の比較

共有ディスクを用いたフェイルオーバー

データベースのコピーを 1つだけ保有すればよいため、共有ディスクを用いたフェイルオーバーは同期によるオーバーヘッドを回避できます。本解法では、複数のサーバが単一のディスクアレイを共有しま

す。主データベースサーバが故障したとき、まるでデータベースの破損から復旧したように、スタンバイサーバが元のデータベースを実装して稼動できます。これはデータの消失がない高速なフェイルオーバーを行うことができます。

ハードウェアを共有するという機能は、ネットワーク上の記憶装置では一般的です。ネットワークファイルシステムの利用も可能ですが、そのファイルシステムがPOSIX仕様を満たしているか注意してください。(18.2.2.1を見てください)。本解法には重大な制約があり、共有ディスクアレイが故障または破損したとき、プライマリサーバもスタンバイサーバも機能しなくなります。また、プライマリサーバが稼動している間は、スタンバイサーバが共有記憶装置にアクセスしてはなりません。

ファイルシステム(ブロックデバイス)レプリケーション

ハードウェア共有機能の改善の一つとしてファイルシステムのレプリケーションをあげることができます。それは、あるファイルシステムに対して行われたすべての変更を他のコンピュータに存在するファイルシステムにミラーリングします。制約はただ一つであり、スタンバイサーバがファイルシステムの一貫したコピーを自身の領域に持つようにミラーリングしなければなりません。具体的には、スタンバイサーバへの書き込みがマスタサーバへの書き込みと同じ順序でおこなわれなければなりません。LinuxにおけるDRBDは、ファイルシステムレプリケーションで広く受けいれられている手法です。

先行書き込みログシッピング

ウォームスタンバイおよびホットスタンバイサーバは、ログ先行書き込み(WAL)のレコードを解読して最新の状態を保持できます。プライマリサーバが故障したとき、スタンバイサーバがプライマリサーバのほぼすべてのデータを保存して、速やかに新しい主データベースを稼動できます。本解法は同期、非同期で行うことができ、データベース全体だけを範囲として処理できます。

スタンバイサーバは、ファイル単位のログシッピング(26.2参照)またはストリーミングレプリケーション(26.2.5参照)または両者の併用を使用して実装できます。ホットスタンバイの情報は 26.5 を参照してください。

論理レプリケーション

論理レプリケーションにより、データベースサーバが他のサーバに、データ更新のストリームを送ることができます。PostgreSQLの論理レプリケーションは、WALから論理的なデータ更新のストリームを構築します。論理レプリケーションでは、個々のテーブルの変更を複製することができます。論理レプリケーションにおいては、特定のサーバをマスターあるいはレプリカに割り当てる必要なしに、複数の方向にデータを流すことができます。論理レプリケーションの更なる情報については、第30章をご覧ください。ロジカルデコーディングインタフェース(第48章)を使って、サードパーティー拡張は同様の機能を提供できます。

トリガベースのマスタ・スタンバイレプリケーション

マスタとスタンバイによるレプリケーションでは、データ更新のすべての問い合わせをマスタサーバに送付します。マスタサーバは更新したデータを非同期でスタンバイサーバに送付します。マスタサーバが稼動している間、スタンバイサーバは読み取り問い合わせだけに応答します。スタンバイサーバはデータウェアハウスへの問い合わせに理想的です。

この種類のレプリケーションの一例はSlony-Iであり、テーブル単位の粒度を持ち、複数のスタンバイサーバが稼動できます。(バッチ処理によって)スタンバイサーバのデータを非同期で更新するため、フェイルオーバーにおけるデータ消失の可能性があります。

SQLに基づいたレプリケーションのミドルウェア

SQLに基づいたレプリケーションのミドルウェアでは、プログラムがすべてのSQL問い合わせを採取して、1つまたはすべてのサーバに送付します。なお、各々のサーバは独立して稼動します。読み書き問い合わせは、すべてのサーバがすべての変更を受け取るように全サーバに送付されなければなりません。しかし、読み取り専用の問い合わせはサーバ全体の読み取り負荷を分散させるために、1つのサーバだけに送付することができます。

問い合わせを修正しないで送付した場合、random()関数による乱数値とCURRENT_TIMESTAMP関数による現在時刻およびシーケンス値が、サーバごとに異なることがあります。その理由は、各サーバが独立して稼動しているため、および、SQL問い合わせの送付では実際に更新した行の識別値を取得できないためです。これが許容できない場合は、ミドルウェアかアプリケーションにおいて1つのサーバにこのような問い合わせを送付し、その結果を書き込み問い合わせで使用しなければなりません。その他の選択肢は従来のマスタとスタンバイによるレプリケーションのオプションを使用するものです。すなわち、データ更新の問い合わせをマスタサーバだけに送付し、ミドルウェアによるレプリケーションを使わずにマスタとスタンバイによるレプリケーションを介してスタンバイサーバに伝達します。トランザクションをコミットするか中断するかについても、全サーバが同一となるよう注意しなければなりません。これには2相コミット（PREPARE TRANSACTION および COMMIT PREPARED）を使用することになるでしょう。Pgpool-IIとContinuent Tungstenがこのレプリケーションの一例です。

非同期マルチマスタレプリケーション

ラップトップやリモートマシンのように、通常は接続されていない、あるいは遅い通信リンクで接続されているサーバ間において、データの一貫性を保持することは挑戦的な課題です。非同期マルチマスタレプリケーションの使用により、全サーバの独立した稼動、およびトランザクションの衝突を識別するための定期的な通信を実現します。トランザクションの衝突は、利用者および衝突回避法によって解決できるでしょう。Bucardoはこの種のレプリケーションの一例です。

同期マルチマスタレプリケーション

同期マルチマスタレプリケーションでは全てのサーバが書き込み要求を受理できます。受理したサーバは更新したデータを、トランザクションをコミットする前に、他の全サーバへ配布します。書き込み負荷が重いとき、ロックの掛かり過ぎやコミットの遅延による作業効率の低下の原因となりえます。読み取り要求はどのサーバにも送付できます。通信による負荷を減らすには、共有ディスクが実装されます。同期マルチマスタレプリケーションは、主に読み取り作業負荷の低減に最適ですが、全てのサーバが書き込み要求を受理できることも大きな利点です。その利点とは、マスタとスタンバイ間で作業負荷を分けなくてよいこと、および更新データが1つのサーバから他のサーバに送付されるため、出力が確定しないrandom()関数などによる問題が起こらないことです。

PostgreSQL では、この種類のレプリケーションを提供しません。しかし、PostgreSQL の 2相コミット（PREPARE TRANSACTIONおよびCOMMIT PREPARED）を使用すれば、アプリケーションのコードまたはミドルウェアにおいて本解法を実装できます。

表 26.1は上述した種々の解法の機能を要約したものです。

表26.1 高可用性、負荷分散およびレプリケーションの特徴

特徴	共有ディスク	ファイルシステムのレプリケーション	先行書き込みログシッピング	論理レプリケーション	トリガに基づいたレプリケーション	SQLに基づいたレプリケーションのミドルウェア	非同期マルチマスタレプリケーション	同期マルチマスタレプリケーション
一般的な例	NAS	DRBD	組み込みストリーミングレプリケーション	組み込み論理レプリケーション、pglogical	Londiste、Slony	pgpool-II	Bucardo	
通信方法	共有ディスク	ディスクブロック	WAL	ロジカルデコーディング	テーブル行	SQL	テーブル行	テーブル行および行ロック
特別なハードウェアが不要		○	○	○	○	○	○	○
複数のマスタサーバが可能				○		○	○	○
マスタサーバにオーバヘッドがない	○		○	○		○		
複数のスレーブサーバを待たない	○		同期が無効の場合	同期が無効の場合	○		○	
マスタの故障によるデータ損失がない	○	○	同期が有効の場合	同期が有効の場合		○		○
レプリカは読み取り専用問い合わせを受理可能			ホットスタンバイ使用時	○	○	○	○	○
テーブルごとの粒度				○	○		○	○
コンフリクトの回避が不要	○	○	○		○			○

上の分類に該当しない解法もあります。

データの分割

データの分割とは、同じテーブルのデータを複数部分に分けることです。各部分に書き込むことができるのは、1つのサーバだけです。例えば、データをロンドンとパリの営業所用に分割でき、サーバをロンドンとパリのどちらにも設置できた状態を考えます。問い合わせにロンドンとパリのデータが混在した場合、アプリケーションは両方のサーバに問い合わせることができます。または、マスタスタンバイレプリケーションを使用して、他の営業所のデータを読み取り専用コピーとして保持できます。

複数サーバによる問い合わせの並列実行

上述した多くの解法は、複数のサーバが複数の問い合わせを処理するものです。処理速度の向上のために、単一の問い合わせに複数のサーバを使用するものはありません。本解法は複数のサーバが単一の問い合わせを共同して実行するものです。その方法は、データをサーバ間で分割し、各サーバが部分的に問い合わせを実行し、各々の結果をプライマリサーバに送付し、プライマリサーバが合体して利用者に返送するものです。これはPL/Proxyツールセットを使用して実装できます。

また、PostgreSQLはオープンソースで、容易に拡張できるので、多くの企業がPostgreSQLをもとにして、独自のフェイルオーバー、レプリケーション、負荷分散機能を備えたクローズドソースの製品を開発していることに注意してください。これらについては、ここでは議論しません。

26.2. ログシッピングスタンバイサーバ

継続的なアーカイブ処理を使用して、プライマリサーバが失敗した場合に操作を引き継ぐ準備がなされた、1つ以上の*スタンバイサーバ*を持つ*高可用性*(HA)クラスタ構成を作成することができます。この機能は*ウォームスタンバイ*または*ログシッピング*として広く知られています。

プライマリサーバとスタンバイサーバは、この機能を提供するために共同して稼動しますが、サーバとサーバはゆるく結合しています。プライマリサーバは継続的アーカイブモードで動作し、各スタンバイサーバはプライマリからWALファイルを読み取る、継続的リカバリモードで動作します。この機能を可能にするために、データベースのテーブル変更は不要です。したがって、他のレプリケーションの解法に比べて、管理にかかるオーバーヘッドが減少します。この構成はプライマリサーバの性能への影響も相対的に減少させます。

あるデータベースサーバから他へ直接WALレコードを移動することは通常、ログシッピングと説明されます。PostgreSQLはファイルベースのログシッピングを実装します。つまりWALレコードはある時点で1つのファイル(WALセグメント)として送信されることを意味します。WALファイル(16MB)は隣り合うシステム、同じサイトの別システム、地球の裏側のシステムなど距離に関わらず、簡単かつ安価に送付することができます。この技法に必要な帯域幅はプライマリサーバのトランザクションの頻度に応じて変動します。レコードベースのログシッピングはより粒度を細かくしたもので、ネットワーク接続を介してWALの変更を増分的に流します(26.2.5参照)。

ログシッピングが非同期であることに注意しなければなりません。つまり、WALレコードはトランザクションがコミットした後に転送されます。結果として、プライマリサーバが災害などの致命的な失敗をうけた場合、送信されていないトランザクションが失われますので、データを損失する空白期間があります。ファイルベースのログシッピングにおけるデータ損失の空白期間量を`archive_timeout`パラメータを用いて制限することができます。これは数秒程度まで小さく設定することができます。しかし、低く設定するとファイル転送に必要な帯域幅が増大します。ストリーミングレプリケーション(26.2.5参照)により、データを損失する期間を非常に小さくすることができます。

リカバリ処理の性能は十分よく、一度実施されれば、スタンバイサーバが完全な状態から逸脱するのは一時的にしかすぎません。結果としてこれは、高可用性を提供するウォームスタンバイ構成と呼ばれます。保管されたベースバックアップからサーバをリストアし、ロールフォワードを行うことはおそらく長時間かかりますので、これは高可用性のための解法とはいえず、災害からのリカバリのための解法です。スタンバイサーバは読み取り専用の問い合わせに使用することもできます。この場合ホットスタンバイサーバと呼ばれます。詳細については26.5を参照してください。

26.2.1. 計画

プライマリサーバとスタンバイサーバを、少なくともデータベースサーバという見地でできる限り同じになるように作成することを通常勧めます。具体的には、テーブル空間に関連するパス名はそのまま渡されますので、テーブル空間機能を使用する場合には、プライマリとスタンバイサーバの両方でテーブル空間用のマウントパスを同じにしておかなければなりません。CREATE TABLESPACEをプライマリで実行する場合、そのコマンドを実行する前に必要な新しいマウントポイントをプライマリとすべてのスタンバイサーバで作成しなければならないことに注意してください。ハードウェアをまったく同じにする必要はありませんが、経験上アプリケーションとシステムの運用期間に渡って2つの同じシステムを管理する方が、異なる2つのシステムを管理するよりも簡単です。いずれにしてもハードウェアアーキテクチャは必ず同じでなければなりません。例えば32ビットシステムから64ビットシステムへのシッピングは動作しません。

一般的に、異なるメジャーリリースレベルのPostgreSQL間でログシッピングはできません。マイナーリリースの更新ではディスク書式を変更しないというのがPostgreSQLグローバル開発グループの方針ですので、プライマリサーバとスタンバイサーバとの間でマイナーリリースレベルの違いがあってもうまく動作するはずです。しかし、この場合、公的なサポートは提供されません。できる限りプライマリサーバとスタンバイサーバとで同じリリースレベルを使用してください。新しいマイナーリリースに更新する場合、もっとも安全な方針はスタンバイサーバを先に更新することです。新しいマイナーリリースは以前のマイナーリリースのWALファイルを読み込むことはできますが、逆はできないかもしれません。

26.2.2. スタンバイサーバの動作

スタンバイモードでは、サーバは継続的にマスタサーバから受け取ったWALを適用します。スタンバイサーバはWALアーカイブ(restore_command参照)から、または直接TCP接続(ストリーミングレプリケーション)を介してマスタサーバから、WALを読み取ることができます。またスタンバイサーバはスタンバイクラスタのpg_walディレクトリにあるすべてのWALをリストアしようと試みます。これはよくサーバの再起動後、スタンバイが再起動前にマスタから流れ込んだWALを再生する時に発生します。しかしまたファイルを再生する任意の時点で、手作業でpg_walにコピーすることもできます。

起動時、スタンバイサーバはrestore_commandを呼び出して、アーカイブ場所にある利用可能なすべてのWALをリストアすることから始めます。そこで利用可能なWALの終端に達し、restore_commandが失敗すると、pg_walディレクトリにある利用可能な任意のWALのリストアを試みます。ストリーミングレプリケーションが設定されている場合、これに失敗すると、スタンバイはプライマリサーバへの接続を試み、アーカイブまたはpg_wal内に存在した最終の有効レコードからWALのストリーミングを開始します。ストリーミングレプリケーションが未設定時にこれに失敗する場合、または、接続が後で切断される場合、スタンバイは最初に戻り、アーカイブからのファイルのリストアを繰り返し行います。このアーカイブ、pg_wal、ストリーミングレプリケーションからという再試行の繰り返しはサーバが停止する、あるいはトリガファイルによるフェイルオーバーが発行されるまで続きます。

pg_ctl promoteが実行された時、pg_promote()が呼び出された時、またはトリガファイル(promote_trigger_file)が存在する時、スタンバイモードは終了し、サーバは通常の動作に切り替わります。フェイルオーバーの前に、アーカイブまたはpg_wal内の即座に利用可能なWALをすべてリストアします。しかし、マスタへの接続を行おうとはしません。

26.2.3. スタンバイサーバのためのマスタの準備

25.3で説明したように、スタンバイからアクセス可能なアーカイブディレクトリに対してプライマリで継続的なアーカイブを設定してください。このアーカイブ場所はマスタが停止した時であってもスタンバイからアクセス可能でなければなりません。つまり、マスタサーバ上ではなく、スタンバイサーバ自身上に存在するか、または他の高信頼性サーバ上に存在しなければなりません。

ストリーミングレプリケーションを使用したい場合、スタンバイサーバ(複数可)からのレプリケーション接続を受け付けるようにプライマリサーバで認証を設定してください。つまり、ロールを作成し適切な項目を提供、あるいは、そのデータベースフィールドとしてreplicationを持つ項目をpg_hba.conf内に設定してください。また、プライマリサーバの設定ファイルにおいてmax_wal_sendersが十分大きな値に設定されていることを確認してください。レプリケーションスロットを使用している場合は、max_replication_slotsも十分に設定されているか確認してください。

25.3.2に記述したように、スタンバイサーバの再起動のために、ベースバックアップを取得してください。

26.2.4. スタンバイサーバの設定

スタンバイサーバを設定するためには、プライマリサーバから取得したベースバックアップをリストアしてください(25.3.4参照)。スタンバイのクラスタデータディレクトリ内にstandby.signalファイルを作成し、standby_modeを有効にしてください。WALアーカイブからファイルをコピーする簡単なコマンドをrestore_commandに設定してください。高可用性のために複数のスタンバイサーバを持たせようとしている場合、recovery_target_timelineをlatestに設定し(デフォルト)、スタンバイサーバが他のスタンバイにフェイルオーバーする時に発生するタイムラインの変更に従うようにします。

> **注記**
>
> ここで説明した組み込みのスタンバイモードといっしょにpg_standbyや類似ツールを使用しないでください。restore_commandはファイルが存在しない場合に即座に終了しなければなりません。サーバが必要に応じてそのコマンドを再度実行します。pg_standbyのようなツールを使用するためには26.4を参照してください。

ストリーミングレプリケーションを使用したい場合には、ホスト名(またはIPアドレス)とプライマリサーバとの接続に必要な追加情報を含む、libpq接続文字列でprimary_conninfoを記述してください。プライマリで認証用のパスワードが必要な場合はprimary_conninfoにそのパスワードも指定する必要があります。

スタンバイサーバを高可用性を目的に設定しているのであれば、スタンバイサーバはフェイルオーバーの後プライマリサーバとして動作しますので、プライマリサーバと同様にWALアーカイブ処理、接続、認証を設定してください。

WALアーカイブを使用している場合、archive_cleanup_commandパラメータを使用してスタンバイサーバで不要となったファイルを削除することで、その容量を最小化することができます。特にpg_archivecleanupユーティリティは、典型的な単一スタンバイ構成(pg_archivecleanup参照)におけるarchive_cleanup_commandと共に使用されるように設計されています。しかし、バックアップを目的にアーカイブを使用している場合には、スタンバイから必要とされなくなったファイルであっても、最新のベース

バックアップの時点からリカバリするために必要なファイルを保持しなければならないことに注意してください。

簡単な設定例を以下に示します。

```
primary_conninfo = 'host=192.168.1.50 port=5432 user=foo password=foopass options=''-c
 wal_sender_timeout=5000'''
restore_command = 'cp /path/to/archive/%f %p'
archive_cleanup_command = 'pg_archivecleanup /path/to/archive %r'
```

スタンバイサーバの台数に制限はありませんが、ストリーミングレプリケーションを使用するなら、プライマリサーバに同時に接続できるようにmax_wal_sendersを十分な数に設定してください。

26.2.5. ストリーミングレプリケーション

ストリーミングレプリケーションによりスタンバイサーバはファイルベースのログシッピングよりもより最近の状態を維持できるようになります。スタンバイは、WALレコードが生成された時にWALファイルがいっぱいになるまで待機せずにWALレコードをスタンバイに流し出すプライマリと接続します。

ストリーミングレプリケーションはデフォルトで非同期で、(26.2.8参照) この場合、プライマリでトランザクションがコミットされてから、その変更がスタンバイ側で参照可能になるまでの間にわずかな遅延がまだあります。しかし、この遅延はファイルベースのログシッピングよりも非常に小さなもので、負荷に追随できる程度の能力があるスタンバイであれば通常は1秒以下です。ストリーミングレプリケーションでは、データ損失期間を減らすためのarchive_timeoutを必要としません。

ファイルベースの継続的アーカイブのないストリーミングレプリケーションを使用している場合、スタンバイが受け取る前に古いWALセグメントを再利用するかもしれません。もし、そうなった場合はスタンバイは新しいベースバックアップから再作成しなければならなくなります。wal_keep_sizeを十分に大きくしたり、レプリケーションスロットにスタンバイを設定することでWALセグメントがすぐに再利用されることを防ぎ、これを防ぐことができます。WALアーカイブをスタンバイからアクセスできる位置に設定する場合は、スタンバイが常にWALセグメントを追随することができるため、これらの解決策は要求されません。

ストリーミングレプリケーションを使用するためには、26.2の説明のようにファイルベースのログシッピングを行うスタンバイサーバを設定してください。ファイルベースのログシッピングを行うスタンバイをストリーミングレプリケーションを行うスタンバイに切り替える手順は、primary_conninfo設定をプライマリサーバを指し示すように設定することです。スタンバイサーバがプライマリサーバ上のreplication疑似データベースに接続できる(26.2.5.1参照)ように、プライマリでlisten_addressesと認証オプション(pg_hba.conf参照)を設定してください。

キープアライブソケットオプションをサポートするシステムでは、tcp_keepalives_idle、tcp_keepalives_intervalおよびtcp_keepalives_countを設定することで、プライマリの接続切断の即時検知に有用です。

スタンバイサーバからの同時接続数の最大値を設定してください(詳細はmax_wal_sendersを参照)。

スタンバイが起動し、primary_conninfoが正しく設定されると、スタンバイはアーカイブ内で利用可能なWALファイルをすべて再生した後にプライマリと接続します。接続の確立に成功すると、スタンバイでwalreceiverが存在し、プライマリで対応するwalsenderが存在します。

26.2.5.1. 認証

信頼できるユーザのみがWALストリームを読み取ることができるように、レプリケーション用のアクセス権限を設定することは非常に重要です。WALから機密情報を取り出すことは簡単だからです。スタンバイサーバはプライマリに対してプライマリのREPLICATION権限を持つアカウントか、スーパーユーザとして認証されなければなりません。レプリケーションのためのREPLICATION権限とLOGIN権限を持つ専用のユーザを作成することをお勧めします。REPLICATION権限は非常に強力な権限なので、SUPERUSERのようにプライマリのデータを変更することを許可されていません。

レプリケーション用のクライアント認証はpg_hba.conf内でそのdatabaseフィールドにreplicationを指定したレコードで制御されます。例えば、スタンバイがIPアドレス192.168.1.100のホストで稼動し、レプリケーション用のアカウントの名前がfooである場合、管理者はプライマリ上のpg_hba.confに以下の行を追加することができます。

```
# 利用者 foo のホスト 192.168.1.100 からプライマリサーバへのレプリケーションスタンバイとしての接続を
# 利用者のパスワードが正しく入力されたならば許可する
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    replication     foo             192.168.1.100/32        md5
```

プライマリサーバのホスト名とポート番号、接続する利用者名およびパスワードは、primary_conninfoで指定します。パスワードはスタンバイサーバの~/.pgpassファイルでも設定できます(databaseフィールドのreplicationを指定します)。例えば、プライマリサーバが稼動するホストのIPアドレスが192.168.1.50でポート番号が5432であり、レプリケーションのアカウント名がfooであり、パスワードがfoopassである場合、管理者はスタンバイサーバのpostgresql.confファイルに次行を追加できます。

```
# プライマリサーバが 192.168.1.50 のホストの 5432ポートで稼動し
# 利用者名が foo でパスワードが foopass とする
primary_conninfo = 'host=192.168.1.50 port=5432 user=foo password=foopass'
```

26.2.5.2. 監視

ストリーミングレプリケーションの重要な健全性尺度は、プライマリサーバで生成されたがスタンバイサーバではまだ適用されていないWALレコードの量です。プライマリサーバの現在のWAL書き込み位置とスタンバイサーバの受理したWALの最終位置を比較すれば、この遅延を計算できます。これらの位置は、プライマリサーバではpg_current_wal_lsnを、スタンバイサーバではpg_last_wal_receive_lsnを使用すれば検索できます(詳細は表 9.85および表 9.86を参照)。スタンバイサーバの最終位置は、psコマンドを使用してWAL受信プロセスの状態としても表示できます(詳細は27.1を参照)。

pg_stat_replicationビューを介してWAL送信処理プロセスのリストを入手することができます。pg_current_wal_lsnとビューのsent_lsnフィールドとの違いが大きい場合、マスタサーバが高負荷状態であることを示している可能性があります。一方でスタンバイサーバ上

のsent_lsnとpg_last_wal_receive_lsnの値の差異は、ネットワーク遅延、またはスタンバイが高負荷状態であることを示す可能性があります。

ホットスタンバイ上では、WAL受信プロセスの状態は、pg_stat_wal_receiverビューを通じて入手することができます。pg_last_wal_replay_lsnとビューのreceived_lsnとの違いが大きい場合、WALのリプレイを上回る速さでWALが受信されていることを示しています。

26.2.6. レプリケーションスロット

レプリケーションスロットは、以下のことを保証する自動的な方法を提供します。全てのスタンバイがWALセグメントを受け取るまでは、マスターがWALセグメントを削除しないこと、また、スタンバイが接続していない際にも、リカバリの競合が発生する可能性がある行をマスターが削除しないこと、です。

レプリケーションスロットを使う代わりに、wal_keep_sizeを使う、あるいはarchive_commandを使用してセグメントをアーカイブに保存することによっても、古いWALセグメントの削除を防ぐことができます。しかし、これらの方法はしばしば要求される以上のWALセグメントを残すことになってしまうのに対し、レプリケーションスロットは必要と判断されたセグメントのみを残します。一方で、レプリケーションスロットはpg_walのための領域を埋め尽くす大量のWALセグメントを残してしまうかも知れません。これらの方法のメリットの一つはpg_walが要求する領域を制限できることです。現時点でレプリケーションスロットを使って同じことをする方法はありません。max_slot_wal_keep_sizeは、レプリケーションスロットによって残されるWALファイルの大きさを制限します。

同様に、hot_standby_feedbackとvacuum_defer_cleanup_ageは必要な行をvacuumが削除するのを防ぐ機能を提供しますが、前者はスタンバイが接続されていない間は行の保護を提供しませんし、後者は適切な保護を提供するために高い値を設定せざるを得ないことがしばしばあります。レプリケーションスロットはこのような短所を克服しています。

26.2.6.1. レプリケーションスロットへの問い合わせと操作

いずれのレプリケーションスロットにも小文字、数字、アンダースコアを含む名前があります。

レプリケーションスロットとその状態はpg_replication_slotsビューより確認できます。

レプリケーションスロットはストリーミングレプリケーションプロトコル(52.4参照)もしくはSQL関数(9.27.6参照)を使用し、作成や削除ができます。

26.2.6.2. 設定の例

以下のような方法でレプリケーションスロットを作成できます。

```
postgres=# SELECT * FROM pg_create_physical_replication_slot('node_a_slot');
  slot_name  | lsn
-------------+-----
 node_a_slot |

postgres=# SELECT slot_name, slot_type, active FROM pg_replication_slots;
```

```
  slot_name  | slot_type | active
-------------+-----------+--------
 node_a_slot | physical  | f
(1 row)
```

スタンバイのレプリケーションスロットを使用できるように設定するためには、primary_slot_nameをスタンバイ側で設定します。以下は単純な設定例です。:

```
primary_conninfo = 'host=192.168.1.50 port=5432 user=foo password=foopass'
primary_slot_name = 'node_a_slot'
```

26.2.7. カスケードレプリケーション

カスケードレプリケーションは、リレーのような振る舞い、つまり、スタンバイサーバから他のスタンバイにレプリケーション接続し、WALレコードを送信することができます。マスターサーバへ直接の接続を減らしたり、 サイト相互の帯域オーバヘッドを最小化するために使用することができます。

カスケードスタンバイとして知られているとおり、スタンバイは受け取り手としても送り手としても振る舞うことができます。よりマスターサーバに近いスタンバイサーバは上流サーバと呼ばれるのに対し、より遠いスタンバイサーバは下流サーバと呼ばれます。カスケードレプリケーションには下流サーバの数に制限は設定されていません。しかし、どのスタンバイサーバも最終的には1つのマスター/プライマリサーバに繋がる1つの上流サーバに接続します。

カスケードスタンバイはマスターから受け取ったWALレコードだけでなく、アーカイブからリストアしたWALレコードも送信します。このため、レプリケーション接続が上流サーバで切断しても、ストリーミングレプリケーションは下流サーバへ新しいWAL レコードがある限り継続します。

カスケードレプリケーションは現時点では非同期です。同期レプリケーション(参照26.2.8)の設定は現時点でカスケードレプリケーションへは影響を与えません。

ホットスタンバイがどの様に配置されていても、ホットスタンバイフィードバックは上流に伝播します。

上流スタンバイサーバが昇格し、新しいマスターサーバになった場合、 recovery_target_timelineが'latest'に設定されていれば、下流サーバは新マスターサーバからのストリーミングレプリケーションを継続します(デフォルトです)。

カスケードレプリケーションを使うためには、カスケードスタンバイをセットアップ、つまり、レプリケーション接続を許可してください。(max_wal_sendersとhot_standbyおよび、クライアント認証を設定してください) また、下流スタンバイがカスケードスタンバイに接続できるために、下流スタンバイではprimary_conninfoを設定する必要があります。

26.2.8. 同期レプリケーション

PostgreSQLのストリーミングレプリケーションはデフォルトで非同期です。プライマリサーバがクラッシュした場合、コミットされた一部のトランザクションがスタンバイサーバに複製されず、データ損失を引き起こす可能性があります。データ損失量はフェイルオーバー時点のレプリケーション遅延に比例します。

同期レプリケーションは、あるトランザクションでなされた変更はすべて、1つ以上の同期スタンバイサーバに転送されていることを確実にする機能を提供します。これはトランザクションコミットで提供される永続性の標準レベルを拡張します。この保護レベルはコンピュータ科学理論では、2-safeレプリケーション、そしてsynchronous_commitがremote_writeに設定されている場合にはgroup-1-safe (group-safeと1-safe) と呼ばれます。

同期レプリケーションを要求する時、書き込みトランザクションのコミットはそれぞれ、そのコミットがプライマリサーバおよびスタンバイサーバの両方で、ディスク上の書き込み先行ログに書き込まれたという確認を受けとるまで待機します。データ損失が起こる可能性は、プライマリサーバとスタンバイサーバが同時にクラッシュしてしまった場合のみです。これは非常に高い永続性を提供することができますが、それはシステム管理者が2つのサーバの設置と管理に関して注意を払っている場合のみです。確認のための待機は、サーバがクラッシュした場合でも変更が失われないということでユーザからの信頼性が大きくなりますが、同時に要求するトランザクションの応答時間も必ず大きくなります。最小待機時間はプライマリとスタンバイの間の往復遅延時間です。

読み取り専用のトランザクションおよびトランザクションのロールバックはスタンバイサーバからの応答を待つ必要はありません。副トランザクションのコミットもスタンバイサーバからの応答を待つことはなく、最上位レベルのコミットのみ待機します。データロード処理やインデックス構築など長時間実行される操作は、最終コミットメッセージまで待機しません。準備およびコミットの両方を含め、二相コミット動作はすべてコミット待機を必要とします。

同期スタンバイは、物理レプリケーションのスタンバイでも、論理レプリケーションのサブスクライバーのどちらでも構いません。また同期スタンバイは、適切なフィードバックメッセージを送信する方法を知っている、物理あるいは論理WALレプリケーションストリームの消費者であっても構いません。組み込みの物理あるいは論理レプリケーションシステムを別にすると、pg_receivewalとpg_recvlogical、それにサードパーティーのレプリケーションシステムとカスタムプログラムが該当します。対応する同期レプリケーションのサポートの詳細に関するドキュメントを参照してください。

26.2.8.1. 基本設定

一度、ストリーミングレプリケーションが設定されている場合、同期レプリケーションの設定には必要な追加設定は1つだけ：synchronous_standby_namesを空でない値に設定することです。またsynchronous_commitはonに設定されていなければなりませんが、これはデフォルト値ですので、通常は変更する必要はありません。(19.5.1 および19.6.2を参照してください) この設定によりスタンバイがそのコミットレコードを信頼できるストレージに書き込んだことが確認できるまで、各コミットが待たされるようになります。synchronous_commitは個々のユーザによって設定することができます。このため、トランザクション単位を基準とした永続性の保証を制御するために、設定ファイルの中で特定のユーザまたはデータベースについて設定することも、アプリケーションによって動的に設定することもできます。

コミットレコードがプライマリ上のディスクに書き出された後、WALレコードがスタンバイに送信されます。スタンバイにてwal_receiver_status_intervalがゼロに設定されていない限り、スタンバイは新しいWALデータの塊がディスクに書き出される度に応答メッセージを返します。synchronous_commitがremote_applyに設定されている場合には、コミットレコードが再生され、そのトランザクションが可視化されたときに応答メッセージを返します。スタンバイが、プライマリ上のsynchronous_standby_namesにしたがって、同期スタンバイとして選ばれた時は、コミットレコードの受領の確認のために待機しているトランザクションをいつ解放すべきかを決めるために、他の同期スタンバイとともにそれらスタンバイからの応答メッセージが考慮されます。これらのパラメータにより、管理者はどのス

タンバイサーバを同期スタンバイとすべきかを指定することができます。同期レプリケーションの設定は主にマスタでなされることに注意してください。指名されたスタンバイは直接マスターサーバに接続される必要があります。つまり、カスケードレプリケーションを使用している下流スタンバイサーバについて、マスターサーバは何も知りません。

synchronous_commitをremote_writeに設定することで、個々のコミットは、スタンバイサーバがコミットされたレコードを受け取り、オペレーティングシステムに書きだしたことが確認できるまで待ちますが、スタンバイ上のディスクに吐き出すまでは待ちません。これは、onと設定するより、提供される永続性は弱くなります。具体的には、スタンバイサーバはオペレーティングシステムがクラッシュした場合にデータを失う可能性がありますが、PostgreSQLがクラッシュした場合にはデータを失いません。しかし、実用的にはこの設定はトランザクションの応答時間を短くすることができるので有用です。データの損失は、プライマリサーバとスタンバイサーバが同時にクラッシュし、かつ、プライマリのデータベースが同時に壊れた場合にのみ発生します。

synchronous_commitをremote_applyに設定することで、現在の同期スタンバイがトランザクションを再生し、ユーザから見えるようにしたと報告するまでは各々のコミットは待たされます。単純なケースでは、因果一貫性を保つ負荷分散を可能にします。

高速シャットダウンが要求された場合、ユーザは待ち状態ではなくなります。しかし非同期レプリケーションを使用している時と同じく、送信中のWALレコードが現在接続しているスタンバイサーバに転送されるまで、サーバは完全に停止しません。

26.2.8.2. 複数の同期スタンバイ

同期レプリケーションは、一つ以上の同期スタンバイサーバをサポートします。同期と見なされるすべてのスタンバイサーバがデータの受領を確認するまで、トランザクションは待機します。トランザクションが応答を待たなければならない同期スタンバイの数は、synchronous_standby_namesで指定されます。また、このパラメータには、スタンバイの名前のリストと、リストされたものから同期スタンバイを選ぶ方法（FIRSTとANY）を指定します。

方法FIRSTは優先度に基づく同期レプリケーションを指定し、優先度に応じて選択された同期スタンバイにWALレコードがレプリケーションされるまで、トランザクションのコミットは待機します。リストの前の方に名前が書いてあるスタンバイにはより高い優先度が与えられ、同期とみなされます。リストの後ろの方に書いてあるスタンバイは、潜在的な同期スタンバイであることを示します。どんな理由であれ、現在のスタンバイのどれかの接続が切断されると、次に優先度が高いスタンバイがとって代わります。

優先度に基づく複数同期スタンバイのためのsynchronous_standby_namesの例を示します。

```
synchronous_standby_names = 'FIRST 2 (s1, s2, s3)'
```

この例では、もし4つのスタンバイサーバs1、s2、s3、s4が稼働中なら、s1とs2が同期スタンバイに選ばれます。それらの名前がスタンバイ名のリストの最初の方にあるからです。s3は潜在的な同期スタンバイで、s1あるいはs2が故障した時に同期スタンバイの役割を取って代わります。このリストに名前が載っていないので、s4は非同期スタンバイです。

方法ANYはクォーラムに基づく同期レプリケーションを指定し、少なくともリスト中で指定された数の同期スタンバイにWALレコードがレプリケーションされるまで、トランザクションのコミットを待たせます

クォーラムに基づく同期スタンバイのためのsynchronous_standby_namesの例を示します。

```
synchronous_standby_names = 'ANY 2 (s1, s2, s3)'
```

この例では、もし4つのスタンバイサーバs1、s2、s3、s4が稼働中なら、トランザクションのコミットは、s1、s2、s3のどれか二つのスタンバイから応答があるまで待たされます。このリストに名前が載っていないので、s4は非同期スタンバイです。

pg_stat_replicationビューを使って、スタンバイサーバの同期状態を見ることができます。

26.2.8.3. 性能に関する考慮

通常、同期レプリケーションは、アプリケーションが満足できる程度に実行されることを確実にするために、注意深くスタンバイサーバを計画し設置しなければなりません。待機のためにシステムリソースを使用することはありませんが、トランザクションロックは転送が確認されるまで継続して保持されます。結果として同期レプリケーションを注意せずに使用すると、応答時間が増加する、および競合がより高くなるため、データベースアプリケーションの性能は低下します。

PostgreSQLではアプリケーション開発者がレプリケーション経由で必要とする永続性レベルを指定することができます。これをシステム全体に対して指定することができますし、特定のユーザ、接続、個々のトランザクションに対してさえ指定することもできます。

例えばアプリケーションの作業量が、重要な顧客詳細の変更が10%、ユーザ間のチャットメッセージなど、あまり重要ではなく、失ったとしても業務をより簡単に戻すことができるようなデータの変更が90% という構成を考えてみます。

(プライマリ上で)アプリケーションレベルで指定する同期レプリケーションオプションを使用して、作業全体を低速化させることなく、最も重要な変更に対して同期レプリケーションを企てることができます。アプリケーションレベルのオプションは、高い性能が求められるアプリケーションで同期レプリケーションの利点が得られる、重要かつ現実的な手段です。

生成されるWALデータの割合よりネットワーク帯域幅が大きくなければならないことを考慮しなければなりません。

26.2.8.4. 高可用性に関する検討

synchronous_commitが、on、remote_apply、remote_writeのいずれかに設定されている場合、synchronous_standby_namesには、コミットされたトランザクションが応答を待つ同期スタンバイの数と名前を指定します。そのようなトランザクションのコミットは、同期スタンバイのどれかがクラッシュすると決して完了しないかもしれません。

高可用性のもっとも良い解決方法は、想定したのと同じ数の同期スタンバイを確実に確保することです。これは、synchronous_standby_namesを使って同期スタンバイ候補を複数指定することによって実現できます。そのリストの最初の方に名前が上がっているスタンバイは、同期スタンバイとして使用されます。その後の方に名前が上がっているスタンバイは、同期スタンバイのどれかが故障した時に、その役割を取って代わります。

優先度に基づく同期レプリケーションでは、リストの前の方に名前が現れるスタンバイが同期スタンバイになります。現在の同期スタンバイのどれかが故障した際には、リストの後の方にあるスタンバイが同期スタンバイの役割を引き継ぎます。

クォーラムに基づく同期レプリケーションでは、リストに現れたすべてのスタンバイが同期スタンバイの候補となります。そのどれかが故障した場合でも、他のスタンバイは引き続き同期スタンバイの候補としての役割を担い続けます。

スタンバイが最初にプライマリに接続された時、それはまだ適切に同期されていません。これはcatchupモードと呼ばれます。一旦スタンバイとプライマリ間の遅延がゼロになると、実時間streaming状態に移ります。追従（catchup）期間はスタンバイが作成された直後は長くなるかもしれません。スタンバイが停止している場合、追従期間はスタンバイの停止期間にしたがって長くなります。スタンバイは、streaming状態に達した後でのみ、同期スタンバイになることができます。この状態は、pg_stat_replicationビューで見ることができます。

コミットが受領通知を待機している間にプライマリが再起動した場合、プライマリデータベースが復旧した後、待機中のトランザクションは完全にコミットされたものと記録されます。すべてのスタンバイがプライマリのクラッシュ時点で送信中のWALデータのすべてを受信したかどうかを確認する方法はありません。トランザクションの一部は、プライマリではコミットされたものと表示されていたとしても、スタンバイではコミットされていないと表示されるかもしれません。PostgreSQLは、WALデータをすべてのスタンバイが安全に受信したことが分かるまで、アプリケーションは明示的なトランザクションコミットの成功に関する受領通知を受けとらないことを保証しています。

要求していた数の同期スタンバイを本当に確保できないときは、トランザクションが応答を待たなければならない同期スタンバイの数を、synchronous_standby_namesから減らしてください（もしくは無効にします）。そして、プライマリサーバの設定ファイルを再読み込みしてください。

プライマリが既存のスタンバイサーバから切り離された場合は、スタンバイサーバの中から最善と思われる候補にフェイルオーバーしてください。

トランザクションの待機中にスタンバイサーバを再作成する必要がある場合、pg_start_backup()およびpg_stop_backup()を実行するコマンドをsynchronous_commit = offであるセッション内で確実に実行してください。さもないとこれらの要求はスタンバイに現れるまで永遠に待機します。

26.2.9. スタンバイにおける継続的アーカイビング

スタンバイにおいてWALの継続的アーカイビングが行われる場合、2つのシナリオが考えられます。WALアーカイブがプライマリとスタンバイで共有されるケースと、スタンバイが自分のWALアーカイブを持つケースです。スタンバイが自分のWALアーカイブを持つケースでは、archive_modeをalwaysに設定しておくことにより、アーカイブからリストアされたWALセグメントであろうと、ストリーミングレプリケーション由来のWALセグメントであろうと、WALセグメントを受信する度にスタンバイはアーカイブコマンドを呼び出します。共有アーカイブのケースも同じように扱えますが、archive_commandはアーカイブしようとしているファイルがすでに存在していて、それが同一内容かどうかのチェックを行う必要があります。このため、archive_commandはより工夫が必要です。というのも、archive_commandは既存のファイルを異なる内容で置き換えてはいけませんし、またまったく同じ内容のファイルを置き換えた場合には成功したと報告しなければならないからです。更に、2つのサーバが同時に同じファイルをアーカイブしようとした時に、競合状態が起きないようにしなければなりません。

archive_modeがonの場合には、リカバリモードあるいはスタンバイモードではアーカイブは有効になりません。スタンバイサーバが昇格すると、昇格後にスタンバイサーバはアーカイブを開始します。しかし、自分が生成しなかったWALやタイムライン履歴ファイルは一切アーカイブしません。完全な一連のWALファイルをアーカイブから取り出すためには、WALがスタンバイに到着する前に、すべてのWALがアーカイブされて

いることを保証しなければなりません。ファイルベースのログシッピングにおいても本質的にはこの通りです。というのも、スタンバイはアーカイブにあるファイルだけをリストアできるからです。ストリーミングレプリケーションが有効ならば、この限りではありません。サーバがリカバリモードでない場合には、onとalwaysのモードの間には違いはありません。

26.3. フェイルオーバー

プライマリサーバに障害が起こると、スタンバイサーバはフェイルオーバー処理を始めなければなりません。

スタンバイサーバが故障した場合、フェイルオーバーは不要です。多少の時間の後に、スタンバイサーバを再起動できれば、再起動可能なリカバリのため、リカバリ処理も即座に再起動することができます。スタンバイサーバを再起動できなければ、新しい完全なスタンバイサーバのインスタンスを作成しなければなりません。

プライマリサーバに障害が起こりスタンバイサーバが新しいプライマリとなり、その後古いプライマリが再起動した場合、もはやプライマリサーバでなくなっていることを古いプライマリに知らせる機構が必要です。これはSTONITH (Shoot the Other Node In The Head)と一部ではいわれています。これは、混乱と最悪はデータ損失をもたらしかねない、両方のシステムが自身をプライマリとして認識してしまう状況を防ぐために必要です。

多くのフェイルオーバーシステムではプライマリとスタンバイといった2つのシステムを使用します。なんらかのハートビート機構でプライマリとスタンバイを接続し、両者の接続性とプライマリの実行能力を継続的に確認します。また、第3のシステム（証言サーバと呼ばれます）を使用して、不適切なフェイルオーバーなどの状況を防ぐこともできます。しかし、さらに複雑になりますので、十分な注意と厳密な検証の元に設定を行わない限り行う意味がありません。

PostgreSQLは、プライマリサーバの障害を識別し、スタンバイデータベースサーバに通知するために必要なシステムソフトウェアを提供しません。こうしたツールは多く存在し、IPアドレスの移行といったフェイルオーバーを成功させるために必要な機能をオペレーティングシステムにうまく統合させています。

スタンバイサーバへのフェイルオーバーが起きた後、運用可能なサーバは1つしかありません。これは縮退状態と呼ばれます。以前のスタンバイサーバはプライマリサーバになり、以前のプライマリは停止し、その後も停止し続けるかもしれません。通常の運用に戻すには、スタンバイサーバを再作成しなければなりません。以前のプライマリサーバが起動できれば、これを使用しても構いませんし、第三のおそらく新規のシステムを使用しても構いません。pg_rewindを使って、大きなクラスタにおける処理を早めることもできます。完了すれば、プライマリとスタンバイの役割が切り替わったとみなすことができます。新しいスタンバイサーバを再作成するまでに第三のサーバを使用して新しいプライマリのバックアップを提供することを選択する人もいますが、これがシステム構成と運用手順を複雑にすることは明らかです。

プライマリサーバからスタンバイサーバへの切り替えは高速ですが、フェイルオーバークラスタを再度準備するのに多少時間が必要です。それぞれのシステムを保守のために定期的に停止することができるので、プライマリからスタンバイへの定期的切り替えは有益です。これは同時に、必要になった時、フェイルオーバー機構が実際に機能するかどうかを確認する試験としても役立ちます。管理手順の文書化を勧めます。

ログシッピングを行うスタンバイサーバのフェイルオーバーを発生させるためには、pg_ctl promoteを実行する、pg_promote()を呼び出す、あるいはpromote_trigger_fileで指定されるファイル名とパスを持つトリガファイルを作成してください。フェイルオーバーのためにpg_ctl promoteを使用する、あるいはpg_promote()を呼び出すつもりならば、promote_trigger_fileは必要ありません。プライマリから読み取

り専用の問い合わせによる負荷を軽減させるためだけに使用し、高可用性を目的としていない、報告処理用サーバを構築する場合は、昇格させる必要はありません。

26.4. この他のログシッピングの方法

これまでの節で説明した組み込みのスタンバイモードの他の方法として、アーカイブ場所を順次問い合わせるrestore_commandを使用する方法があります。これはバージョン8.4以前では唯一の利用可能な選択肢でした。このリファレンス実装としてpg_standbyを参照してください。

このモードでは、サーバは1度に1つのWALファイルを適用することに注意してください。このため問い合わせ用にスタンバイサーバを使用する場合(ホットスタンバイを参照)、マスタにおける動作とそれがスタンバイで可視になるまでの間に、WALファイルをみたすために必要とする時間に相当する、遅延が存在します。archive_timeoutを使用して遅延を短くすることができます。また、この方法とストリーミングレプリケーションと組み合わせることができないことにも注意してください。

プライマリおよびスタンバイサーバの両方で発生する操作は通常の継続的なアーカイブ処理とリカバリ処理です。2つのデータベースサーバが連携する唯一の点は、両者が共有するWALファイルのアーカイブです。プライマリがアーカイブに書き出し、スタンバイがアーカイブから読み取ります。注意して他のプライマリサーバ由来のWALアーカイブが混在しないことを確実にしなければなりません。さもないと混乱が発生します。スタンバイ操作でのみ必要なものですので、アーカイブは必ずしも巨大になりません。

2つの疎結合サーバを協調させる秘訣は簡単で、スタンバイサーバにて使用されるrestore_commandです。これは次のWALファイルを問い合わせ、それをプライマリから利用可能になるまで待機します。通常のリカバリ処理はWALアーカイブからファイルを要求し、ファイルが利用できなければ失敗を報告します。スタンバイ処理では、次のWALファイルを入手できないことは異常ではありませんので、スタンバイは利用可能になるまで待機しなければなりません。.historyで終わるファイルについては、待機する必要はなく、非ゼロの終了コードを返さなければなりません。restore_commandを待機させるには、次のWALファイルの存在を確認した後にループする独自のスクリプトを作成することで実現できます。また、restore_commandに割り込み、ループを終了させ、ファイルが存在しないというエラーをスタンバイサーバに返す、フェイルオーバーを発生させる何らかの方法がなければなりません。これがリカバリ処理を停止しますので、スタンバイサーバは通常のサーバになります。

restore_commandの擬似コードの一例は以下です。

```
triggered = false;
while (!NextWALFileReady() && !triggered)
{
    sleep(100000L);         /* wait for ~0.1 sec */
    if (CheckForExternalTrigger())
        triggered = true;
}
if (!triggered)
        CopyWALFileForRecovery();
```

待機を行うrestore_commandの実例はpg_standbyモジュール内で提供されています。これは上記のロジックをどのように正確に実装するかについての参照として使用すべきです。また、これを特定の設定または環境をサポートするため必要に応じて拡張することができます。

フェイルオーバーを通知する手段は計画・設計段階で重要な部分です。考えられる選択肢の1つはrestore_commandです。これは各WALファイルに対して1度実行されるものですが、restore_commandを実行するプロセスは各ファイルに対して起動・終了します。このようにデーモンやサーバプロセスはありませんので、シグナルやシグナルハンドラを使用することはできません。したがって、restore_commandはフェイルオーバーの通知には適していません。特にプライマリサーバ上の既知のarchive_timeout設定と連係して使用できるならば、単純なタイムアウト機能を使用することができます。しかし、これはネットワーク障害や高負荷なプライマリサーバによりフェイルオーバーが始まってしまうため、どちらかというとエラーになりやすいものです。実現可能ならば、明示的な通知用ファイルの作成などの通知機構の方が理想的です。

26.4.1. 実装

この代替方式を使用してスタンバイサーバを構築する短めの手順を以下に示します。各段階の詳細については、注記していますので、前の節を参照してください。

1. できる限り同じようにプライマリシステムとスタンバイシステムを設定してください。同じリリースレベルのPostgreSQLの同一コピーの導入も含みます。

2. プライマリサーバで、継続的アーカイブをスタンバイサーバ上のディレクトリ上にWALをアーカイブするように設定してください。プライマリサーバで、archive_mode、archive_commandおよびarchive_timeoutが適切に設定されていることを確認してください(25.3.1を参照してください)。

3. プライマリサーバでベースバックアップを作成(25.3.2を参照してください)し、スタンバイサーバでこのデータをロードしてください。

4. スタンバイサーバで、上記の通り待機を行うrestore_commandを使用して、ローカルなWALアーカイブからリカバリ処理を実行してください(25.3.4を参照してください)。

リカバリ処理はWALアーカイブを読み取り専用として扱います。このため、WALファイルがスタンバイシステムにコピーされた後、スタンバイデータベースサーバによる読み取りと同時にWALファイルをテープにコピーすることができます。このように、高可用性スタンバイサーバの実行を、災害からのリカバリを目的とした長期的な保管と同時に行うことができます。

試験のためにプライマリサーバとスタンバイサーバを同じシステムで稼動させることができます。これによりサーバ堅牢性が向上することも、高可用性と呼べることもありません。

26.4.2. レコードベースのログシッピング

この代替手法を用いたレコード単位のログシッピングの実装も可能ですが、利用者側の開発が必要です。さらに、完全なWALファイルが転送された後のみで、変更がホットスタンバイ問い合わせで可視になります。

外部プログラムはpg_walfile_name_offset()関数(9.27を参照)を呼び出して、WALの現在の終了点のファイル名と正確なバイトオフセットを見つけ出すことができます。そして、WALファイルに直接アクセスし、直前の既知のWAL終了点から現在の終了点までのデータをスタンバイサーバにコピーすることができます。この方法では、データ損失期間はコピー処理プログラムの実行周期となります。非常に短くすることができますし、部分的に使用されたセグメントファイルを強制的にアーカイブするため無駄な帯域もありません。スタンバイサーバのrestore_commandスクリプトがWALファイル全体しか扱うことができないことに注意してください。このため、逐次的にコピーしたデータは通常はスタンバイサーバで利用することができません。プライ

マリサーバが停止した時のみこれを使用します。その場合、プライマリサーバが立ち上がる前に、最後の部分的なWALファイルがセカンダリサーバに渡されます。この処理の正しい実装では、データコピープログラムとrestore_commandスクリプトとの連係が必要です。

PostgreSQLバージョン9.0から、同じ利点をより少ない設定で実現できるストリーミングレプリケーション(26.2.5参照)を使用することができます。

26.5. ホットスタンバイ

ホットスタンバイという単語は、サーバがアーカイブリカバリを実行している最中にサーバに接続し読み取り専用の問い合わせを実行することができる機能を説明するために使われます。これは、レプリケーションという目的およびバックアップからのリストアの両方で高い精度で好ましい状態にするために有用です。ホットスタンバイという単語はまた、ユーザが問い合わせを実行しながら、または、開いている接続を維持しながら、またはその両方で、サーバをリカバリ状態から通常の動作に移すことができる機能も示すものです。

ホットスタンバイモードにおける問い合わせは、通常の問い合わせに類似していますが、利用上および管理上の差異が多少あり、以下に説明します。

26.5.1. ユーザのための概説

スタンバイサーバでhot_standbyパラメータが真に設定されている場合、リカバリによりシステムが一貫性を持つようになった後接続を受け付け始めます。こうした接続はすべて読み取り専用に限定されます。一時テーブルであっても書き込むことはできません。

スタンバイ上のデータはプライマリサーバから届くまでに多少の時間がかかります。このため、プライマリとスタンバイの間にはある程度の遅延があります。したがって、同じ問い合わせをほとんど同時にプライマリとスタンバイに対して実行すると、異なる結果が返る可能性があります。スタンバイ上のデータはプライマリに対して最後には一貫性を持つといいます。あるトランザクションのコミットレコードがスタンバイ上で再生されると、そのトランザクションでなされた変更はスタンバイで獲得されるすべての新規スナップショットで可視になります。現在のトランザクション分離レベルに応じて、スナップショットは各問い合わせの開始時または各トランザクションの開始時に獲得されます。詳細については13.2を参照してください。

ホットスタンバイ中に開始されたトランザクションは以下のコマンドを発行することができます。

- 問い合わせによるアクセス: SELECTおよびCOPY TO
- カーソルコマンド: DECLAREとFETCHとCLOSE
- 設定の操作: SHOWとSETとRESET
- トランザクション管理コマンド:
  - BEGINとENDとABORTとSTART TRANSACTION
  - SAVEPOINTとRELEASEとROLLBACK TO SAVEPOINT
  - EXCEPTIONブロックおよびこの他の内部サブトランザクション

- LOCK TABLE。なお、以下のモードが明示された場合に限ります。ACCESS SHAREまたはROW SHAREまたはROW EXCLUSIVE
- 計画と資源: PREPAREとEXECUTEとDEALLOCATEとDISCARD
- プラグインと拡張: LOAD
- UNLISTEN
- UNLISTEN

ホットスタンバイ中に開始したトランザクションではトランザクションIDを割り当てられません。また、システムのログ先行書き込みに書き出すことができません。このため、以下の動作はエラーメッセージを生成します。

- データ操作言語(DML): INSERT、UPDATE、DELETE、COPY FROMおよびTRUNCATE。リカバリ中にトリガ内で実行されてしまう場合でも許されていない動作であることに注意してください。現在のホットスタンバイ環境では行うことができないトランザクションIDの割り当てを行うことなく、テーブル行の読み書きを行うことができませんので、この制限は一時テーブルであっても適用されます。
- データ定義言語(DDL): CREATE、DROP、ALTERおよびCOMMENT。この制約は一時テーブルに対しても適用されます。これらの操作の実行がシステムカタログテーブルの更新を必要とするためです。
- SELECT ... FOR SHARE | UPDATE。背後のデータファイルを更新することなく行ロックを獲得することはできないためです。
- データ操作言語のコマンドを生成するSELECT文のルール
- ROW EXCLUSIVE MODEより高いモードを明示的に要求するLOCK
- 短いデフォルト構文のLOCK。これはACCESS EXCLUSIVE MODEを要求するためです。
- 読み取り専用でない状態を明示的に設定するトランザクション処理コマンド
  - BEGIN READ WRITEとSTART TRANSACTION READ WRITE
  - SET TRANSACTION READ WRITEとSET SESSION CHARACTERISTICS AS TRANSACTION READ WRITE
  - SET transaction_read_only = off
- 二相コミットコマンド: PREPARE TRANSACTION、COMMIT PREPAREDおよびROLLBACK PREPARED。読み取り専用トランザクションでも、プリペア相(二相コミットの第1相)においてWALの書き込みが必要だからです。
- シーケンス更新の関数: nextval()とsetval()
- LISTEN、NOTIFY

通常の操作では、「読み取り専用」トランザクションにはLISTENとNOTIFYの使用が許可されています。ホットスタンバイセッションの操作では、通常の読み取り専用セッションよりも少し厳しい制約を受けます。将来のリリースではこの制約の一部が緩和されるかもしれません。

ホットスタンバイ中は、transaction_read_onlyパラメータは常に真であり、変更することはできません。しかし、データベースを変更するような試行がない限り、ホットスタンバイ中の接続は他のデータベース接続

とほとんど同じように動作します。もし、フェイルオーバーまたはスイッチオーバが発生すると、データベースは通常処理モードに切り替わります。サーバのモードが変わってもセッションは接続を保持します。ホットスタンバイが完了すると、読み書き可能なトランザクションを(ホットスタンバイ中に始まったセッションからであっても)始められるようになります。

ユーザはSHOW transaction_read_onlyを発行することで、そのセッションが読み取り専用かどうかを調べることができます。さらに、ユーザがスタンバイサーバに関する情報にアクセスできる関数群(表 9.86)があります。これらによりデータベースの現状認識を行うプログラムを作成することができます。これらを使用して、リカバリの進行状況を監視するために使用したり、データベースを特定の状態にリストアする複雑なプログラムを作成したりすることができます。

26.5.2. 問い合わせコンフリクトの処理

プライマリサーバとスタンバイサーバは、多方面でゆるく結合しています。プライマリサーバの動作はスタンバイサーバに影響します。その結果、負の相互作用またはコンフリクトの可能性があります。最も分かりやすいコンフリクトは性能です。プライマリサーバで巨大なデータがロードされた場合、スタンバイサーバにおいて同様に巨大なWALレコードが生成されるので、スタンバイサーバにおける問い合わせは互いにI/Oなどのシステム資源を奪い合います。

ホットスタンバイで発生する可能性があるコンフリクトの種類には他にもあります。これらのコンフリクトは、問い合わせをキャンセルしなければならない可能性があり、解消させるためにはセッションの接続を閉ざすことになる場合もあるため、*致命的なコンフリクト*です。ユーザにはこうしたコンフリクトを扱うための複数の方法が提供されます。コンフリクトする状況には以下があります。

- プライマリサーバで獲得されたアクセス排他ロックは、スタンバイの問い合わせにおけるテーブルアクセスとコンフリクトします。明示的なLOCKコマンドおよび各種DDL操作を含みます。
- プライマリでテーブル空間を削除することは、一時作業ファイル用にそのテーブル空間を使用するスタンバイ側の問い合わせとコンフリクトします。
- プライマリでデータベースを削除することは、スタンバイ側でそのデータベースに接続するセッションとコンフリクトします。
- WALからのバキュームクリーンアップレコードの適用は、その適用により削除される行のどれか1つでも「見る」ことができるスナップショットを持つスタンバイでのトランザクションとコンフリクトします。
- WALからのバキュームクリーンアップレコードは、消去されるデータが可視か否かに関係なく、スタンバイで対象ページにアクセスする問い合わせとコンフリクトします。

プライマリサーバでは、こうした状況は単に待たされるだけです。ユーザはコンフリクトする操作をキャンセルすることを選ぶことができます。しかし、スタンバイ側には選択肢がありません。WALに記録された操作はすでにプライマリで発生したものですので、スタンバイではその適用に失敗してはなりません。さらに、適用したいWALを無制限に待機させることを許すことは、まったく望まない結果になってしまうかもしれません。なぜなら、スタンバイの状態がプライマリの状態とだんだんとかけ離れてしまうからです。したがって適用すべきWALレコードとコンフリクトするスタンバイの問い合わせを強制的に取り消す仕組みが用意されています。

この問題の例として、スタンバイサーバで現在問い合わせ対象となっているテーブルをプライマリサーバでDROP TABLEを行う管理者を考えてみます。スタンバイでDROP TABLEが適用されたら問い合わせを継続で

きないことは明確です。プライマリ上でこうした状況が発生した場合は、他の問い合わせが終わるまでDROP TABLEは待機させられます。しかし、DROP TABLEがプライマリで実行された時、プライマリ側でスタンバイで稼動する問い合わせに関する情報がありませんので、スタンバイ側のこうした問い合わせを待機させることはできません。スタンバイ側で問い合わせが実行している時にWALの変更レコードがスタンバイに届けば、コンフリクトが発生します。スタンバイサーバはWALレコードの適用を遅延させる(およびその後の適用すべても遅延させる)か、DROP TABLEを適用できるようにコンフリクトする問い合わせを取り消すかのいずれかを行わなければなりません。

コンフリクトする問い合わせが短ければ、適用したいWALを多少遅延させることで、問い合わせを完了させることが通常望まれます。しかし、WALの適用が長く遅延することはたいていは望まれません。したがって、取り消し機能はmax_standby_archive_delayとmax_standby_streaming_delayというパラメータを持ちます。これらはWAL適用に許される遅延を定義するものです。コンフリクトする問い合わせは、何らかの新しく受信したWALデータを適用するための各種遅延設定を超えたら取り消されます。アーカイブからWALデータを読み取る場合(つまりベースバックアップからの初期リカバリや大きく遅延したスタンバイサーバの「追従」)とストリーミングレプリケーションとで異なる遅延値を指定することができるように2つのパラメータが存在します。

主に高可用性のために存在するスタンバイサーバでは、スタンバイ側の問い合わせによって発生する遅延のためにプライマリと大きく遅延が発生することがないように、遅延パラメータを相対的に短く設定することが最善です。しかし、スタンバイサーバが長時間かかる問い合わせを実行するためのものであれば、長い遅延もしくは制限を設けないことが好まれるかもしれません。しかし、長時間かかる問い合わせがWALレコードの適用を遅延させてしまう場合、スタンバイサーバ上の他のセッションがプライマリにおける最近の変更を参照することができなくなることは覚えておいてください。

max_standby_archive_delayまたはmax_standby_streaming_delayで指定した遅延を超えると、コンフリクトする問い合わせは取り消されます。通常これは単なる取り消しエラーという結果となりますが、DROP DATABASEを再生する場合では、コンフリクトするセッション全体が終了します。また、コンフリクトが待機中のトランザクションで保持されるロックについてのものであれば、そのコンフリクトするセッションが終了します(この動作は将来変更されるかもしれません)。

ユーザは取り消された問い合わせをすぐに再試行するかもしれません(もちろん新規のトランザクション開始後に)。問い合わせの取り消しは、再生されるWALレコードの性質に依存するので、取り消された問い合わせが再度実行された場合には正常に動作するかもしれません。

遅延パラメータはスタンバイサーバでWALデータを受信してからの経過時間と比べられることに注意してください。したがって、スタンバイ上で任意の問い合わせに許される猶予期間は、この遅延パラメータよりも大きくなることは決してありません。これまでの問い合わせを完了させるために待機した結果、あるいは、大量の更新負荷に追従することができなくなった結果、スタンバイがすでに遅延している場合は相当小さくなります。

スタンバイ側の問い合わせとWAL再生の間でもっともよくあるコンフリクト理由は「早すぎる消去」です。通常PostgreSQLはMVCC規則にしたがって正確なデータの可視性を確実にするために、古い行バージョンを参照するトランザクションが存在しない場合それらを消去することが許されています。しかし、この規則はマスタ上で実行するトランザクションのみに適用させることができます。したがって、スタンバイ上のトランザクションでまだ可視である行バージョンを、マスタ上の消去処理が削除してしまう可能性があります。

熟練したユーザは、行バージョンの消去と行バージョンの凍結の両方ともスタンバイ側の問い合わせとコンフリクトする可能性があることに気づくはずです。手作業でのVACUUM FREEZEは、更新または削除された行がないテーブルであったとしてもコンフリクトを発生し易いものです。

プライマリサーバにおいて規則的かつ頻繁に更新されるテーブルは、スタンバイサーバにおける問い合わせの取り消しの原因になりやすいことを利用者は理解するべきです。そのような場合、max_standby_archive_delayまたはmax_standby_streaming_delayの設定値はstatement_timeoutの設定と同様に考えることができます。

スタンバイにおける問い合わせの中断が受け入れがたいほど多い場合、この問題を改善する方法が用意されています。1つ目の選択肢は、hot_standby_feedbackパラメータを設定することです。これはVACUUMによる最近不要になった行の削除を防止しますので、消去によるコンフリクトが発生しません。これを行う場合、プライマリで不要になった行の消去が遅延することに注意が必要です。望まないテーブルの膨張が発生してしまうかもしれません。しかし、スタンバイ側で行うべき問い合わせをプライマリサーバ上で直接実行することと比べ、こうした消去に関する問題を優先する価値はありません。また、スタンバイに実行負荷を分散できるという利点があります。スタンバイサーバが接続、切断を頻繁に繰り返す場合、hot_standby_feedbackによるフィードバックが提供されていなければ、その値を調整したいと思うでしょう。例えば、max_standby_archive_delayが増大し、切断している期間WALアーカイブのコンフリクト発生による問い合わせの中断が速やかに行われないことを考えてみてください。また、再接続後に速やかに問い合わせが中断されることを避けるためにmax_standby_streaming_delayを大きくすることを考えてみてください。

他の選択肢は、不要になった行が通常よりも早く消去されないようにプライマリサーバでvacuum_defer_cleanup_ageを増やすことです。これにより、max_standby_streaming_delayを長くすることなく、スタンバイでキャンセルが起こるようになる前により多くの時間、問い合わせを実行することができます。しかし、vacuum_defer_cleanup_ageはプライマリサーバ上で実行されたトランザクションを単位に測定されますので、この方法では特定の実行期間を保証することは困難です。

問い合わせキャンセルの個数とその原因はスタンバイサーバ上のpg_stat_database_conflictsシステムビューを用いて参照することができます。またpg_stat_databaseシステムビューには要約された情報が含まれます。

26.5.3. 管理者のための概説

postgresql.confにおいてhot_standbyがonで（これはデフォルトです）、かつstandby.signalが存在すれば、サーバはホットスタンバイモードで稼動します。しかし、サーバはまず問い合わせが実行できる程度の一貫性を持つ状態を提供するために十分なリカバリを完了させなければなりませんので、ホットスタンバイでの接続が有効になるまでに多少の時間がかかるかもしれません。サーバの準備ができたことを確認するために、アプリケーションで接続試行を繰り返すか、サーバログに以下のメッセージがあるかどうかを確認します。

```
LOG:  entering standby mode


... 多少時間が経過して ...

LOG:  consistent recovery state reached
LOG:  database system is ready to accept read only connections
```

一貫性に関する情報はプライマリでチェックポイント毎に一回記録されます。プライマリでwal_levelがreplicaもしくはlogicalに設定されていなかった期間に書き込まれたWALを読み取っている

間は、ホットスタンバイを有効にすることはできません。また、一貫性のある状態への到達は、以下の両方が存在する間遅延することがあります。

- サブトランザクション数が64を超える書き込みトランザクション
- 非常に長く実行される書き込みトランザクション

ファイルベースのログシッピング(「ウォームスタンバイ」)を実行しているのであれば、次のWALファイルが届く、長くともプライマリのarchive_timeout設定まで待機しなければなりません。

プライマリサーバにおける設定値を変更した場合、スタンバイサーバにおいて数個のパラメータの再設定が必要です。スタンバイサーバにおける設定値は、プライマリサーバにおける設定値以上でなければなりません。ですから、これらの値を増やしたいなら、プライマリで設定を変更する前に、まずスタンバイで設定変更するべきです。逆にこれらの値を減らしたいなら、スタンバイで設定を変更する前に、まずプライマリで設定変更するべきです。これらのパラメータが所定値未満の設定の場合、スタンバイは起動を取りやめます。所定値以上の設定により、スタンバイサーバは再起動してリカバリが再び開始されます。このパラメータは以下です。

- max_connections
- max_prepared_transactions
- max_locks_per_transaction
- max_wal_senders
- max_worker_processes

max_standby_archive_delayおよびmax_standby_streaming_delayの値が適切であるように管理者が選択することが重要です。最善の選択は業務上の優先順位によって変化します。例えば、サーバが主に高可用性を目的としたサーバとして作業するものであれば、短い遅延を設定したいでしょう。非常に積極的な設定ですが、ゼロにしたいかもしれません。スタンバイサーバが意思決定支援のための問い合わせ用の追加サーバとして作業するものであれば、数時間程度の最大の遅延値の設定、あるいは問い合わせの完了を永遠に待つことを意味する-1という設定でさえ、許容範囲であるかもしれません。

プライマリ側で「ヒントビット」として書き出されたトランザクション状態はWALに記録されません。このためスタンバイ側のデータはスタンバイ側でヒントを再度書き出すことになります。ユーザは大規模なソート用の一時ファイルを書き出し、relcache情報ファイルを再作成します。したがって、ホットスタンバイモードではデータベースのすべてが本当に読み取り専用ではありません。また、ローカルでは読み取り専用のトランザクションであってもdblinkモジュールを使用したリモートデータベースへの書き出しや、その他のPL関数を使用したデータベース外部への操作が可能であることに注意してください。

リカバリモードの間、下記の管理者用コマンドは受理されません。

- データ定義言語: 例えばCREATE INDEX
- 権限および所有権: GRANTとREVOKEとREASSIGN
- 保守コマンド: ANALYZEとVACUUMとCLUSTERとREINDEX

繰り返しますが、これらのコマンドの一部は、プライマリサーバにおける「読み取り専用」モードのトランザクションで実際に許可されていることに注意してください。

その結果、スタンバイ側にのみ存在する追加のインデックスやスタンバイ側にのみ存在する統計情報を作成することはできません。これらの管理者用コマンドが必要な場合、プライマリ側で実行しなければなりません。最終的にこの変更はスタンバイ側に伝播します。

pg_cancel_backend()とpg_terminate_backend()は利用者のバックエンドでは実行できますが、リカバリを実行する起動プロセスでは実行できません。pg_stat_activityはリカバリ中のトランザクションをアクティブとして表示しません。その結果、リカバリの間pg_prepared_xactsは常に空となります。調査が必要な準備されたトランザクションがある場合は、プライマリサーバにおいてpg_prepared_xactsを表示し、その場でトランザクションを解決するか、リカバリが終わるのを待ってからトランザクションを解決します。

pg_locksは通常通りバックエンドで保持されるロックを示します。pg_locksはまた、リカバリによって再生されているトランザクションで保持されるAccessExclusiveLocksのすべてを所有する、起動プロセスで管理される仮想トランザクションも表示します。起動プロセスはデータベースの変更を行うためのロックを獲得しません。このため起動プロセスにおいてAccessExclusiveLocks以外のロックはpg_locksでは表示されません。これらは存在することを想定されているだけです。

存在を検知する情報が単純なので、Nagiosプラグインは稼動します。一部の報告値が異なった、混乱を招く結果となりますが、check_postgresの監視スクリプトも動作します。それでも、プライマリで行われるバキュームはその変更をスタンバイに送信します。

リカバリの間WALの制御コマンドは稼動しません。例えば、pg_start_backupやpg_switch_walなどです。

pg_stat_statementsも含み、動的に読み込み可能なモジュールは稼動します。

デッドロック検出を含むアドバイザリロックは、通常リカバリにおいて稼動します。アドバイザリロックはWALに決して記録されないので、プライマリサーバでもスタンバイサーバでもWALの再実行においてコンフリクトが起こらないことに注意してください。プライマリサーバでアドバイザリロックを取得して、スタンバイサーバで同様のアドバイザリロックを掛けることはできません。アドバイザリロックは取得したサーバだけに関係するものです。

SlonyやLondisteやBucardoのようにトリガに基づいたレプリケーションシステムは、スタンバイサーバで全く稼動しません。しかし、それによる変更がスタンバイサーバに送られるまでは、プライマリサーバにおいて問題なく稼動します。WALの再実行はトリガに基づいたものではありません。したがって、データベースへの付加的な書き込みを必要とするか、トリガの使用に依存するものを、スタンバイサーバを中継して他のシステムへ送ることはできません。

一部のUUIDジェネレータは、データベースに新しい状態を書き出すことに依存していない限り動作可能ですが、新しいOIDを割り当てることはできません。

現時点では、読み取り専用のトランザクションでは一時テーブルの作成は許されません。このため既存のスクリプトが正しく動作しない場合があります。この制限は将来のリリースで緩和されるかもしれません。これは、標準SQLとの互換性の問題でもあり、技術的な問題でもあります。

テーブル空間が空の場合だけ、DROP TABLESPACEが成功します。一部のスタンバイ側のユーザはtemp_tablespacesパラメータを介してテーブル空間を活発に使用しているかもしれません。テーブル空間に一時ファイルが存在する場合、一時ファイルを確実に削除するためすべての問い合わせが取り消されます。このため、WAL再生を続けながらテーブル空間を削除することができます。

プライマリサーバにおけるDROP DATABASEまたはALTER DATABASE ... SET TABLESPACEの実行により、スタンバイサーバのデータベースに接続するすべてのユーザを強制的に接続を切断させることになるWALエ

ントリを生成します。これはmax_standby_streaming_delayの設定にかかわらず、直ちに起こります。ALTER DATABASE ... RENAMEはユーザを切断しないので大部分の場合は気がつきませんが、プログラムがデータベースの名称に依存するときは混乱の原因となることに注意してください。

通常の(リカバリ以外の)モードで、ログイン権限を持つロールが接続している間にそのロールにDROP USERまたはDROP ROLEを発行した場合、接続中のユーザには何も起こらず、接続し続けます。しかし、そのユーザは再接続できません。この振舞いはリカバリモードでも適用されます。このためプライマリ側でDROP USERされたとしても、スタンバイ側のユーザの接続は切断されません。

リカバリの間も統計情報は収集されます。すべてのスキャン、読み取り、ブロック、インデックスの使用などは、スタンバイサーバにおいて正常に記録されます。再実行によりプライマリサーバの結果が重複して収集されることはないので、行の挿入によりpg_stat_user_tablesの挿入列の値は増加しません。リカバリの開始時点で統計情報ファイルが削除されるので、プライマリサーバとスタンバイサーバで統計情報は異なります。これは将来どうするか検討中であり、バグではありません。

リカバリの間は自動バキュームは稼動しません。リカバリが終わると正常に起動します。

リカバリの間、チェックポイントプロセスとバックグラウンドライタプロセスは稼動しています。チェックポイントプロセスは(プライマリサーバにおけるチェックポイントに類似した)リスタートポイントを設定し、通常のブロック消去を行います。これはスタンバイサーバに保存されるヒントビット情報の更新を含むことができます。リカバリの間CHECKPOINTコマンドは受理されますが、新規のチェックポイントではなくてリスタートポイントが設定されます。

26.5.4. ホットスタンバイパラメータリファレンス

種々のパラメータが上記26.5.2および26.5.3で述べられています。

プライマリサーバでは、wal_levelおよびvacuum_defer_cleanup_ageのパラメータを使用できます。プライマリサーバにmax_standby_archive_delayおよびmax_standby_streaming_delayを設定しても無効です。

スタンバイサーバではhot_standbyとmax_standby_archive_delayとmax_standby_streaming_delayのパラメータを使用できます。サーバがスタンバイモードの間vacuum_defer_cleanup_ageを設定しても無効です。しかし、スタンバイサーバがプライマリサーバになった場合、意味を持つようになります。

26.5.5. 警告

ホットスタンバイには幾つかの制限があります。将来のリリースでは改善されると思われます。

- スナップショットを取ることができるようになる前に、実行中のトランザクションについての完全な知識が要求されます。(現時点では64を超える)多くのサブトランザクションを使用するトランザクションでは、実行中の最長の書き込みトランザクションが完了するまで、読み取り専用の接続の開始は遅延されます。この状況が起こると、それを説明するメッセージがサーバログに記録されます。
- スタンバイ問い合わせ用の有効な起動ポイントは、マスタにおけるチェックポイント毎に生成されます。マスタが停止状態にある時にスタンバイが停止した場合、プライマリが起動し、さらに起動ポイントをWALログに生成するまで再度ホットスタンバイになることができないことがあります。この状況は、通常考

えられる状態では問題ではありません。一般的に、プライマリが停止し利用できなくなった場合、それはスタンバイに対して新しいプライマリに切り替わることを要求するような深刻な失敗が原因であることが多いはずです。また、プライマリを意図的に停止させるような状況では、それに伴いスタンバイが新しいプライマリになめらかに切り替わることも普通の手順です。

- リカバリの終了において、準備されたトランザクションが保持するAccessExclusiveLocksには、通常の2倍のロックテーブルへのエントリ数が必要です。通常AccessExclusiveLocksを取るプリペアドトランザクションを大量に同時実行させる、または、多くのAccessExclusiveLocksを取る大規模なトランザクションを1つ実行させることを考えている場合、max_locks_per_transactionの値を、おそらくプライマリサーバのパラメータ値の倍程度に大きくすることを勧めます。max_prepared_transactionsの設定が0ならば、これを検討する必要はまったくありません。

- シリアライザブルトランザクション分離レベルはまだホットスタンバイでは利用できません。(13.2.3および13.4.1参照) ホットスタンバイにおいてトランザクションをシリアライザブルトランザクション分離レベルに設定しようとすると、エラーになります。

第27章 データベース活動状況の監視

データベース管理者はよく、「システムは今現在何をしているか」を気にします。本章ではそれを知る方法について説明します。

データベース活動状況の監視と性能解析用のツールはいくつか存在します。本章の大部分はPostgreSQLの統計情報コレクタの説明に費されていますが、psやtop、iostat、vmstatなどの通常のUnix監視プログラムを無視すべきではありません。また、性能が悪い問い合わせであると認知された問い合わせは、その後、PostgreSQLのEXPLAINコマンドを使用して調査を行う必要が発生します。14.1では、個々の問い合わせの振舞いを理解するための、EXPLAINやその他の方法について記載しています。

27.1. 標準的なUnixツール

ほとんどのUNIXプラットフォームでは、PostgreSQLは、個々のサーバプロセスが容易に識別できるように、psによって報告されるコマンドタイトル部分を変更します。以下に表示例を示します。

```
$ ps auxww | grep ^postgres
postgres  15551  0.0  0.1  57536  7132 pts/0    S    18:02   0:00 postgres -i
postgres  15554  0.0  0.0  57536  1184 ?        Ss   18:02   0:00 postgres: background writer
postgres  15555  0.0  0.0  57536   916 ?        Ss   18:02   0:00 postgres: checkpointer
postgres  15556  0.0  0.0  57536   916 ?        Ss   18:02   0:00 postgres: walwriter
postgres  15557  0.0  0.0  58504  2244 ?        Ss   18:02   0:00 postgres: autovacuum launcher
postgres  15558  0.0  0.0  17512  1068 ?        Ss   18:02   0:00 postgres: stats collector
postgres  15582  0.0  0.0  58772  3080 ?        Ss   18:04   0:00 postgres: joe runbug 127.0.0.1
 idle
postgres  15606  0.0  0.0  58772  3052 ?        Ss   18:07   0:00 postgres: tgl regression
 [local] SELECT waiting
postgres  15610  0.0  0.0  58772  3056 ?        Ss   18:07   0:00 postgres: tgl regression
 [local] idle in transaction
```

(psの適切な呼び出し方はプラットフォームによって異なります。同様に、何が詳細に表示されるのかも異なります。この例は最近のLinuxシステムのものです。) この一覧の最初のプロセスはマスタサーバプロセスです。表示されているコマンド引数は、起動時に使用されたものと同じものです。次の5つのプロセスは、マスタプロセスから自動的に起動されるバックグラウンドワーカプロセスです。(システムを統計情報コレクタが起動しないように設定していた場合は「統計情報コレクタ」はありません。同様に「自動バキュームランチャ」を無効にできます。) 残るプロセスはそれぞれ、1つのクライアント接続を取り扱うサーバプロセスです。それぞれのプロセスは、次の形式のコマンドライン表示を設定します。

```
postgres: user database host activity
```

ユーザ、データベース、(クライアント)ホストという項目はクライアント接続の存続期間中変更されることはありませんが、活動状況を示す部分は変わります。活動状況は、idle(つまり、クライアントからのコマンド待ち状態)、idle in transaction(BEGINブロックの内側でのクライアントの待ち状態)、またはSELECTのようなコマンド種類名のいずれかとなります。また、そのサーバプロセスが他のセッションによって保持されたロックを待っている状態の場合は、waitingが追加されます。上の例では、プロセス15606はプロセス

15610におけるトランザクションの完了とそれに伴うロックの解放を待っていると推測できます。(他に実行中のセッションがありませんので、プロセス15610がブロックしている側であるはずです。もっと複雑な場合ではpg_locksシステムビューを検索し、どのプロセスがどのプロセスをブロックしているか決定しなければなりません。)

cluster_nameが設定されていれば、psの出力でクラスタ名も表示されます。

```
$ psql -c 'SHOW cluster_name'
 cluster_name
--------------
 server1
(1 row)

$ ps aux|grep server1
postgres   27093  0.0  0.0  30096  2752 ?        Ss   11:34   0:00 postgres: server1: background
 writer
...
```

update_process_titleを無効にした場合、活動情報を示す部分は更新されません。新しいプロセスが起動した時に一度だけ、プロセスの表題は設定されます。プラットフォームの中には、これによりコマンドごとのオーバヘッドをかなり抑えられるものもありますし、まったく意味がないものもあります。

> **ヒント**
>
> Solarisでは特別な取り扱いが必要です。/bin/psではなく、/usr/ucb/psを使用しなければなりません。また、wフラグを1つではなく2つ使用しなければなりません。さらに、元のpostgresの呼び出しに関するpsのステータス表示は、各サーバプロセスに関するステータス表示よりも短くなければなりません。この3条件を全て満たさないと、各サーバプロセスのpsの出力は、元のpostgresのコマンドラインのものになってしまいます。

27.2. 統計情報コレクタ

PostgreSQLの統計情報コレクタはサーバの活動状況に関する情報を収集し、報告するサブシステムです。現在、コレクタはテーブルとインデックスへのアクセスをディスクブロックおよび個々の行単位で数えることができます。またこれは、各テーブル内の総行数、および、各テーブルでのバキュームやアナライズの実施情報を追跡します。また、ユーザ定義関数の呼ばれた回数、それぞれの消費した総時間を数えます。

また、PostgreSQLは他のサーバプロセスによって現在実行されている正確なコマンドなど現在システム内で起きていること、またシステム内にどんな他の接続が存在するかということについての動的情報を正確に報告する機能を持ちます。これはコレクタプロセスから独立している機能です。

27.2.1. 統計情報収集のための設定

統計情報の収集によって問い合わせの実行に少しオーバーヘッドが加わりますので、システムは情報を収集するようにもしないようにも設定できます。これは通常はpostgresql.conf内で設定される、設定パラメータによって制御されます。(設定パラメータの設定についての詳細は第19章を参照してください。)

track_activitiesパラメータにより、すべてのサーバプロセスで現在実行されているコマンドを監視することができます。

track_countsパラメータは、テーブルおよびインデックスアクセスに関する統計情報を収集するかどうか制御します。

track_functionsパラメータは、ユーザ定義関数の使用状況を追跡するかどうかを指定できます。

track_io_timingパラメータは、ブロック読み取りおよび書き込み回数の監視するかどうかを指定できます。

通常、これらの変数は全てのサーバプロセスに適用できるようにpostgresql.conf内で設定されます。しかし、SETコマンドを使用して、個別のセッションで有効または無効にできます。(一般ユーザがその活動を管理者から隠すことを防止するために、スーパーユーザのみがSETを使用してこれらのパラメータを変更できます。)

統計情報コレクタは収集した情報を他のPostgreSQLプロセスに一時ファイルを介して送信します。これらのファイルはstats_temp_directoryパラメータで指名されたディレクトリ、デフォルトはpg_stat_tmp内に格納されます。性能を向上させるために、stats_temp_directoryをRAMベースのファイルシステムを指し示すようにして、物理的なI/O要求を減らすことができます。サーバが正しくシャットダウンした際は、統計情報がサーバの再起動を跨がって保持されるように、統計情報データの永続的なコピーがpg_statサブディレクトリに格納されます。サーバ起動時にリカバリが実施される場合(例えば、即時シャットダウンやサーバクラッシュ、ポイントインタイムリカバリの後)、統計カウンタはすべてリセットされます。

27.2.2. 統計情報の表示

システムの現在の状態を表示するために、いくつかの定義済みのビューがあり、表 27.1に一覧されています。また、統計情報の収集結果を表示するために、他にもいくつかのビューがあり、表 27.2に一覧されています。あるいはまた、27.2.20で説明する、基礎的な統計情報関数を使用した独自のビューを構築することもできます。

この統計情報を使用して、収集されるデータを監視する場合、この情報は即座に更新されないことを認識することが重要です。個別のサーバプロセスは、待機状態になる直前に、新しい統計情報に関する数をコレクタに送信します。ですので、実行中の問い合わせやトランザクションは表示上の総和には影響を与えません。また、コレクタ自体はPGSTAT_STAT_INTERVALミリ秒(サーバ構築時に変更しない限り500 ms)に多くても一度新しい報告を出力します。ですので、表示上の情報は実際の活動から遅れて表示されます。しかし、track_activitiesで収集される現在の問い合わせの情報は常に最新です。

この他の重要なポイントは、いつサーバプロセスが統計情報を表示するように尋ねられるかです。サーバプロセスは、まずコレクタプロセスによって発行された最も最近の報告を取り出します。そして、現在のトランザクションが終わるまで、全ての統計情報ビューと関数においてこのスナップショットを使用し続けます。ですから、現在のトランザクションを続けている間、統計情報は一定の情報を示します。同様に、全セッションの現在の問い合わせに関する情報も、そうした情報がトランザクションで最初に要求された時に収集され、そのトランザクションの間同じ情報が表示されます。これはバグではなく、特徴です。なぜなら、これにより、知らない間に値が変わるのを考慮することなく、統計情報に対して複数の問い合わせを実行し、その結果を相関することができるからです。しかし、各問い合わせで新しい結果を取り出したい場合は、確実にトランザクションブロックの外側でその問い合わせを行ってください。あるいはまた、pg_stat_clear_snapshot()を呼び出すこともできます。これは現在のトランザクションの統計情報スナップショットを(もしあれば)破棄します。次に統計情報を使用する場合に新しいスナップショットを取り出すことになります。

トランザクションからは、pg_stat_xact_all_tables、pg_stat_xact_sys_tables、pg_stat_xact_user_tables、およびpg_stat_xact_user_functionsを通じて、自身の統計情報(まだコレクタに送られていない)も参照することができます。これらの数値はトランザクション中に継続的に更新されていくため上記の様な(静的な情報を示す)振る舞いとはなりません。

表 27.1で表示される動的な統計ビューの情報の中にはセキュリティ制限があるものがあります。一般ユーザは自身のセッション(メンバとなっているロールに属するセッション)に関する全情報だけを参照できます。他セッションに関する行では多くの列がNULLになるでしょう。しかしながら、セッションの存在とセッションのユーザとデータベースなどの一般的な属性は全ユーザに可視であることに注意してください。スーパーユーザと組み込みロールpg_read_all_stats(21.5も参照してください)のメンバも全セッションに関する全情報を参照できます。

表27.1 動的統計情報ビュー

ビュー名	説明
pg_stat_activity	サーバプロセスあたり1行の形式で、状態や現在の問い合わせ等のプロセスの現在の活動状況に関連した情報を表示します。詳細についてはpg_stat_activityを参照してください。
pg_stat_replication	WAL送信プロセス毎に1行の形式で、送信サーバが接続したスタンバイサーバへのレプリケーションに関する統計情報を表示します。詳細についてはpg_stat_replicationを参照してください。
pg_stat_wal_receiver	1行の形式で、受信サーバが接続したサーバからWAL受信サーバに関する統計情報を表示します。詳細についてはpg_stat_wal_receiverを参照してください。
pg_stat_subscription	1つのサブスクリプションにつき少なくとも1行の形式で、サブスクリプションワーカに関する情報を表示します。詳細についてはpg_stat_subscriptionを参照してください。
pg_stat_ssl	接続(通常およびレプリケーション)あたり1行の形式で、接続に使われるSSLの情報を表示します。詳しくはpg_stat_sslを参照してください。
pg_stat_gssapi	接続(通常およびレプリケーション)あたり1行の形式で、接続に使われるGSSAPI認証と暗号化に関する情報を表示します。詳しくはpg_stat_gssapiを参照してください。
pg_stat_progress_analyze	ANALYZEを実行している(自動バキュームワーカプロセスを含んだ)各バックエンドごとに1行の形式で、現在の進捗を表示します。27.4.1を参照してください。
pg_stat_progress_create_index	CREATE INDEXまたはREINDEXを実行している各バックエンドごとに1行の形式で、現在の進捗を表示します。27.4.2を参照してください。
pg_stat_progress_vacuum	VACUUMを実行している(自動バキュームワーカプロセスを含んだ)各バックエンドごとに1行の形式で、現在の進捗を表示します。27.4.3を参照してください。

ビュー名	説明
pg_stat_progress_cluster	CLUSTERまたはVACUUM FULLを実行している各バックエンドごとに1行の形式で、現在の進捗を表示します。27.4.4を参照してください。
pg_stat_progress_basebackup	ベースバックアップをストリームしている各WAL送信プロセスごとに1行の形式で、現在の進捗を表示します。27.4.5を参照してください。

表27.2 収集済み統計情報ビュー

ビュー名	説明
pg_stat_archiver	WALアーカイバプロセスの活動状況に関する統計情報を1行のみで表示します。詳細についてはpg_stat_archiverを参照してください。
pg_stat_bgwriter	バックグラウンドライタプロセスの活動状況に関する統計情報を1行のみで表示します。詳細についてはpg_stat_bgwriterを参照してください。
pg_stat_database	データベース毎に1行の形式で、データベース全体の統計情報を表示します。詳細についてはpg_stat_databaseを参照してください。
pg_stat_database_conflicts	データベース毎に1行の形式で、スタンバイサーバにおける復旧との競合のためにキャンセルされた問い合わせについてのデータベース全体の統計情報を表示します。詳細についてはpg_stat_database_conflictsを参照してください。
pg_stat_all_tables	現在のデータベースの各テーブルごとに1行の形式で、特定のテーブルへのアクセスに関する統計情報を示します。詳細についてはpg_stat_all_tablesを参照してください。
pg_stat_sys_tables	システムテーブルのみが表示される点を除き、pg_stat_all_tablesと同じです。
pg_stat_user_tables	ユーザテーブルのみが表示される点を除き、pg_stat_all_tablesと同じです。
pg_stat_xact_all_tables	pg_stat_all_tablesと似ていますが、現在のトランザクションにて実施された処理結果をカウントします。(数値が見える時点では、これらの数値はpg_stat_all_tablesと関連するビューに含まれていません。) このビューでは、有効行数、無効行数、およびバキュームやアナライズの活動は表示しません。
pg_stat_xact_sys_tables	システムテーブルのみが表示される点を除き、pg_stat_xact_all_tablesと同じです。
pg_stat_xact_user_tables	ユーザテーブルのみが表示される点を除き、pg_stat_xact_all_tablesと同じです。

ビュー名	説明
pg_stat_all_indexes	現在のデータベースのインデックスごとに1行の形式で、特定のインデックスへのアクセスに関する統計情報を示します。詳細についてはpg_stat_all_indexesを参照してください。
pg_stat_sys_indexes	システムテーブルのインデックスのみが表示される点を除き、pg_stat_all_indexesと同じです。
pg_stat_user_indexes	ユーザテーブルのインデックスのみが表示される点を除き、pg_stat_all_indexesと同じです。
pg_statio_all_tables	現在のデータベース内のテーブルごとに1行の形式で、特定のテーブルに対するI/Oに関する統計情報を示します。詳細についてはpg_statio_all_tablesを参照してください。
pg_statio_sys_tables	システムテーブルのみが表示される点を除き、pg_statio_all_tablesと同じです。
pg_statio_user_tables	ユーザテーブルのみが表示される点を除き、pg_statio_all_tablesと同じです。
pg_statio_all_indexes	現在のデータベース内のインデックスごとに1行の形式で、特定のインデックスに対するI/Oに関する統計情報を示します。詳細についてはpg_statio_all_indexesを参照してください。
pg_statio_sys_indexes	システムテーブルのインデックスのみが表示される点を除き、pg_statio_all_indexes と同じです。
pg_statio_user_indexes	ユーザテーブルのインデックスのみが表示される点を除き、pg_statio_all_indexesと同じです。
pg_statio_all_sequences	現在のデータベース内のシーケンスごとに1行の形式で、特定のシーケンスに対するI/Oに関する統計情報を示します。詳細についてはpg_statio_all_sequencesを参照してください。
pg_statio_sys_sequences	システムシーケンスのみが表示される点を除き、pg_statio_all_sequencesと同じです（現時点では、システムシーケンスは定義されていませんので、このビューは常に空です）。
pg_statio_user_sequences	ユーザシーケンスのみが表示される点を除き、pg_statio_all_sequencesと同じです。
pg_stat_user_functions	追跡された関数ごとに1行の形式で、関数の実行に関する統計情報を示します。詳細についてはpg_stat_user_functionsを参照してください。
pg_stat_xact_user_functions	pg_stat_user_functionsと似ていますが、現在のトランザクション中に呼び出されたものだけをカウントします。(数値が見える時点では、これらの数値はpg_stat_user_functionsに含まれていません。)
pg_stat_slru	SLRUごとに1行の形で、操作に関する統計情報を示します。詳細はpg_stat_slruを参照してください。

インデックス単位の統計情報は、どのインデックスが使用され、どの程度効果があるのかを評価する際に、特に有用です。

pg_statio_ビューは主に、バッファキャッシュの効率を評価する際に有用です。実ディスク読み取りの数がバッファヒットの数よりもかなり少ないのであれば、そのキャッシュはカーネル呼び出しを行うことなく、ほとんどの読み取り要求を満足させています。しかし、PostgreSQLバッファキャッシュに存在しないデータはカーネルのI/Oキャッシュにある可能性があり、そのため、物理的な読み取りを行うことなく取り出される可能性があるというPostgreSQLのディスクI/Oの取り扱いのため、これらの統計情報は、完全な論拠を提供しません。PostgreSQLのI/O動作に関するより詳細な情報を入手したいのであれば、PostgreSQL統計情報コレクタとカーネルのI/Oの取り扱いの監視を行うオペレーティングシステムユーティリティを組み合わせることを勧めます。

27.2.3. pg_stat_activity

pg_stat_activityはサーバプロセス毎に、そのプロセスの現在の活動に関連する情報を表示する1行を持ちます。

表27.3 pg_stat_activityビュー

列 型 説明
datid oid バックエンドが接続するデータベースのOIDです。
datname name バックエンドが接続するデータベースの名前です。
pid integer バックエンドのプロセスIDです。
leader_pid integer このプロセスがパラレルクエリワーカであればパラレルグループリーダーのプロセスIDです。このプロセスがパラレルグループリーダーであるか、パラレルクエリに参加していないのであればNULLです。
usesysid oid バックエンドにログインしたユーザのOIDです。
usename name バックエンドにログインしたユーザの名前です。
application_name text バックエンドに接続したアプリケーションの名前です。
client_addr inet バックエンドに接続したクライアントのIPアドレスです。このフィールドがNULLである場合、これはクライアントがサーバマシン上のUnixソケット経由で接続されたか、自動バキュームなど内部プロセスであることを示しています。
client_hostname text client_addrのDNS逆引き検索により報告された、接続クライアントのホスト名です。IP接続、かつlog_hostnameが有効である場合にのみこのフィールドは非NULLになります。
client_port integer クライアントがバックエンドとの通信に使用するTCPポート番号、もしくはUnixソケットを使用する場合は-1です。このフィールドがNULLであれば、内部のサーバプロセスであることを示しています。
backend_start timestamp with time zone プロセスが開始した時刻です。クライアントのバックエンドについては、クライアントがサーバに接続した時刻です。

列 型 説明
xact_start timestamp with time zone プロセスの現在のトランザクションが開始した時刻です。活動中のトランザクションがない場合はNULLです。現在の問い合わせがトランザクションの先頭である場合、この列はquery_start列と同じです。
query_start timestamp with time zone 現在活動中の問い合わせが開始した時刻です。もしstateがactiveでない場合は直前の問い合わせが開始した時刻です。
state_change timestamp with time zone stateの最終変更時刻です。
wait_event_type text バックエンドが待機しているイベントがあれば、その型、なければNULLとなります。表 27.4を参照してください。
wait_event text バックエンドが現在待機している場合は待機イベント名、そうでなければNULLとなります。表 27.5から表 27.13までを参照してください。
state text 現在のバックエンドの総体的な状態です。以下のいずれかの値を取ることができます。 • active: バックエンドは問い合わせを実行中です。 • idle: バックエンドは新しいクライアントからのコマンドを待機しています。 • idle in transaction: バックエンドはトランザクションの内部にいますが、現在実行中の問い合わせがありません。 • idle in transaction (aborted): この状態はidle in transactionと似ていますが、トランザクション内のある文がエラーになっている点が異なります。 • fastpath function call: バックエンドは近道関数を実行中です。 • disabled: この状態は、このバックエンドでtrack_activitiesが無効である場合に報告されます。
backend_xid xid もしあれば、このバックエンドの最上位のトランザクション識別子です。
backend_xmin xid 現在のバックエンドのxminです。
query text バックエンドの最も最近の問い合わせテキストです。stateがactiveの場合、このフィールドは現在実行中の問い合わせを示します。その他のすべての状態では、実行済みの最後の問い合わせを示します。デフォルトでは問い合わせのテキストは1024バイトで切り詰められますが、この値はパラメータtrack_activity_query_sizeにより変更できます。
backend_type text 現在のバックエンドの種別です。取り得る値はautovacuum launcher、autovacuum worker、logical replication launcher、logical replication worker、parallel worker, background writer、client backend、checkpointer、startup、walreceiver、walsender、walwriterです。これに加えて、拡張によって登録されたバックグラウンドワーカは追加の型を持つかも知れません。

> **注記**
>
> wait_eventとstate列は独立しています。バックエンドがactive状態である場合、いくつかのイベントではwaitingかもしれませんし、そうでないかもしれません。状態がactiveであり、wait_eventがNULLでない場合、問い合わせは実行中ですが、システム内のどこかでブロックされていることを意味します。

表27.4 待機イベント型

待機イベント型	説明
Activity	サーバプロセスはアイドル状態です。このイベント型はプロセスがメインの処理ループ内で活動を待機していることを示します。wait_eventによりその待機点が特定できます。表 27.5を参照してください。
BufferPin	サーバプロセスは、データバッファに排他的アクセスをするために待機しています。バッファピン待機は、他のプロセスが該当のバッファから最後に読み込んだデータのオープンカーソルを保持している場合に長引かされることがあります。表 27.6を参照してください。
Client	サーバプロセスはユーザアプリケーションに接続しているソケット上での活動を待機しています。それゆえ、サーバはその内部プロセスとは無関係の何かが起きることを期待しています。wait_eventによりその待機点が特定できます。表 27.7を参照してください。
Extension	サーバプロセスは拡張モジュールにより定義された条件を待機しています。表 27.8を参照してください。
IO	サーバプロセスは入出力が完了するのを待機しています。wait_eventによりその待機点が特定できます。表 27.9を参照してください。
IPC	サーバプロセスは、他のサーバプロセスとの相互作用を待機しています。wait_eventによりその待機点が特定できます。表 27.10を参照してください。
Lock	サーバプロセスは重量ロックを待機しています。ロックマネージャロックや単にロックとしても知られている重量ロックは、主にテーブルのようなSQLで可視なオブジェクトを保護します。しかし、それらはリレーション拡張のような、なんらかの内部操作のために相互排他を確実にするためにも使用されます。wait_eventは、待たせているロックの型を識別します。表 27.11を参照してください。
LWLock	サーバプロセスは軽量ロックを待機しています。ほとんどのこのようなロックは、共有メモリ内の特定のデータ構造を保護します。wait_eventには軽量ロックの目的を特定する名前が入ります。(特定の名前がついたロックもあれば、似たような目的のロックのグループの一部となっているものもあります。) 表 27.12を参照してください。

待機イベント型	説明
Timeout	サーバプロセスはタイムアウトが満了するのを待機しています。wait_eventによりその待機点が特定できます。表 27.13を参照してください。

表27.5 Activity型の待機イベント

Activity待機イベント	説明
ArchiverMain	アーカイバプロセスのメインループ内で待機しています。
AutoVacuumMain	自動バキュームのランチャプロセスのメインループ内で待機しています。
BgWriterHibernate	バックグラウンドライタプロセス内で待機し、休止状態になっています。
BgWriterMain	バックグラウンドライタプロセスのメインループ内で待機しています。
CheckpointerMain	チェックポインタプロセスのメインループ内で待機しています。
LogicalApplyMain	論理レプリケーション適用プロセスのメインループ内で待機しています。
LogicalLauncherMain	論理レプリケーションランチャプロセスのメインループ内で待機しています。
PgStatMain	統計情報収集プロセスのメインループ内で待機しています。
RecoveryWalStream	ストリーミングリカバリ中に、起動プロセスのメインループ内でWALが到着するのを待機しています。
SysLoggerMain	syslogプロセスのメインループ内で待機しています。
WalReceiverMain	WAL受信プロセスのメインループ内で待機しています。
WalSenderMain	WAL送信プロセスのメインループ内で待機しています。
WalWriterMain	WAL書き込みプロセスのメインループ内で待機しています。

表27.6 BufferPin型の待機イベント

BufferPin待機イベント	説明
BufferPin	バッファ上の排他ピンを獲得するのを待機しています。

表27.7 Client型の待機イベント

Client待機イベント	説明
ClientRead	クライアントからのデータの読み込みを待機しています。
ClientWrite	クライアントへのデータの書き込みを待機しています。
GSSOpenServer	GSSAPIセッションを確立する際にクライアントからのデータ読み込みを待機しています。
LibPQWalReceiverConnect	WAL受信プロセス内でリモートサーバへの接続が確立するのを待機しています。

Client待機イベント	説明
LibPQWalReceiverReceive	WAL受信プロセス内でリモートサーバからデータを受信するのを待機しています。
SSLOpenServer	接続試行中にSSLを待機しています。
WalReceiverWaitStart	起動プロセスがストリーミングレプリケーションの初期データを送信するのを待機しています。
WalSenderWaitForWAL	WAL送信プロセス内でWALがフラッシュされるのを待機しています。
WalSenderWriteData	WAL送信プロセス内でWAL受信者からの応答を処理している時に、何らかの活動を待機しています。

表27.8 Extension型の待機イベント

Extension待機イベント	説明
Extension	拡張内で待機しています。

表27.9 IO型の待機イベント

IO待機イベント	説明
BufFileRead	バッファファイルからの読み取りを待機しています。
BufFileWrite	バッファファイルへの書き込みを待機しています。
ControlFileRead	pg_controlファイルからの読み取りを待機しています。
ControlFileSync	pg_controlファイルが永続的ストレージに到達するのを待機しています。
ControlFileSyncUpdate	pg_controlファイルの更新が永続的ストレージに到達するのを待機しています。
ControlFileWrite	pg_controlファイルへの書き込みを待機しています。
ControlFileWriteUpdate	pg_controlファイルの更新の書き込みを待機しています。
CopyFileRead	ファイルコピーの操作の間、読み込みを待機しています。
CopyFileWrite	ファイルコピーの操作の間、書き込みを待機しています。
DSMFillZeroWrite	動的共有メモリの背後のファイルにゼロのバイトを書き込むのを待機しています。
DataFileExtend	リレーションのデータファイルが拡張されるのを待機しています。
DataFileFlush	リレーションのデータファイルが永続的ストレージに到達するのを待機しています。
DataFileImmediateSync	リレーションのデータファイルが永続的ストレージに即座に同期されるのを待機しています。
DataFilePrefetch	リレーションのデータファイルからの非同期プリフェッチを待機しています。
DataFileRead	リレーションのデータファイルからの読み込みを待機しています。

IO待機イベント	説明
DataFileSync	リレーションのデータファイルへの変更が永続的ストレージに到達するのを待機しています。
DataFileTruncate	リレーションのデータファイルが切り詰められるのを待機しています。
DataFileWrite	リレーションのデータファイルへの書き込みを待機しています。
LockFileAddToDataDirRead	データディレクトリのロックファイルに行を追加する間の読み込みを待機しています。
LockFileAddToDataDirSync	データディレクトリのロックファイルに行を追加する間、データが永続的ストレージに到達するのを待機しています。
LockFileAddToDataDirWrite	データディレクトリのロックファイルに行を追加する間の書き込みを待機しています。
LockFileCreateRead	データディレクトリのロックファイルを作成する間の読み込みを待機しています。
LockFileCreateSync	データディレクトリのロックファイルを作成する間、データが永続的ストレージに到達するのを待機しています。
LockFileCreateWrite	データディレクトリのロックファイルを作成する間の書き込みを待機しています。
LockFileReCheckDataDirRead	データディレクトリのロックファイルを再検査する間の読み込みを待機しています。
LogicalRewriteCheckpointSync	チェックポイントの間に、論理的な再書き込みのマッピングが永続的ストレージに到達するのを待機しています。
LogicalRewriteMappingSync	論理的な再書き込みの間に、マッピングデータが永続的ストレージに到達するのを待機しています。
LogicalRewriteMappingWrite	論理的な再書き込みの間に、マッピングデータの書き込みを待機しています。
LogicalRewriteSync	論理的な再書き込みのマッピングが永続的ストレージに到達するのを待機しています。
LogicalRewriteTruncate	論理的な再書き込みの際にマッピングデータが切り詰められるのを待機しています。
LogicalRewriteWrite	論理的な再書き込みのマッピングの書き込みを待機しています。
RelationMapRead	リレーションのマップファイルの読み込みを待機しています。
RelationMapSync	リレーションのマップファイルが永続的ストレージに到達するのを待機しています。
RelationMapWrite	リレーションのマップファイルの書き込みを待機しています。
ReorderBufferRead	並べ替えのバッファ管理の間に読み込みを待機しています。
ReorderBufferWrite	並べ替えのバッファ管理の間に書き込みを待機しています。
ReorderLogicalMappingRead	並べ替えのバッファ管理の間に、論理マッピングの読み込みを待機しています。

IO待機イベント	説明
ReplicationSlotRead	レプリケーションスロットの制御ファイルからの読み込みを待機しています。
ReplicationSlotRestoreSync	レプリケーションスロットの制御ファイルをメモリにリストアする間、それが永続的ストレージに到達するのを待機しています。
ReplicationSlotSync	レプリケーションスロットの制御ファイルが永続的ストレージに到達するのを待機しています。
ReplicationSlotWrite	レプリケーションスロットの制御ファイルへの書き込みを待機しています。
SLRUFlushSync	チェックポイントまたはデータベースのシャットダウン中に、SLRUデータが永続的ストレージに到達するのを待機しています。
SLRURead	SLRUページの読み込みを待機しています。
SLRUSync	ページ書き込みの後、SLRUデータが永続的ストレージに到達するのを待機しています。
SLRUWrite	SLRUページの書き込みを待機しています。
SnapbuildRead	シリアライズされた通時的カタログのスナップショットの読み込みを待機しています。
SnapbuildSync	シリアライズされた通時的カタログのスナップショットが永続的ストレージに到達するのを待機しています。
SnapbuildWrite	シリアライズされた通時的カタログのスナップショットの書き込みを待機しています。
TimelineHistoryFileSync	ストリーミングレプリケーションを経由して受け取ったタイムラインの履歴ファイルが永続的ストレージに到達するのを待機しています。
TimelineHistoryFileWrite	ストリーミングレプリケーションを経由して受け取ったタイムラインの履歴ファイルの書き込みを待機しています。
TimelineHistoryRead	タイムラインの履歴ファイルの読み込みを待機しています。
TimelineHistorySync	新しく作成されたタイムラインの履歴ファイルが永続的ストレージに到達するのを待機しています。
TimelineHistoryWrite	新しく作成されたタイムラインの履歴ファイルの書き込みを待機しています。
TwophaseFileRead	二相の状態ファイルの読み込みを待機しています。
TwophaseFileSync	二相の状態ファイルが永続的ストレージに到達するのを待機しています。
TwophaseFileWrite	二相の状態ファイルの書き込みを待機しています。
WALBootstrapSync	ブートストラップ時にWALが永続的ストレージに到達するのを待機しています。
WALBootstrapWrite	ブートストラップ時にWALページの書き込みを待機しています。

IO待機イベント	説明
WALCopyRead	既存のWALセグメントをコピーして新しいWALセグメントを作成する時に読み込みを待機しています。
WALCopySync	既存のWALセグメントをコピーして作成した新しいWALセグメントが永続的ストレージに到達するのを待機しています。
WALCopyWrite	既存のWALセグメントをコピーして新しいWALセグメントを作成する時に書き込みを待機しています。
WALInitSync	新しく初期化されたWALファイルが永続的ストレージに到達するのを待機しています。
WALInitWrite	新しいWALファイルを初期化している時に書き込みを待機しています。
WALRead	WALファイルからの読み込みを待機しています。
WALSenderTimelineHistoryRead	WAL送信サーバのタイムラインコマンドで、タイムラインの履歴ファイルの読み込みを待機しています。
WALSync	WALファイルが永続的ストレージに達するのを待機しています。
WALSyncMethodAssign	新しいWALの同期方法を割り当てている時にデータが永続的ストレージに到達するのを待機しています。
WALWrite	WALファイルへの書き込みを待機しています。

表27.10 IPC型の待機イベント

IPC待機イベント	説明
BackupWaitWalArchive	バックアップに必要なWALファイルがアーカイブに成功するのを待機しています。
BgWorkerShutdown	バックグラウンドワーカがシャットダウンするのを待機しています。
BgWorkerStartup	バックグラウンドワーカが起動するのを待機しています。
BtreePage	パラレルB-treeスキャンを継続するのに必要なページ番号が利用可能になるのを待機しています。
CheckpointDone	チェックポイントが完了するのを待機しています。
CheckpointStart	チェックポイントが開始するのを待機しています。
ExecuteGather	Gather計画ノードの実行時に子プロセスの活動を待機しています。
HashBatchAllocate	選ばれたパラレルハッシュ参加者がハッシュテーブルを獲得するのを待機しています。
HashBatchElect	ハッシュテーブルを獲得するパラレルハッシュ参加者を選ぶのを待機しています。
HashBatchLoad	他のパラレルハッシュ参加者がハッシュテーブルのロードを完了させるのを待機しています。
HashBuildAllocate	選ばれたパラレルハッシュ参加者が初期ハッシュテーブルを獲得するのを待機しています。

IPC待機イベント	説明
HashBuildElect	初期ハッシュテーブルを獲得するパラレルハッシュ参加者を選ぶのを待機しています。
HashBuildHashInner	他のパラレルハッシュ参加者がインナーリレーションのハッシュを完了させるのを待機しています。
HashBuildHashOuter	他のパラレルハッシュ参加者がアウターリレーションのパーティショニングを完了させるのを待機しています。
HashGrowBatchesAllocate	選ばれたパラレルハッシュ参加者が追加バッチを獲得するのを待機しています。
HashGrowBatchesDecide	将来のバッチの増加を決めるパラレルハッシュ参加者を選ぶのを待機しています。
HashGrowBatchesElect	追加バッチを獲得するパラレルハッシュ参加者を選ぶのを待機しています。
HashGrowBatchesFinish	選ばれたパラレルハッシュ参加者が将来のバッチの増加を決めるのを待機しています。
HashGrowBatchesRepartition	他のパラレルハッシュ参加者がリパーティショニングを完了させるのを待機しています。
HashGrowBucketsAllocate	選ばれたパラレルハッシュ参加者が追加バケット獲得を完了するのを待機しています。
HashGrowBucketsElect	追加バケットを獲得するパラレルハッシュ参加者を選ぶのを待機しています。
HashGrowBucketsReinsert	他のパラレルハッシュ参加者が新しいバケットに対するタプル挿入を完了させるのを待機しています。
LogicalSyncData	論理レプリケーションのリモートサーバが最初のテーブル同期のためのデータを送信するのを待機しています。
LogicalSyncStateChange	論理レプリケーションのリモートサーバが状態を変更するのを待機しています。
MessageQueueInternal	他のプロセスが共有メッセージキューにアタッチされるのを待機しています。
MessageQueuePutMessage	共有メッセージキューにプロトコルのメッセージを書くのを待機しています。
MessageQueueReceive	共有メッセージキューからバイトを受信するのを待機しています。
MessageQueueSend	共有メッセージキューにバイトを送信するのを待機しています。
ParallelBitmapScan	パラレルビットマップスキャンが初期化されるのを待機しています。
ParallelCreateIndexScan	パラレルCREATE INDEXワーカがヒープスキャンを完了するのを待機しています。
ParallelFinish	パラレルワーカが計算を完了するのを待機しています。

IPC待機イベント	説明
ProcArrayGroupUpdate	グループリーダーが並列操作の最後にトランザクションIDをクリアするのを待機しています。
ProcSignalBarrier	バックエンドすべてでバリアイベントが処理されるのを待機しています。
Promote	スタンバイの昇格を待機しています。
RecoveryConflictSnapshot	バキュームクリーンアップに対するリカバリ競合の解決を待機しています。
RecoveryConflictTablespace	テーブル空間の削除に対するリカバリ競合の解決を待機しています。
RecoveryPause	リカバリが再開するのを待機しています。
ReplicationOriginDrop	レプリケーションオリジンが削除できるよう非活動状態になるのを待機しています。
ReplicationSlotDrop	レプリケーションスロットが削除できるよう非活動状態になるのを待機しています。
SafeSnapshot	READ ONLY DEFERRABLEのトランザクションに対する有効なスナップショットの獲得を待機しています。
SyncRep	同期レプリケーション中に、リモートサーバからの確認を待機しています。
XactGroupUpdate	グループリーダーが並列操作の最後にトランザクション状態を更新するのを待機しています。

表27.11 Lock型の待機イベント

Lock待機イベント	説明
advisory	勧告的ユーザロックを獲得するのを待機しています。
extend	リレーションを拡張するのを待機しています。
frozenid	pg_database.datfrozenxidとpg_database.datminmxidを更新するのを待機しています。
object	非リレーションデータベースオブジェクト上のロックを獲得するのを待機しています。
page	リレーションのページ上のロックを獲得するのを待機しています。
relation	リレーション上のロックを獲得するのを待機しています。
spectoken	投機的挿入ロックを獲得するのを待機しています。
transactionid	トランザクションが終了するのを待機しています。
tuple	タプル上のロックを獲得するのを待機しています。
userlock	ユーザロックを獲得するのを待機しています。
virtualxid	仮想トランザクションIDロックを獲得するのを待機しています。

表27.12 LWLock型の待機イベント

LWLock待機イベント	説明
AddinShmemInit	共有メモリの拡張の領域確保を管理するのを待機しています。
AutoFile	postgresql.auto.confファイルを更新するのを待機しています。
Autovacuum	自動バキュームワーカの現在の状態の読み込み、または更新を待機しています。
AutovacuumSchedule	自動バキューム対象として選定されたテーブルが、まだバキューム処理が必要であることを確認するのを待機しています。
BackgroundWorker	バックグラウンドワーカ状態の読み込み、または更新を待機しています。
BtreeVacuum	B-treeインデックスのバキュームに関連した情報の読み込み、または更新を待機しています。
BufferContent	メモリ内のデータページへアクセスするのを待機しています。
BufferIO	データページ上のI/Oを待機しています。
BufferMapping	データブロックをバッファプール内のバッファと関連付けるのを待機しています。
Checkpoint	チェックポイントを開始するのを待機しています。
CheckpointerComm	fsyncリクエストを管理するのを待機しています。
CommitTs	トランザクションコミットタイムスタンプのために設定された最新の値の読み込み、または更新を待機しています。
CommitTsBuffer	コミットタイムスタンプSLRUバッファでのI/Oを待機しています。
CommitTsSLRU	コミットタイムスタンプSLRUキャッシュにアクセスするのを待機しています。
ControlFile	pg_controlファイルの読み込みもしくは更新、または新しいWALファイルの作成を待機しています。
DynamicSharedMemoryControl	動的共有メモリの割り当て情報の読み込み、または更新を待機しています。
LockFastPath	プロセスのファストパスロック情報の読み込み、または更新を待機しています。
LockManager	「重量」ロックに関する情報の読み込み、または更新を待機しています。
LogicalRepWorker	論理レプリケーションワーカの状態の読み込み、または更新を待機しています。
MultiXactGen	共有マルチトランザクション状態の読み込み、または更新を待機しています。
MultiXactMemberBuffer	マルチトランザクションメンバSLRUバッファでのI/Oを待機しています。
MultiXactMemberSLRU	マルチトランザクションメンバSLRUキャッシュにアクセスするのを待機しています。

LWLock待機イベント	説明
MultiXactOffsetBuffer	マルチトランザクションオフセットSLRUバッファでのI/Oを待機しています。
MultiXactOffsetSLRU	マルチトランザクションオフセットSLRUキャッシュにアクセスするのを待機しています。
MultiXactTruncation	マルチトランザクション情報の読み込み、または切り詰めを待機しています。
NotifyBuffer	NOTIFYメッセージSLRUバッファでのI/Oを待機しています。
NotifyQueue	NOTIFYメッセージの読み込み、または更新を待機しています。
NotifyQueueTail	NOTIFYメッセージストレージの制限が更新されるのを待機しています。
NotifySLRU	NOTIFYメッセージSLRUキャッシュにアクセスするのを待機しています。
OidGen	新しいOIDを割り当てるのを待機しています。
OldSnapshotTimeMap	古いスナップショット制御情報の読み込み、または更新を待機しています。
ParallelAppend	パラレルアペンド計画を実行中に次のサブプランの選択を待機しています。
ParallelHashJoin	パラレルハッシュ結合計画を実行中に、ワーカの同期を待機しています。
ParallelQueryDSA	パラレルクエリの動的共有メモリ割り当てを待機しています。
PerSessionDSA	パラレルクエリの動的共有メモリ割り当てを待機しています。
PerSessionRecordType	複合型に関するパラレルクエリの情報にアクセスするのを待機しています。
PerSessionRecordTypmod	匿名レコード型を特定する型修飾子に関するパラレルクエリの情報にアクセスするのを待機しています。
PerXactPredicateList	パラレルクエリの間に、現在のシリアライザブルトランザクションによって保持された述語ロックの一覧へアクセスするのを待機しています。
PredicateLockManager	シリアライザブルトランザクションによって使われる述語ロックの情報にアクセスするのを待機しています。
ProcArray	(典型的には、スナップショットを得たりセッションのトランザクションIDを報告するために)共有のプロセスごとのデータ構造にアクセスするのを待機しています。
RelationMapping	(特定のシステムカタログのファイルノードの割り当てを追跡するのに使われる)pg_filenode.mapファイルの読み込み、または更新を待機しています。
RelCacheInit	pg_internal.initリレーションキャッシュ初期化ファイルの読み込み、または更新を待機しています。

LWLock待機イベント	説明
ReplicationOrigin	レプリケーションオリジンの作成、削除、または使用を待機しています。
ReplicationOriginState	あるレプリケーションオリジンの進捗の読み込み、または更新を待機しています。
ReplicationSlotAllocation	レプリケーションスロットの割り当て、または解放を待機しています。
ReplicationSlotControl	レプリケーションスロット状態の読み込み、または更新を待機しています。
ReplicationSlotIO	レプリケーションスロットでのI/Oを待機しています。
SerialBuffer	シリアライザブルトランザクション競合SLRUバッファでのI/Oを待機しています
SerializableFinishedList	完了したシリアライザブルトランザクションの一覧へアクセスするのを待機しています。
SerializablePredicateList	シリアライザブルトランザクションによって保持された述語ロックの一覧へアクセスするのを待機しています。
SerializableXactHash	シリアライザブルトランザクションに関する情報の読み込み、または更新を待機しています。
SerialSLRU	シリアライザブルトランザクション競合SLRUキャッシュにアクセスするのを待機しています。
SharedTidBitmap	パラレルビットマップインデックススキャンの間に、共有TIDにアクセスするのを待機しています。
SharedTupleStore	パラレルクエリの間に共有タプルストアにアクセスするのを待機しています。
ShmemIndex	共有メモリ内に領域を発見する、もしくは割り当てるのを待機しています。
SInvalRead	共有カタログ無効化キューからメッセージを取り出すのを待機しています。
SInvalWrite	共有カタログ無効化キューにメッセージを追加するのを待機しています。
SubtransBuffer	サブトランザクションSLRUバッファのI/Oを待機しています。
SubtransSLRU	サブトランザクションSLRUキャッシュにアクセスするのを待機しています。
SyncRep	同期レプリケーションの状態に関する情報を読み込む、または更新するの待機しています。
SyncScan	同期テーブルスキャンの開始位置を選ぶのを待機しています。
TablespaceCreate	テーブルスペースの作成、または削除を待機しています。
TwoPhaseState	プリペアドトランザクションの状態の読み込み、または更新を待機しています。
WALBufMapping	WALバッファ内のページの置き換えを待機しています。

LWLock待機イベント	説明
WALInsert	WALデータをメモリバッファに挿入するのを待機しています。
WALWrite	WALバッファがディスクに書き込まれるのを待機しています。
WrapLimitsVacuum	トランザクションIDとマルチトランザクションの消費の制限が更新されるのを待機しています。
XactBuffer	トランザクション状態SLRUバッファでのI/Oを待機しています。
XactSLRU	トランザクション状態SLRUキャッシュにアクセスするのを待機しています。
XactTruncation	pg_xact_statusを実行する、またはその関数で利用可能な最古のトランザクションIDを更新するのを待機しています。
XidGen	新しいトランザクションIDを割り当てるのを待機しています。

注記

拡張は表 27.12に示す一覧にLWLock型を追加できます。拡張によって割り当てられた名前がすべてのサーバプロセスでは利用可能でない場合があります。そのためLWLock待機イベントは、拡張が割り当てた名前ではなく単に「extension」と報告されるかもしれません。

表27.13 Timeout型の待機イベント

Timeout待機イベント	説明
BaseBackupThrottle	スロットル活動時にベースバックアップで待機しています。
PgSleep	pg_sleepまたは同系列の関数を呼び出したため待機しています。
RecoveryApplyDelay	遅延設定によりリカバリ時のWAL適用を待機しています。
RecoveryRetrieveRetryInterval	WALデータがまだあらゆる種類のソース(pg_wal、アーカイブまたはストリーム)から得られない時にリカバリで待機しています。
VacuumDelay	コストに基づくバキューム遅延ポイントで待機しています。

以下に、待機イベントが表示される例を示します。

```
SELECT pid, wait_event_type, wait_event FROM pg_stat_activity WHERE wait_event is NOT NULL;
 pid  | wait_event_type | wait_event
------+-----------------+------------
 2540 | Lock            | relation
 6644 | LWLock          | ProcArray
(2 rows)
```

27.2.4. pg_stat_replication

pg_stat_replicationビューには、WAL送信プロセス毎に、送信処理に接続したスタンバイサーバへのレプリケーションに関する統計情報を示す1行を保持します。直接接続されたスタンバイサーバのみが一覧表示されます。下流のスタンバイサーバに関する情報はありません。

表27.14 pg_stat_replicationビュー

列 型 説明
pid integer WAL送信プロセスのプロセスIDです。
usesysid oid WAL送信プロセスにログインしたユーザのOIDです。
usename name WAL送信プロセスにログインしたユーザの名前です。
application_name text WAL送信処理に接続したアプリケーションの名前です。
client_addr inet WAL送信処理に接続したクライアントのIPアドレスです。このフィールドがNULLの場合、クライアントがサーバマシン上のUnixソケット経由で接続したことを示します。
client_hostname text client_addrのDNS逆引き検索により報告された、接続クライアントのホスト名です。IP接続、かつlog_hostnameが有効である場合にのみ、このフィールドは非NULLになります。
client_port integer クライアントがWAL送信処理との通信に使用するTCPポート番号、もしUnixソケットを使用する場合は-1です。
backend_start timestamp with time zone プロセスが開始、つまりクライアントがWAL送信処理に接続した時刻です。
backend_xmin xid hot_standby_feedbackにより報告されたこのスタンバイのxminです。
state text WAL送信サーバの現在の状態です。取り得る値は以下の通りです。 • startup: このWAL送信サーバは起動するところです。 • catchup: このWAL送信サーバが接続しているスタンバイはプライマリに追いつこうとしています。 • streaming: このWAL送信サーバは、接続先のスタンバイサーバがプライマリに追いついた後、変更をストリームしています。 • backup: このWAL送信サーバはバックアップを送信しています。 • stopping: このWAL送信サーバは停止するところです。
sent_lsn pg_lsn この接続で送信された最後の先行書き込みログの位置です。
write_lsn pg_lsn このスタンバイサーバによってディスクに書き出された最後の先行書き込みログの位置です。
flush_lsn pg_lsn このスタンバイサーバによってディスクに吐き出された最後の先行書き込みログの位置です。
replay_lsn pg_lsn このスタンバイサーバ上のデータベースに再生された最後の先行書き込みログの位置です。

列 型 説明
write_lag interval 最近のWALをローカルに吐き出してから、このスタンバイサーバがそれを書き出した（が、まだ吐き出したり適用したりしていない）ことの通知を受け取るまでの経過時間です。このサーバが同期スタンバイとして設定されているとして、コミット時にsynchronous_commitレベルのremote_writeが起こした遅延を正確に測定するために、これを使用することができます。
flush_lag interval 最近のWALをローカルに吐き出してから、このスタンバイサーバがそれを書き出して吐き出した（が、まだ適用していない）ことの通知を受け取るまでの経過時間です。このサーバが同期スタンバイとして設定されているとして、コミット時にsynchronous_commitレベルのonが起こした遅延を正確に測定するために、これを使用することができます。
replay_lag interval 最近のWALをローカルに吐き出してから、このスタンバイサーバがそれを書き出し、吐き出し、そして適用したことの通知を受け取るまでの経過時間です。このサーバが同期スタンバイとして設定されているとして、コミット時にsynchronous_commitレベルのremote_applyが起こした遅延を正確に測定するために、これを使用することができます。
sync_priority integer 優先度に基づく同期レプリケーションで、このスタンバイサーバが同期スタンバイとして選択される優先度です。クォーラムに基づく同期レプリケーションでは効果がありません。
sync_state text このスタンバイサーバの同期状態です。取り得る値は以下の通りです。 • async: このスタンバイサーバは非同期です。 • potential: このスタンバイサーバは現在非同期ですが、現在同期中のサーバの一つが故障すると同期になる可能性があります。 • sync: このスタンバイサーバは同期です。 • quorum: このサーバはクォーラムのスタンバイの候補とみなされています。
reply_time timestamp with time zone スタンバイサーバから受け取った最後の応答メッセージの送信時刻です。

pg_stat_replicationビューで報告される経過時間は、最近のWALが書き込まれ、吐き出され、再生されるのに要した時間の測定結果であり、また、送信サーバがそれを知るためのものです。リモートサーバが同期スタンバイとして設定されている場合、これらの時間は、同期コミットの各レベルによって引き起こされた（あるいは引き起こされたであろう）コミットの遅延を表します。非同期スタンバイの場合は、replay_lag列は最近のトランザクションが問い合わせに対して可視になったときまでの遅延を近似します。スタンバイサーバが送信サーバに完全に追いつき、WALの活動がなくなった状態のときは、最も直近に測定された経過時間が短い間、表示され続け、その後はNULLとなります。

経過時間は物理レプリケーションの場合は自動的に機能します。ロジカルデコーディングのプラグインはオプションで追跡メッセージを発することができますが、そうしなければ追跡機能は単にNULLの経過時間を表示します。

> **注記**
>
> 報告される経過時間は、現在の再生速度の前提でスタンバイが送信サーバに追いつくのに要する時間を予測するものではありません。そのようなシステムでは、新しいWALが生成されている間は類似

した時間を示しますが、送信サーバがアイドル状態になると異なるものになるでしょう。特に、スタンバイが完全に追いついたとき、pg_stat_replicationは、一部のユーザが期待するゼロではなく、最も最近に報告されたWAL位置を書き込み、吐き出し、再生するのに要した時間を示します。これは最近の書き込みトランザクションについて同期コミットおよびトランザクションの可視性の遅延を測定するという目的と首尾一貫しています。経過時間について異なるモデルを期待するユーザの混乱を抑えるため、完全に再生されてアイドルになったシステムでは、経過時間の列は短い時間の後、NULLに戻ります。監視システムでは、これをデータなしとする、ゼロとする、あるいは最後の既知の値を表示し続けるという選択をすることになります。

27.2.5. pg_stat_wal_receiver

pg_stat_wal_receiverビューは、1行のみの形式で、受信サーバが接続したサーバからWALレシーバに関する統計情報を表示します。

表27.15 pg_stat_wal_receiverビュー

列 型 説明
pid integer WALレシーバプロセスのプロセスIDです。
status text WALレシーバプロセスの活動状態です。
receive_start_lsn pg_lsn WALレシーバが開始された時に使われる先行書き込みログの最初の位置です。
receive_start_tli integer WALレシーバが開始された時に使われる初期タイムライン番号です。
written_lsn pg_lsn すでに受信し、ディスクに書き出されたもののまだフラッシュされていない先行書き込みログの最新位置です。これはデータの完全性の確認のためには使うべきではありません。
flushed_lsn pg_lsn すでに受信し、ディスクにフラッシュされた先行書き込みログの最新位置です。この列の初期値は、WALレシーバが開始された時に使用される、最初のログ位置です。
received_tli integer 受信済みでディスクにフラッシュされた先行書き込みログの最新位置のタイムライン番号です。この列の初期値は、WALレシーバが開始された時に使用される、最初のログ位置のタイムライン番号です。
last_msg_send_time timestamp with time zone オリジンWAL送信サーバから受け取った最後のメッセージの送信時刻です。
last_msg_receipt_time timestamp with time zone オリジンWAL送信サーバから受け取った最後のメッセージの受信時刻です。
latest_end_lsn pg_lsn オリジンWAL送信サーバに最後に報告された先行書き込みログ位置です。
latest_end_time timestamp with time zone オリジンWAL送信サーバへ最新の先行書き込みログ位置が報告された時間です。

列 型 説明
slot_name text WALレシーバによって使用されたレプリケーションスロット名です。
sender_host text WALレシーバーが接続しているPostgreSQLインスタンスのホストです。これはホスト名、IPアドレス、あるいはUNIXソケットで接続している場合はディレクトリのパスです。（パスは、常に/で始まる絶対パスなので、パスであることを識別できます。）
sender_port integer WALレシーバーが接続しているPostgreSQLインスタンスのポート番号です。
conninfo text セキュリティに重要な値が難読化された文字列を含む、WALレシーバによって使用された接続文字列です。

27.2.6. pg_stat_subscription

pg_stat_subscriptionビューには、各サブスクリプションのメインワーカに対して1行が含まれ（ワーカが実行中でないときはPIDがNULLになります）、さらにサブスクライブされたテーブルの初期データコピーを処理するワーカについて別の行があります。

表27.16 pg_stat_subscriptionビュー

列 型 説明
subid oid サブスクリプションのOIDです。
subname name サブスクリプションの名前です。
pid integer サブスクリプションのワーカプロセスのプロセスIDです。
relid oid ワーカが同期しているリレーションのOIDです。メインの適用ワーカの場合はNULLです。
received_lsn pg_lsn 最後に受け取った先行書き込みログ位置です。このフィールドの初期値は0です。
last_msg_send_time timestamp with time zone オリジンWAL送信サーバから受け取った最後のメッセージの送信時刻です。
last_msg_receipt_time timestamp with time zone オリジンWAL送信サーバから受け取った最後のメッセージの受信時刻です。
latest_end_lsn pg_lsn オリジンWAL送信サーバに最後に報告された先行書き込みログ位置です。
latest_end_time timestamp with time zone オリジンWAL送信サーバに最後の先行書き込みログ位置が報告された時刻です。

27.2.7. pg_stat_ssl

pg_stat_sslビューは、バックエンドプロセスおよびWAL送信プロセスごとに1行を保持し、接続上でのSSLの使用に関する統計情報を示します。pg_stat_activityまたはpg_stat_replicationとpid列で結合することで、接続に関するより詳細な情報を取得できます。

表27.17 pg_stat_sslビュー

列 型 説明
pid integer バックエンドプロセスまたはWAL送信プロセスのプロセスIDです。
ssl boolean この接続でSSLが使用されていれば真になります。
version text 使用されているSSLのバージョンです。この接続でSSLが使用されていなければNULLになります。
cipher text 使用されているSSL暗号の名前です。この接続でSSLが使用されていなければNULLになります。
bits integer 使用されている暗号アルゴリズムのビット数です。この接続でSSLが使用されていなければNULLになります。
compression boolean SSL圧縮が使用されていれば真、使用されていなければ偽です。この接続でSSLが使用されていなければNULLになります。
client_dn text 使用されているクライアント証明書の識別名(DN)フィールドです。クライアント証明書が提供されなかった場合、およびこの接続でSSLが使用されていない場合はNULLになります。このフィールドは、DNフィールドがNAMEDATALEN(標準ビルドでは64文字)より長いと切り詰められます。
client_serial numeric クライアント証明書のシリアル番号です。この接続でクライアント証明書が提供されていないかSSLが使われていない場合にNULLになります。証明書のシリアル番号と証明書発行者の組み合わせは(発行者が誤ってシリアル番号を再使用しない限り)証明書を一意に識別します。
issuer_dn text クライアント証明書の発行者のDNです。この接続でクライアント証明書が提供されていないかSSLが使われていない場合にNULLになります。このフィールドはclient_dnと同様に切り詰められます。

27.2.8. pg_stat_gssapi

pg_stat_gssapiビューはバックエンド毎に1行で構成され、接続でのGSSAPI使用に関する情報を表示します。接続に関する更なる詳細を得るため、これをpg_stat_activityやpg_stat_replicationとpid列で結合できます。

表27.18 pg_stat_gssapiビュー

列 型 説明
pid integer バックエンドのプロセスIDです。

列 型 説明
gss_authenticated boolean この接続にGSSAPI認証が使われていたなら真です。
principal text この接続の認証に使われているプリンシパルです。接続の認証にGSSAPIが使われていない場合にはNULLです。このフィールドはプリンシパルがNAMEDATALEN(標準ビルドでは64文字)よりも長い場合には切り詰められます。
encrypted boolean この接続でGSSAPI暗号化が使われているなら真です。

27.2.9. pg_stat_archiver

pg_stat_archiverビューは常に、クラスタのアーカイバプロセスに関するデータを含む1つの行を持ちます。

表27.19 pg_stat_archiverビュー

列 型 説明
archived_count bigint アーカイブに成功したWALファイルの数です。
last_archived_wal text アーカイブに成功した最後のWALファイルの名前です。
last_archived_time timestamp with time zone 最後に成功したアーカイブ操作の時刻です。
failed_count bigint WALファイルのアーカイブに失敗した回数です。
last_failed_wal text 最後にアーカイブ操作に失敗したWALファイルの名前です。
last_failed_time timestamp with time zone 最後にアーカイブ操作に失敗した時刻です。
stats_reset timestamp with time zone 統計情報がリセットされた最終時刻です。

27.2.10. pg_stat_bgwriter

pg_stat_bgwriterビューは常に、クラスタのグローバルデータに関する1つの行を持ちます。

表27.20 pg_stat_bgwriterビュー

列 型 説明
checkpoints_timed bigint これまでに実行された、スケジュールされたチェックポイントの個数です。
checkpoints_req bigint

列 型 説明
これまでに実行された、要求されたチェックポイントの個数です。
checkpoint_write_time double precision チェックポイント処理におけるディスクにファイルを書き出す部分に費やされた、ミリ秒単位の総時間です。
checkpoint_sync_time double precision チェックポイント処理におけるディスクにファイルを同期する部分に費やされた、ミリ秒単位の総時間です。
buffers_checkpoint bigint チェックポイント期間に書き出されたバッファ数です。
buffers_clean bigint バックグラウンドライタにより書き出されたバッファ数です。
maxwritten_clean bigint バックグラウンドライタが書き出したバッファ数が多過ぎたために、整理用スキャンを停止した回数です。
buffers_backend bigint バックエンドにより直接書き出されたバッファ数です。
buffers_backend_fsync bigint バックエンドが独自にfsync呼び出しを実行しなければならなかった回数です。(通常は、バックエンドが独自に書き込んだ場合であっても、バックグラウンドライタがこれらを扱います。)
buffers_alloc bigint 割当られたバッファ数です。
stats_reset timestamp with time zone 統計情報がリセットされた最終時刻です。

27.2.11. pg_stat_database

pg_stat_databaseビューには、クラスタ内のデータベース毎に1行と加えて共有オブジェクトのための1行が含まれ、データベース全体の統計情報を示します。

表27.21 pg_stat_databaseビュー

列 型 説明
datid oid データベースのOIDです。共有リレーションに属するオブジェクトについては0になります。
datname name データベース名です。共有オブジェクトについてはNULLになります。
numbackends integer 現在データベースに接続しているバックエンドの個数です。共有オブジェクトについてはNULLになります。これは、このビューの中で、現在の状態を反映した値を返す唯一の列です。他の列はすべて、最後にリセットされてから累積された値を返します。
xact_commit bigint データベース内でコミットされたトランザクション数です。

列 型 説明
xact_rollback bigint データベース内でロールバックされたトランザクション数です。
blks_read bigint データベース内で読み取られたディスクブロック数です。
blks_hit bigint バッファキャッシュに既にあることが分かっているためにディスクブロックの読み取りが不要だった回数です（これにはPostgreSQLのバッファキャッシュにおけるヒットのみが含まれ、オペレーティングシステムのファイルシステムキャッシュは含まれません）。
tup_returned bigint データベース内の問い合わせで返された行数です。
tup_fetched bigint データベース内の問い合わせで取り出された行数です。
tup_inserted bigint データベース内の問い合わせで挿入された行数です。
tup_updated bigint データベース内の問い合わせで更新された行数です。
tup_deleted bigint データベース内の問い合わせで削除された行数です。
conflicts bigint データベース内のリカバリで競合したためキャンセルされた問い合わせ数です。(競合はスタンバイサーバ上でのみ起こります。詳細についてはpg_stat_database_conflictsを参照してください。)
temp_files bigint データベース内の問い合わせによって書き出された一時ファイルの個数です。一時ファイルが作成された理由（ソート処理やハッシュ処理）やlog_temp_filesの設定に関わらず、すべての一時ファイルが計上されます。
temp_bytes bigint データベース内の問い合わせによって一時ファイルに書き出されたデータ量です。一時ファイルが作成された理由やlog_temp_filesの設定に関わらず、すべての一時ファイルが計上されます。
deadlocks bigint データベース内で検知されたデッドロック数です。
checksum_failures bigint データベース（あるいは共有オブジェクト）内で検出されたデータページチェックサムの検査失敗数です。データチェックサムが無効の場合にはNULLです。
checksum_last_failure timestamp with time zone データベース（または共有オブジェクト）内で最後にデータページチェックサムの検査失敗が検知された時刻です。データチェックサムが無効の場合にはNULLです。
blk_read_time double precision データベース内でバックエンドによりデータファイルブロックの読み取りに費やされた、ミリ秒単位の時間です（track_io_timingが有効な場合。そうでなければゼロです）。
blk_write_time double precision

列 型 説明
データベース内でバックエンドによりデータファイルブロックの書き出しに費やされた、ミリ秒単位の時間です (track_io_timingが有効な場合。そうでなければゼロです)。
stats_reset timestamp with time zone 統計情報がリセットされた最終時刻です。

27.2.12. pg_stat_database_conflicts

pg_stat_database_conflictsビューは、データベース毎に1行を保持し、スタンバイサーバでのリカバリと競合するためにキャンセルされた問い合わせに関するデータベース全体の統計情報を示します。マスタサーバでは競合は発生しませんので、スタンバイサーバ上の情報のみが保持されます。

表27.22 pg_stat_database_conflictsビュー

列 型 説明
datid oid データベースのOIDです。
datname name データベースの名前です。
confl_tablespace bigint データベースにおいて、削除されたテーブル空間のためにキャンセルされた問い合わせの個数です。
confl_lock bigint データベースにおいて、ロック時間切れのためにキャンセルされた問い合わせの個数です。
confl_snapshot bigint データベースにおいて、古いスナップショットのためにキャンセルされた問い合わせの個数です。
confl_bufferpin bigint データベースにおいて、ピンが付いたバッファのためにキャンセルされた問い合わせの個数です。
confl_deadlock bigint データベースにおいて、デッドロックのためにキャンセルされた問い合わせの個数です。

27.2.13. pg_stat_all_tables

pg_stat_all_tablesビューは現在のデータベース内のテーブル(TOASTテーブルを含む)毎に1行の形式で、特定のテーブルへのアクセスに関する統計情報を表示します。pg_stat_user_tablesおよびpg_stat_sys_tablesビューにも同じ情報が含まれますが、それぞれユーザテーブルとシステムテーブルのみにフィルタされています。

表27.23 pg_stat_all_tablesビュー

列 型 説明
relid oid テーブルのOIDです。

列 型 説明
schemaname name テーブルが存在するスキーマの名前です。
relname name テーブルの名前です。
seq_scan bigint テーブル上で初期化されたシーケンシャルスキャンの個数です。
seq_tup_read bigint シーケンシャルスキャンによって取り出された有効行の個数です。
idx_scan bigint テーブル上で開始されたインデックススキャンの実行回数です。
idx_tup_fetch bigint インデックススキャンによって取り出された有効行の個数です。
n_tup_ins bigint 挿入された行数です。
n_tup_upd bigint 更新された行数です。(HOT更新された行数を含みます)
n_tup_del bigint 削除された行数です。
n_tup_hot_upd bigint HOT更新(つまりインデックスの更新を別途必要としない)された行数です。
n_live_tup bigint 有効行数の推定値です。
n_dead_tup bigint 無効行数の推定値です。
n_mod_since_analyze bigint このテーブルが最後に解析されてから変更された行数の推定値です。
n_ins_since_vacuum bigint このテーブルが最後にバキュームされてから挿入された行数の推定値です。
last_vacuum timestamp with time zone テーブルが手作業でバキュームされた最終時刻です(VACUUM FULLは含まれません)。
last_autovacuum timestamp with time zone 自動バキュームデーモンによりテーブルがバキュームされた最終時刻です。
last_analyze timestamp with time zone テーブルが手作業で解析された最終時刻です。
last_autoanalyze timestamp with time zone 自動バキュームデーモンによりテーブルが解析された最終時刻です。
vacuum_count bigint テーブルが手作業でバキュームされた回数です。(VACUUM FULLは含まれません)。

列 型 説明
autovacuum_count bigint テーブルが自動バキュームデーモンによりバキュームされた回数です。
analyze_count bigint テーブルが手作業で解析された回数です。
autoanalyze_count bigint テーブルが自動バキュームデーモンによって解析された回数です。

27.2.14. pg_stat_all_indexes

pg_stat_all_indexesビューは、現在のデータベース内のインデックス毎に、特定のインデックスへのアクセスに関する統計情報を示す1行を保持します。pg_stat_user_indexesとpg_stat_sys_indexesも同じ情報を保持しますが、ユーザ向けのインデックスとシステム向けのインデックスに対する行のみを保持するようにフィルタ処理されています。

表27.24 pg_stat_all_indexesビュー

列 型 説明
relid oid このインデックスに対応するテーブルのOIDです。
indexrelid oid インデックスのOIDです。
schemaname name インデックスが存在するスキーマの名前です。
relname name このインデックスに対応するテーブルの名前です。
indexrelname name インデックスの名前です。
idx_scan bigint インデックスに対して開始されたインデックススキャンの実行回数です。
idx_tup_read bigint インデックスに対するスキャンにより返されたインデックス項目の個数です。
idx_tup_fetch bigint インデックスを使用する単純なインデックススキャンによって取り出された有効テーブル行数です。

単純なインデックススキャン、「ビットマップ」インデックススキャン、あるいはオプティマイザによりインデックスが使用されることがあります。ビットマップスキャンでは、複数のインデックスの出力をANDやOR規則で組み合わせることができます。このため、ビットマップスキャンが使用される場合、特定インデックスと個々のヒープ行の取り出しとを関連づけることが困難です。したがってビットマップスキャンでは、使用したインデックスのpg_stat_all_indexes.idx_tup_read個数を増やし、そのテーブルのpg_stat_all_tables.idx_tup_fetch個数を増やしますが、pg_stat_all_indexes.idx_tup_fetchを変更

しません。オプティマイザもインデックスにアクセスし、提供された定数値がオプティマイザの統計情報に記録された範囲の外側にあるときに、それを検査します。これはオプティマイザの統計情報が古いかもしれないからです。

> **注記**
>
> idx_tup_readとidx_tup_fetch個数は、ビットマップスキャンがまったく使用されていない場合でも異なります。idx_tup_readはインデックスから取り出したインデックス項目を計上し、idx_tup_fetchはテーブルから取り出した有効行を計上するからです。インデックスを用いて無効行やまだコミットされていない行が取り出された場合やインデックスオンリースキャン法によりヒープの取り出しが回避された場合に、後者は減少します。

27.2.15. pg_statio_all_tables

pg_statio_all_tablesビューは現在のデータベース内のテーブル（TOASTテーブルを含む）ごとに、特定のテーブルのI/Oに関する統計情報を示す1行を保持します。pg_statio_user_tablesとpg_statio_sys_tablesには同じ情報が保持されますが、ユーザテーブルとシステムテーブルに関する行のみを持つようにフィルタ処理がなされています。

表27.25 pg_statio_all_tablesビュー

列 型 説明
relid oid テーブルのOIDです。
schemaname name テーブルが存在するスキーマの名前です。
relname name テーブルの名前です。
heap_blks_read bigint テーブルから読み取られたディスクブロック数です。
heap_blks_hit bigint テーブル内のバッファヒット数です。
idx_blks_read bigint テーブル上のすべてのインデックスから読み取られたディスクブロック数です。
idx_blks_hit bigint テーブル上のすべてのインデックス内のバッファヒット数です。
toast_blks_read bigint テーブルのTOASTテーブル（もしあれば）から読み取られたディスクブロック数です。
toast_blks_hit bigint テーブルのTOASTテーブル（もしあれば）におけるバッファヒット数です。
tidx_blks_read bigint テーブルのTOASTテーブルのインデックス（もしあれば）から読み取られたディスクブロック数です。

列 型 説明
tidx_blks_hit bigint テーブルのTOASTテーブルのインデックス（もしあれば）におけるバッファヒット数です。

27.2.16. pg_statio_all_indexes

pg_statio_all_indexesビューは、現在のデータベース内のインデックス毎に、特定のインデックスへのI/Oに関する統計情報を持つ1行を保持します。pg_statio_user_indexesとpg_statio_sys_indexesも同じ情報を保持しますが、それぞれユーザ向けのインデックスとシステム向けのインデックスに対する行のみを保持するようにフィルタ処理されています。

表27.26 pg_statio_all_indexesビュー

列 型 説明
relid oid このインデックスに対応するテーブルのOIDです。
indexrelid oid インデックスのOIDです。
schemaname name インデックスが存在するスキーマの名前です。
relname name このインデックスに対応するテーブルの名前です。
indexrelname name インデックスの名前です。
idx_blks_read bigint インデックスから読み取られたディスクブロック数です。
idx_blks_hit bigint インデックスにおけるバッファヒット数です。

27.2.17. pg_statio_all_sequences

pg_statio_all_sequencesビューは現在のデータベース内のシーケンスごとに、特定シーケンスにおけるI/Oに関する統計情報を示す1行を保持します。

表27.27 pg_statio_all_sequencesビュー

列 型 説明
relid oid シーケンスのOIDです。
schemaname name

列 型 説明
シーケンスが存在するスキーマの名前です。
relname name シーケンスの名前です。
blks_read bigint シーケンスから読み取られたディスクブロック数です。
blks_hit bigint シーケンスにおけるバッファヒット数です。

27.2.18. pg_stat_user_functions

pg_stat_user_functionsビューは追跡された関数毎に、その関数の実行に関する統計情報を1行保持します。track_functionsパラメータは関数が追跡されるかどうかを正確に制御します。

表27.28 pg_stat_user_functionsビュー

列 型 説明
funcid oid 関数のOIDです。
schemaname name 関数が存在するスキーマの名前です。
funcname name 関数の名前です。
calls bigint 関数が呼び出された回数です。
total_time double precision 関数とその関数から呼び出されるその他の関数で費やされた、ミリ秒単位の総時間です。
self_time double precision その関数から呼び出されるその他の関数で費やされた時間を含まない、関数自身で費やされた、ミリ秒単位の総時間です。

27.2.19. pg_stat_slru

PostgreSQLは*SLRU*(simple least-recently-used)キャッシュ経由で特定のディスク上の情報にアクセスします。pg_stat_slruビューは、追跡されたSLRUキャッシュ毎にキャッシュされたページへのアクセスに関する統計情報を1行保持します。

表27.29 pg_stat_slruビュー

列 型 説明
name text

列 型 説明
SLRUの名前です。
blks_zeroed bigint 初期化中にゼロにされたブロックの数です。
blks_hit bigint SLRUに既にあることが分かっているためにディスクブロックの読み取りが不要だった回数です（これにはSLRUにおけるヒットのみが含まれ、オペレーティングシステムのファイルシステムキャッシュは含まれません）。
blks_read bigint SLRUから読み取られたディスクブロック数です。
blks_written bigint SLRUに書き込まれたディスクブロック数です。
blks_exists bigint SLRUで存在を検査されたブロック数です。
flushes bigint SLRUでのダーティデータのフラッシュ数です。
truncates bigint SLRUでの切り詰めの数です。
stats_reset timestamp with time zone 統計情報がリセットされた最終時刻です。

27.2.20. 統計情報関数

統計情報を参照する他の方法は、上述の標準ビューによって使用される基礎的な統計情報アクセス関数と同じ関数を使用した問い合わせを作成することで設定することができます。こうした関数の名前などに関する詳細については、標準ビューの定義を参照してください。（例えばpsqlでは\d+ pg_stat_activityを発行してください。）データベースごとの統計情報についてのアクセス関数は、どのデータベースに対して報告するのかを識別するためにデータベースのOIDを取ります。テーブルごと、インデックスごとの関数はテーブルの、もしくはインデックスのOIDを取ります。関数ごとの統計情報の関数は、関数のOIDを取ります。これらの関数を使用して参照できるテーブルとインデックス、および関数は現在のデータベース内のものだけであることに注意してください。

その他の統計情報収集に関連した関数を表 27.30に示します。

表27.30 その他の統計情報関数

関数 説明
pg_backend_pid () → integer 現在のセッションにアタッチされたサーバプロセスのプロセスIDを返します。
pg_stat_get_activity (integer) → setof record 指定されたプロセスIDに該当するバックエンドの情報のレコードを、NULLが指定された場合はシステム上のアクティブな各バックエンドに関するレコードを返します。返される情報内容はpg_stat_activityの一部と同じです。

関数 説明
pg_stat_get_snapshot_timestamp () → timestamp with time zone 現在の統計情報のスナップショットのタイムスタンプを返します。
pg_stat_clear_snapshot () → void 現在の統計情報スナップショットを破棄します。
pg_stat_reset () → void 現在のデータベースに関する統計情報カウンタすべてをゼロにリセットします。 この関数はデフォルトでスーパーユーザに限定されていますが、関数を実行できるように他のユーザにEXECUTE権限を付与できます。
pg_stat_reset_shared (text) → void 引数に応じて、クラスタ全体の統計情報カウンタの一部をゼロにリセットします。引数にbgwriterを指定すると、pg_stat_bgwriterビューで示されるカウンタがすべてリセットされ、archiverを指定するとpg_stat_archiverビューで示されるカウンタがすべてリセットされます。 この関数はデフォルトでスーパーユーザに限定されていますが、関数を実行できるように他のユーザにEXECUTE権限を付与できます。
pg_stat_reset_single_table_counters (oid) → void 現在のデータベース内にある、ひとつのテーブルあるいはインデックスの統計情報をゼロにリセットします。 この関数はデフォルトでスーパーユーザに限定されていますが、関数を実行できるように他のユーザにEXECUTE権限を付与できます。
pg_stat_reset_single_function_counters (oid) → void 現在のデータベース内にある、ひとつの関数の統計情報をゼロにリセットします。 この関数はデフォルトでスーパーユーザに限定されていますが、関数を実行できるように他のユーザにEXECUTE権限を付与できます。
pg_stat_reset_slru (text) → void ひとつのSLRUキャッシュ、またはクラスタ内のすべてのSLRUの統計情報をゼロにリセットします。引数がNULLであれば、pg_stat_slruビューで示されているすべてのSLRUキャッシュに対するカウンタがリセットされます。引数は、そのエントリのみに対応するカウンタをリセットするようCommitTs、MultiXactMember、MultiXactOffset、Notify、Serial、Subtrans、Xactの1つを指定できます。引数がother(実際のところは、認められていない名前であれば何でも)であれば、拡張が定義したキャッシュのような、それ以外のSLRUキャッシュに対するカウンタがリセットされます。 この関数はデフォルトでスーパーユーザに限定されていますが、関数を実行できるように他のユーザにEXECUTE権限を付与できます。

pg_stat_activityビューの基礎となるpg_stat_get_activity関数は、各バックエンドプロセスに関して利用可能な情報をすべて含むレコード集合を返します。この情報の一部のみを入手することがより簡便である場合があるかもしれません。このような場合、表 27.31に示す、古めのバックエンド単位の統計情報アクセス関数を使用することができます。これらのアクセス関数は、1から現在活動中のバックエンドの個数までの値を取る、バックエンドID番号を使用します。pg_stat_get_backend_idset関数は、これらの関数を呼び出すために、活動中のバックエンド毎に1行を生成する簡便な方法を提供します。例えば以下はすべてのバックエンドについてPIDと現在の問い合わせを示します。

```
SELECT pg_stat_get_backend_pid(s.backendid) AS pid,
       pg_stat_get_backend_activity(s.backendid) AS query
    FROM (SELECT pg_stat_get_backend_idset() AS backendid) AS s;
```

表27.31 バックエンド単位の統計情報関数

関数 説明
pg_stat_get_backend_idset () → setof integer 現在活動中のバックエンドID番号（1から活動中のバックエンドの個数まで）の集合を返します。
pg_stat_get_backend_activity (integer) → text バックエンドが最後に行った問い合わせテキストを返します。
pg_stat_get_backend_activity_start (integer) → timestamp with time zone バックエンドの最後の問い合わせが開始された時刻を返します。
pg_stat_get_backend_client_addr (integer) → inet バックエンドに接続したクライアントのIPアドレスを返します。
pg_stat_get_backend_client_port (integer) → integer クライアントが通信に使用しているTCPポート番号を返します。
pg_stat_get_backend_dbid (integer) → oid バックエンドが接続するデータベースのOIDを返します。
pg_stat_get_backend_pid (integer) → integer バックエンドのプロセスIDを返します。
pg_stat_get_backend_start (integer) → timestamp with time zone プロセスが開始された時刻を返します。
pg_stat_get_backend_userid (integer) → oid バックエンドにログインしたユーザのOIDを返します。
pg_stat_get_backend_wait_event_type (integer) → text バックエンドが現在待機中であれば、待機イベント型名を、さもなくばNULLを返します。詳細は表 27.4を参照してください。
pg_stat_get_backend_wait_event (integer) → text バックエンドが現在待機中であれば、待機イベント名を、さもなくばNULLを返します。詳細は表 27.5から表 27.13までを参照してください。
pg_stat_get_backend_xact_start (integer) → timestamp with time zone バックエンドの現在のトランザクションが開始された時刻を返します。

27.3. ロックの表示

この他に、データベース活動状況の監視に役立つツールとしてpg_locksシステムテーブルがあります。これにより、データベース管理者はロックマネージャ内の未解決のロックに関する情報を参照することができます。例えば、この機能を使用すると以下のことができます。

- 現在未解決のロック、特定データベース内のリレーション上のロック、特定のリレーションのロック、または特定のPostgreSQLセッションが保持するロックを全て表示する。
- 最も許可されにくいロック（データベースクライアント間で競合の原因になる可能性がある）を持つ、現在のデータベースにおけるリレーションを表示する。

- 競合によって変動するデータベースの全トラフィックの範囲に加えて、全体的なデータベースの性能に対するロック競合の影響を判断する。

pg_locksビューの詳細は、51.73にあります。PostgreSQLのロックと同時実行性についての詳細は、第13章を参照してください。

27.4. 進捗状況のレポート

PostgreSQLは、何らかのコマンドの実行中に進捗状況をレポートする能力があります。現在、進捗状況のレポートをサポートしているのは、ANALYZE、CLUSTER、CREATE INDEX、VACUUM、および、BASE_BACKUP(すなわち、pg_basebackupがベースバックアップのために発行するレプリケーションコマンド)のみです。将来的にサポートされるコマンドが拡大される可能性があります。

27.4.1. ANALYZEの進捗状況のレポート

ANALYZEが実行されているときにはいつでも、pg_stat_progress_analyzeビューには現在コマンドを実行している各バックエンドごとの行が含まれます。以下の表は、報告される情報を説明し、どのように解釈するかの情報を提供します。

表27.32 pg_stat_progress_analyzeビュー

列 型 説明
pid integer バックエンドのプロセスIDです。
datid oid バックエンドが接続されているデータベースのOIDです。
datname name バックエンドが接続されているデータベース名です。
relid oid 解析されているテーブルのOIDです。
phase text 現在処理中のフェーズです。表 27.33を参照してください。
sample_blks_total bigint サンプルされるヒープブロックの総数です。
sample_blks_scanned bigint スキャンされたヒープブロックの数です。
ext_stats_total bigint 拡張統計情報の個数です。
ext_stats_computed bigint 計算された拡張統計情報の個数です。このカウンタはフェーズがcomputing extended statisticsの時にのみ増加します。
child_tables_total bigint 子テーブルの数です。

列 型 説明
child_tables_done bigint スキャンされた子テーブルの数です。このカウンタはフェーズがacquiring inherited sample rowsの時にのみ増加します。
current_child_table_relid oid 現在スキャンされている子テーブルのOIDです。このフィールドはフェーズがacquiring inherited sample rowsの時のみ有効です。

表27.33 ANALYZEのフェーズ

フェーズ	説明
initializing	コマンドはヒープをスキャンし始める準備をしています。このフェーズは非常に短時間であると予想されます。
acquiring sample rows	コマンドはサンプル行を得るため、relidで指定されたテーブルを現在スキャンしています。
acquiring inherited sample rows	コマンドはサンプル行を得るため、子テーブルを現在スキャンしています。列child_tables_total、child_tables_done、current_child_table_relidはこのフェーズの進捗情報を含みます。
computing statistics	コマンドはテーブルスキャンの間に得られたサンプルから統計情報を計算しています。
computing extended statistics	コマンドはテーブルスキャンの間に得られたサンプルから拡張統計情報を計算しています。
finalizing analyze	コマンドはpg_classを更新しています。このフェーズが完了すれば、ANALYZEは終わります。

> **注記**
>
> ANALYZEがパーティションテーブルで実行される場合は、そのパーティションテーブルのすべてもANALYZEに書かれているように再帰的に解析されることに注意してください。その場合、ANALYZEの進捗はまず親テーブルについて報告され、それによってその継承の統計情報が集められ、各パーティションの報告が続きます。

27.4.2. CREATE INDEXの進捗状況のレポート

CREATE INDEXやREINDEXが実行中であるときにはいつでも、pg_stat_progress_create_indexビューには現在インデックスを作成している各バックエンドごとの行が含まれます。以下の表は、報告される情報を説明し、どのように解釈するかの情報を提供します。

表27.34 pg_stat_progress_create_indexビュー

列 型 説明
pid integer バックエンドのプロセスIDです。

列 型 説明
datid oid バックエンドが接続されているデータベースのOIDです。
datname name バックエンドが接続されているデータベースの名前です。
relid oid インデックスが作られているテーブルのOIDです。
index_relid oid 作成または再作成されているインデックスのOIDです。同時作成ではないCREATE INDEXのときは、これは0です。
command text 実行しているコマンドで、CREATE INDEX、CREATE INDEX CONCURRENTLY、REINDEXまたはREINDEX CONCURRENTLYです。
phase text 現在処理中のインデックス作成のフェーズです。表 27.35を参照してください。
lockers_total bigint 該当するときに、待機するロック取得者の総数です。
lockers_done bigint 既に待機したロック取得者の数です。
current_locker_pid bigint 現在待機しているロック取得者のプロセスIDです。
blocks_total bigint 現在のフェーズで処理されることになっているブロックの総数です。
blocks_done bigint 現在のフェーズで既に処理されたブロック数です。
tuples_total bigint 現在のフェーズで処理されることになっているタプルの総数です。
tuples_done bigint 現在のフェーズで既に処理されたタプル数です。
partitions_total bigint パーティションテーブル上のインデックスを作成するとき、この列にはインデックスが作られることになっているパーティションの総数が設定されます。
partitions_done bigint パーティションテーブル上のインデックスを作成するとき、この列にはインデックス作成が既に完了しているパーティションの数が設定されます。

表27.35 CREATE INDEXのフェーズ

フェーズ	説明
initializing	CREATE INDEXやREINDEXはインデックスを作る準備をしています。このフェーズはごく短時間になると予想されます。
waiting for writers before build	CREATE INDEX CONCURRENTLYやREINDEX CONCURRENTLYは、潜在的にテーブルを参照するかもしれない書き込みロックを伴うトランザクションが終了するのを待機しています。

フェーズ	説明
	本フェーズは同時モードでないときには省かれます。列lockers_total、lockers_done、および、current_locker_pidには本フェーズの進行情報が入ります。
building index	インデックスがアクセスメソッド固有のコードにより作成されています。本フェーズでは、進捗レポートをサポートするアクセスメソッドが自身の進捗データを記入し、また、サブフェーズはこの列で示されます。典型的には、blocks_totalとblocks_doneが、さらにあるいはtuples_totalとtuples_doneも、進捗データを含みます。
waiting for writers before validation	CREATE INDEX CONCURRENTLYやREINDEX CONCURRENTLYは、潜在的にテーブルに書き込みするかもしれない書き込みロックを伴うトランザクションが終了するのを待機しています。本フェーズは同時モードでないときには省かれます。列lockers_total、lockers_done、および、current_locker_pidには本フェーズの進行情報が入ります。
index validation: scanning index	CREATE INDEX CONCURRENTLYは確認が必要なタプルに対するインデックス検索をスキャンしています。本フェーズは同時モードでないときには省かれます。列blocks_total(インデックスの総サイズが設定される)とblocks_doneに本フェーズの進行情報が入ります。
index validation: sorting tuples	CREATE INDEX CONCURRENTLYはインデックスをスキャンするフェーズ(scanning index)の出力をソートしています。
index validation: scanning table	CREATE INDEX CONCURRENTLYは、前の2フェーズで収集されたインデックスのタプルを確認するためテーブルをスキャンしています。本フェーズは同時モードでないときには省かれます。列blocks_total(テーブルの総サイズが設定される)とblocks_doneに本フェーズの進行情報が入ります。
waiting for old snapshots	CREATE INDEX CONCURRENTLYやREINDEX CONCURRENTLYは、潜在的にテーブルを参照するかもしれないトランザクションがそれらのスナップショットを解放するのを待機しています。本フェーズは同時モードでないときには省かれます。列lockers_total、lockers_done、および、current_locker_pidには本フェーズの進行情報が入ります。
waiting for readers before marking dead	REINDEX CONCURRENTLYは、古いインデックスに無効と印付けする前に、テーブルへの読み取りロックを伴うトランザクションが終了するのを待機しています。本フェーズは同時モードでないときには省かれます。列lockers_total、lockers_done、および、current_locker_pidには本フェーズの進行情報が入ります。
waiting for readers before dropping	REINDEX CONCURRENTLYは、古いインデックスを削除する前に、テーブルへの読み取りロックを伴うトランザクションが終了するのを待機しています。本フェーズは同時モードでないときには省かれます。列lockers_total、lockers_done、および、current_locker_pidには本フェーズの進行情報が入ります。

27.4.3. VACUUMの進捗状況のレポート

VACUUMを実行するときはいつでも、pg_stat_progress_vacuumビューは、現在バキューム処理している(自動バキュームワーカプロセスを含む)それぞれのバックエンドごとに1行を格納します。以下の表は、報告される情報を説明し、どのように解釈するかの情報を提供します。VACUUM FULLコマンドの進捗はpg_stat_progress_clusterでレポートされます。これは、通常のVACUUMはテーブル内を書き換えするのみである一方、VACUUM FULLとCLUSTERはいずれもテーブルを再作成するためです。27.4.4を参照してください。

表27.36 pg_stat_progress_vacuumビュー

列 型 説明
pid integer バックエンドのプロセスIDです。
datid oid バックエンドが接続されているデータベースのOIDです。
datname name バックエンドが接続されているデータベースの名前です。
relid oid バキューム処理が行われているテーブルのOIDです。
phase text 現在処理しているバキュームのフェーズです。表 27.37を参照してください。
heap_blks_total bigint テーブルのヒープブロックの総数です。この数字は、スキャンの開始を基点としてレポートされます。後に追加されるブロックは、このVACUUMによって処理されません(必要もありません)。
heap_blks_scanned bigint スキャンされたヒープブロックの数です。可視性マップがスキャンを最適化するために使用されるため、いくつかのブロックが検査されずに読み飛ばされます。読み飛ばされたブロックはこの総数に含まれ、そのためこの数字はバキューム処理が完了した時に、最終的にheap_blks_totalと同じになります。このカウンタは、フェーズがscanning heapの時にのみ増加します。
heap_blks_vacuumed bigint バキューム処理されたヒープブロックの数です。テーブルにインデックスが1つでも存在するなら、このカウンタはフェーズがvacuuming heapの時にのみ増加します。無効なタプルが含まれていないブロックは読み飛ばされ、それゆえカウンタは時々大きな増加量で早送りされます。
index_vacuum_count bigint 完了したインデックスバキュームサイクルの数です。
max_dead_tuples bigint インデックスバキュームサイクルの実行に必要となる前に格納できる、maintenance_work_memに基づいた、無効なタプルの数です。
num_dead_tuples bigint 最後のインデックスバキュームサイクルから収集された無効タプルの数です。

表27.37 VACUUMのフェーズ

フェーズ	説明
initializing	VACUUMは、ヒープをスキャンし始める準備をしています。このフェーズは、非常に短時間であると予想されます。
scanning heap	VACUUMは、現在ヒープをスキャン中です。必要であればそれぞれのページを切り取り、デフラグし、場合によってはフリーズ活動を実行します。スキャンの進捗状況の監視にheap_blks_scanned列が使用できます。
vacuuming indexes	VACUUMは、現在インデックスをバキューム処理中です。テーブルにインデックスが存在する場合、ヒープが完全にスキャンされた後に、バキューム実行ごとに少なくとも1回発生

フェーズ	説明
	します。maintenance_work_memが、発見された無効タプルの数量を格納するのに不十分な場合は、バキューム実行ごとに複数回発生する可能性があります。
vacuuming heap	VACUUMは、現在ヒープをバキューム処理中です。ヒープのバキュームは、ヒープのスキャンと異なり、インデックスをバキューム処理するそれぞれのインスタンスの後に発生します。heap_blks_scannedがheap_blks_totalより少ない場合、システムはこのフェーズの完了後にヒープのスキャン処理に戻ります。さもなければ、このフェーズの完了後にインデックスの整理を始めます。
cleaning up indexes	VACUUMは、現在インデックスの整理処理中です。これは、ヒープが完全にスキャンされ、インデックスとヒープが完全にすべてバキューム処理された後に発生します。
truncating heap	VACUUMは、現在リレーションの終点の空のページをオペレーティングシステムに戻すためにヒープを切り詰めています。これは、インデックスの整理処理後に発生します。
performing final cleanup	VACUUMは、最終クリーンアップ処理をしています。このフェーズの間、VACUUM空き領域マップをバキュームし、pg_classの統計情報を更新し、統計情報コレクタに統計情報を報告します。このフェーズが完了した時、VACUUMは終了します。

27.4.4. CLUSTERの進捗状況のレポート

CLUSTERやVACUUM FULLが実行されているときにはいつでも、pg_stat_progress_clusterビューには現在いずれかのコマンドを実行している各バックエンドごとの行が含まれます。以下の表は、報告される情報を説明し、どのように解釈するかの情報を提供します。

表27.38 pg_stat_progress_clusterビュー

列 型 説明
pid integer バックエンドのプロセスIDです。
datid oid バックエンドが接続されているデータベースのOIDです。
datname name バックエンドが接続されているデータベースの名前です。
relid oid クラスタ化されているテーブルのOIDです。
command text 実行しているコマンドです。CLUSTERかVACUUM FULLのいずれかです。
phase text 現在処理しているフェーズです。表 27.39を参照してください。
cluster_index_relid oid テーブルがインデックスを使ってスキャンされているのであれば、これは使われているインデックスのOIDで、さもなくばゼロです。
heap_tuples_scanned bigint

列 型 説明
スキャンされたヒープタプルの数です。このカウンタは、フェーズがseq scanning heap、index scanning heap、または、writing new heapであるときのみ増加します。
heap_tuples_written bigint 書かれたヒープタプルの数です。このカウンタは、フェーズがseq scanning heap、index scanning heap、または、writing new heapであるときのみ増加します。
heap_blks_total bigint テーブル内のヒープブロックの総数です。この数にはseq scanning heapの開始時の値が報告されます。
heap_blks_scanned bigint スキャンされたヒープブロックの数です。このカウンタは、フェーズがseq scanning heapであるときのみ増加します。
index_rebuild_count bigint インデックス再作成の数です。このカウンタはフェーズがrebuilding indexであるときのみ増加します。

表27.39 CLUSTERとVACUUM FULLのフェーズ

フェーズ	説明
initializing	コマンドはヒープのスキャンを開始する準備をしています。本フェーズはごく短時間になると予想されます。
seq scanning heap	コマンドは現在、テーブルをシーケンシャルスキャンを使ってスキャンしています。
index scanning heap	CLUSTERは現在、インデックススキャンを使ってテーブルをスキャンしています。
sorting tuples	CLUSTERは現在、タプルをソートしています。
writing new heap	CLUSTERが新しいヒープに書き込んでいます。
swapping relation files	コマンドは現在、新たに構築したファイルを置き換えて設置しています。
rebuilding index	コマンドは現在、インデックスを再構築しています。
performing final cleanup	コマンドは現在、最終クリーンアップを実行中です。このフェーズが完了すると、CLUSTERやVACUUM FULLは終了します。

27.4.5. ベースバックアップの進捗状況のレポート

pg_basebackupのようなアプリケーションがベースバックアップを取る時はいつでも、pg_stat_progress_basebackupビューには現在BASE_BACKUPレプリケーションコマンドを実行し、バックアップをストリームしている各WAL送信プロセスごとの行が含まれます。以下の表は、報告される情報を説明し、どのように解釈するかの情報を提供します。

表27.40 pg_stat_progress_basebackupビュー

列 型 説明
pid integer WAL送信プロセスのプロセスIDです。
phase text 現在処理中のフェーズです。表 27.41を参照してください。

列 型 説明
backup_total bigint ストリームされるデータの総量です。これは推定され、streaming database filesフェーズの最初に報告されます。データベースはstreaming database filesフェーズの間に変化するかもしれませんし、WALログが後ほどバックアップに含められますので、これは近似でしかないことに注意してください。ストリームされたデータ量が推定された総量を超えたら、これは常にbackup_streamedと同じ値です。pg_basebackupで推定が無効にされて(すなわち、--no-estimate-sizeオプションが指定されて)いれば、NULLです。
backup_streamed bigint ストリームされるデータの量です。このカウンタはフェーズがstreaming database filesまたはtransferring wal filesの時にのみ増加します。
tablespaces_total bigint ストリームされるテーブル空間の総数です。
tablespaces_streamed bigint ストリームされたテーブル空間の数です。このカウンタはフェーズがstreaming database filesの時にのみ増加します。

表27.41 ベースバックアップのフェーズ

フェーズ	説明
initializing	WAL送信プロセスはバックアップを開始する準備をしています。このフェーズはごく短時間になると予想されます。
waiting for checkpoint to finish	WAL送信プロセスは、ベースバックアップを取る準備をするために現在pg_start_backupを実行し、バックアップ開始チェックポイントが完了するのを待っています。
estimating backup size	WAL送信プロセスは、ベースバックアップとしてストリームされるデータベースファイルの総量を現在推定しています。
streaming database files	WAL送信プロセスはデータベースファイルをベースバックアップとして現在ストリームしています。
waiting for wal archiving to finish	WAL送信プロセスは、バックアップを終了するために現在pg_stop_backupを実行し、ベースバックアップに必要なWALファイルすべてのアーカイブに成功するのを待っています。pg_basebackupで--wal-method=noneか--wal-method=streamのいずれかが指定されれば、バックアップはこのフェーズが完了したら終了します。
transferring wal files	WAL送信プロセスはバックアップ中に生成されたWALログをすべて現在転送しています。pg_basebackupで--wal-method=fetchが指定されていれば、このフェーズがwaiting for wal archiving to finishの次に来ます。バックアップはこのフェーズが完了したら終了します。

27.5. 動的追跡

PostgreSQLは、データベースサーバの動的追跡をサポートする機能を提供します。これにより、外部ユーティリティをコードの特定のポイントで呼び出すことができ、追跡を行うことができるようになります。

多くの追跡やプローブ用のポイントは、すでにソースコード内部に存在します。これらのプローブはデータベースの開発者や管理者が使うことを意図しています。デフォルトでは、これらのプローブはPostgreSQLに

コンパイルされません。ユーザは明示的にconfigureスクリプトでプローブを有効にするように設定する必要があります。

現在、これを書いている時点ではSolaris、macOS、FreeBSD、NetBSD、Oracle Linuxで利用可能なDTrace[1]ユーティリティがサポートされています。SystemTap[2]プロジェクトではDTrace相当の機能をLinux向けに提供しており、それを使うこともできます。他の動的追跡ユーティリティのサポートは、src/include/utils/probes.h内のマクロ定義を変更することで、理論上は可能です。

27.5.1. 動的追跡のためのコンパイル

デフォルトでは、プローブは有効ではありません。そのため、PostgreSQLでプローブが利用できるようにするためにconfigureスクリプトで明示的に設定しなければなりません。DTraceサポートを含めるには、configureに--enable-dtraceを指定します。詳細は16.4を参照してください。

27.5.2. 組み込み済みのプローブ

表 27.42で示されるように、多くの標準的なプローブがソースコード内で提供されています。表 27.43はプローブで使用している型を示しています。また、PostgreSQL内の可観測性を強化するためのプローブ追加が可能です。

表27.42 組み込み済みのDTraceプローブ

名前	パラメータ	説明
transaction-start	(LocalTransactionId)	新しいトランザクションの開始を捕捉するプローブです。arg0はトランザクションIDです。
transaction-commit	(LocalTransactionId)	トランザクションの正常終了を捕捉するプローブです。arg0はトランザクションIDです。
transaction-abort	(LocalTransactionId)	トランザクションの異常終了を捕捉するプローブです。arg0はトランザクションIDです。
query-start	(const char *)	問い合わせ処理の開始を捕捉するプローブです。arg0は問い合わせ文字列です。
query-done	(const char *)	問い合わせ処理の正常終了を捕捉するプローブです。arg0は問い合わせ文字列です。
query-parse-start	(const char *)	問い合わせのパース処理の開始を捕捉するプローブです。arg0は問い合わせ文字列です。
query-parse-done	(const char *)	問い合わせのパース処理の正常終了を捕捉するプローブです。arg0は問い合わせ文字列です。
query-rewrite-start	(const char *)	問い合わせの書き換え処理の開始を捕捉するプローブです。arg0は問い合わせ文字列です。
query-rewrite-done	(const char *)	問い合わせの書き換え処理の正常終了を捕捉するプローブです。arg0は問い合わせ文字列です。

[1] https://en.wikipedia.org/wiki/DTrace

[2] https://sourceware.org/systemtap/

名前	パラメータ	説明
query-plan-start	()	問い合わせのプランナ処理の開始を捕捉するプローブです。
query-plan-done	()	問い合わせのプランナ処理の正常終了を捕捉するプローブです。
query-execute-start	()	問い合わせの実行(エクゼキュータ)処理の開始を捕捉するプローブです。
query-execute-done	()	問い合わせの実行(エクゼキュータ)処理の正常終了を捕捉するプローブです。
statement-status	(const char *)	サーバプロセスによるpg_stat_activity.statusの状態の更新を捕捉するプローブです。arg0は新しい状態の文字列です。
checkpoint-start	(int)	チェックポイントの開始を捕捉するプローブです。arg0はチェックポイントの種類の違い(shutdown、immediate、force)を区別するためのビットフラグを持っています。
checkpoint-done	(int, int, int, int, int)	チェックポイントの正常終了を捕捉するプローブです。(以下に示すプローブはチェックポイント進行に従い順番に捕捉されます。) arg0は書き込まれたバッファ数、arg1はバッファの総数、arg2、3、4はそれぞれ追加、削除、再利用されたWALファイルの数です。
clog-checkpoint-start	(bool)	CLOG部分のチェックポイントの開始を捕捉するプローブです。arg0がtrueならば通常のチェックポイントであり、falseならばシャットダウン時のチェックポイントを示します。
clog-checkpoint-done	(bool)	CLOG部分のチェックポイントの正常終了を捕捉するプローブです。arg0はclog-checkpoint-startと同じ意味を持ちます。
subtrans-checkpoint-start	(bool)	サブトランザクション部分のチェックポイントの開始を捕捉するプローブです。arg0がtrueならば通常のチェックポイントであり、falseならばシャットダウン時のチェックポイントを示します。
subtrans-checkpoint-done	(bool)	サブトランザクション部分のチェックポイントの正常終了を捕捉するプローブです。arg0はsubtrans-checkpoint-startと同じ意味を持ちます。
multixact-checkpoint-start	(bool)	マルチトランザクション部分のチェックポイントの開始を捕捉するプローブです。arg0がtrueならば通常のチェックポイントであり、falseならばシャットダウン時のチェックポイントを示します。
multixact-checkpoint-done	(bool)	マルチトランザクション部分のチェックポイントの正常終了を捕捉するプローブです。arg0はmu

名前	パラメータ	説明
		ltixact-checkpoint-startと同じ意味を持ちます。
buffer-checkpoint-start	(int)	チェックポイントのバッファ書き込み部分の開始を捕捉するプローブです。arg0はチェックポイントの種類の違い(shutdown、immediate、force)を区別するためのビットフラグを持っています。
buffer-sync-start	(int, int)	チェックポイント中のダーティバッファの書き出し開始を捕捉するプローブです(どのバッファが書き出す必要があるのかを判定した後です)。arg0はバッファの総数で、arg1は現在ダーティであり、書き出す必要のあるバッファ数です。
buffer-sync-written	(int)	チェックポイント中のそれぞれのバッファの書き出し後を捕捉するプローブです。arg0はバッファのIDを示します。
buffer-sync-done	(int, int, int)	全てのダーティバッファの書き出し後を捕捉するプローブです。arg0はバッファの総数です。arg1はチェックポイント処理により実際に書き出されたバッファ数です。arg2は書き出されるであろうと見積もられたバッファ数(buffer-sync-startのarg1相当)です。違いはチェックポイント中に他のプロセスがバッファを書き出したことを反映しています。
buffer-checkpoint-sync-start	()	カーネルへのダーティバッファの書き出し処理発行の後、そして同期書き出し要求を開始する前を捕捉するプローブです。
buffer-checkpoint-done	()	バッファからディスクへの同期書き出し処理の終了を捕捉するプローブです。
twophase-checkpoint-start	()	二相コミット部分のチェックポイントの開始を捕捉するプローブです。
twophase-checkpoint-done	()	二相コミット部分のチェックポイントの正常終了を捕捉するプローブです。
buffer-read-start	(ForkNumber, BlockNumber, Oid, Oid, Oid, int, bool)	バッファ読み込みの開始を捕捉するプローブです。arg0とarg1は読み込みページのフォーク番号とブロック番号です(ただし、リレーションの拡張要求であれば、arg1は-1になるでしょう)。arg2、arg3、arg4は対象のリレーションを識別するテーブル空間、データベース、そしてリレーションのOIDです。arg5は一時テーブルをローカルバッファに作成していればそのバックエンドのIDであり、InvalidBackendId(-1)であれは共有バッファを指します。arg6はtrueならばリレーションの拡張要求、falseは通常の読み込みを示します。

名前	パラメータ	説明
buffer-read-done	(ForkNumber, BlockNumber, Oid, Oid, Oid, int, bool, bool)	バッファ読み込みの終了を捕捉するプローブです。arg0とarg1は読み込みページのフォーク番号とブロック番号です(もしリレーションの拡張要求であれば、arg1は新たに追加されたブロックの番号を含みます)。arg2、arg3、arg4は対象のテーブルを識別するテーブル空間、データベース、そしてテーブルのOIDです。arg5は一時テーブルをローカルバッファに作成していればそのバックエンドのIDであり、InvalidBackendId(-1)であれは共有バッファを指します。arg6はtrueならばリレーションの拡張要求、falseは通常の読み込みを示します。arg7はtrueならばバッファがプール内にある、falseはプール内に無かったことを示します。
buffer-flush-start	(ForkNumber, BlockNumber, Oid, Oid, Oid)	共有バッファへの書き込み要求開始を捕捉するプローブです。arg0とarg1はそのページのフォーク番号とブロック番号です。arg2、arg3、arg4は対象のリレーションを識別するテーブル空間、データベース、そしてテーブルのOIDです。
buffer-flush-done	(ForkNumber, BlockNumber, Oid, Oid, Oid)	書き込み要求の終了を捕捉するプローブです。(これはカーネルへデータを渡したタイミングのみを反映していることに注意してください。大抵、この時点ではまだ実際にディスクへ書き込まれていません。) 引数はbuffer-flush-startと同じです。
buffer-write-dirty-start	(ForkNumber, BlockNumber, Oid, Oid, Oid)	サーバプロセスによるダーティバッファの書き出し開始を捕捉するプローブです。(もしこれが頻発するようでしたら、shared_buffersが少な過ぎるか、バックグラウンドライタ制御のパラメータの調節が必要なことを意味します。) arg0とarg1はそのページのフォーク番号とブロック番号です。arg2、arg3、arg4は対象のリレーションを識別するテーブル空間、データベース、そしてリレーションのOIDです。
buffer-write-dirty-done	(ForkNumber, BlockNumber, Oid, Oid, Oid)	ダーティバッファの書き出しの終了を捕捉するプローブです。引数はbuffer-write-dirty-startと同じです。
wal-buffer-write-dirty-start	()	WALバッファ領域の不足によるサーバプロセスのダーティなWALバッファの書き出しを捕捉するプローブです。(もしこれが頻発するようでしたら、wal_buffersが小さすぎることを意味します。)
wal-buffer-write-dirty-done	()	ダーティなWALバッファの書き出し終了を捕捉するプローブです。

名前	パラメータ	説明
wal-insert	(unsigned char, unsigned char)	WALレコードの挿入を捕捉するプローブです。arg0はレコードのリソースマネージャ(rmid)です。arg1は情報フラグです。
wal-switch	()	WALセグメントのスイッチ要求を捕捉するプローブです。
smgr-md-read-start	(ForkNumber, BlockNumber, Oid, Oid, Oid, int)	リレーションからのブロック読み込みの開始を捕捉するプローブ。arg0とarg1はそのページのフォーク番号とブロック番号です。arg2、arg3、arg4は対象のリレーションを識別するテーブル空間、データベース、そしてリレーションのOIDです。arg5は一時テーブルをローカルバッファに作成していればそのバックエンドのIDであり、InvalidBackendId(-1)であれは共有バッファを指します。
smgr-md-read-done	(ForkNumber, BlockNumber, Oid, Oid, Oid, int, int, int)	ブロックの読み込み終了を捕捉するプローブです。arg0とarg1はそのページのフォーク番号とブロック番号です。arg2、arg3、arg4は対象のリレーションを識別するテーブル空間、データベース、そしてリレーションのOIDです。arg5は一時テーブルをローカルバッファに作成していればそのバックエンドのIDであり、InvalidBackendId(-1)であれは共有バッファを指します。arg6は実際に読み込んだバイト数、arg7はリクエストされた読み込みバイト数です(もし、これらに差異があった場合、何らかの問題があることを示します)。
smgr-md-write-start	(ForkNumber, BlockNumber, Oid, Oid, Oid, int)	リレーションへのブロック書き出しの開始を捕捉するプローブです。arg0とarg1はそのページのフォーク番号とブロック番号です。arg2、arg3、arg4は対象のリレーションを識別するテーブル空間、データベース、そしてリレーションのOIDです。arg5は一時テーブルをローカルバッファに作成していればそのバックエンドのIDであり、InvalidBackendId(-1)であれは共有バッファを指します。
smgr-md-write-done	(ForkNumber, BlockNumber, Oid, Oid, Oid, int, int, int)	ブロックの書き出し終了を捕捉するプローブです。arg0とarg1はそのページのフォーク番号とブロック番号です。arg2、arg3、arg4は対象のリレーションを識別するテーブル空間、データベース、そしてリレーションのOIDです。arg5は一時テーブルをローカルバッファに作成していればそのバックエンドのIDであり、InvalidBackendId(-1)であれは共有バッファを指します。arg6は実際に書き出したバイト数、arg7はリクエストされた書き出しバイト数です(もし、これらに差異があった場合、何らかの問題があることを示します)。

名前	パラメータ	説明
sort-start	(int, bool, int, int, bool, int)	ソート処理の開始を捕捉するプローブです。arg0は対象データがヒープ、インデックス、またはdatumのどれかを示します。arg1はtrueならば一意性を必要としていることを示します。arg2はカラムのキー数です。arg3は許容されている作業メモリ(work_mem)のキロバイト数です。arg4はtrueならばソート結果に対するランダムアクセスが要求されていることを示します。arg5は、0ならばシリアル、1ならばパラレルワーカ、2ならばパラレルリーダーであることを示します。
sort-done	(bool, long)	ソート処理の終了を捕捉するプローブです。arg0はtrueならば外部ソート、falseは内部ソートを示します。arg1は外部ソートで使用されたディスクブロック数、もしくは内部ソートで使用されたメモリーのキロバイト数を示します。
lwlock-acquire	(char *, LWLockMode)	LWLockの獲得を捕捉するプローブです。arg0はLWLockのトランシェを示します。arg1は要求されたロックモード(排他または共有)を示します。
lwlock-release	(char *)	LWLockの解放を捕捉するプローブです(ただし、解放された待機状態のものにはまだ通知されていないことに注意してください)。arg0はLWLockのトランシェを示します。
lwlock-wait-start	(char *, LWLockMode)	LWLockが即座には獲得できず、ロックが利用可能になるまでサーバプロセスが待機を開始したことを捕捉するプローブです。arg0はLWLockのトランシェを示します。arg1は要求されたロックモード(排他または共有)を示します。
lwlock-wait-done	(char *, LWLockMode)	サーバプロセスがLWLockの待機から解放されたことを捕捉するプローブです(まだ実際にはロックを取得していません)。arg0はLWLockのトランシェを示します。arg1は要求されたロックモード(排他または共有)を示します。
lwlock-condacquire	(char *, LWLockMode)	呼び出し元が待機しないことを指定した際の、LWLockの獲得成功を捕捉するプローブです。arg0はLWLockのトランシェを示します。arg1は要求されたロックモード(排他または共有)を示します。
lwlock-condacquire-fail	(char *, LWLockMode)	呼び出し元が待機しないことを指定した際の、LWLockの獲得失敗を捕捉するプローブです。arg0はLWLockのトランシェを示します。arg1は要求されたロックモード(排他または共有)を示します。
lock-wait-start	(unsigned int, unsigned int, unsigned int, unsigned int, unsigned int, LOCKMODE)	重量ロック(lmgr lock)を即座に取得できなかったため、サーバプロセスがロックを利用できるま

名前	パラメータ	説明
		でロック待ち状態になった際の開始を捕捉するプローブです。arg0からarg3はロックされたオブジェクトの識別用タグ領域です。arg4はロックされたオブジェクトのタイプを示します。arg5は要求されたロックの種類を示します。
lock-wait-done	(unsigned int, unsigned int, unsigned int, unsigned int, unsigned int, LOCKMODE)	重量ロック(lmgr lock)要求の待機終了を捕捉するプローブです(つまりロックを取得した)。引数はlock-wait-startと同じです。
deadlock-found	()	デッドロック検知器によるデッドロックの発見を捕捉するプローブです。

表27.43 プローブパラメータで使われる型の定義

型	定義
LocalTransactionId	unsigned int
LWLockMode	int
LOCKMODE	int
BlockNumber	unsigned int
Oid	unsigned int
ForkNumber	int
bool	unsigned char

27.5.3. プローブの利用

以下の例では、性能試験前後でpg_stat_databaseのスナップショットを取る代わりに、システムにおけるトランザクション数を解析するDTraceスクリプトを示します。

```
#!/usr/sbin/dtrace -qs

postgresql$1:::transaction-start
{
      @start["Start"] = count();
      self->ts  = timestamp;
}

postgresql$1:::transaction-abort
{
      @abort["Abort"] = count();
}

postgresql$1:::transaction-commit
/self->ts/
```

```
{
      @commit["Commit"] = count();
      @time["Total time (ns)"] = sum(timestamp - self->ts);
      self->ts=0;
}
```

実行すると、例のDスクリプトは以下のような出力をします。

```
# ./txn_count.d `pgrep -n postgres` or ./txn_count.d <PID>
^C

Start                                          71
Commit                                         70
Total time (ns)                          2312105013
```

> **注記**
>
> 基本となる追跡ポイントの互換性はありますが、SystemTapはDTraceと異なる追跡スクリプトの表記を用います。表記に関して特に注意すべき点として、SystemTapでは参照する追跡ポイント名のハイフンの代わりに二重のアンダースコアを用いる必要があります。これは将来的なSystemTapのリリースで修正されることを期待しています。

DTraceスクリプトの作成には注意が必要であり、デバッグが必要であることは忘れないでください。さもないと、収集される追跡情報の意味がなくなるかもしれません。ほとんどの場合、見つかる問題はシステムではなく使用方法の間違いです。動的追跡を使用して見つかった情報に関して議論を行う際には、スクリプトの検査や議論もできるようにスクリプトも含めるようにしてください。

27.5.4. 新規プローブの定義

開発者が望めばコード内に新しくプローブを定義することができます。しかし、これには再コンパイルが必要です。下記は、新規プローブの定義の手順です。

1. プローブの名前とプローブの処理を通じて取得可能とするデータを決めます
2. src/backend/utils/probes.dにプローブの定義を追加します
3. もし、プローブポイントを含むモジュールがpg_trace.hをインクルードしていなければそれをインクルードし、ソースコード中のプローブを行いたい場所にTRACE_POSTGRESQLマクロを挿入します
4. 再コンパイルを行い、新規プローブが利用できるか確認します

例: これはトランザクションIDを用いて新規トランザクションを追跡するプローブ追加の仕方の例です。

1. プローブ名をtransaction-startとし、LocalTransactionId型のパラメータを必要とすることを決めます。
2. src/backend/utils/probes.dにプローブの定義を追加します:

```
probe transaction__start(LocalTransactionId);
```

プローブ名に二重のアンダースコアを使用する場合は注意してください。DTraceスクリプトでプローブを用いる場合、二重のアンダースコアをハイフンに置き換える必要があります。そのため、transaction-startがユーザ向けの文書に記載される名前となります。

3. コンパイル時に、transaction__startはTRACE_POSTGRESQL_TRANSACTION_STARTと呼ばれるマクロに変換されます(ここではアンダースコアはひとつになります)。このマクロは、pg_trace.hをインクルードすることにより使用可能となります。このマクロをソースコード中の適切な箇所へ追加していきます。この場合、以下の様になります。

```
TRACE_POSTGRESQL_TRANSACTION_START(vxid.localTransactionId);
```

4. 再コンパイル後に新しいバイナリでサーバを起動し、下記の様なDTraceコマンドの実行により新たに追加したプローブが利用可能かチェックします。下記の様な出力が確認できるはずです:

```
# dtrace -ln transaction-start
   ID    PROVIDER          MODULE           FUNCTION NAME
18705 postgresql49878     postgres     StartTransactionCommand transaction-start
18755 postgresql49877     postgres     StartTransactionCommand transaction-start
18805 postgresql49876     postgres     StartTransactionCommand transaction-start
18855 postgresql49875     postgres     StartTransactionCommand transaction-start
18986 postgresql49873     postgres     StartTransactionCommand transaction-start
```

Cのソースコードに追跡用のマクロを追加する際、いくつかの注意点があります:

- プローブの引数に指定したデータ型がマクロで使用される変数のデータ型と一致するよう注意しなければなりません。でなければ、コンパイル時にエラーとなるでしょう。

- ほとんどのプラットフォームでは、もしPostgreSQLが--enable-dtrace付きでビルドされた場合、何の追跡もされなかったとしても、制御がマクロを通過する際はいつでも追跡用マクロの引数が評価されます。ごく少数のローカルな変数を報告するような場合はそれほど心配はいりません。ただし、高価な関数呼び出しを引数にする場合は注意してください。もしそのようにする必要がある場合、追跡が実際に有効かどうかをチェックしてマクロを保護することを考慮してください:

```
if (TRACE_POSTGRESQL_TRANSACTION_START_ENABLED())
    TRACE_POSTGRESQL_TRANSACTION_START(some_function(...));
```

各追跡マクロは対応するENABLEDマクロを持っています。

第28章 ディスク使用量の監視

本章ではPostgreSQLデータベースシステムのディスク使用量を監視する方法について説明します。

28.1. ディスク使用量の決定

各テーブルには、データの大部分が格納されるプライマリヒープディスクファイルが備わっています。もしテーブルが、長くなる可能性のある値を持つ列を持つ時は、テーブルに関連付けられたTOASTファイルもあるかもしれません。このファイルは、メインテーブルに収納するには大き過ぎる値をテーブルに格納するために使用されます（68.2を参照してください）。TOASTテーブルが存在する場合は、そのテーブルに有効なインデックスが1つあります。基本テーブルに関連付けられたインデックスが存在することもあります。テーブルとインデックスはそれぞれ別のディスクファイルに格納されます。このファイルが1ギガバイトを超える場合は、複数のファイルになります。これらのファイルの命名規約について68.1で説明します。

ディスクスペースの監視は、次の3つの方法で行えます。表 9.90にあるSQL関数を使用する方法とoid2nameモジュールを使用する方法、およびシステムカタログを手動で調べる方法です。SQL関数を使用する方法が、一般的に一番簡単な方法です。本セクションの残りの部分で、システムカタログを調査することによりこの方法を示します。

バキュームされて間もないデータベース、もしくは解析されたデータベース上でpsqlを使用することにより、どのようなテーブルでもディスクの使用量を調べる問い合わせを発行できます。

```
SELECT pg_relation_filepath(oid), relpages FROM pg_class WHERE relname = 'customer';

 pg_relation_filepath | relpages
----------------------+----------
 base/16384/16806     |       60
(1 row)
```

1ページは通常8キロバイトです（relpagesはVACUUMとANALYZE、さらにCREATE INDEXといったいくつかのDDLによってのみ更新されることに注意してください）。もしテーブルのディスクファイルを直接調べるときは、ファイルのパス名称に注目して下さい。

TOASTテーブルで使用されている容量を示すには、以下のような問い合わせを使用してください。

```
SELECT relname, relpages
FROM pg_class,
     (SELECT reltoastrelid
      FROM pg_class
      WHERE relname = 'customer') AS ss
WHERE oid = ss.reltoastrelid OR
      oid = (SELECT indexrelid
             FROM pg_index
             WHERE indrelid = ss.reltoastrelid)
ORDER BY relname;
```

```
       relname        | relpages
----------------------+----------
 pg_toast_16806       |        0
 pg_toast_16806_index |        1
```

インデックスサイズについても、以下のように簡単に表示できます。

```
SELECT c2.relname, c2.relpages
FROM pg_class c, pg_class c2, pg_index i
WHERE c.relname = 'customer' AND
      c.oid = i.indrelid AND
      c2.oid = i.indexrelid
ORDER BY c2.relname;

      relname      | relpages
-------------------+----------
 customer_id_index |       26
```

この情報を使用して、以下のように簡単に最大のテーブルとインデックスを見つけ出すことができます。

```
SELECT relname, relpages
FROM pg_class
ORDER BY relpages DESC;

       relname        | relpages
----------------------+----------
 bigtable             |     3290
 customer             |     3144
```

28.2. ディスク容量不足による問題

データベース管理者の最も重要なディスク監視作業は、ディスクが容量不足になっていないことを確認することです。容量不足となったディスクにより、データが破壊されることはありません。しかし、データ破壊が起こりやすくなる可能性があります。もしWALファイルを持つディスクが一杯になった場合は、データベースサーバはパニックを起こし結果的にシャットダウンします。

他のデータを削除しても、ディスクに空き容量を用意できない場合、テーブル空間を使用することによって、 データベースファイルのいくつかを 他のファイルシステムに移動させることができます。詳細は 22.6を参照してください。

> **ヒント**
>
> 一部のファイルシステムは、容量がほぼ一杯になっている場合にパフォーマンスが悪くなります。ですから、ディスクがほぼ一杯になる前に余裕をもって対策を取ってください。

システムでユーザ単位のディスククォータをサポートしている場合、当然ながらデータベースもサーバを実行するユーザに割り当てられたクォータに従います。クォータを超えた場合、ディスク容量が完全になくなった時と同じ悪影響が発生します。

第29章 信頼性とログ先行書き込み

本章では、効率的かつ信頼できる運用を得るためにログ先行書き込みがどのように使用されているかについて説明します。

29.1. 信頼性

信頼性は、すべての本格的なデータベースシステムで重要な特性です。PostgreSQLは信頼できる操作を保証するためにできることは何でもします。信頼できる操作の一面は、コミットされたトランザクションにより記録されたデータはすべて不揮発性の領域に格納され、電源断、オペレーティングシステムの障害、ハードウェアの障害（当然ですが、不揮発性の領域自体の障害は除きます。）があっても安全であるという点です。通常、コンピュータの永続的格納領域（ディスク装置など）へのデータ書き込みの成功がこの条件を満たします。実際、コンピュータに致命的な障害が発生したとしても、もしディスク装置が無事ならば、類似のハードウェアを持つ別のコンピュータに移すことができ、コミットされたトランザクションを元通りに復元できます。

データを周期的にディスクプラッタに書き出すことは簡単な操作に思われるかもしれませんが、そうではありません。ディスク装置は主メモリ、CPU、コンピュータの主メモリとディスクプラッタの間にある各種のキャッシュ層と比べ非常に低速であるからです。まず、オペレーティングシステムのバッファキャッシュが存在します。これは頻繁にアクセス要求があるディスクブロックをキャッシュし、ディスクへの書き込みをまとめます。好運にもすべてのオペレーティングシステムがバッファキャッシュをディスクに強制書き込みさせる方法をアプリケーションに提供しています。PostgreSQLはこの機能を使用します。（これを調整する方法についてはwal_sync_methodパラメータを参照してください。）

次に、ディスク装置のコントローラキャッシュが存在する可能性があります。特に、RAIDコントローラカードでは、これは一般的です。これらの中には*write-through*キャッシュがあり、つまり、データが届いた時に即座に書き込みがディスク装置に対して行なわれます。他には*write-back*キャッシュがあり、多少遅れて書き込みがディスク装置に対して行なわれます。こうしたキャッシュでは、ディスクコントローラキャッシュが揮発性で、電源障害の際にその内容が失われてしまい、信頼性に関して致命的な問題になる可能性があります。より優れたコントローラカードにはバッテリバックアップ付き装置(BBUs)があり、システムの電源が落ちた場合もキャッシュに電源を供給します。後で電源が復旧した後に、データがディスク装置に書き出されます。

最後に、ほとんどのディスク装置がキャッシュを持っています。一部はwrite-throughであり、一部はwrite-backです。ディスクコントローラキャッシュの場合と同様にwrite-backのディスク装置キャッシュの場合にはデータが損失する恐れがあります。一般消費者向けのIDEおよびSATA装置では、電源障害時にデータが残らないwrite-backキャッシュを使用している可能性がとりわけ高いです。多くのソリッドステートドライブ(SSD)も同様に揮発性のwrite-backキャッシュを持っています。

これらのキャッシュは、大抵は無効にできます。しかしながらオペレーティングシステムやドライブの種類によってその方法は異なります。

- Linux上で`hdparm -I`を使用することでIDEおよびSATAドライブのキャッシュについて調べることができます。`Write cache`の次に *があれば書き込みキャッシュが有効になっています。`hdparm -W 0`により書き込みキャッシュを無効にできます。SCSIドライブであればsdparm[1]を使うことで調査が可能です。`sdparm --get=WCE`によりキャッシュが有効かどうかの確認ができ、`sdparm --clear=WCE`により無効にすることができます。

[1] http://sg.danny.cz/sg/sdparm.html

- FreeBSDでは、IDEドライブに対してatacontrolにより確認ができ、そして書き込みキャッシュを無効にするには/boot/loader.confのhw.ata.wc=0を利用します。SCSIドライブに対してはcamcontrol identifyを確認に使用することができ、sdparmを使用できる場合にはそれを用いて書き込みキャッシュの確認と変更が可能です。

- Solarisでは、ディスクの書き込みキャッシュはformat -eで制御できます。(SolarisのZFSファイルシステムは、独自のディスクキャッシュ書き出しコマンドを発行しているため、ディスクの書き込みキャッシュを有効にしても安全です。)

- Windowsでは、もしwal_sync_methodがopen_datasync(デフォルト)の場合、My Computer\Open \disk drive\Properties\Hardware\Properties\Policies\Enable write caching on the diskのチェックを外すことで、書き込みキャッシュを無効にできます。もう一つの方法としては、wal_sync_methodをfsyncかfsync_writethroughに設定し、書き込みキャッシュを使用しないようにします。

- macOSでは、wal_sync_methodをfsync_writethroughに設定することで書き込みキャッシュを使用しないようにします。

最近のSATAドライブ(ATAPI-6またはそれ以降)はドライブキャッシュの書き出しコマンド(FLUSH CACHE EXT)を提供している一方、SCSIドライブでは従来から類似のSYNCHRONIZE CACHEコマンドをサポートしていました。これらのコマンドは、直接PostgreSQLに発行されませんが、いくつかのファイルシステム(例えばZFSやext4)では、それらをwrite-backが有効なドライブへデータを書き出すために使います。不幸なことに、このようなwriteバリアを持つファイルシステムは、バッテリバックアップ付き装置 (BBU)のディスクコントローラと組み合わせた際に、好ましい動作をしません。このような処理の流れにおいて、同期コマンドはコントローラキャッシュにあるデータを全てディスクへ強制的に書き込みを行うため、BBUのメリットの大半を失わせています。pg_test_fsyncプログラムを 使うことで、あなたの環境が影響を受けるかどうかを確認できます。もし影響を受けるようであれば、ファイルシステムのwriteバリアを無効にするか、(オプションがあれば)ディスクコントローラを再設定することで、BBUによる性能上の効果を得ることできるでしょう。もしwriteバリアを無効にした場合は、バッテリが 動作していることを確認しておきましょう。バッテリの欠陥はデータロスの可能性に繋がります。ファイルシステムやディスクコントローラの設計者が、いずれはこの動作を修正してくれることが望まれます。

オペレーティングシステムが、ストレージハードウェアに書き込み要求を送信した時、データが不揮発性のストレージ領域に本当に届いたかどうかを確認することはほぼできません。ですので、全てのストレージ構成品がデータとファイルシステムのメタデータの整合性を保証することをよく確認しておくことは、管理者の責任です。バッテリバックアップされた書き込みキャッシュを持たないコントローラの使用は避けてください。装置レベルでは、もし装置が停止前にデータが書き出されることを保証できないのであれば、write-backキャッシュを無効にしてください。もしSSDを使っている場合、多くのドライブはデフォルトでキャッシュ書き出しコマンドを無視することに注意して下さい。diskchecker.pl[2]を使うことで、I/Oサブシステムの動作の信頼性をテストすることができます。

ディスクプラッタの書き込み操作自体によってもデータ損失が発生することがあります。ディスクプラッタは、通常512バイトのセクタに分割されています。物理的な読み込み操作、書き込み操作はすべて、セクタ全体を処理します。書き込み要求がディスクに達した時、その要求は512バイトの倍数になるでしょう(PostgreSQLでは大抵一度に8192バイトすなわち16セクタを書き込みます)。そして電源断により、任意のタイミングで書き込み処理が失敗することがありえます。これは一部の512バイトのセクタに書き込みが行なわれたのに、残りのセクタには書き込みが行なわれていない状況を意味します。こうした問題の対策とし

[2] https://brad.livejournal.com/2116715.html

て、PostgreSQLは、ディスク上の実際のページを変更する前に定期的にページ全体のイメージを永続的なWAL格納領域に書き出します。これにより、PostgreSQLはクラッシュリカバリ時に部分的に書き出されたページをWALから復旧させることができます。もし、部分的なページ書き込みを防止できるファイルシステムソフトウェア（例えばZFS）を使うのであれば、full_page_writesを無効にしてページイメージ作成を無効にすることができます。バッテリバックアップ付き（BBU）のディスクコントローラでは、フルページ（8kB）がBBUへ書き込まれることを保証できなければ、部分的なページ書き出しを防止できません。

さらにPostgreSQLは、ハードウェアエラーや経時変化によるメディア障害により発生する、ごみデータ読み書きしてしまうようなストレージ装置内のある種のデータ破損を防ぎます。

- WALファイルのそれぞれのレコードは、レコードの内容が正確かどうかを伝えるためCRC-32 (32-bit) チェックにより保護されています。CRCの値はそれぞれのWALレコードを書き込み、クラッシュ回復の過程で検証され、アーカイブの回復とレプリケーション時に設定されます。

- 今のところ、デフォルトではデータページはチェックサム計算はされませんが、WALレコードに記録されているページ全体のイメージは保護されます。詳細はinitdbを参照してください。

- pg_xact、pg_subtrans、pg_multixact、pg_serial、pg_notify、pg_stat、pg_snapshotsのような内部データ構造は直接チェックサム計算もされず、全ページ書き込みによる保護もされていません。しかし、そのようなデータ構造が持続する場所は、WALレコードはクラッシュ回復時に正確に最新の変更を行えるようWALレコードが書き出され、それらのWALレコードは上記のように保護されます。

- pg_twophaseにある個別の状態ファイルはCRC-32で保護されています。

- 大きな問い合わせの中でソート、具現化、および中間結果用に使用される暫定的なデータファイルは現在チェックサム計算されず、それらのファイルに対する変更もWALレコードに書き込まれません。

PostgreSQLは修復可能なメモリーエラーに対して保護を行いません。業界標準の誤り検出訂正（Error Correcting Codes -ECC-）またはそれ以上の保護付きのRAM使用が想定されています。

29.2. ログ先行書き込み(WAL)

ログ先行書き込み（WAL）はデータの一貫性を確実にするための標準的な手法です。詳細については、トランザクション処理について書かれた(すべてとは言いませんが)たいていの書籍に記載されています。簡単に言うと、WALの基本的な考え方は、（テーブルやインデックスがある）データファイルへの変更は、ログへの記録、つまり、変更内容を記述したログレコードが永続格納領域に書き出された後にのみ書き出されなければならないということです。このような手順に従って処理を行えば、たとえクラッシュが起きてもログを使ってデータベースをリカバリすることができるため、トランザクションのコミットの度にデータページをディスクに吐き出す必要がなくなります。リカバリの時点では、まず、データページに対してまだ行われていない変更分はログレコードを使って再実行されます（これがREDOとして知られているロールフォワードリカバリです）。

> **ヒント**
>
> WALによりデータベースファイルの中身を障害後にリストアするため、信頼性のある格納領域にあるデータファイルやWALファイルに対しては、ジャーナルファイルシステムは必要ありません。実際、特に、もしファイルシステムのデータをディスクにフラッシュさせている場合には、ジャーナリングのオーバーヘッドは性能を劣化させることがあります。幸運なことに、ジャーナリング中のデータのフラッ

> シュをマウントオプションにより無効にできることが多いです。例えばLinuxのext3ファイルシステムでは、data=writebackと指定します。ジャーナルファイルシステムは障害後の起動速度を改善します。

WALを使用することでディスクへの書き込み回数が大幅に減少します。と言うのも、トランザクションがコミットされたことを保証するために、そのトランザクションで変更された全てのデータファイルではなく、ログファイルだけをディスクに吐き出す必要があるからです。ログファイルへの書き込みはシーケンシャルに行われるため、データページを吐き出すコストに比べログファイルの同期はずっと低コストになります。これは特に、データ格納領域の様々な部分を変更する小さなトランザクションを多く扱うサーバで顕著に現れます。さらに、サーバが小規模なトランザクションを同時に多く処理する時、ログファイルを一度fsyncすることで、多くのトランザクションをコミットすることができる場合もあります。

また、WALにより、25.3で説明するオンラインバックアップとポイントインタイムリカバリをサポートすることができます。WALのデータを保持することにより、そのWALデータが範囲内とする任意の時点に戻すことができます。単純にデータベースの主となる物理バックアップをインストールし、WALログを目的の時点まで単に再生することで実現できます。さらに、物理バックアップはインスタンス化可能なデータベース状態のスナップショットである必要もありません。ある程度の時間を経過して作成されたバックアップであっても、その期間用のWALを再生することにより、内部の不整合を修復します。

29.3. 非同期コミット

非同期コミットとは、トランザクションをより高速に完了することができるオプションです。もっとも最近のトランザクションがデータベースがクラッシュしてしまった場合に失われるという危険があります。これは、多くのアプリケーションで受け入れられるトレードオフです。

前節で説明した通り、通常トランザクションのコミットは同期的です。サーバはトランザクションのWALレコードが永続的格納領域に記録されるまで、クライアントに成功したことを通知することを待機します。従って、直後にサーバクラッシュといった障害があったとしても、コミットされたと報告されたトランザクションは保持されることをクライアントは保証されます。しかし、短期のトランザクションでは、この遅延はトランザクションの処理時間の大半を占める要素となります。非同期コミットモードを選択することは、サーバがWAL記録が実際に作成された通りにディスクに書き込まれるより前に、トランザクションの論理的な完了をもって成功したと通知することを意味します。これにより、小規模なトランザクションでスループットがかなり向上します。

非同期コミットにはデータ損失の危険があります。トランザクションの完了をクライアントに通知してからトランザクションが本当に完了する（つまり、サーバクラッシュしても損失がないことが保証される）までの間にわずかな時間が存在します。したがって、クライアントがトランザクションが記録されているという仮定を元に外部的な動作を行う場合は、非同期コミットを使用すべきではありません。例えば、銀行では、ATMの現金分配を記録するトランザクションで非同期コミットを使用してはいけません。しかし、イベント記録など多くのシナリオでは、この種の保証を持って格納する必要はありません。

非同期コミットによりもたらされる危険性は、データの破壊ではなくデータの損失です。データベースがクラッシュした場合、最後にフラッシュされた記録までWALを再生することで復旧が行われます。このため、データベースは内部で一貫性を持った状態に復旧されますが、ディスクにフラッシュされていないトランザクションはすべてそこには反映されません。したがって、影響を受けるのは、最後に行われたいくつかのトランザクションの損失です。トランザクションはコミットされた順に再生されますので、一貫性が失われることはありません。例えば、トランザクションBが以前に行われたトランザクションAの結果に依存した変更を行った場合、Bの影響が保存されている限り、Aの影響が失われることは起こり得ません。

ユーザは各トランザクションでコミットモードを選択することができます。このため、同時実行されるトランザクションを同期的、および非同期の両方でコミットさせることができます。これにより、性能とトランザクションの信頼性の確実性との間で柔軟な選択を行うことができます。コミットモードはユーザによる設定が可能なパラメータsynchronous_commitで制御されます。このパラメータは、設定パラメータを設定することができる全ての方法で変更することが可能です。あるひとつのトランザクションで使用されるモードは、トランザクションのコミットが始まった時のsynchronous_commitの値に依存します。

例えばDROP TABLEなどの特定のユーティリティコマンドでは、synchronous_commitの設定に関わらず、強制的に同期的コミットが行われます。これにより、サーバのファイルシステムとデータベースの論理的な状態との間の一貫性が保証されます。PREPARE TRANSACTIONなどの2相コミットをサポートするコマンドもまた、常に同期的です。

もし非同期コミットとそのトランザクションのWAL記録の書き込みの間の危険期間にデータベースがクラッシュしたとすると、そのトランザクションでなされた変更は失われるでしょう。バックグラウンドプロセス（「WALライタ」）が未書き込みのWAL記録をwal_writer_delayミリ秒毎にディスクに吐き出しますので、この危険期間は制限されます。WALライタは稼働中に一回ページ全体を書き込むように設計されているため、危険期間の実際の最大の長さはwal_writer_delayの3倍です。

> **注意**
>
> 即時モードのシャットダウンはサーバクラッシュと同じことですので、吐き出されていない非同期コミットが失われることになります。

非同期コミットではfsync = offという設定とは異なる動作になります。fsyncはサーバ全体に関する設定であり、すべてのトランザクションの動作を変更します。これは、PostgreSQLにおける、データベースの別の場所への同期書き込みの試行に関するすべてのロジックを無効にします。このため、システムクラッシュ（PostgreSQL自体の失敗ではなくハードウェアやオペレーティングシステムのクラッシュ）の結果、予測できないデータベース状態の破壊が起こります。非同期コミットはデータ破壊の危険性はなく、多くの状況ではfsyncを無効にした場合に得られる性能向上とほぼ同等の性能を提供します。

またcommit_delayも非同期コミットと類似のように見えますが、これは実のところ同期コミットの一方法です。（実際、非同期コミット時commit_delayは無視されます。）トランザクションがWALをディスクに吐き出す直前に、こうしたトランザクションによって実行される一度の吐き出しにより、ほぼ同時期にコミットを行う他のトランザクションの分も処理できるようにすることを目的とした遅延がcommit_delayにより発生します。この設定は、一回のフラッシュに参画するグループに、複数のトランザクションの中でフラッシュのコストを償却し、加わることが可能なトランザクションの時間的猶予を広げる方法の一つとして考えることができます。

29.4. WALの設定

データベースの性能に影響するようなWALに関連した設定パラメータが複数あります。本節では、その使い方を説明します。サーバ設定パラメータの設定方法についての詳細は第19章を参照してください。

チェックポイントは、一連のトランザクションにおいて、そのチェックポイント以前に書かれた全ての情報によりヒープとインデックスファイルがすでに更新されていることを保証する場所です。チェックポイントの時刻において、全てのダーティページデータはディスクにフラッシュされ、特殊なチェックポイントレコードがログファイルに書き込まれます。(変更されたレコードは以前にWALフラッシュされています。) クラッシュした

時、クラッシュからの復旧処理は最後のチェックポイントレコードを見つけ、ログの中でどのレコード（これはredoレコードと呼ばれています）から復旧処理がREDOログ操作を開始すべきかを決定します。このチェックポイント以前になされたデータの変更は、すでにディスク上にあることが保証されています。従って、チェックポイント後、redoレコード内のそのチェックポイント以前のログセグメントは不要となり、再利用または削除することができます（WALアーカイブが行われる場合、このログセグメントは削除もしくは再利用される前に保存されなければなりません）。

チェックポイント処理は、全てのダーティデータページをディスクへ書き出すため、大きなI/O負荷を発生させます。チェックポイント処理においては、I/Oはチェックポイント開始時に始まり、次のチェックポイントが開始する前に完了するように調節されます。これは、チェックポイント処理中の性能劣化を極力抑える効果があります。

サーバのチェックポインタプロセスは、自動的にチェックポイントを時々実行します。checkpoint_timeout秒が経過するか、またはmax_wal_sizeに達するか、どちらかの条件が最初に満たされるとチェックポイントが開始されます。デフォルトの設定では、それぞれ5分と1GBとなっています。前回のチェックポイント以降書き出すWALがない場合、checkpoint_timeoutが経過したとしても新しいチェックポイントが飛ばされます。(WALアーカイブ処理を使用しており、かつ、データ損失の可能性を限定するためにファイルのアーカイブ頻度に対する下限を設定したい場合、チェックポイント関連のパラメータよりも、archive_timeoutパラメータを調節するべきです。) また、CHECKPOINT SQLコマンドで強制的にチェックポイントを作成することもできます。

checkpoint_timeoutまたはmax_wal_size、あるいはその両者を減少させると、チェックポイントはより頻繁に行われます。これにより、やり直しに要する処理量が少なくなるので、クラッシュ後の修復は高速になります。しかし、変更されたデータページの吐き出しがより頻繁に行われることにより増大するコストとバランスを考えなければなりません。full_page_writesが設定されている（デフォルトです）場合、他に考慮しなければならない点があります。データページの一貫性を保証するために、各チェックポイント後の最初に変更されるデータページは、そのページ全体の内容がログに保存されることになります。このような場合、チェックポイントの間隔を少なくすることは、WALログへの出力を増加させ、間隔を短くする目的の一部を無意味にします。また、確実により多くのディスクI/Oが発生します。

チェックポイントはかなり高価なものです。1番の理由は、この処理は現時点の全てのダーティバッファを書き出す必要があること、2番目の理由は、上記のようにその後に余計なWALの書き込みが発生することです。そのため、チェックポイント用のパラメータを高くし、チェックポイントがあまりにも頻発することがないようにすることを勧めます。簡単なチェックポイント用のパラメータの健全性検査として、checkpoint_warningパラメータを設定することができます。チェックポイントの発生間隔がcheckpoint_warning秒未満の場合、max_wal_sizeの増加を勧めるメッセージがサーバのログに出力されます。このメッセージが稀に現れたとしても問題にはなりませんが、頻出するようであれば、チェックポイントの制御パラメータを増加させるべきです。max_wal_sizeを十分高く設定していないと、大規模なCOPY転送などのまとまった操作でこうした警告が多く発生するかもしれません。

ページ書き出しの集中による入出力システムの溢れを防ぐために、チェックポイント期間のダーティバッファの書き出しは一定の期間に分散されます。この期間はcheckpoint_completion_targetにより制御され、チェックポイント間隔の割合として指定されます。I/Oの割合は、チェックポイントの起動時からcheckpoint_timeout秒が経過した時、あるいはmax_wal_sizeを超えた時、このどちらかが発生するとすぐに、チェックポイントが完了するように調整されます。デフォルトの0.5という値では、PostgreSQLは、次のチェックポイントが始まるまでのおよそ半分の時間で各チェックポイントが完了するものと想定できることになります。通常の操作においてほぼ最大のI/Oスループットに近いようなシステムでは、チェックポイントにおけるI/O負荷を減らすためにcheckpoint_completion_targetを増やすことを勧めます。この欠点は、延長さ

れたチェックポイントがリカバリ時に影響をあたえることです。リカバリ時に使用できるように、より多くのWALセグメントを保持する必要があるためです。checkpoint_completion_targetを最大の1.0に設定することもできますが、より低く抑えること（おそらく最大で0.9）が最善です。チェックポイントには、ダーティバッファを書き出す以外の活動も含まれているからです。1.0という設定は、ある時点でチェックポイントが完了しなくなるという結果に陥ります。これは必要なWALセグメント数が想定以上に変動することになり、性能の劣化が発生することになります。

LinuxおよびPOSIXプラットフォームでは、チェックポイントによって書かれたページを、設定したバイト数の後にディスクに吐き出すようにcheckpoint_flush_afterを使ってOSに強制させることができます。この設定がない場合はこのページはOSのページキャッシュに保持されるかもしれず、チェックポイントの最後にfsyncが発行された際の速度低下を招きます。この設定は、しばしばトランザクションの遅延を減少させるのに役立ちます。しかし、とりわけワークロードがshared_buffersよりも大きく、かつOSのページキャッシュよりも小さい場合には性能上不利になることもあります。

pg_walディレクトリ内のWALセグメントファイルの数は、min_wal_size、max_wal_size、それに前回のチェックポイントで生成されたWALの量に依存します。古いログセグメントファイルが不要になると、削除または再利用(連番のうち、今後利用される予定の番号に名前が変更されます)されます。ログの出力レートが短期間にピークを迎えたためにmax_wal_sizeを超えた場合、この制限以下になるまで不要なセグメントファイルが削除されます。この制限以下になると、次のチェックポイントまでは、システムは見積もりを満たすだけのWALファイルを再利用します。この見積は、前回のチェックポイントの際に使用されたWALファイルの移動平均に基づいています。もし実際の使用量が見積もりを上回ると、移動平均は直ちに増加します。これにより、平均需要というよりは、ピーク時の需要をある程度満たすことができるわけです。min_wal_sizeは、今後のために再利用されるWALファイル数の最小値を設定します。システムがアイドル状態にあり、WALの使用量を見積った結果、少ないWALしか必要ないとなったとしても、こうした量のWALファイルは必ず再利用されます。

max_wal_sizeに関わらず、最新のwal_keep_sizeメガバイトのWALファイルに加えて、もう一つのWALファイルが常に保持されます。また、WALアーカイブを利用している場合は、古いセグメントは、アーカイブされるまでは削除も再利用もされません。WALが生成されるペースにWALのアーカイブ処理が追いつかなかったり、archive_commandが連続して失敗すると、事態が解決するまでWALファイルはpg_walの下に蓄積されていきます。レプリケーションスロットを使用しているスタンバイサーバが低速だったり、失敗すると、同じ現象が起きます(26.2.6を参照のこと)

アーカイブからのリカバリもしくはスタンバイモードにおいて、サーバでは定期的に通常運用でのチェックポイント処理と似た再開始点処理を行います。これは、すでに再生されたWALを再度読み込む必要がないよう、ディスクに現在の状態を強制的に書き込み、pg_controlファイルを更新します。またpg_walディレクトリの中の古いログセグメントを再利用できるようにします。再開始点処理はチェックポイントレコードに対してしか実施されないので、マスタ側のチェックポイント処理よりも発生頻度が多いということはありません。再開始点は、最後の再開始点より少なくともcheckpoint_timeout秒が経過しているか、あるいはmax_wal_sizeを超えそうな場合に起動されます。しかし、再開始点が実施できるための制約事項により、リカバリの際には1回のチェックポイント分のWALを上限に、max_wal_sizeを超えてしまいがちです。(どのみちmax_wal_sizeはハードリミットではないので、ディスクスペースを使い尽くしてしまわないように、常に十分な余裕を持っておくべきです)

よく使われる2つの内部用WAL関数があります。XLogInsertRecordとXLogFlushです。XLogInsertRecordは共有メモリ上のWALバッファに新しいレコードを挿入します。新しいレコードを挿入する余地がない時は、XLogInsertRecordは、満杯になったWALバッファを書き込み（カーネルキャッシュに移動）しなければいけません。これは望ましいことではありません。なぜなら、データベースへの低レベルの変更（例えば行の挿

入）の度にXLogInsertRecordが呼ばれますが、そのような場合には変更を受けたページに対して排他ロックがかかっており、それゆえこの操作は可能な限り高速に実行されなければなりません。さらに悪いことには、WALバッファへの書き込みの際に、さらに時間がかかる、強制的な新しいログセグメントの生成が必要となるかもしれません。通常、WALの書き込み、吐き出しはXLogFlush要求で実施されます。これはたいていの場合、トランザクションコミットの際に永続的な記憶領域にトランザクションレコードが吐き出されることを保証するために行われます。ログ出力が大量に行われるシステムでは、XLogInsertRecordによって必要となる書き込みを防ぐほどにはXLogFlush要求が頻繁に起こらないかもしれません。そういうシステムでは、wal_buffersパラメータを変更してWALバッファの数を増やしてください。full_page_writesが設定され、かつ、システムが高負荷状態である場合、wal_buffersを高くすることで、各チェックポイントの直後の応答時間を滑らかにすることができます。

commit_delayパラメータは、XLogFlush内でロックを取得してからグループコミット上位者が何マイクロ秒休止するかを定義します。一方、グループコミット追従者は上位者の後に並びます。すべてが上位者の結果として生ずる同期操作によりフラッシュされるように、この遅延は他のサーバプロセスがそれらのコミットレコードをWALバッファに追加することを許容します。fsyncが有効でないか、またはcommit_siblingsより少ない他のセッションがその時点で活動しているトランザクションであれば休止は行われません。他の何らかのセッションが直ぐにでもコミットするという起こりそうにない時の休止を避けるものです。いくつかのプラットフォームにおいて、休止要求の分解能は10ミリ秒で、1から10000マイクロ秒の間のcommit_delayの設定は、どの値でも同じ効果となることを覚えておいてください。いくつかのプラットフォームで、休止操作はパラメータによって要求された時間よりわずかに長くなることも覚えておいてください。

commit_delayの目的は、それぞれのフラッシュ操作のコストを並列にコミット中のトランザクションに（潜在的にはトランザクションの待ち時間と引き換えに）分散させることにあり、うまく設定を行うためには、まずそのコストを測る必要があります。そのコストが高ければ高いほど、トランザクションのスループットがある程度向上するという意味において、commit_delayの効果がより増すことが期待できます。pg_test_fsyncプログラムは、一つのWALフラッシュが必要とするマイクロ秒単位の平均時間を計測するために使用可能です。プログラムが報告する単一の8kB書き込み操作のあとのフラッシュ平均時間の2分の1の値は、しばしばcommit_delayの最も効果的な設定です。従って、この値は特定の作業負荷のための最適化を行うときに使用するための手始めとして推奨されます。WALログが高遅延の回転ディスクに格納されているときは、commit_delayのチューニングは特に有効ですが、半導体ドライブまたはバッテリー・バックアップされている書き込みキャッシュ付きのRAIDアレーのような、特に同期時間が高速な格納メディア上であっても大きなメリットがある場合があります。しかし、このことは、代表的作業負荷に対してきちんと検証しておくべきです。commit_siblingsの高い値は、これらの状況で使用すべきで、一方より小さなcommit_siblingsの値は高遅延メディア上でしばしば有用です。余りにも高い値のcommit_delayを設定すると、トランザクション遅延を増加させかねないことになり、トランザクションの総スループットが低下します。

commit_delayが(デフォルトの)ゼロに設定されても、グループコミットが起こることがあります。しかし、それぞれのグループは前回のフラッシュ操作（あった場合）が発生していた期間中に、それぞれのコミットレコードをフラッシュする必要に至ったセッションのみから成ります。クライアントが多い状況では、「gangway effect」が起こる傾向があり、そのためcommit_delayがゼロであってもグループコミットの効果が著しく、従って、commit_delayを明示的に設定しても役立ちません。commit_delayの設定は（1）複数の同時にコミット中のトランザクションが存在すること、そして（2）コミット頻度によりある程度までスループットが制限されている場合に役立ちます。しかし、回転待ち時間が長い場合、この設定はわずか二つのクライアントにおいてさえトランザクションスループットを向上させる効果があるかもしれません（言いかえれば、一つの兄弟（sibling）トランザクションを所有する単一のコミット中のクライアントです）。

wal_sync_methodパラメータはPostgreSQLがカーネルに対してWAL更新のディスクへの書き込みを要求する方法を決定します。fsync_writethroughを除き、どういう設定でも信頼性は同じはずで

す。fsync_writethroughは他のオプションがそうしないときでも、時々ディスクキャッシュの書き出しを強制することができます。しかしながら、プラットフォームによってどれが一番速いのかがまったく違います。pg_test_fsyncプログラムを使って異なるオプションの速度テストを行うことができます。ちなみに、このパラメータはfsyncが無効になっている場合は役に立ちません。

wal_debug設定パラメータを有効にすることで、XLogInsertRecordとXLogFlushというWAL呼び出しは毎回サーバログにログが残ります（このパラメータをサポートするようにPostgreSQLをコンパイルする必要があります）。将来このオプションはより一般的な機構に置き換わる可能性があります。

29.5. WALの内部

WALは自動的に有効になります。WALログが必要とするディスク容量を確保すること、そして必要ならばチューニングすることを除いては（29.4を参照）、管理者は何もする必要はありません。

新しいレコードが作成されるごとに、WALレコードがWALログに追加されます。挿入位置はログシーケンス番号(LSN)によって記録されます。LSNはログのバイトオフセットで、単調増加します。LSN値は、pg_lsnデータ型として返されます。2つのLSN値を比較することでWALデータの差分量を計算することができるので、レプリケーションやリカバリーの進捗状況を測定できます。

WALログは、データディレクトリ以下のpg_walディレクトリに、通常16メガバイトのサイズを持つセグメントファイルの集合として格納されています(ただし、このサイズはinitdbの--with-wal-segsizeオプションで変更できます)。各セグメントは通常8キロバイトのページに分割されます(このサイズは--with-wal-blocksizeというconfigureオプションで変更できます)。ログレコード用のヘッダはaccess/xlogrecord.hに記述されています。レコードの内容は、ログの対象となる事象の種類によって異なります。セグメントファイルは名前として000000010000000000000001から始まる、常に増加する数が与えられています。数字は巡回しませんが、利用可能な数字を使い尽くすには非常に長い時間がかかるはずです。

主要なデータベースファイルが置いてあるディスクとは別のディスクにログを置くと利点があります。これはpg_walディレクトリを別の場所に（もちろんサーバを終了しておいてから）移動し、主データディレクトリ以下の元々の場所から新しい場所へのシンボリックリンクを張ることによって可能となります。

WALの目的である、確実にデータベースレコードが変更される前にログが書き出されることは、実際にはキャッシュにしかデータがなく、ディスクには格納されていない時にディスクドライブが格納に成功したとカーネルに虚偽の報告を行うことによって失われる可能性があります。そのような状況では、電源が落ちた際に、復旧不可能なデータの破壊が起こることがあります。管理者は、PostgreSQLのWALログを保持しているディスク装置がそのような嘘の報告をしないように保証するべきです。(29.1を参照して下さい。)

チェックポイントが実行され、ログが吐き出された後、チェックポイントの位置はpg_controlに保存されます。したがって、リカバリの開始の際は、サーバはまずpg_controlを読み、次にチェックポイントレコードを読みます。そして、チェックポイントレコード内で示されたログの位置から前方をスキャンすることでREDO処理を行います。データページの内容全体は、チェックポイント後の最初のページ変更時にログ内に保存されますので(full_page_writesパラメータが無効にされていないという前提です)、そのチェックポイント以降に変更された全てのページは一貫した状態に復旧されます。

pg_controlが壊れた場合に備え、ログセグメントを逆順に読み（すなわち新しいものから古いものへと）、最終チェックポイントを見つける方法を実際には実装した方が良いと思われます。まだこれはできていません。pg_controlはかなり小さなもの（1ディスクページ未満）ですので、一部のみ書き込みされるという問題は起

こりません。またこの書き込みの時点では、pg_control自体の読み込みができないことによるデータベースエラーという報告はありません。このため、pg_controlは理屈では弱点ですが、実質問題になりません。

第30章 論理レプリケーション

論理レプリケーションとは、レプリケーションアイデンティティ(replication identity)（通常は主キーです）に基づき、データオブジェクトと、それに対する変更を複製する手法です。この論理という用語は、正確なブロックアドレスを使い、バイト同士の複製を行う物理レプリケーションと対比的に使用しています。PostgreSQLは両方の仕組みを同時にサポートします。第26章をご覧ください。論理レプリケーションにより、データの複製とセキュリティに対するきめの細かい制御が可能になります。

論理レプリケーションは、ひとつの*パブリッシャー(publisher)*ノード上の一つ以上の*パブリケーション(publications)*を購読する一つ以上の*サブスクライバー(subscribers)*を伴う、*パブリッシュ(publish)*と*サブスクライブ(subscribe)*モデルを使用します。サブスクライバーは、サブスクライブするパブリケーションからデータを取得し、再パブリッシュしてカスケードレプリケーションや、更に複雑な構成を構築することができます。

テーブルの論理レプリケーションは、通常、パブリッシャーのデータベース上のデータのスナップショットを取り、サブスクライバーにコピーすることから始まります。それが完了したあとは、パブリッシャーにおける変更は、発生した時にリアルタイムでサブスクライバーに送られます。サブスクライバーはパブリッシャーと同じ順にデータを適用します。そのため、一つのサブスクリプション内のパブリケーションに対するトランザクションの一貫性が保証されます。この方式によるデータレプリケーションは、トランザクショナルレプリケーション(transactional replication)と呼ばれることがあります。

典型的な論理レプリケーションの利用例には、以下のようなものがあります。

- 一つのデータベース、あるいはデータベースの一部に起こった更新の差分を、発生都度サブスクライバーに送る。
- サブスクライバーに更新が到着した時に、それぞれの更新に対してトリガーを起動する。
- 複数のデータベースを一つのデータベースに統合する。（たとえば分析目的で。）
- 異なるメジャーバージョンのPostgreSQL間でレプリケーションする。
- 異なるプラットフォーム上のPostgreSQLインスタンス間（たとえばLinuxからWindows）でレプリケーションする。
- 異なるユーザのグループに対して、複製されたデータにアクセスさせる。
- 複数のデータベース間でデータベースの一部を共有する。

サブスクライバーのデータベースは、他のPostgreSQLインスタンスと同様に振る舞い、自分用のパブリケーションを定義することにより、他のデータベースに対するパブリッシャーとして利用できます。アプリケーションがそのサブスクライバーを読み取り専用として取り扱うときには、単独のサブスクリプションからはコンフリクトは発生しません。一方、アプリケーションあるいは他のサブスクライバーから同じテーブルに書き込みが起こるとすると、コンフリクトが発生する可能性があります。

30.1. パブリケーション

*パブリケーション*は、どのような物理レプリケーションのマスターにも定義できます。パブリケーションが定義されたノードは、*パブリッシャー*と呼ばれます。パブリケーションは、テーブルか、テーブルのグループから生

成された更新の集合であると同時に、更新セットあるいはレプリケーションセットであるとも言えます。一つのパブリケーションは一つのデータベースにのみ存在します。

パブリケーションはスキーマとは異なり、テーブルがどのようにアクセスされるかには影響しません。必要ならば、テーブルを複数のパブリケーションに追加できます。今のところパブリケーションはテーブルのみを含むことができます。パブリケーションがALL TABLESで作られた場合を除き、オブジェクトは明示的に追加されなければなりません。

パブリケーションは、生成される更新を、INSERT、UPDATE、DELETE、TRUNCATEのうちのどのような組み合わせにも制限することができます。これはトリガーが特定のイベント型によって起動されることに似ています。デフォルトでは、すべての操作タイプがレプリケーションされます。

パブリッシュされたテーブルは、UPDATEとDELETEをレプリケーションできるようにするために、「レプリカアイデンティティ」の設定を含んでいなければなりません。そうすることにより、サブスクライバー側で更新または削除する対象の正しい行が特定できるようになります。デフォルトでは主キーがあれば、それがレプリカアイデンティティになります。他に、ユニークキー（追加の要件を伴います）もレプリカアイデンティティにできます。テーブルに適当なキーがなければ、レプリカアイデンティティを「full」にできます。これは、行全体がキーになることを意味します。しかし、これは非常に非効率なので、他の解決方法がない場合のみの代替手段にすべきです。「full」以外のレプリカアイデンティティがパブリッシャー側に設定されている場合、同じか、より少ない列を含むレプリカアイデンティティがサブスクライバー側に設定されていなければなりません。レプリカアイデンティティを設定する詳細な方法については、REPLICA IDENTITYをご覧ください。UPDATEあるいはDELETE操作をレプリケーションするパブリケーションに、レプリカアイデンティティがないテーブルが追加されると、以後UPDATEあるいはDELETE操作が行われるとパブリッシャー側でエラーが発生します。INSERT操作は、レプリカアイデンティティの設定に関わらず実行されます。

すべてのパブリケーションは、複数のサブスクライバーを持つことができます。

パブリケーションは、CREATE PUBLICATIONコマンドで作成し、対応するコマンドで変更や削除ができます。

個々のテーブルはALTER PUBLICATIONで動的に追加削除できます。ADD TABLEおよびDROP TABLE操作はトランザクションの対象です。ひとたびトランザクションがコミットされれば、正しいスナップショットでテーブルのレプリケーションが開始あるいは終了されます。

30.2. サブスクリプション

サブスクリプションは論理レプリケーションの下流側です。サブスクリプションが定義されたノードはサブスクライバーとして参照されます。サブスクリプションは他のデータベースへの接続と、サブスクリプション対象の一つ以上のパブリケーションの集合を定義します。

サブスクライバーのデータベースは、他のPostgreSQLインスタンスと同様に振る舞い、自分用のパブリケーションを定義することにより、他のデータベースに対するパブリッシャーとして利用できます。

サブスクライバーノードは、必要ならば複数のサブスクリプションを持つことができます。一組のパブリッシャーとサブスクライバーの間で複数のサブスクリプションを定義することもできますが、サブスクライブしたパブリケーションオブジェクトが重複しないように注意が必要です。

各々のサブスクリプションは、一つのレプリケーションスロット（26.2.6を参照）を通じて更新が通知されます。既存のテーブルデータを初期同期するために、追加で一時的なレプリケーションスロットが必要になることもあります。

論理レプリケーションのサブスクリプションは、同期レプリケーション（26.2.8参照）のスタンバイであっても構いません。スタンバイ名称はデフォルトではサブスクリプション名となります。サブスクリプションのコネクション情報の中のapplication_nameを別の名前として指定することもできます。

現在のユーザがスーパーユーザならば、サブスクリプションはpg_dumpでダンプできます。そうでない場合には、警告が出力され、サブスクリプションはスキップされます。非スーパーユーザはすべてのサブスクリプション情報を、pg_subscriptionカタログから読み出せないからです。

サブスクリプションはCREATE SUBSCRIPTIONで追加し、ALTER SUBSCRIPTIONを使って、いつでも停止、再開でき、そしてDROP SUBSCRIPTIONで削除できます。

サブスクリプションが削除され、そして再作成されると、同期情報は失われます。このことは、後でデータを再同期しなければならないことを意味します。

スキーマ定義情報はレプリケーションされないので、パブリッシュするテーブルはサブスクライバーに存在しなければなりません。通常のテーブルだけがレプリケーションの対象です。たとえば、ビューはレプリケーションできません。

パブリッシャーとサブスクライバーの間でのテーブルの照合は、完全修飾されたテーブル名に基づいて行われます。サブスクライバーで異なる名前になっているテーブルに対するレプリケーションは、サポートされていません。

テーブルの列も名前で照合されます。サブスクライバーのテーブルでの列の順序はパブリッシャーと一致している必要はありません。データのテキスト表現列が対象の型に変換可能である限り、列のデータ型も一致している必要がありません。例えば、integer型の列からbigint型の列にレプリケーションすることができます。対象テーブルはパブリッシュされたテーブルにない追加の列を持つこともできます。そうした列には対象テーブルの定義の指定に従ってデフォルト値が挿入されます。

30.2.1. レプリケーションスロットの管理

前述のように、各々の（有効な）サブスクリプションは、リモート（パブリッシュしている）側のレプリケーションスロットに対する変更を受信します。通常、リモートのレプリケーションスロットは、CREATE SUBSCRIPTIONによりサブスクリプションが作られた時に自動的に作られ、DROP SUBSCRIPTIONによりサブスクリプションが削除された時に自動的に削除されます。しかし、状況によっては、サブスクリプションとそれが依拠しているレプリケーションスロットを別々に操作する方が良かったり、あるいはそうした必要性が出てくることもあるかもしれません。そうしたシナリオを示します。

- サブスクリプションを作る際、レプリケーションスロットがすでに存在しています。この場合、create_slot = falseオプションを使ってサブスクリプションを作成し、既存のスロットと関連付けることができます。

- サブスクリプションを作成する際に、リモートホストが接続できない状態にあるか、不明な状況にあります。こうした時は、connect = falseを使ってサブスクリプションを作成することができます。リモートホストにはまったく接続しません。これは、pg_dumpが使っている方法です。サブスクリプションを有効にする前に、リモートホストのレプリケーションスロットを手動で作成しなければなりません。

- サブスクリプションを削除する際に、レプリケーションスロットを維持する必要があります。サブスクライバーのデータベースが別のホストに移動中で、移動後にそこからデータベースを起動するときに有効です。この場合、サブスクリプションを削除する前に、ALTER SUBSCRIPTIONでそのスロットを切り離します。

- サブスクリプションを削除する際に、リモートホストに接続できません。この場合、サブスクリプションを削除する前に、ALTER SUBSCRIPTIONでそのスロットを切り離しを試みます。リモートデータベースインスタンスが存在しない場合は、これ以上の操作は必要ありません。しかし、単にリモートデータベースに接続できない状態ならば、レプリケーションスロットを手動で削除する必要があります。そうでなければ、WALが保存され続け、いずれディスクを埋め尽くすかもしれません。そのような状態は注意深く調査する必要があります。

30.3. コンフリクト

サブスクライバーノードでローカルにデータが変更された場合でも、データが更新されるという点では、論理レプリケーションは通常のDML操作と同じように振る舞います。到着したデータが制約に違反すると、レプリケーションは停止します。これは、コンフリクトと呼ばれます。UPDATEあるいはDELETE操作をレプリケーションする場合は、存在しないデータによってコンフリクトは起こらず、そのような操作は単にスキップされます。

コンフリクトはエラーを生じさせ、レプリケーションを停止させます。コンフリクトはユーザが手動で解消しなければなりません。コンフリクトの詳細は、サブスクライバーのサーバーログに出力されます。

コンフリクトの解消は、到着した更新とコンフリクトしないようにサブスクライバーのデータを変更するか、既存のデータとコンフリクトしているトランザクションをスキップさせることで達成できます。トランザクションは、pg_replication_origin_advance()関数にサブスクリプション名に関連するnode_nameと位置を引数で渡すことによりスキップできます。オリジンの現在位置はpg_replication_origin_statusシステムビューで参照できます。

30.4. 制限事項

論理レプリケーションには、以下の制限事項とサポートされていない機能があります。将来のリリースでは、これらは対処されるかもしれません。

- データベーススキーマおよびDDLコマンドはレプリケーションされません。初期スキーマは、pg_dump --schema-onlyを使ってコピーすることができます。以後のスキーマ変更の同期は手動で行ないます。(なお、両者でスキーマ名は完全に同じである必要はないことに留意してください。) 稼働中のスキーマ定義変更に対して、論理レプリケーションは頑健です。スキーマがパブリッシャー側で変更され、複製データがサブスクライバー側に到着し始めたものの、データがテーブルスキーマに合致しない場合は、スキーマが変更されるまではレプリケーションはエラーとなります。多くの場合、間欠的なエラーは、サブスクライバーに先に追加的なスキーマ変更を行うことで避けることができます。

- シーケンスデータはレプリケーションされません。シーケンスによって裏付けされたSERIAL型や識別列のデータは、もちろんテーブルの一部としてレプリケーションされます。しかし、シーケンス自体は、サブスクライバーがスタートした時の値のままです。サブスクライバーが読み取り専用のデータベースとして使われているなら、通常は問題になりません。しかし、サブスクライバーのデータベースをスイッチオーバーやフェイルオーバーするつもりなら、パブリッシャーから現在のデータをコピーするか(おそらくpg_dumpを使います)、テーブル自身から十分に大きな値を決定し、シーケンスを最新の値に更新しなければなりません。

- TRUNCATEコマンドのレプリケーションはサポートされますが、外部キーで結びついたテーブル群を削除する場合には注意が必要です。削除処理をレプリケーションするとき、サブスクライバーはパブリッシャーで

明示的に指定され削除された、もしくはCASCADEにより暗黙的に削除されたテーブル群から、サブスクリプションの一部ではないテーブルを除いたテーブル群を削除します。この処理は、外部キーで関連付けられた全てのテーブルが同一のサブスクリプションの一部であれば、正常に動作します。しかし、サブスクライバーで削除されるテーブルが同一のサブスクリプションの一部でないテーブルと外部キーで接続されていた場合、サブスクライバー上の削除処理は失敗します。

- ラージオブジェクト（第34章参照）はレプリケーションされません。通常のテーブルにデータを格納する以外に回避方法はありません。
- レプリケーションは、パーティションテーブルを含むテーブルでのみサポートされています。ビュー、マテリアライズドビュー、外部テーブルのような、その他の種類のリレーションをレプリケーションしようとすると、エラーになります。
- パーティションテーブル間でレプリケーションする場合には、実際のレプリケーションは、デフォルトでは、パブリッシャー側の末端のパーティションから開始します。ですので、パブリッシャー側のパーティションがサブスクライバー側にも有効な対象テーブルとして存在していなければなりません。(対象テーブルは、それ自身が末端のパーティションかもしれませんし、さらにサブパーティション化されているかもしれません。独立したテーブルであっても構いません。) パブリケーションは、変更が実際に開始された個々の末端のパーティションのIDとスキーマの代わりに、パーティションのルートのテーブルのIDとスキーマを使って指定することもできます(CREATE PUBLICATIONを参照してください)。

30.5. アーキテクチャ

論理レプリケーションは、パブリッシャー側のデータベース上のデータのスナップショットをコピーすることから始まります。それが完了したあとは、パブリッシャーにおける変更は、発生した時にリアルタイムでサブスクライバーに送られます。サブスクライバーはパブリッシャーでコミットが発生した順にデータを適用します。そのため、どの単一のサブスクリプションにおいても、パブリケーションに対するトランザクションの一貫性が保証されます。

論理レプリケーションは物理ストリーミングレプリケーション(26.2.5参照)と似たアーキテクチャで構成されています。「WAL送信」プロセスと「適用」プロセスで実装されています。walsenderプロセスはWALのロジカルデコーディング（第48章に記載）を開始し、標準のロジカルデコーディングプラグイン（pgoutput）をロードします。このプラグインは、WALから読み込んだ更新を論理レプリケーションプロトコル（52.5参照）に変換します。そして、パブリケーションの指定にしたがってフィルターします。データは次に、ストリーミングレプリケーションプロトコルを使って継続的に適用ワーカーに転送されます。適用ワーカーは、データをローカルテーブルにマップし、更新を受信すると正しいトランザクション順に個々の更新を適用します。

サブスクライバーデータベースの適用プロセスは、常にsession_replication_roleをreplicaにセットして実行されます。これによりトリガーと制約で通常の効果を生成します。

今のところ、論理レプリケーション適用プロセスは行トリガーだけを起動し、文トリガーは起動しません。ただし、初期テーブル同期はCOPYコマンドのように実装されているので、INSERTの行と文トリガーの両方を起動します。

30.5.1. 初期スナップショット

既存のサブスクライブされたテーブル中の初期データのスナップショットが取得され、特殊な適用プロセスの並列インスタンスにコピーされます。このプロセスは自身の一時レプリケーションスロットを作成し、既存の

データをコピーします。既存のデータのコピーが終わると、ワーカーは同期モードに入ります。このモードでは、初期データのコピー中に起こった更新を標準の論理レプリケーションを使ってストリーミングすることにより、テーブルが主適用プロセスと同期状態になることを保証します。ひとたび同期が完了すれば、テーブルのレプリケーションの制御は主適用プロセスに戻され、レプリケーションは通常通り継続されます。

30.6. 監視

論理レプリケーションは物理ストリーミングレプリケーションと類似のアーキテクチャに基づいているので、パブリケーションノードの監視は、物理レプリケーションのマスター(26.2.5.2参照)の監視と似ています。

サブスクリプションに関する監視情報はpg_stat_subscriptionで見ることができます。このビューは、個々のサブスクリプションワーカー毎に1つの行を含んでいます。サブスクリプションは状態により、0以上のアクティブなサブスクリプションワーカーを持つことができます。

有効なサブスクリプションのために通常は一つの適用プロセスが実行中です。無効なサブスクリプション、あるいはクラッシュしたサブスクリプションはこのビュー中に0個の行を持ちます。テーブルの初期データの同期が進行中なら、同期中のテーブルのための追加ワーカーが存在するでしょう。

30.7. セキュリティ

サブスクライバ側のテーブルのスキーマを変更できるユーザは任意のコードをスーパーユーザとして実行することができます。そのようなテーブルの所有権とTRIGGER権限はスーパーユーザが信頼するロールにのみ付与するように制限してください。さらに信用できないユーザがテーブルを作成できる場合は、テーブルを明示的にリストしているパブリケーションのみを使用してください。つまり、スーパーユーザが全てのユーザにパブリッシャやサブスクライバに非一時テーブルを作成することを信用している場合にのみ、FOR ALL TABLESサブスクリプションを作成してください。

レプリケーション接続のために使われるロールには、REPLICATION属性が付与されている(もしくはスーパーユーザである)必要があります。ロールに SUPERUSERとBYPASSRLSがない場合は、パブリッシャは行セキュリティポリシーを実行できます。ロールが全てのテーブルの所有者を信頼していない場合、接続文字列にoptions=-crow_security=offを含めてください。テーブルの所有者が行セキュリティポリシーを追加した場合、ポリシーが実行されるのではなく、レプリケーションが停止します。接続のためのロールはpg_hba.confで設定され、LOGIN属性を持つ必要があります。

テーブルの初期データをコピーできるためには、レプリケーション接続に使用されるロールは、パブリッシュされるテーブルに対してSELECT権限を持っていなければなりません。(あるいはスーパーユーザでなければなりません。)

パブリケーションを作成するためには、ユーザはデータベース中のCREATE権限を持っていなければなりません。

テーブルをパブリケーションに追加するためには、ユーザはテーブルの所有権限を持っていなければなりません。自動的にすべてのテーブルにパブリッシュするパブリケーションを作成するには、ユーザはスーパーユーザでなければなりません。

サブスクリプションを作成するためには、ユーザはスーパーユーザでなければなりません。

ローカルデータベースで実行されるサブスクリプション適用プロセスは、スーパーユーザ権限で実行されます。

権限は、レプリケーション接続の開始時に一度だけチェックされます。パブリッシャーから更新レコードを読む際、あるいは個々の更新を適用する際には再チェックされません。

30.8. 構成設定

論理レプリケーションにはいくつかの設定オプションの設定が必要です。

パブリッシャー側では、wal_levelがlogicalに、max_replication_slotsには少なくとも接続予定のサブスクリプション数に加えてテーブル同期のための予備が設定されなければなりません。また、max_wal_sendersは少なくともmax_replication_slotsに加えて 同時に接続する物理レプリカ数が設定されなければなりません。

また、サブスクライバーではmax_replication_slotsの設定が必要です。この場合、少なくともサブスクライバーに追加する予定のサブスクリプション数が設定されている必要があります。パブリッシャーと同様に、max_logical_replication_workersは、少なくともサブスクリプション数に加えてテーブル同期のための予備が設定されていなければなりません。加えて、レプリケーションワーカーを収容するために、max_worker_processesを少なくとも(max_logical_replication_workers + 1)に調整する必要があるかもしれません。ある種の拡張とパラレルクエリは、max_worker_processesからワーカースロットを使うことに留意してください。

30.9. 簡単な設定

まずpostgresql.confの設定オプションを設定してください。

```
wal_level = logical
```

基本的な設定のためには、それ以外の設定はデフォルトのままで十分です。

pg_hba.confはレプリケーションを許可するために調整が必要です。(ここで示した値は、実際のネットワーク設定と、接続に使用するユーザにより異なります。)

```
host     all     repuser     0.0.0.0/0     md5
```

次にパブリッシャーデータベースで以下を実行します。

```
CREATE PUBLICATION mypub FOR TABLE users, departments;
```

サブスクライバーデータベースでは次を実行します。

```
CREATE SUBSCRIPTION mysub CONNECTION 'dbname=foo host=bar user=repuser' PUBLICATION mypub;
```

上記により、テーブルusersとdepartmentsの初期内容の同期プロセスが起動されます。その後、これらのテーブルへの増分変更のレプリケーションが開始します。

第31章 実行時コンパイル(JIT)

本章では、実行時コンパイル(just-in-time compilation)とは何か、そしてPostgreSQLでそれをどのように設定できるかを説明します。

31.1. JITコンパイルとは何か?

実行時(JIT)コンパイルとは、ある形式のインタプリタプログラムの評価をネイティブプログラムに変換する過程であり、かつそれを実行時に行うことを指します。たとえば、`WHERE a.col = 3`のような特定のSQL述語を評価するために、任意のSQL式を評価できる汎用目的のコードを使う代わりに、その式専用の関数を生成し、CPUによってネイティブに実行して速度向上をもたらすことができます。

PostgreSQLが`--with-llvm`でビルドされている場合、PostgreSQLにはLLVM[1]を使ってJITコンパイルを実行するためのサポートが組み込まれます。

さらなる詳細は`src/backend/jit/README`をご覧ください。

31.1.1. JITにより高速化される処理

今の所、PostgreSQLのJIT実装は、式評価とタプルデフォーミング(tuple deforming)の高速化をサポートしています。将来は他の操作も高速化されるかも知れません。

式評価は、`WHERE`句、ターゲットリスト、集約、射影を評価するために使用されます。それぞれのケースに応じたコードを生成することによって高速化することができます。

タプルデフォーミングは、ディスク上のタプル(68.6.1参照)をメモリ上の表現に変換する処理です。これはテーブルレイアウトと抽出するカラム数に特化した関数を作ることによって高速化可能です。

31.1.2. インライン展開(Inlining)

PostgreSQLは拡張性が高く、新しいデータ型、関数、演算子、その他のデータベースオブジェクトを定義することが可能です。第37章を参照してください。実際、組み込みオブジェクトは似た機構を使って実装されています。この拡張性は、たとえば関数呼び出し(37.3参照)により、幾分のオーバーヘッドをもたらします。このオーバーヘッドを軽減するために、JITコンパイルは、小さな関数の本体をそれを使っている式にインライン展開することができます。これにより、オーバーヘッドのかなりの部分を最適化によって解消することができます。

31.1.3. 最適化

LLVMは、生成したコードの最適化をサポートしています。ある最適化はJITが使用される際に常に適用できるほど安価ですが、長時間実行する問い合わせのときだけ有利になるようなものもあります。最適化についてのさらなる詳細は、https://llvm.org/docs/Passes.html#transform-passesをご覧ください。

[1] https://llvm.org/

31.2. どんなときにJITを使うべきか?

JITコンパイルは、主に長時間実行するCPUバウンドの問い合わせに有益です。短い問い合わせでは、JITコンパイルを行うことにより加わるオーバーヘッドはしばしばそれによって短縮できる時間よりも大きくなるでしょう。

JITコンパイルを使うべきかどうかを決めるために、問い合わせの合計見積もり実行時間(第70章と19.7.2を参照)が使用されます。問い合わせの見積もりコストはjit_above_costの設定と比較されます。もしもそのコストが大きければ、JITコンパイルが実行されます。さらなる二つの決定が必要になります。まず、見積もりコストがjit_inline_above_costの設定よりも大きければ、問い合わせ中で使用される短い関数と演算子がインライン展開されます。次に、見積もりコストがjit_optimize_above_costの設定よりも大きければ、生成コードを改善するために、高価な最適化が適用されます。これらのオプションはJITコンパイルのオーバーヘッドを大きくしますが、かなりクエリの実行時間を短縮します。

これらのコストに基づく決定は実行時ではなく、プラン時に行われます。このことは、準備された文が使われ、汎用プラン(PREPARE参照)が用いられるときには、実行時ではなく、準備時に参照される設定パラメータの値が決定を左右することを意味します。

> **注記**
>
> jitがoffか、JIT実装が適用外(たとえばサーバが--with-llvm付きでコンパイルされていない)場合は、たとえ上記の基準からは有益であったとしてもJITは実行されません。jitをoffにすると、プラン時と実行時の両方に影響を与えます。

EXPLAINを使ってJITが使われているかどうかを確認できます。JITを使っていない例を示します。

```
=# EXPLAIN ANALYZE SELECT SUM(relpages) FROM pg_class;
                                                     QUERY PLAN
---------------------------------------------------------------------
----------------------------------------
 Aggregate  (cost=16.27..16.29 rows=1 width=8) (actual time=0.303..0.303 rows=1 loops=1)
   ->  Seq Scan on pg_class  (cost=0.00..15.42 rows=342 width=4) (actual time=0.017..0.111
 rows=356 loops=1)
 Planning Time: 0.116 ms
 Execution Time: 0.365 ms
(4 rows)
```

プランに与えられたコストによれば、JITが使われないのは完全に合理的です。JITのコストは潜在的な節約よりも大きいのです。コスト上限を調整すると、JITが使われるようになります。

```
=# SET jit_above_cost = 10;
SET
=# EXPLAIN ANALYZE SELECT SUM(relpages) FROM pg_class;
                                                     QUERY PLAN
---------------------------------------------------------------------
----------------------------------------
```

```
Aggregate  (cost=16.27..16.29 rows=1 width=8) (actual time=6.049..6.049 rows=1 loops=1)
  ->  Seq Scan on pg_class  (cost=0.00..15.42 rows=342 width=4) (actual time=0.019..0.052
rows=356 loops=1)
Planning Time: 0.133 ms
JIT:
  Functions: 3
  Options: Inlining false, Optimization false, Expressions true, Deforming true
  Timing: Generation 1.259 ms, Inlining 0.000 ms, Optimization 0.797 ms, Emission 5.048 ms,
Total 7.104 ms
Execution Time: 7.416 ms
```

これを見るとわかるように、JITは使われていますが、インライン展開と高価な最適化は行われていません。加えてjit_inline_above_costあるいはjit_optimize_above_costを小さくすれば、これは変わることでしょう。

31.3. 設定

設定パラメータのjitは、JITが有効か無効化を決定します。有効ならば、設定値jit_above_cost、jit_inline_above_cost、jit_optimize_above_costは問い合わせでJITコンパイルが実行されるかどうか、どの程度の努力がJITコンパイルに払われるのかを決定します。

jit_providerはどのJIT実装が使われるのかを決定します。めったに変更する必要はありません。31.4.2を参照してください。

19.17にあるように、開発とデバッグ目的のために少数の追加設定パラメータがあります。

31.4. 拡張性

31.4.1. 拡張のためのインライン展開サポート

PostgreSQLのJIT実装は、Cとinternal型の関数の本体をインライン展開できます。そうした関数に基づく演算子も同様です。拡張の関数に同じことを行うには、関数の定義が入手可能である必要があります。LLVM JITサポートがコンパイルされているサーバに対してPGXSを使って拡張をビルドする際に、関連するファイルは自動的にビルドされ、インストールされます。

関連するファイルは$pkglibdir/bitcode/$extension/に、そのサマリは$pkglibdir/bitcode/$extension.index.bcにインストールされなければなりません。ここで、$pkglibdirは、pg_config --pkglibdirが返すディレクトリで、$extensionは拡張の共有ライブラリのベース名です。

> **注記**
>
> PostgreSQL自身に組み込まれた関数については、ビットコードが$pkglibdir/bitcode/postgresにインストールされます。

31.4.2. プラグ可能JITプロバイダ

PostgreSQLはLLVMに基づいたJIT実装を提供します。JITプロバイダのインタフェースはプラグ可能で、プロバイダは再コンパイルすることなく変更できます。(ただし今のところ、ビルドプロセスはLLVM用のインライン展開サポートデータのみを提供しています。) 有効なプロバイダはjit_providerの設定で選択できます。

31.4.2.1. JITプロバイダインタフェース

名前付きの共有ライブラリをロードすることにより、JITは動的にロードされます。ライブラリを特定するために通常のライブラリサーチパスが使用されます。必要なJITプロバイダコールバックを提供し、かつそのライブラリが実際にJITプロバイダであることを示すために、_PG_jit_provider_initという名前のC関数を提供する必要があります。この関数には構造体が渡され、その構造体には各々の動作用のコールバック関数へのポインタが設定される必要があります。

```
struct JitProviderCallbacks
{
    JitProviderResetAfterErrorCB reset_after_error;
    JitProviderReleaseContextCB release_context;
    JitProviderCompileExprCB compile_expr;
};

extern void _PG_jit_provider_init(JitProviderCallbacks *cb);
```

第32章 リグレッションテスト

リグレッションテストとは、PostgreSQLのSQL実装についての包括的なテストの集まりです。リグレッションテストでは、標準SQLの操作に加えてPostgreSQLの拡張SQL機能もテストします。

32.1. テストの実行

リグレッションテストは既にインストールされ稼働中のサーバや、ビルドツリー内の一時的なインストレーションに対して実行することができます。さらに、テストの実行には「並行」と「連続」モードがあります。連続モードでは個々のテストスクリプトを単独で実行し、並行モードでは複数のサーバプロセスを実行し、テストをグループ化して並行的に実行します。並行テストではプロセス間通信とロック機能が正常に作動しているかをテストします。

32.1.1. 一時的なインストレーションに対するテストの実行

構築後、インストール前に並行リグレッションテストを行う場合には、最上位のディレクトリで以下のように入力してください。

```
make check
```

（または、src/test/regressディレクトリに移動して、そこで実行してください。）終了したら以下のような表示がされるはずです。

```
=======================
 All 193 tests passed.
=======================
```

これが表示されなければ、テストは失敗したことになります。「失敗」を深刻な問題であると推測する前に、以下の 32.2 を参照してください。

この試験方法では、一時的にサーバを起動するので、rootユーザとして構築を行なった場合には動作しません。サーバがrootでは起動しないからです。rootで構築をしないこと、もしくはインストール完了後に試験を実施することをお薦めします。

古いPostgreSQLのインストレーションが既に存在している場所にPostgreSQLをインストールするように構築した場合、新しいバージョンをインストールする前にmake checkを行うと、新しいプログラムがインストール済みの共有ライブラリを使用しようとするために試験が失敗することになります。（典型的な症状は、未定義シンボルに関するエラーメッセージです。）古いインストレーションを上書きする前に試験を行いたいのであれば、configure --disable-rpathで構築する必要があります。しかし、このオプションを最終的なインストレーションで使用することは推奨しません。

並行リグレッションテストは、実行したユーザのユーザIDを使用して相当数のプロセスを起動します。現在、 最大で20個の並行テストスクリプトが同時に実行されますが、これは合計40個のプロセスが実行されることを意味します。各テストスクリプトに対して、1つのサーバプロセスと1つのpsqlプロセスが存在するためです。

ですので、使用するシステムでユーザ当たりのプロセス数に制限を加えている場合は、その上限が少なくとも50程度であることを確認してください。さもないと、並行テストにおいて、ランダムに発生しているように見える失敗が発生するかもしれません。この上限を変更できない場合は、MAX_CONNECTIONSパラメータを編集して、並行度を減らすことができます。例えば、以下は同時実行数を10以下で実行します。

```
make MAX_CONNECTIONS=10 check
```

32.1.2. 既存のインストレーションに対するテストの実行

インストール(第16章を参照)後にテストを実行するには、第18章で説明したようにデータディレクトリを初期化し、サーバを起動し、そして以下を入力してください。

```
make installcheck
```

もしくは、並行テストの場合は以下を入力してください。

```
make installcheck-parallel
```

テストでは、PGHOST環境変数とPGPORT環境変数で指定がない限り、ローカルホストのサーバに接続し、デフォルトのポート番号を使用します。テストはregressionという名前のデータベースで行なわれます。この名前の既存のデータベースはすべて削除されます。

テストは、ロールやテーブル空間、サブスクリプションのようなクラスタ全体にわたるオブジェクトも一時的に作成します。このオブジェクトの名前はregress_で始まります。実際のグローバルオブジェクトがそのように名付けられたインストレーションでinstallcheckモードを使う場合には注意してください。

32.1.3. 追加のテストスイート

make checkとmake installcheckコマンドは「コア」リグレッションテストだけを実行します。そのテストはPostgreSQLサーバに組み込まれている機能のみをテストします。ソース配布には、オプションとなっている手続き言語のような追加機能とその多くが関係のある追加のテストスイートが多く含まれています。

コアテストを含む、構築するよう選択されたモジュールに適用できるテストスイートをすべて実行するにはビルドツリーの最上位で以下のコマンドの一つを入力して下さい。

```
make check-world
make installcheck-world
```

make checkとmake installcheckで以前述べたように、このコマンドは、それぞれ、一時的なサーバもしくは既にインストールされているサーバを使ってテストを行ないます。それ以外に考慮すべきことはそれぞれのところで以前述べたことと同じです。make check-worldはテストするモジュール毎に別のインスタンス(一時的なデータディレクトリ)を構築しますので、make installcheck-worldよりもずっとより多くの時間とディスク容量が必要です。

複数のCPUコアがあり、オペレーティングシステムの厳しい制限のない最近のマシンでは、並列処理でかなり速くできます。ほとんどのPostgreSQL開発者がテストをすべて実行するのに実際に使っている方法は

```
make check-world -j8 >/dev/null
```

のようなものです。ここで-jの範囲は利用可能なコアの数に近い、もしくはそれより少し多い値です。stdoutを捨てることで、成功を検証する時には興味のない出力を除きます。(失敗した場合、どこをより詳細に調べるべきか決めるにはstderrメッセージでたいてい十分です。)

代わりに、構築ツリーの適切なサブディレクトリでmake checkまたはmake installcheckと入力することで個々のテストスイートを実行することもできます。make installcheckはコアサーバだけでなく、関係のあるモジュールもインストール済みであると仮定することを覚えておいて下さい。

このように実行できる追加のテストには以下のものが含まれます。

- オプションとなっている手続き言語のリグレッションテスト。これはsrc/plにあります。
- contribの下にあるcontribモジュールのリグレッションテスト。すべてのcontribモジュールにテストがあるわけではありません。
- ECPGインタフェースライブラリのリグレッションテスト。src/interfaces/ecpg/testにあります。
- コアがサポートする認証方式のテスト。src/test/authenticationにあります。(認証に関連する追加のテストについては下記を参照してください。)
- 同時実行中のセッションの振舞いの負荷テスト。src/test/isolationにあります。
- クラッシュリカバリと物理レプリケーションのテスト。src/test/recoveryにあります。
- 論理レプリケーションのテスト。src/test/subscriptionにあります。
- src/bin以下のクライアントプログラムのテスト。

installcheckモードを使う場合には、上記のテストは名前regressionを含むテストデータベースを破壊します。例えば、pl_regression、contrib_regressionです。非テストデータベースがそのように名付けられたインストレーションでinstallcheckモードを使う場合には注意してください。

この補助的なテストスイートの中には32.4で説明するTAP基盤を使うものがあります。オプション--enable-tap-testsを指定してPostgreSQLを構築した場合にのみ、TAPベースのテストが行なわれます。これは開発にはお薦めですが、適切なPerlのインストレーションがない場合には省略できます。

マルチユーザシステムにおいて安全に実行できない、あるいは、特別なソフトウェアを必要とする、といういずれかの理由により、一部のテストスイートはデフォルトでは実行されません。どのテストスイートを追加で実行するかをmakeや環境変数PG_TEST_EXTRAに空白区切りのリストを設定することで決定できます。以下に例を示します。

```
make check-world PG_TEST_EXTRA='kerberos ldap ssl'
```

現在、以下の値がサポートされます。

kerberos

src/test/kerberos以下のテストスイートを実行します。これはMIT Kerberosのインストールを必要とし、TCP/IPリッスンソケットを開きます。

ldap

src/test/ldap以下のテストスイートを実行します。これはOpenLDAPのインストールを必要としTCP/IPリッスンソケットを開きます。

ssl

src/test/ssl以下のテストスイートを実行します。これはTCP/IPリッスンソケットを開きます。

現在のビルド設定ではサポートされない機能のテストは、PG_TEST_EXTRAに記述されていても、実行されません。

さらに、make check-worldでは実行されますが、make installcheck-worldでは実行されないテストがsrc/test/modulesにあります。これは、実運用向けではない拡張をインストールしたり、実運用のインストレーションには望ましくない副作用があったりするためです。お望みとあらば、そのサブディレクトリの1つでmake installとmake installcheckを使うことはできますが、テスト用でないサーバでそうすることはお勧めしません。

32.1.4. ロケールと符号化方式

デフォルトでは、一時的なインストレーションを使うテストは、現在の環境で定義されたロケールとinitdbで決定される対応するデータベース符号化方式を使用します。異なるロケールを試験する際は、以下の例のように適切な環境変数を設定することが有用です。

```
make check LANG=C
make check LC_COLLATE=en_US.utf8 LC_CTYPE=fr_CA.utf8
```

実装上の理由のため、LC_ALLはこの目的には動作しません。この他のロケール関連の環境変数は動作します。

既存のインストレーションに対するテストでは、ロケールは既存のデータベースクラスタによって決まり、テスト実行時に別の値に設定することができません。

また、以下の例のようにENCODING変数を設定することで明示的にデータベース符号化方式を選択することができます。

```
make check LANG=C ENCODING=EUC_JP
```

この方法でデータベース符号化方式を設定することは、通常ロケールがCだった場合にのみ意味があります。この他の場合、ロケールから自動的に符号化方式が選択されます。ロケールと一致しない符号化方式を指定してもエラーになるだけです。

データベース符号化方式は一時的なインストレーションに対するテストでも既存のインストレーションに対するテストでも設定することができます。ただし、後者の場合にはインストレーションのロケールと互換性がなければなりません。

32.1.5. 追加のテスト

プラットフォームに依存する、または非常に時間がかかる可能性があるという理由で、コアリグレッションテスト一式にはデフォルトでは動作しないテストがいくつか含まれています。EXTRA_TESTS変数を設定すること

でこれらの追加テストやその他のテストを実行することができます。例えば、numeric_bigテストを以下のように実行します。

```
make check EXTRA_TESTS=numeric_big
```

32.1.6. ホットスタンバイのテスト

ソース配布は、ホットスタンバイの静的な挙動に対するリグレッションテストも含んでいます。これらのテストは、稼働しているプライマリサーバと、(ファイルベースのログ転送、またはストリーミングレプリケーションによって)プライマリからの新規のWALの変更を受け付けられる稼働中のスタンバイサーバを必要とします。これらのサーバは、自動的に作成されませんし、ここにはレプリケーション設定ドキュメントもありません。必要なコマンドや関連する問題について記述されている、ドキュメントのさまざまなセクションを参照してください。

ホットスタンバイテストを実行するには、最初にregressionという名前のデータベースをプライマリに作成します。

```
psql -h primary -c "CREATE DATABASE regression"
```

次に、準備のためのスクリプトsrc/test/regress/sql/hs_primary_setup.sqlをプライマリのregressionデータベース上で以下のように実行します。

```
psql -h primary -f src/test/regress/sql/hs_primary_setup.sql regression
```

この変更がスタンバイに伝搬するようにします。

ここで、デフォルトデータベース接続がスタンバイサーバの試験環境になるように(例えば、PGHOSTとPGPORT環境変数を設定することで)手配してください。最後に、リグレッションテスト用のディレクトリからmake standbycheckを実行してください。

```
cd src/test/regress
make standbycheck
```

いくつかの極端な挙動を、スタンバイのテストのための挙動を実現するスクリプトsrc/test/regress/sql/hs_primary_extremes.sqlを用いることでプライマリサーバ上で生成することができます。

32.2. テストの評価

適正にインストールされ、かつ、すべての機能が使用できるようなPostgreSQLインストレーションであっても、浮動小数点の表現やメッセージ内の単語順など、プラットフォーム特有の誤差のために「失敗」することがあります。現在のテストは単純に、(基準となる)参照用システムで生成した出力とのdiffを取ることで結果の検証を行っているため、システムの些細な違いにも反応します。結果が「失敗」となった場合は、予測された結果と実際の結果との差分を必ず評価してください。それらの差異が重大ではないことが判明することもあります。なお、すべてのテストが成功するように、サポートするすべてのプラットフォームに対する正確な参照ファイルの保守に努めています。

実際のリグレッションテストの出力は、src/test/regress/resultsディレクトリ内のファイルに書き込まれます。テストスクリプトはdiffを使用して、各出力ファイルとsrc/test/regress/expectedディレクトリ内の参照用出力とを比較します。あらゆる差異は調査用にsrc/test/regress/regression.diffsに保存されます。(コアテスト以外のテストスイートを実行する場合には、上記のファイルはもちろんsrc/test/regressではなく適切なサブディレクトリに現れます。)

デフォルトで利用されているdiffオプションが気に入らなければ、例えばPG_REGRESS_DIFF_OPTS='-c'のように、環境変数PG_REGRESS_DIFF_OPTSを設定して下さい。(あるいは、自分でdiffを実行することもできます。)

何らかの理由で、特定のプラットフォームが指定した試験で「失敗」し、その出力の調査により結果の方が有効であると確信できた場合、新しい比較用ファイルを追加し、今後の試験で失敗の報告が発生しないようにすることができます。詳細は32.3を参照してください。

32.2.1. エラーメッセージの違い

リグレッションテストのいくつかは、意図的に無効な入力値を使用します。エラーメッセージはPostgreSQLのコードによるもの、または使用しているプラットフォームの関数によるものがあります。後者の場合、プラットフォームによって違いがあるかもしれませんが、似たような内容になるはずです。これらのメッセージの違いによりリグレッションテストは「失敗」する可能性がありますが、これらは検査で確認できます。

32.2.2. ロケールの違い

Cロケール以外の照合順序のロケールで初期化されたサーバに対してテストを実行する際には、ソート順やその後に発生する失敗に違いが生じる可能性があります。リグレッションテストスイートはこの問題を解決するために、多くのロケールを処理するための代替の結果ファイルを提供するように設定されています。

一時的なインストレーションを使用して、異なるロケールのテストを実行するためには、以下の例のように、makeコマンドラインに適切なロケール関連の環境変数を渡してください。

```
make check LANG=de_DE.utf8
```

(リグレッションテストのドライバはLC_ALLを設定しないため、この変数を使ってロケールを選択することはできません。) ロケール無しを使用するためには、すべてのロケール関連の環境変数を設定しない(または、それらをCに設定する)か、もしくは以下の特殊な起動を行います。

```
make check NO_LOCALE=1
```

既存のインストレーションに対してテストを実行する場合は、ロケール設定は既存のインストレーションによって決まります。変更するためには、initdbに適切なオプションを渡して、異なるロケールでデータベースクラスタを初期化してください。

実際に実運用で使用されるロケールおよび符号化方式に関連した部分のコードが検証されますので、一般的には、実運用で使用されるロケール設定でリグレッションテストを実行することを推奨します。オペレーティングシステム環境に依存して、結果が失敗する場合もありますが、少なくとも実際のアプリケーションを実行する時に想定されるロケール固有の動作を知ることができます。

32.2.3. 日付と時刻の違い

日付と時刻の結果のほとんどはタイムゾーンの環境に依存します。参照ファイルはタイムゾーン PST8PDT (Berkeley, California) 用に生成されているため、このタイムゾーン設定で実行されていないテストは明らかに失敗します。リグレッションテストのドライバは、適切な結果を保証するために、環境変数PGTZにPST8PDTを設定します。

32.2.4. 浮動小数点数の違い

いくつかのテストでは、64ビット(double precision型)の浮動小数点数値をテーブルの列から取り出して計算を行います。double precision列における数学演算関数では、異なった結果が発生する場合があることが知られています。float8とgeometryテストは特に、プラットフォーム間、またはコンパイラの最適化の設定による小さな違いが起こりやすくなります。これらの違い、通常は小数点以下10桁目以降の相違の、実際の影響度を判断するためには、人間の目で実際に確認する必要があります。

いくつかのシステムではマイナス0を-0と表示することがあり、その他のシステムでは単に0と表示します。

いくつかのシステムでは、現在のPostgreSQLのコードが想定しているメカニズムと異なるために、pow()とexp()でエラーを発生する場合があります。

32.2.5. 行の順序の違い

同じ行の出力が、参照ファイルで記述されている順序とは異なっている場合があります。ほとんどの場合、これは厳密に言ってバグではありません。ほとんどのリグレッションテストは、各SELECT文に対してORDER BYを使用するほど規則に厳しくなく、そのため、結果の行の順序はSQLの仕様に従って、明確に決まっていません。実際には、同じソフトウェアで同じデータを用いて同じ問い合わせで参照しているので、すべてのプラットフォームで同じ順序の結果が返されるため、ORDER BYがないことは問題ではないと言えます。しかし、問い合わせによっては、プラットフォーム間の順序の違いが起こる可能性があります。インストール済みのサーバに対して試験を行う場合、C以外のロケール、独自のwork_memや独自のプランナ用のコストパラメータなどデフォルト以外のパラメータ設定により順序の違いが生じる可能性があります。

したがって、順序の違いを見つけた場合、問い合わせにORDER BYが含まれていて順序が影響を及ぼす場合以外は、気にする必要はありません。ただし、その場合にもとにかくご一報ください。特定の問い合わせから偽の「失敗」を取り除くために、将来のリリースにおいて、ORDER BYを追加します。

このような問題を避けるために、なぜ我々がすべてのリグレッションテストに対して明示的に順序を指定しないのか、疑問に思うかもしれません。その理由は、ソートが必要がない場合であってもソートされた結果を生成する問い合わせ計画を実行しようとすることによって、リグレッションテストの意義が増すわけではなく、むしろ減るからです。

32.2.6. スタック長の不足

errorsテストがselect infinite_recurse()コマンドでサーバをクラッシュさせた場合、プラットフォームのプロセススタックサイズがmax_stack_depthパラメータが示す値よりも小さいことを意味します。これは、

スタックサイズ制限を高くして（デフォルトのmax_stack_depthでの推奨値は4メガバイト）サーバを実行することで修正することができます。これを行うことができない場合、max_stack_depthの値を少なくすることが代替方法です。

getrlimit()をサポートするプラットフォームでは、サーバは自動的にmax_stack_depthの安全な値を選ぶべきです。この設定を手で上書きしたのでない限り、この種の失敗は報告する価値のあるバグです。

32.2.7.「乱数」テスト

randomテストスクリプトは、無作為な結果を生成することを目的としています。非常に稀ですが、これがリグレッションテストが失敗する原因になることがあります。次のように、

```
diff results/random.out expected/random.out
```

と入力すると、ほんの数行だけの差異が生じるはずです。繰り返し失敗しない限り、気に留める必要はありません。

32.2.8. 設定パラメータ

既存のインストレーションに対してテストを実行する場合、デフォルトでないパラメータ設定はテストが失敗する原因になり得ます。例えば、enable_seqscanやenable_indexscanのようなパラメータの変更は、EXPLAINを使うテストの結果に影響する計画の変更の原因となり得ます。

32.3. 各種の比較用ファイル

試験の中には必然的に環境に依存した結果となるものがありますので、「expected」結果ファイルの代替を指定する方法を用意しています。各リグレッションテストは、異なるプラットフォームで出力される可能性がある、複数の比較用ファイルを持つことができます。各試験に対してどの比較用ファイルを使用するかを決定する方法には、独立した2つの機構があります。

1つ目のメカニズムにより、特定のプラットフォームのための比較用ファイルを選ぶことができます。関連付けを行うsrc/test/regress/resultmapというファイルがあり、どの比較用ファイルがどのプラットフォームで使用されるのかを定義します。特定のプラットフォームにおいて試験の「失敗」の誤検知を防ぐためには、まず結果ファイルを選ぶ、あるいは結果ファイルを作成してから、resultmapファイルに1行加えてください。

マッピングファイルの各行の書式は下記の通りです。

```
testname:output:platformpattern=comparisonfilename
```

testnameは特定のリグレッションテストのモジュール名です。outputの値は、どの出力ファイルを検査するのかを示します。標準のリグレッションテストでは、これは常にoutです。この値は出力ファイルの拡張子に対応します。platformpatternとは、expr Unixツールスタイル（最初に暗黙的な^がある正規表現）のパターンです。これはconfig.guessによって出力されるプラットフォーム名と比較されます。comparisonfilenameは置き換える結果比較ファイルの（ディレクトリ部分を除いた）名前です。

以下に例を示します。システムの中には、動作するstrtof関数がないものがあり、そのため私たちの回避策がfloat4リグレッションテストでの丸め誤差の原因となります。そのため、float4-misrounded-input.outという異なる比較ファイルを用意し、そこにこういったシステムでの期待される値を記述します。HP-UX 10プラットフォームにおいて偽の「失敗」メッセージ出力を行わせないようにするために、resultmapに以下を含めます。

```
float4:out:hppa.*-hp-hpux10.*=float4-misrounded-input.out
```

これは、config.guessの出力がhppa.*-hp-hpux10.*に一致するすべてのマシンに対して適用されます。resultmapのその他の行は、他のプラットフォーム向けの適切な比較ファイルを選択します。

2つ目の比較用ファイルの選択の仕組みはかなり自動化されています。これは単純に、提供されている各種比較用ファイルの中から「もっとも一致するもの」を使用します。リグレッションテストのドライバスクリプトは、 試験において、標準の比較用ファイルtestname.outとtestname_digit.out（ここでdigitは0-9のいずれかからなる1つの数字です）という名前の別のファイルの両方を考慮します。もしこの中のいずれかのファイルが正確に一致した場合、試験が成功したものとみなします。さもなくば、生成されたdiffの結果がもっとも小さかった結果ファイルを選択して、失敗報告を生成します。（resultmapに特定の試験用の項目が含まれていると、resultmap内の名前が元となるtestnameに置き換えられます。）

例えば、charの試験では、比較用ファイルchar.outにはCロケールとPOSIXロケールで想定される結果が含まれています。一方、char_1.outファイルには、他の多くのロケールで現れる結果がソートされて含まれています。

この最善一致の仕組みは、ロケールに依存した結果に対応できるように考え出されました。しかし、この仕組みはプラットフォームの名前だけでは簡単に予測できる試験結果とならないような、任意の状況で使用することができます。この仕組みの制約は、現在の環境でどの種類の比較ファイルが本当に「正しい」のかが試験ドライバでは分からないという点です。単にもっともうまく動いていそうなものを選択しているだけだからです。したがって、すべての文脈で平等に有効とみなすことができるような種類の結果においてのみ利用するのが、もっとも安全です。

32.4. TAPテスト

様々なテスト、特にsrc/bin以下のクライアントプログラムテストはPerl TAPツールを使い、Perlテストプログラムproveを使って実行されます。make変数PROVE_FLAGSを設定することでproveコマンドラインオプションを渡すことができます。例えば、

```
make -C src/bin check PROVE_FLAGS='--timer'
```

詳細な情報はproveのマニュアルページを参照してください。

デフォルトのt/*.plに替えて、テストの指定サブセットを実行するために、make変数PROVE_TESTSを使用できます。proveを起動するMakefileからの相対パスの空白区切りのリストを指定します。以下に例を示します。

```
make check PROVE_TESTS='t/001_test1.pl t/003_test3.pl'
```

TAPテストはPerlモジュールIPC::Runが必要です。このモジュールはCPANもしくはオペレーティングシステムのパッケージから入手可能です。

一般的に言って、TAPテストは、make installcheckとした場合には以前インストールしたインストレーションツリーの実行ファイルをテストし、make checkとした場合には現在のソースから新しいインストレーションツリーを構築します。どちらの場合も、ローカルインスタンス(データディレクトリ)を初期化し、その中で一時的にサーバを実行します。テストの中には2つ以上のサーバを実行するものがあります。従って、このテストはかなりリソース集約的になる可能性があります。

make installcheckとした場合でも、TAPテストはテストサーバを開始することを理解しておくことは重要です。これは伝統的な非TAPテスト基盤とは異なります。非TAPテスト基盤ではその場合、既に動作しているテストサーバを使うことが期待されます。PostgreSQLのサブディレクトリには、伝統的な形式のテストとTAP形式のものの両方を含むものがありますので、make installcheckは一時的なサーバと既に動作しているテストサーバからの結果を寄せ集めることになります。

32.5. テストが網羅する範囲の検証

PostgreSQLソースコードは、カバレッジテストツールとともにコンパイルすることができるため、リグレッションテスト、あるいはその他のテストスイートによって、コードのどの部分が網羅されているかを評価することができます。これは現在、GCCを使用してコンパイルした時にサポートされ、gcovおよびlcovプログラムを必要とします。

典型的な作業の流れは以下のようになります。

```
./configure --enable-coverage ... OTHER OPTIONS ...
make
make check # or other test suite
make coverage-html
```

そして、HTMLブラウザでcoverage/index.htmlを参照します。gmakeコマンドはサブディレクトリ内においても同様に動作します。

lcovが無い、あるいは、HTMLレポートよりもテキスト出力を好むなら、make coverage-htmlの代わりに以下を実行してください。

```
make coverage
```

これはテストに関連する各ソースファイルに対して.gcov出力ファイルを生成します。(make coverageとmake coverage-htmlは互いのファイルを上書きしますので、混用は混乱をひき起こすかもしれません)

複数回の試験を実行する時、実行回数をリセットするためには以下を実行します。

```
make coverage-clean
```

パート IV. クライアントインタフェース

ここではPostgreSQLに附属するクライアントプログラミングインタフェースについて説明します。各章は独立して読むことができます。クライアントプログラムには、この他にも様々なプログラミングインタフェースがありますが、これらのインタフェースは独自の資料とともに個別に配布されていますのでご注意ください（付録Hに人気があるインタフェースの一部を列挙しています）。読者は、データベースの操作や問い合わせを行うためのSQLコマンド（パート IIを参照）、また、当然ながら、インタフェースが使用するプログラミング言語にも慣れ親しんでいることが必要です。

目次

第33章 libpq — C ライブラリ

libpqは、C言語によるアプリケーションプログラマ用のPostgreSQLインタフェースです。libpqは、クライアントプログラムからPostgreSQLのバックエンドサーバに問い合わせを渡し、その結果を受け取るためのライブラリ関数の集合です。

libpqは、C++、Perl、Python、Tcl、ECPGなどを含む、PostgreSQLの他の各種アプリケーションインタフェースを支えるエンジンでもあります。従って、libpqの動作は、これらのパッケージを使用する人にとって重要なものになります。特に、33.14、33.15および33.18にて、libpqを使用するすべてのアプリケーションのユーザから見える動作を説明します。

本章の最後に、libpqの使い方を示す、いくつかの短いプログラム(33.21)があります。また、ソースコード配布物内のsrc/test/examplesディレクトリに、libpqを利用したアプリケーションプログラム一式の例があります。

libpqを使用してフロントエンドプログラムを作成するには、libpq-fe.hヘッダファイルのインクルードと、libpq ライブラリとのリンクが必要です。

33.1. データベース接続制御関数

PostgreSQLのバックエンドサーバとの接続を作成するには、以下の関数を使用します。アプリケーションプログラムはバックエンドとの接続を一度に複数個開くことができます。(そのようにする1つの理由として、複数のデータベースへのアクセスが挙げられます。) 個々の接続は、PQconnectdb、PQconnectdbParamsまたはPQsetdbLogin関数を呼び出すことで得られるPGconnオブジェクトによって表されます。なお、これらの関数は、PGconnオブジェクトに割り当てるほんのわずかなメモリの余裕さえもない場合を除き、NULLではなく常にオブジェクトのポインタを返します。また、この接続オブジェクトを通じて問い合わせを送る前に、PQstatus関数を呼び出して、データベースとの接続に成功したか戻り値を検査しなければなりません。

警告

信頼できないユーザが、安全なスキーマ使用パターンを適用していないデータベースへアクセスする際には、セッション開始時にsearch_pathから、第三者が書き込みができるスキーマを削除してください。これはoptionsパラメータキーワードに値-csearch_path=を設定することで可能となります。別の方法としては、接続後にPQexec(conn, "SELECT pg_catalog.set_config('search_path', '', false)")を発行しても構いません。このような配慮は、libpqに限ったことではありません。任意のSQLコマンドを実行するすべてのインタフェースに当てはまります。

警告

Unix上で、libpq接続を開いたプロセスのフォークは、親と子のプロセスが同じソケットとオペレーティングシステムの資源を共有するため、予期せぬ結果を招くことがあります。この理由により、新規実行形式を子プロセスが読み込むためexecを行うことが安全と言っても、このような使用方法は推奨されません。

PQconnectdbParams

新たにデータベースサーバへの接続を作成します。

```
PGconn *PQconnectdbParams(const char * const *keywords,
                          const char * const *values,
                          int expand_dbname);
```

この関数は、2つのNULL終端の配列から取得したパラメータを使用して、データベースとの接続を新たに1つ確立します。1つ目は文字列配列として定義されるkeywordsで、それぞれがキーワードとなります。2つ目はvaluesで、各キーワードの値を提供します。後述のPQsetdbLoginとは異なり、関数のシグネチャを変更せずにパラメータ集合を拡張できますので、アプリケーションプログラムを新たに作成する際には、この関数(もしくは非ブロックモードでよく似た処理をするPQconnectStartParamsとPQconnectPoll)を使用することをお勧めします。

現在有効なパラメータキーワードを33.1.2に示します。

expand_dbnameが非ゼロの場合、dbnameキーワードの値を接続文字列として認識させることができます。最初に出現したdbnameだけがこのように展開され、後続のdbname値は通常のデータベース名として処理されます。接続文字列の取り得る書式に関する詳細については33.1.1を参照してください。

空の配列を渡してすべてデフォルトパラメータを使用することができます。また渡される配列に1つ以上のパラメータ設定を持たせることもできます。これらの長さは一致しなければなりません。keywords配列の最初のNULL要素で処理は停止します。

パラメータがNULLや空文字列の場合には、対応する環境変数(33.14参照)が検査されます。環境変数も設定されていない場合は、組み込みのデフォルト値が使用されます。

一般的にキーワードはこれらの配列の先頭からインデックス順で処理されます。この影響はキーワードが繰り返された場合で、最後に処理された値が残ることになります。このため、dbnameキーワードの記述場所に注意することで、conninfo文字列により何が上書きされるか、何が上書きされないかを決定することができます。

PQconnectdb

新たにデータベースサーバへの接続を作成します。

```
PGconn *PQconnectdb(const char *conninfo);
```

この関数はconninfo文字列から取得されるパラメータを使用して、新しいデータベース接続を開きます。

空の文字列を渡してすべてデフォルトパラメータを使用することができます。また空白文字で区切ることで1つ以上のパラメータ設定を持たせることもできます。さらにURIを含めることができます。詳細については33.1.1を参照してください。

PQsetdbLogin

新たにデータベースサーバへの接続を作成します。

```
PGconn *PQsetdbLogin(const char *pghost,
                     const char *pgport,
                     const char *pgoptions,
                     const char *pgtty,
                     const char *dbName,
                     const char *login,
                     const char *pwd);
```

これはパラメータ群を固定したPQconnectdbの前身です。設定できないパラメータが常にデフォルト値になる点を除き、同一の機能を持ちます。固定のパラメータに対してNULLもしくは空文字列とすると、それはデフォルトを使用することになります。

dbName内に=記号が含まれる場合、または有効な接続URI接頭辞を持つ場合、PQconnectdbに渡された場合とまったく同じ扱いでconninfo文字列として扱われます。その後残りのパラメータがPQconnectdbParamsの指定のように適用されます。

PQsetdb

新たにデータベースサーバへの接続を作成します。

```
PGconn *PQsetdb(char *pghost,
                char *pgport,
                char *pgoptions,
                char *pgtty,
                char *dbName);
```

これは、loginとpwdにNULLポインタを設定するPQsetdbLoginを呼び出すマクロです。非常に古いプログラムへの後方互換性のために提供されています。

PQconnectStartParams
PQconnectStart
PQconnectPoll

ブロックしない方法で、データベースサーバへの接続を作成します。

```
PGconn *PQconnectStartParams(const char * const *keywords,
                             const char * const *values,
                             int expand_dbname);

PGconn *PQconnectStart(const char *conninfo);

PostgresPollingStatusType PQconnectPoll(PGconn *conn);
```

これら3つの関数は、リモートI/Oの実行時にアプリケーションスレッドの実行がブロックされないようなデータベースサーバへの接続を作成するために使われます。この手法の利点は、I/Oの終了待ちがPQconnectdbParamsまたはPQconnectdb内部ではなく、アプリケーションプログラムのメインループでできることにあります。これによって、アプリケーションは他の処理と並行してこの処理を管理することができます。

PQconnectStartParamsでは、上でPQconnectdbParamsで説明したように、データベース接続はkeywordsおよびvalues配列から取得され、expand_dbnameによって制御されたパラメータを使用して確立します。

PQconnectStartでは、上でPQconnectdbで説明したように、conninfo文字列から取得されたパラメータを使用してデータベース接続を確立します。

PQconnectStartParams、PQconnectStartとPQconnectPollのどちらも以下の制限に適合する場合ブロックしません。

- hostaddrパラメータは、DNS問い合わせが発生するのを防ぐように適切に使用されなければいけません。詳細については33.1.2内のパラメータ説明を参照してください。
- PQtraceを呼び出す場合は、トレースに使用するストリームオブジェクトがブロックされないことが保証されていなくてはなりません。
- プログラマ自身が、後に示すように、PQconnectPollを呼び出す前にソケットが適切な状態にあることを保証しなくてはいけません。

非ブロック接続要求を始めるにはまず、PQconnectStartかPQconnectStartParamsを呼び出します。その結果がNULLの場合、libpqは新たなPGconn構造体を割り当てられませんでした。そうでない場合は、適切なPGconnへのポインタが返されます（ただし、未だデータベースへの有効な接続を示しているわけではありません）。次にPQstatus(conn)を呼び出します。もし、結果がCONNECTION_BADであった場合、接続の試みは失敗しています。典型的には無効な接続パラメータに因ります。

PQconnectStartあるいはPQconnectStartParamsが成功したら、次は接続シーケンスを進めるために、libpqをポーリングします。データベース接続の背後にあるソケットの記述子を取り出すには、PQsocket(conn)を使用します。（注意：複数のPQconnectPoll呼び出しでソケットが同じままであると思わないでください。）以下の繰り返しです。直前のPQconnectPoll(conn)がPGRES_POLLING_READINGの場合、（select()やpoll()などのシステム関数で示されて）ソケットの読み込み準備が整うまで待機します。そして、再度PQconnectPoll(conn)を呼び出します。反対に直前のPQconnectPoll(conn)がPGRES_POLLING_WRITINGの場合、ソケットの書き込み準備が整うまで待機し、その後、PQconnectPoll(conn)を再度呼び出します。繰り返しの最初、すなわち、未だPQconnectPollを呼び出していない場合、最後にPGRES_POLLING_WRITINGを返したかのように振舞います。この繰り返しをPQconnectPoll(conn)が、接続手続きの失敗を示すPGRES_POLLING_FAILED、もしくは、接続確立に成功したことを示すPGRES_POLLING_OKを返すまで継続します。

接続している間は、いつでもPQstatusを呼び出すことで、接続の状態を検査することができます。この関数呼び出しがCONNECTION_BADを返す場合、接続手続きは失敗しており、CONNECTION_OKを返す場合、接続が確立しています。上述のように、このいずれの状態も、PQconnectPollの戻り値から同様に検出できます。これ以外の状態は、非同期の接続手続きの間（のみに）現れることがあります。これらは、接続手続きの現在の段階を示すものであり、例えばユーザへのフィードバックを提供することに使用できます。以下の状態があります。

CONNECTION_STARTED

接続の確立待ち状態です。

CONNECTION_MADE

接続はOKです。送信待ち状態です。

CONNECTION_AWAITING_RESPONSE

サーバからの応答待ち状態です。

CONNECTION_AUTH_OK

認証済みです。バックエンドの起動待ち状態です。

CONNECTION_SSL_STARTUP

SSL暗号化の調停状態です。

CONNECTION_SETENV

環境が提供するパラメータ設定の調停状態です。

CONNECTION_CHECK_WRITABLE

接続が書き込みトランザクションを扱えるかどうかを調べています。

CONNECTION_CONSUME

接続の残りの応答メッセージを消費しています。

これらの定数は（互換性を保つため）なくなることはありませんが、アプリケーションは、これらが特定の順で出現したり、本書に書いてある値のどれかに必ずステータス値が該当するということを決して当てにしてはいけません。アプリケーションは、以下に示すようにするべきです。

```
switch(PQstatus(conn))
{
        case CONNECTION_STARTED:
            feedback = "Connecting...";
            break;

        case CONNECTION_MADE:
            feedback = "Connected to server...";
            break;
.
.
.
        default:
            feedback = "Connecting...";
}
```

PQconnectPollを使用する場合、connect_timeout接続パラメータは無視されます。経過時間が長過ぎるかどうかの判定はアプリケーションの責任で行ないます。さもないと、PQconnectStartの後のPQconnectPollの繰り返しがPQconnectdbと同じになります。

PQconnectStartやPQconnectStartParamsが非NULLポインタを返した場合、処理を終了する際には、構造体や関連するメモリブロックを始末するために、PQfinishを呼び出さなくてはならないことに注意し

てください。この処理は、接続試行が失敗した場合やその試行を中断する場合にも、必ず実行されなければいけません。

PQconndefaults

デフォルトの接続オプションを返します。

```
PQconninfoOption *PQconndefaults(void);

typedef struct
{

    char   *keyword;   /* このオプションのキーワード */
    char   *envvar;    /* 代替となる環境変数の名前 */
    char   *compiled;  /* 代替となるコンパイル時に組み込まれたデフォルト値 */
    char   *val;       /* オプションの現在値、もしくは、NULL */
    char   *label;     /* 接続ダイアログ内の当該フィールドのラベル */
    char   *dispchar;  /* 接続ダイアログ内の当該フィールドをどのように表示するかの指示
                          値:
                          ""          入力された値をそのまま表示
                          "*"         値を隠すパスワードフィールド用
                          "D"         デバッグオプション。デフォルトで何も表示しません */
    int     dispsize;  /* ダイアログ用のフィールドの大きさ(文字数単位) */
} PQconninfoOption;
```

接続オプションの配列を返します。これは、使用可能なPQconnectdb用オプションのすべてや、その時点でのデフォルト値を決定するために使用することができます。戻り値は、PQconninfoOption構造体の配列へのポインタで、keywordポインタがヌルとなる項目が配列の末尾にきます。メモリが確保できなかった場合にはヌルポインタを返します。現在のデフォルト値(val フィールド)は、環境変数や他のコンテキストに依存します。呼び出し側では、接続オプションの情報は、読み込み専用として取り扱わなければいけません。

オプションの配列を処理した後は、それをPQconninfoFreeに渡して解放します。この処理をしないと、PQconndefaultsが呼び出されるたびに少しずつメモリリークが発生します。

PQconninfo

所在する接続で使用される接続オプションを返します。

```
PQconninfoOption *PQconninfo(PGconn *conn);
```

接続オプション配列を返します。これは全ての可能性のあるPQconnectdbオプションとサーバに接続するのに使用される値を確定するために使用することができます。返り値はPQconninfoOption構造体の配列を指し示します。それはnull keyword ポインタを持つ項目で終結します。PQconndefaultsに対する上記の全ての注釈はまたPQconninfoの結果に適用されます。

PQconninfoParse

提供された接続文字列から構文解析された接続オプションを返します。

```
PQconninfoOption *PQconninfoParse(const char *conninfo, char **errmsg);
```

接続文字列の構文解析を行い、配列として結果オプションを返すか、または接続文字列に問題があった場合にNULLを返します。この関数を提供された接続文字列の中のPQconnectdbオプションを取り出すために使用することができます。戻り値はPQconninfoOption構造体の配列を指し示し、それはヌルのkeywordポインタを持つ項目で終結します。

正規なオプションはすべて、結果配列内に現れます。しかし接続文字列内に現れない、何らかのオプション用のPQconninfoOptionはNULLに設定されたvalを持ちます。デフォルトは挿入されません。

errmsgが非NULLであれば、成功した場合*errmsgはNULLに設定され、そうでなければ、問題を説明したmallocされたエラー文字列になります。(*errmsgがNULLに設定され、かつ、この関数がNULLを返すこともあり得ます。これはメモリ不足状態を意味します。)

オプション配列を処理した後、それをPQconninfoFreeに渡して解放してください。これが行われない場合、PQconninfoParseへのそれぞれの呼び出しに対してメモリーリークが起こります。反対に、エラーが起こり、そしてerrmsgが非NULLであれば、PQfreememを使用してエラー文字列を必ず解放してください。

PQfinish

サーバとの接続を閉ざします。また、PGconnオブジェクトが占めるメモリも解放します。

```
void PQfinish(PGconn *conn);
```

たとえサーバへの接続試行が失敗しても(PQstatusで調べます)、アプリケーションはPQfinishを呼び出しPGconnオブジェクトが占めるメモリを解放するべきです。そしてPQfinishを呼び出したら、もうPGconnへのポインタを使ってはいけません。

PQreset

サーバへの通信チャンネルをリセットします。

```
void PQreset(PGconn *conn);
```

この関数はサーバへの接続を閉じ、以前使用したパラメータをすべて使用して、同一のサーバへ新しく接続を確立します。これは、作業中の接続が失われた場合のエラーの修復に役立つでしょう。

PQresetStart
PQresetPoll

非ブロッキング方式で、サーバへの接続チャンネルをリセットします。

```
int PQresetStart(PGconn *conn);

PostgresPollingStatusType PQresetPoll(PGconn *conn);
```

これらの関数はサーバへの接続を閉じ、それから再度、以前使用したパラメータをすべて使用して、同じサーバと新たな接続を確立しようとします。これらは作業中の接続が失われた場合のエラー修復に役立つでしょう。PQreset(前述)との違いは、この2つの関数が非ブロック方式で動作することです。また、これらの関数はPQconnectStartParams、PQconnectStartおよびPQconnectPollと同じ制限を受けます。

接続のリセットを始めるためにはPQresetStartを呼び出します。この関数がゼロを返す場合、リセットに失敗しています。戻り値が1ならば、PQconnectPollを使って接続を確立した時とまったく同じに、PQresetPollを使用してリセットのポーリングを行います。

PQpingParams

PQpingParamsはサーバの状態を報告します。この関数は上述のPQconnectdbParamsと同じ接続パラメータを受け付けます。サーバの状態を得るために正しいユーザ名、パスワード、データベース名を提供する必要はありません。しかし、不適切な値が供給されると、サーバは不成功に終わった接続の試みをログに残します。

```
PGPing PQpingParams(const char * const *keywords,
                    const char * const *values,
                    int expand_dbname);
```

この関数は以下の値のいずれかを返します。

PQPING_OK

サーバは稼動中で、接続を受け付けているようです。

PQPING_REJECT

サーバは稼動中ですが、接続を許可しない状態（起動処理中、停止処理中、クラッシュリカバリ中）です。

PQPING_NO_RESPONSE

サーバと通信できません。これは、サーバが稼動中ではない、指定した接続パラメータの何か（例えばポート番号の間違い）が間違っている、ネットワーク接続性の問題（例えば接続要求をブロックするファイアウォール）があることを示しているかもしれません。

PQPING_NO_ATTEMPT

指定されたパラメータが明らかに間違っている、または、（メモリ不足など）クライアント側の問題があったため、サーバとの通信を試行しませんでした。

PQping

PQpingはサーバの状態を報告します。この関数は上述のPQconnectdbと同じ接続パラメータを受け付けます。サーバの状態を得るために正しいユーザ名、パスワード、データベース名を提供する必要はありません。しかし、不適切な値が供給されると、サーバは不成功に終わった接続の試みをログに残します。

```
PGPing PQping(const char *conninfo);
```

戻り値はPQpingParamsと同じです。

PQsetSSLKeyPassHook_OpenSSL

PQsetSSLKeyPassHook_OpenSSLはアプリケーションにsslpasswordや対話的なプロンプトを使ってlibpqの暗号化されたクライアント証明書キーファイルのデフォルト処理を置き換えさせます。

```
void PQsetSSLKeyPassHook_OpenSSL(PQsslKeyPassHook_OpenSSL_type hook);
```

アプリケーションは以下のシグネチャを持つコールバック関数のポインタを渡します。

```
int callback_fn(char *buf, int size, PGconn *conn);
```

libpqはデフォルトのPQdefaultSSLKeyPassHook_OpenSSLハンドラの代わりに、これを呼び出します。コールバックはキーに対するパスワードを決定して、それをsizeの大きさを持つ結果バッファbufにコピーすべきです。buf内の文字列はNULL終端でなければなりません。コールバックはbufに格納されたパスワードのヌル終端子を除いた長さを返さなければなりません。失敗した場合、コールバックはbuf[0] = '\0'をセットし、0を返すべきです。例として、libpqのソースコードのPQdefaultSSLKeyPassHook_OpenSSLを参照してください。

ユーザが明示的なキー位置を指定した場合、コールバックが実行されたときにそのパスがconn->sslkeyに含まれます。デフォルトのキーパスが使われている場合、これは空になります。エンジン指定子であるキーに対して、OpenSSLパスワードコールバックを使うか固有の処理を定義するかは、エンジン実装によります。

アプリケーションのコールバックが対応していない場合についてPQdefaultSSLKeyPassHook_OpenSSLに委託したり、最初に呼び出して0が返った場合に何らか他のことを試みたり、あるいは完全に上書きしたりすることにしても良いです。

コールバックは例外、longjmp(...)などで通常のフロー制御から脱出してはいけません。正常にリターンしなければなりません。

PQgetSSLKeyPassHook_OpenSSL

PQgetSSLKeyPassHook_OpenSSLは現在のクライアント証明書のキーパスワードのフックを、あるいは、設定されていない場合にNULLを返します。

```
PQsslKeyPassHook_OpenSSL_type PQgetSSLKeyPassHook_OpenSSL(void);
```

33.1.1. 接続文字列

複数のlibpq関数は、接続パラメータを得るためにユーザが指定した文字列の解析を行います。この文字列として、単純なkeyword = value文字列とURIという2種類の書式が受け付けられます。URIは通常RFC3986[1]に従いますが、以下で詳細を説明する複数ホスト接続文字列が使用できるところが例外です。

33.1.1.1. キーワード/値形式の接続文字列

最初の書式では、各パラメータ設定はkeyword = valueという形式です。等号記号の前後の空白文字は省略可能です。空の値を書く、または空白文字を含む値を書くためには、keyword = 'a value'のように単一引用符で値を括ります。値内部の単一引用符とバックスラッシュはバックスラッシュでエスケープしなければなりません。つまり\'と\\です。

以下に例を示します。

[1] https://tools.ietf.org/html/rfc3986

```
host=localhost port=5432 dbname=mydb connect_timeout=10
```

有効なパラメータキーワードを33.1.2に示します。

33.1.1.2. 接続URI

接続URIの一般的な形式を以下に示します。

```
postgresql://[user[:password]@][netloc][:port][,...][/dbname][?param1=value1&...]
```

URIスキーム指示子はpostgresql://またはpostgres://のいずれかを取ることができます。個々のURI部品は省略可能です。以下の例で有効なURI構文の使用例を示します。

```
postgresql://
postgresql://localhost
postgresql://localhost:5433
postgresql://localhost/mydb
postgresql://user@localhost
postgresql://user:secret@localhost
postgresql://other@localhost/otherdb?connect_timeout=10&application_name=myapp
postgresql://host1:123,host2:456/somedb?target_session_attrs=any&application_name=myapp
```

URIの階層部品の要素をパラメータとして与えることができます。以下に例を示します。

```
postgresql:///mydb?host=localhost&port=5433
```

接続URIは、その中のどこかの部分に特別な意味を持つシンボルを含む場合、パーセントエンコーディング[2]でエンコードされている必要があります。以下は等号(=)が%3D、空白が%20で置き換えられた例です。

```
postgresql://user@localhost:5433/mydb?options=-c%20synchronous_commit%3Doff
```

33.1.2に示されたキーワードに対応しない接続パラメータは無視され、これに関する警告メッセージがstderrに書き出されます。

JDBCの接続URI構文との互換性を高めるために、ssl=trueパラメータインスタンスはsslmode=requireに変換されます。

ホスト部分にはホスト名またはIPアドレスを書くことができます。IPv6ホストアドレスを指定するためには角括弧で括ります。

```
postgresql://[2001:db8::1234]/database
```

ホスト要素はパラメータhostで説明したように解釈されます。具体的には、ホスト部が空または絶対パス名のように見える場合、Unixドメインソケット接続が選択され、さもなければTCP/IP接続で初期化されます。しかしURIの階層部ではスラッシュが予約された文字であることに注意してください。このため、標準以外の

[2] https://tools.ietf.org/html/rfc3986#section-2.1

Unixドメインソケットディレクトリを指定するためには、URIからホスト指定を省き、パラメータとしてホストを指定するか、URIのホスト要素内のパスをパーセントエスケープするかどちらかを行ってください。

```
postgresql:///dbname?host=/var/lib/postgresql
postgresql://%2Fvar%2Flib%2Fpostgresql/dbname
```

単一のURIの中に、オプションのポート要素を伴う複数のホスト要素を指定することができます。postgresql://host1:port1,host2:port2,host3:port3/という形式のURIは、host=host1,host2,host3 port=port1,port2,port3という形式の接続文字列と同じです。接続の確立に成功するまで、各々のホストが順番に試されます。

33.1.1.3. 複数ホストの指定

接続先に複数のホストを指定することができ、指定された順に試されます。キーワード/値形式では、host、hostaddr、portオプションは、カンマで区切った値のリストを受け付けます。指定された各々のオプションでは、同じ数の要素を与えなければなりません。たとえば、最初のhostaddrは最初のホスト名に関連付けられ、二番目のhostaddrは二番目のホスト名に関連付けられる、という具合です。例外として、一つのportだけが指定された場合には、すべてのホストにそれが適用されます。

接続URI形式では、host要素中にカンマで区切って複数のhost:portペアを指定できます。

いずれの形式でも、単一ホスト名は複数のネットワークアドレスに変換されることがあります。これの一般的な例はIPv4とIPv6のアドレスを両方持つホストです。

複数のホスト名が指定された場合、あるいは単一のホスト名が複数のアドレスに変換された場合、そのうちの一つが成功するまで、すべてのホストとアドレスがその順に試されます。どのホストも到達可能でなければ、接続は失敗します。接続の確立に成功しても、認証に失敗すると、リスト中の残りのホストは試されません。

パスワードファイルが使用される場合は、異なるホストに対して異なるパスワードを使用できます。他の接続オプションは、リスト中のすべてのホストで同じです。たとえば、異なるユーザ名を異なるホストに指定することはできません。

33.1.2. パラメータキーワード

現時点で有効なパラメータのキーワードは以下に示す通りです。

host

接続するホスト名を指定します。ホスト名が絶対パス名のように見えるなら、それはTCP/IPによる通信ではなく、Unixドメインの通信を示します。その場合、この値はソケットファイルを格納するディレクトリの名前になります。(Unixでは絶対パス名はスラッシュから始まります。Windowsではドライブ文字から始まるパスも認められます。) hostが指定されなかったり、空の場合のデフォルトの振る舞いは、/tmp(または、PostgreSQLの構築時に指定したソケットディレクトリ)にあるUnixドメインのソケットに接続することです。Unixドメインソケットを持たないマシンにおけるデフォルトは、localhostに接続することです。

カンマで区切ったホスト名も受け付けます。この場合、リスト中のホスト名が順に試されます。リスト中の空の項目には、上で説明したデフォルトの挙動が適用されます。詳細は33.1.1.3をご覧ください。

hostaddr

接続するホストのIPアドレスを指定します。これは、172.28.40.9といった標準的なIPv4アドレス書式でなければなりません。使用するマシンでIPv6をサポートする場合は、そのアドレスを使用することもできます。このパラメータに空以外の文字列が指定されると、TCP/IP通信が常に使用されます。パラメータが指定されない場合、対応するIPアドレスを探すためにhostの値が調べられます。あるいは、hostでIPアドレスを指定している場合、その値が直接使われます。

hostaddrを使用することで、アプリケーションがホスト名の検索を行なわずに済みます。特に時間的制約があるアプリケーションでは重要になるでしょう。しかし、GSSAPI、SSPI認証方式では、ホスト名が必要になります。verify-fullSSL証明書検証を行う場合も同様です。以下の規則が使用されます。

- hostaddrを使わずにhostを指定した場合は、ホスト名の検索が発生します。(PQconnectPollを使う場合、PQconnectPollが最初にホスト名を考慮するときに、PQconnectPollをかなり長い時間、ブロックさせてしまうかもしれません。)

- hostを使わずにhostaddrを指定した場合、hostaddrの値はサーバのネットワークアドレスとなります。認証方式がホスト名を必要する場合は接続試行が失敗します。

- hostとhostaddrの両方を指定した場合、hostaddrがサーバのネットワークアドレスとなります。hostの値は認証方式で必要とされない限り無視され、必要とされる場合にはホスト名として使用されます。

hostがhostaddrネットワークアドレスに対応するマシンの名前と一致しない場合は、認証に失敗する可能性があるので注意してください。また、hostとhostaddrの両方が指定されると、hostがパスワードファイル(33.15を参照)での接続の識別に使用されます。

カンマ区切りのhostaddr値のリストも受け付けます。この場合、リスト中のホストが順に試されます。リスト中の空の項目には、対応するホスト名が使用されます。そのホスト名も空の場合は、デフォルトのホスト名が使用されます。詳細は33.1.1.3をご覧ください。

ホスト名もホストのアドレスも用いない場合、libpqはローカルのUnixドメインソケットを使用して接続します。ただし、WindowsとUnixドメインソケットを持たないマシンでは、localhostへの接続を試みます。

port

サーバホストでの接続用のポート番号、または、Unixドメイン接続の場合は、ソケットファイルの拡張子を指定します。もし複数のホストがhostあるいはhostaddrパラメータで与えられると、このパラメータで同じ長さのポートのリストを与えることができます。あるいは、一つのポート番号をすべてのホストに指定することもできます。空文字、あるいはカンマ区切りリスト中の空の項目は、PostgreSQLが構築されたときに設定されたデフォルトポート番号を指定します。

dbname

データベース名を指定します。デフォルトはユーザ名と同じです。特定の文脈では、この値は拡張書式で検査されます。詳細については33.1.1を参照してください。

user

データベースへ接続するPostgreSQLユーザ名を指定します。デフォルトは、そのアプリケーションを実行しているユーザのオペレーティングシステム上の名前と同じです。

password

サーバがパスワードによる認証を必要とした場合に使用されるパスワードを指定します。

passfile

パスワードを格納するファイル名を指定します。(33.15参照。) デフォルトは~/.pgpassまたは、Microsoft Windowsでは%APPDATA%\postgresql\pgpass.confです。(このファイルが存在しなくてもエラーは報告されません。)

channel_binding

このオプションはクライアントのチャンネルバインディングの使用を制御します。require設定では接続はチャンネルバイデンディングを使わなければならず、preferではクラアイントが可能であればチャンネルバインディングを使い、disableではチャンネルバイディングを使用させません。PostgreSQLがSSLサポートを伴ってコンパイルされている場合のデフォルトはpreferで、そうでなければデフォルトはdisableです。

チャンネルバインディングはサーバが自身が信頼できることをクライアントに証明する方法です。これは、SCRAM認証方式を使ったPostgreSQL 11以降のサーバとのSSL接続上でのみサポートされます。

connect_timeout

接続中の最大待機時間を秒単位(10進整数で記述してください、10など)で指定します。ゼロ、負値、もしくは未設定は、無期限の待機を意味します。許容される最小のタイムアウトは2秒です。したがって、1は2と解釈されます。このタイムアウトは各ホスト名やIPアドレスに別々に適用されます。例えば、二つのホストを指定して、connect_timeoutが5であるなら、各ホストが5秒以内に接続できないときにタイムアウトして、接続を待つ合計所要時間は10秒近くになるかもしれません。

client_encoding

接続用のclient_encoding設定パラメータを設定します。対応するサーバオプションで受け付けられる値の他に、クライアントにおける現在のロケール(Unixシステムの場合はLC_CTYPE環境変数)から正しい符号化方式を決定するautoを使用することができます。

options

接続開始時にサーバに送信するコマンドラインオプションを指定します。例えば、これを-c geqo=offに設定すると、geqoパラメータのセッション値はoffになります。この文字列中の空白はバックスラッシュ(\)でエスケープされていなければコマンド行引数の区切りであるとみなされます。リテラルのバックスラッシュを表すには\\と書いて下さい。利用可能なオプションに関する詳細については第19章を参照してください。

application_name

application_name設定パラメータの値を指定します。

fallback_application_name

application_name設定パラメータの予備値を指定します。接続パラメータまたはPGAPPNAME環境変数によりapplication_nameの値が指定されない場合に、この値が使用されます。予備の名前を指定することは、デフォルトのアプリケーション名を設定したいが、ユーザにもそれを上書きできるようにしておきたい、一般的なユーティリティプログラムで有用です。

keepalives

クライアント側におけるTCPキープアライブの使用を制御します。デフォルト値は1であり、有効であることを意味します。しかしキープアライブを望まない場合は、無効であることを意味するゼロに設定することができます。このパラメータはUnixドメインソケット経由の接続では無視されます。

keepalives_idle

TCPがサーバにキープアライブメッセージを送信した後に活動を行わない期間を秒単位で制御します。ゼロという値ではシステムのデフォルトを使用します。Unixドメインソケット経由でなされた接続の場合もしくはキープアライブが無効な場合、このパラメータは無視されます。これはTCP_KEEPIDLEまたは同等のソケットオプションが利用できるシステムおよびWindowsでのみサポートされます。他のシステムでは効果がありません。

keepalives_interval

TCPキープアライブメッセージに対する応答がサーバからない場合に、何秒後に再送を行うかを制御します。ゼロという値ではシステムのデフォルトを使用します。Unixドメインソケット経由でなされた接続の場合、またはキープアライブを無効にしている場合、このパラメータは無視されます。これはTCP_KEEPINTVLまたは同等のソケットオプションが利用できるシステムおよびWindowsでのみサポートされます。他のシステムでは効果がありません。

keepalives_count

サーバへのクライアント接続が不要になったとみなすまで、何回キープアライブの欠落を認めるかを制御します。ゼロという値ではシステムのデフォルトを使用します。Unixドメインソケット経由でなされた接続の場合、またはキープアライブを無効にしている場合、このパラメータは無視されます。これはTCP_KEEPCNTまたは同等のソケットオプションが利用できるシステムでのみサポートされます。他のシステムでは効果がありません。

tcp_user_timeout

接続が強制的に閉じられるまで、送信されたデータに対して応答がない状況をどれだけ認めるかをミリ秒単位で制御します。値0はシステムのデフォルトを使用します。Unixドメインソケット経由でなされた接続の場合、このパラメータは無視されます。TCP_USER_TIMEOUTが利用可能なシステムでのみサポートされます。他のシステムでは効果がありません。

tty

無視されます(以前は、これはサーバデバッグ出力を送信する場所を指定するものでした)。

replication

このオプションは接続が通常プロトコルの代わりにレプリケーションプロトコルを使うかどうかを決めます。これはPostgreSQLのレプリケーション接続やpg_basebackupなどのツールが内部的に使うものですが、サードパーティアプリケーションからも使われることがあります。レプリケーションプロトコルについての説明は52.4を参照してください。

以下の値がサポートされます。これらは大文字小文字を区別しません。

true, on, yes, 1

接続は物理レプリケーションモードになります。

database

接続は論理レプリケーションモードになり、dbnameパラメータで指定されたデータベースに接続します。

false, off, no, 0

接続は通常のものになります。これがデフォルトの振る舞いです。

物理あるいは論理レプリケーションモードでは、簡易問い合わせプロトコルのみが使用できます。

gssencmode

このオプションは、GSSによる安全なTCP/IP接続をサーバと調停するか、するのならどの優先度で調停するかを決定します。3つのモードがあります。

disable

非GSSAPI暗号化接続のみ試行

prefer (デフォルト)

GSSAPI認証情報が(すなわち認証情報キャッシュに)存在すれば、まずGSSAPI暗号化接続を試行します。その試行に失敗した場合、もしくは認証情報がない場合には非GSSAPI暗号化接続を試行します。これがPostgreSQLをGSSAPIサポートを有効にしてコンパイルした場合のデフォルトです。

require

GSSAPI暗号化接続のみ試行

gssencmodeはUnixドメインソケット通信では無視されます。PostgreSQLがGSSAPIなしでコンパイルされた場合、requireオプションを使うとエラーになります。一方、preferは受け付けられますが、libpqは実際にはGSSAPI暗号化接続を試行しません。

sslmode

このオプションは、どのSSLによる安全なTCP/IP接続の優先度でサーバと調停するかを決定します。6つのモードがあります。

disable

非SSL接続のみ試行

allow

最初に非SSL接続を試行し、失敗したら、SSL接続を試行

prefer（デフォルト）

最初にSSL接続を試行し、失敗したら、非SSL接続を試行

require

SSL接続のみ試行。ルートCAファイルが存在する場合、verify-caが指定された場合と同じ方法で証明書が検証されます。

verify-ca

SSL接続のみ試行し、サーバ証明書が信用された認証局（CA）から発行されたかを検証

verify-full

SSL接続のみ試行し、サーバ証明書が信用されたCAから発行されたか、およびそのサーバホスト名が証明書内のものと一致するかを検証

これらのオプションがどのように動くのかについては33.18を参照してください。

sslmodeはUnixドメインソケット通信では無視されます。SSLサポートなしでPostgreSQLがコンパイルされた場合に、require、verify-ca、verify-fullを使用するとエラーになります。一方、allowとpreferは使用できますが、実際にlibpqはSSL接続を受け付けません。

requiressl

このオプションはsslmode設定を支持する観点から廃止予定になっています。

1に設定することで、サーバへのSSL接続が必要になります (これはsslmodeのrequireと同じです)。サーバがSSL接続を受け付けない場合、libpqは接続を拒絶します。0(デフォルト)に設定することで、サーバと接続形式の調停を行います。(sslmodeのpreferと同じです。) SSLサポート付きでPostgreSQLをコンパイルした場合にのみ、このオプションが利用できます。

sslcompression

1に設定することで、SSL接続越えで送信されるデータは圧縮されます。0に設定すると、圧縮が無効になります。デフォルトは0です。このパラメータはSSLを使わない接続では無視されます。

SSL圧縮は今日では安全ではないと考えられていて、もはや使用は推奨されません。OpenSSL 1.1.0はデフォルトでは圧縮を無効にしており、多くのOSディストリビューションでもこれまでのバージョンで無効化しています。そのため、サーバが圧縮を受け付けない場合、本パラメータをonに設定しても効果がありません。一方で、1.0.0以前のOpenSSLは圧縮の無効化をサポートしていませんので、これらのバージョンでは本パラメータは無視されて、圧縮が使用されるかはサーバ次第です。

セキュリティが主要な関心でないなら、ネットワークがボトルネックであるとき圧縮でスループットを改善できます。CPU性能が律速要素であるなら、圧縮を無効化することで応答時間とスループットを改善できます。

sslcert

このパラメータは、~/.postgresql/postgresql.crtというデフォルトを置き換えるクライアントSSL証明書のファイル名を指定します。このパラメータはSSL接続が確立していない場合は無視されます。

sslkey

このパラメータはクライアント証明書に対して使用される秘密鍵の場所を指定します。デフォルトの~/.postgresql/postgresql.keyの代わりに使用されるファイル名、または外部「エンジン」（エンジンとはOpenSSLロード可能なモジュール）から得られるキーを指定することも可能です。外部エンジンの指定にはコロンで区切ったエンジン名とエンジン特有の鍵識別子を含んでいなければなりません。SSL接続が確立していない場合このパラメータは無視されます。

sslpassword

このパラメータはsslkeyで指定される秘密鍵に対するパスワードを指定して、対話的なパスフレーズ入力が現実的でないときにも、クライアント証明書のプライベートキーを暗号化された形式でディスクに格納できるようにします。

空でない値でこのパラメータを指定することで、暗号化されたクライアント証明書キーがlibpqに供給されるときにOpenSSLがデフォルトで出すEnter PEM pass phrase:プロンプトを抑止します。

キーが暗号化されていない場合、このパラメータは無視されます。OpenSSLエンジンがプロンプトにOpenSSLパスワードコールバックの仕組みを使わない限り、このパラメータはエンジンで指定されたキーに影響しません。

このオプションと同等の環境変数、および、パスワードを.pgpassから探す機能はありません。このオプションはサービスファイルの接続定義で使用できます。より高度な使用法を用いるユーザは、OpenSSLエンジンとPKCS#11やUSB暗号オフロードデバイスといったツールの利用を検討すべきです。

sslrootcert

このパラメータはSSL認証局(CA)の証明書のファイル名を指定します。このファイルが存在する場合、サーバ証明書はこれらの認証局の1つで署名されているかどうか検証されます。デフォルトは~/.postgresql/root.crtです。

sslcrl

このパラメータはSSL証明書失効リスト(CRL)のファイル名を指定します。このファイルに列挙された証明書が存在した場合、それはサーバ証明書を承認しようとする時に拒絶されます。デフォルトは~/.postgresql/root.crlです。

requirepeer

このパラメータは、例えばrequirepeer=postgresのようにサーバのオペレーティングシステムのユーザ名を指定します。Unixドメインソケット接続を確立する時に、このパラメータが設定された場合、クライアントは接続開始時にサーバプロセスが指定されたユーザ名で稼動しているか検査し、稼動していない場合は接続をエラーとして中断します。このパラメータは、TCP/IP接続においてSSL証明書で実現するようなサーバ認証を実現するために使用することができます。(Unixドメインソケットが/tmpなどの誰にでも書き込むことができる場所にある場合、誰でもそこで接続を監視するサーバを起動できることに注意してください。信頼できるユーザが起動したサーバに接続することを確実に行うために、このパラメータを使用してください。) このオプションはpeer認証方式が実装されたプラットフォームでのみでサポートされます。20.9を参照してください。

ssl_min_protocol_version

このパラメータは接続で許容されるSSL/TLSプロトコルの最小バージョンを指定します。有効な値はTLSv1、TLSv1.1、TLSv1.2、および、TLSv1.3です。対応しているプロトコルは使われているOpenSSLバージョンに依存し、より古いバージョンでは最新プトロコルバージョンに対応していません。指定しない場合、デフォルトはTLSv1.2で、これは執筆時点では業界標準を満たします。

ssl_max_protocol_version

このパラメータは接続で許容されるSSL/TLSプロトコルの最大バージョンを指定します。有効な値はTLSv1、TLSv1.1、TLSv1.2、および、TLSv1.3です。対応しているプロトコルは使われているOpenSSL

バージョンに依存し、より古いバージョンでは最新のプロトコルバージョンに対応していません。設定しない場合、このパラメータは無視されて、接続ではバックエンドで定義されている最大範囲が、もし定義されているなら、使われます。テストや、一部コンポーネントがより新しいプロトコルでの動作に問題がある場合に対して、大概はプロトコルの最大バージョンを設定することが有用です。

krbsrvname

GSSAPIの認証時に使われるKerberosサービス名です。成功するためには、これはサーバのKerberos認証設定のサービス名と一致していなければなりません。(20.6も参照してください。)

gsslib

GSSAPI認証で使用されるGSSライブラリです。これは今のところ、GSSAPIとSSPIの両方のサポートを含むWindowsビルド版を除いて無視されます。その場合、認証にデフォルトのSSPIではなく、GSSAPIライブラリを使うようlibpqに強制するには、これをgssapiに設定してください。

service

追加のパラメータ用に使用されるサービス名です。pg_service.conf内の追加的な接続パラメータを保持するサービス名を指定します。これによりアプリケーションはサービス名だけを指定でき、接続パラメータを集中的に保守できるようになります。33.16を参照してください。

target_session_attrs

このパラメータがread-writeなら、読み書きトランザクションがデフォルトで許容される接続だけが受付可能になります。接続に成功すると、問合せSHOW transaction_read_onlyが送られ、onが返ると接続は閉じられます。接続文字列で複数のホストが指定されている場合は、あたかも接続の試みが失敗したかのように、残りのサーバが試されます。このパラメータのデフォルト値はanyで、すべての接続が受付可能であると見なされます。

33.2. 接続状態関数

これらの関数を使用して、既存のデータベース接続オブジェクトの状態を調べることができます。

> **ヒント**
>
> libpqアプリケーションのプログラマは注意してPGconnという抽象化を維持してください。PGconnの内容は以下に挙げるアクセス用関数を使って取り出してください。PGconn構造体中のフィールドは将来予告なく変更されることがありますので、libpq-int.hを使用したフィールドの参照は避けてください。

以下の関数は、接続で確立したパラメータの値を返します。これらの値は接続期間中固定されます。複数ホストの接続文字列が使用されている場合、同じPGconnオブジェクトを使用して新しい接続が確立されると、PQhost、PQport、PQpassの値は変わる可能性があります。他の変数はPGconnの存在期間中固定されます。

PQdb

接続したデータベース名を返します。

```
char *PQdb(const PGconn *conn);
```

PQuser

接続したユーザ名を返します。

```
char *PQuser(const PGconn *conn);
```

PQpass

接続したパスワードを返します。

```
char *PQpass(const PGconn *conn);
```

PQpassは、接続パラメータで指定されたパスワードを返します。もし接続パラメータにパスワードがなくて、パスワードファイルからパスワードを取得できる場合には、そのパスワードを返します。この場合、接続パラメータに複数のホストが指定されていると、接続が確立するまでは、PQpassの結果を当てにすることはできません。接続の状態は、関数PQstatusで確認できます。

PQhost

実際に接続したサーバホスト名を返します。これはホスト名、IPアドレス、あるいはUnixソケット経由で接続している場合はディレクトリパスになります。（パスの場合は必ず/で始まる絶対パスになるので、他と区別できます。）

```
char *PQhost(const PGconn *conn);
```

hostとhostaddrの両方が指定されると、PQhostは、そのhost情報を返します。hostaddrだけが指定されると、それが返されます。接続パラメータ中に複数のホストが指定された場合には、PQhostは実際に接続しているホストの情報を返します。

conn引数がNULLならば、PQhostはNULLを返します。そうでない場合、もしホスト情報の生成中エラーとなったら（おそらくコネクションがまだ完全には確立されていないか、なんらかのエラーがある場合です）、空文字が返ります。

接続パラメータ中に複数のホストが指定されると、接続が確立するまではPQhostの結果を当てにすることはできません。接続の状態は、PQstatus関数で確認できます。

PQhostaddr

実際に接続したサーバIPアドレスを返します。これはホスト名を解決したアドレス、あるいはhostaddrパラメータ経由で与えられたIPアドレスになります。

```
char *PQhostaddr(const PGconn *conn);
```

conn引数がNULLならば、PQhostaddrはNULLを返します。そうでない場合、もしホスト情報の生成がエラーとなったら（おそらくコネクションがまだ完全には確立されていないか、なんらかのエラーがある場合です）、空文字が返ります。

PQport

実際に接続したポートを返します。

```
char *PQport(const PGconn *conn);
```

接続パラメータ中に複数のポートが指定された場合には、PQportは実際に接続しているポートを返します。

conn引数がNULLならば、PQportはNULLを返します。そうでない場合、もしホスト情報の生成がエラーとなったら(おそらくコネクションがまだ完全には確立されていないか、なんらかのエラーがある場合です)、空文字が返ります。

接続パラメータ中に複数のポートが指定されると、接続が確立するまではPQportの結果を当てにすることはできません。接続の状態は、PQstatus関数で確認できます。

PQtty

接続のデバッグ用TTYを返します。(これは廃れたものです。サーバはもはやTTY設定を参照しません。後方互換性のためにこの関数が残っています。)

```
char *PQtty(const PGconn *conn);
```

PQoptions

接続要求時に渡されたコマンドラインオプションを返します。

```
char *PQoptions(const PGconn *conn);
```

以下の関数は、PGconnオブジェクトに対して操作を行うことで変更可能な状態データを返します。

PQstatus

接続の状態を返します。

```
ConnStatusType PQstatus(const PGconn *conn);
```

この状態は多くの値の中の1つとなるはずです。しかし非同期接続手順の外部からは、その中でたった2つ、CONNECTION_OKとCONNECTION_BADだけが現れます。データベースへの接続に問題がなければ、CONNECTION_OK状態になります。接続に失敗している場合はCONNECTION_BAD状態となります。通常、OK状態はPQfinishまで維持されますが、通信失敗のために早まってCONNECTION_BADになることもあります。その場合、アプリケーションはPQresetを呼び出して修復を試みることができます。

返される可能性があるその他の状態コードについてはPQconnectStartParams、PQconnectStartおよびPQconnectPollの項目を参照してください。

PQtransactionStatus

サーバの現在のトランザクション内部状態を返します。

```
PGTransactionStatusType PQtransactionStatus(const PGconn *conn);
```

この状態は、PQTRANS_IDLE (現在待機中)、PQTRANS_ACTIVE (コマンド実行中)、PQTRANS_INTRANS (有効なトランザクションブロック内で待機中)、PQTRANS_INERROR (無効なトランザクションブロック内で待機中)となり得ます。接続に問題がある場合のみPQTRANS_UNKNOWNが報告されます。サーバへ問い合わせが送信されたが、まだ完了していない場合のみPQTRANS_ACTIVEが報告されます。

PQparameterStatus

サーバの現在のパラメータ設定を検索します。

```
const char *PQparameterStatus(const PGconn *conn, const char *paramName);
```

あるパラメータ値は、接続開始時に、もしくは、その値が変更された時は常にサーバによって自動的に報告されます。PQparameterStatusはそれらの設定の調査に役立ちます。パラメータの現在値がわかればその値を、わからない場合はNULLを返します。

現在のリリースで報告されるパラメータには、server_version、server_encoding、client_encoding、application_name、is_superuser、session_authorization、DateStyle、IntervalStyle、TimeZone、integer_datetimesおよびstandard_conforming_stringsがあります。(8.0より前ではserver_encoding、TimeZoneおよびinteger_datetimesが、8.1より前ではstandard_conforming_stringsが、そして8.4より前ではIntervalStyleが、9.0より前ではapplication_nameが報告されませんでした。) server_version、server_encodingおよびinteger_datetimesは起動後変更できないことに注意してください。

プロトコル3.0より前のサーバはパラメータ設定を報告しません。しかし、libpqにはserver_versionとclient_encodingの値を取り出す仕組みがとりあえずあります。アプリケーションは、付け焼き刃なコードでこれらの値を決定するのではなく、PQparameterStatusを使用することが求められています。(しかし、3.0より前の接続では、接続開始後にSETによるclient_encodingの変更はPQparameterStatusに反映されないことに注意してください。) server_versionについては、この情報をより比較し易い数値形式で返すPQserverVersionも参照してください。

standard_conforming_stringsの値がないと報告された場合、アプリケーションはoffと推測することができます。つまり、バックスラッシュは文字リテラル中のエスケープ文字として扱います。また、このパラメータが存在すると、エスケープ文字構文(E'...')が受付けられることを意味するものと取られます。

返されるポインタはconstと宣言されていますが、実際にはPGconn構造体に関連付けされた変化する領域を指し示します。このポインタが諸問い合わせに渡って有効なままであるとみなすのは賢明ではありません。

PQprotocolVersion

使用されるフロントエンド/バックエンドプロトコルを調査します。

```
int PQprotocolVersion(const PGconn *conn);
```

ある機能がサポートされているかどうかを決定するために、アプリケーションはこの関数を使用することができます。現在、取り得る値は2(2.0プロトコル)、3(3.0プロトコル)、あるいは0(接続不良)です。このプロトコルバージョンは接続の開始が完了した後で変更することはできません。しかし、理論的には接続のリセット時に変更可能です。PostgreSQL 7.4以降での通信時、通常3.0プロトコルが使用されます。7.4より前のサーバでは2.0プロトコルのみをサポートします。(1.0プロトコルは廃止され、libpqではサポートされていません。)

PQserverVersion

サーバのバージョンの整数表現を返します。

```
int PQserverVersion(const PGconn *conn);
```

この関数を使用してアプリケーションは接続したデータベースサーバのバージョンを決定することができます。返却値の形式は、メジャーバージョン番号に10000を掛け、マイナーバージョン番号を加えたものです。例えば、バージョン10.1では100001を返し、バージョン11.0では110000を返します。接続不良の場合は0が返されます。

バージョン10よりも前では、PostgreSQLでは、最初の2つの部分がメジャーバージョンを表す、3つの部分からなるバージョン番号が使われていました。これらのバージョンでは、PQserverVersionはそれぞれの部分に2桁の数字を使います。たとえば、バージョン9.1.5では90105が返され、バージョン9.2.0では90200が返されます。

ですから、機能の互換性を見極めるのが目的なら、アプリケーションはPQserverVersionの結果を10000ではなく、100で割り、論理的なメジャーバージョンを求めるべきです。すべてのリリースで、最後の2桁だけがマイナーリリースで異なります。(バグ修正リリースです。)

PQerrorMessage

接続における操作において、最も最近に生成されたエラーメッセージを返します。

```
char *PQerrorMessage(const PGconn *conn);
```

ほとんどすべてのlibpq関数は、失敗時にPQerrorMessage用のメッセージを設定します。libpqでの決まりとして、空でないPQerrorMessageの結果は複数行に渡ることも可能で、最後に改行が含まれることがある点に注意してください。呼び出し元はこの結果を直接解放してはいけません。関連するPGconnハンドルがPQfinishに渡された時にこれは解放されます。PGconn構造体への操作を跨って、この結果文字列が同一であると想定してはいけません。

PQsocket

サーバとの接続ソケットに対するファイル記述子番号を得ます。有効な記述子なら値は0以上です。-1の場合は、サーバとの接続がまだ開いていないことを示します。(これは通常の操作では変更することはできません。接続設定中やリセット中に変更されます。)

```
int PQsocket(const PGconn *conn);
```

PQbackendPID

接続を処理するバックエンドのプロセスID(PID)を返します。

```
int PQbackendPID(const PGconn *conn);
```

バックエンドのPIDは、デバッグする場合やNOTIFYメッセージ(これは通知を発行したバックエンドプロセスのPIDを含んでいます)の比較に便利です。このPIDはデータベースサーバホスト上で実行されているプロセスのものであり、ローカルホスト側のものではありません! 注意してください。

PQconnectionNeedsPassword

接続認証方式がパスワードを要求し、利用可能なパスワードがない場合真(1)を返します。さもなくば偽(0)を返します。

```
int PQconnectionNeedsPassword(const PGconn *conn);
```

この関数を、接続試行に失敗した後でユーザにパスワード入力を促すかどうかを決定するために適用することができます。

PQconnectionUsedPassword

接続認証方式でパスワードを使用する場合は真(1)、さもなくば偽(0)を返します。

```
int PQconnectionUsedPassword(const PGconn *conn);
```

この関数は、接続の試みが失敗したか成功したかの後に、サーバがパスワードを要求したかどうかを検出するために適用できます。

以下の関数はSSLに関連した情報を返します。この情報は通常、接続の確立後には変更されません。

PQsslInUse

接続がSSLを使っていれば真(1)、使っていなければ偽(0)を返します。

```
int PQsslInUse(const PGconn *conn);
```

PQsslAttribute

接続におけるSSL関連の情報を返します。

```
const char *PQsslAttribute(const PGconn *conn, const char *attribute_name);
```

利用可能な属性のリストは使用されているSSLライブラリおよび接続の種類に依存して変わります。属性が利用可能でないときはNULLが返されます。

一般的には、以下の属性が利用可能です。

library

使用されているSSLの実装の名前です。(現在は"OpenSSL"だけが実装されています。)

protocol

使用されているSSL/TLSのバージョンです。一般的な値は、"TLSv1"、"TLSv1.1"、"TLSv1.2"ですが、他のプロトコルが使用されれば、異なる文字列が返されるかもしれません。

key_bits

暗号アルゴリズムで使用されている鍵のビット数です。

cipher

使用されている暗号スイートの短縮名、例えば"DHE-RSA-DES-CBC3-SHA"です。この名前は各SSLの実装に固有のものです。

compression

SSL圧縮が使用されている場合、圧縮アルゴリズムの名前を返します。圧縮は使われているがアルゴリズムが不明という場合を"on"を返します。圧縮が使われていない場合は"off"を返します。

PQsslAttributeNames

利用可能なSSL属性名の配列を返します。配列の最後のメンバにはNULLポインタが入ります。

```
const char * const * PQsslAttributeNames(const PGconn *conn);
```

PQsslStruct

接続を説明するSSLの実装に固有のオブジェクトへのポインタを返します。

```
void *PQsslStruct(const PGconn *conn, const char *struct_name);
```

利用可能な構造体は、使用されるSSLの実装に依存します。OpenSSLでは、"OpenSSL"の名前の下に利用可能な構造体が1つあり、OpenSSLのSSL構造体へのポインタを返します。この関数を使用するには、以下のようなプログラムが利用できます。

```
#include <libpq-fe.h>
#include <openssl/ssl.h>

...

    SSL *ssl;

    dbconn = PQconnectdb(...);
    ...

    ssl = PQsslStruct(dbconn, "OpenSSL");
    if (ssl)
    {

        /* sslにアクセスするためOpenSSLの関数を使う */
    }
```

この構造体は、暗号化レベルの確認、サーバ証明書の検証、その他に使用できます。この構造体に関する情報についてはOpenSSLのドキュメントを参照して下さい。

PQgetssl

接続で使用されているSSLの構造体を返します。SSLが使われていなければNULLを返します。

```
void *PQgetssl(const PGconn *conn);
```

この関数はPQsslStruct(conn, "OpenSSL")と同等です。返される構造体はOpenSSLに固有のもので他のSSL実装が利用されていると使用できないので、新しく作成するアプリケーションでは使うべきではありません。接続がSSLを使用しているかどうかを調べるには、代わりにPQsslInUseを呼び出して下さい。また、接続に関するより詳細についてはPQsslAttributeを使って下さい。

33.3. コマンド実行関数

いったんデータベースサーバへの接続の確立が成功すれば、本節で説明する関数を使ってSQLの問い合わせやコマンドを実行します。

33.3.1. 主要な関数

PQexec

コマンドをサーバに送信し、結果を待機します。

```
PGresult *PQexec(PGconn *conn, const char *command);
```

戻り値はPGresultへのポインタ、場合によってはヌルポインタです。メモリ不足の状態、あるいはサーバへのコマンド送信が不可能といった深刻なエラーの場合を除けば、通常非ヌルのポインタが返ります。PQresultStatus関数を呼び出して、何かエラー(ヌルポインタ値を含むエラー。この場合はPGRES_FATAL_ERRORが返されます)がないか戻り値を検査しなければなりません。こうしたエラーの詳しい情報はPQerrorMessageで得ることができます。

コマンド文字列には(セミコロンで区切られた)複数のSQLコマンドを含めることができます。単一のPQexec呼び出しで送信された複数の問い合わせは、単一トランザクションで処理されます。ただし、問い合わせ文字列内に明示的なBEGIN/COMMITコマンドがある場合は、複数のトランザクションに分離されます。(サーバがどのように複数問い合わせを処理するかの更なる詳細は52.2.2.1を参照してください。) しかし、返されるPGresult構造体には、その文字列内で最後に実行されたコマンドの結果のみが含まれることに注意してください。そのコマンドの1つが失敗したとすると、文字列の処理はそこで中断し、エラー条件が含まれるPGresultが返されます。

PQexecParams

サーバにコマンドを送信し、結果を待ちます。ただし、SQLコマンドテキストとは別にパラメータを渡すことができます。

```
PGresult *PQexecParams(PGconn *conn,
                       const char *command,
                       int nParams,
                       const Oid *paramTypes,
                       const char * const *paramValues,
                       const int *paramLengths,
                       const int *paramFormats,
```

```
                        int resultFormat);
```

PQexecParamsとPQexecは似ていますが、前者は次の機能が追加されています。パラメータ値をコマンド文字列とは別に適切に指定することができ、また、問い合わせの結果をテキスト書式としてでもバイナリ書式としてでも要求できます。PQexecParamsはプロトコル3.0以降でのみサポートされ、プロトコル2.0で使用した場合は失敗します。

この関数の引数を以下に示します。

conn

接続オブジェクトです。これを通してコマンドを送信します。

command

実行させるSQLコマンド文字列です。パラメータが使用される場合は、コマンド文字列内で$1、$2などのように参照されます。

nParams

提供されるパラメータ数です。これは配列paramTypes[]、paramValues[]、paramLengths[]、paramFormats[]の要素数です。(この配列ポインタは、nParamsが0の場合、NULLとすることができます。)

paramTypes[]

パラメータシンボルに代入されるデータ型をOIDで指定したものです。paramTypesがNULL、または、ある配列要素が0の場合、サーバは、型指定のないリテラル文字列に対して行う推定方法と同じ方法を使用して、パラメータシンボルのデータ型を推定します。

paramValues[]

パラメータの実際の値を指定します。配列内のヌルポインタは対応するパラメータがNULLであることを意味します。さもなくば、このポインタはゼロ終端のテキスト文字列(テキスト書式)、または、サーバで想定している書式によるバイナリデータ(バイナリ書式)を指し示します。

paramLengths[]

バイナリ書式のパラメータの実データ長を指定します。NULLパラメータおよびテキスト書式のパラメータでは無視されます。バイナリパラメータが存在しない場合、この配列ポインタはヌルとしてもかまいません。

paramFormats[]

パラメータがテキスト(パラメータに対応する配列要素に0を設定)か、バイナリ(パラメータに対応する配列要素に1を設定)かを指定します。この配列ポインタがヌルの場合、すべてのパラメータはテキスト文字列であると仮定されます。

バイナリ書式で渡された値は、バックエンドが想定する内部表現の知識を必要とします。例えば、整数はネットワークバイト順に渡されなければなりません。numericによる値は、src/backend/utils/adt/numeric.c::numeric_send()およびsrc/backend/utils/adt/numeric.c::numeric_recv()で実装されたようにサーバストレージ書式の知識を必要とします。

resultFormat

結果をテキスト書式で取り出したい場合は0を、バイナリ書式で取り出したい場合は1を指定します。（現時点では、プロトコル内部では実現可能ですが、結果の列ごとに異なる書式を指定して取り出す機構は存在しません。）

PQexecに対するPQexecParamsの主要な利点は、コマンド文字列とパラメータ値を分離することができることです。これにより、面倒でエラーを招きやすい引用符付けやエスケープ処理を行なう必要がなくなります。

PQexecと異なり、PQexecParamsは、文字列内に最大でも1つのSQLコマンドを入れることができます。（セミコロンを入れることはできますが、空でないコマンドを2つ以上入れることはできません。）これは、プロトコル自体の制限ですが、SQLインジェクション攻撃に対する追加の防御となりますので、多少役に立ちます。

ヒント

OID経由のパラメータ型の指定は、特にプログラムの中で特定のOID値がソースに直接書き込まれることを好まない場合には退屈です。しかしながら、パラメータの型をサーバ自身で決定できない場合や、望む型と異なる型を選択する場合であっても、これを避けることができます。SQLコマンドテキストでどのデータ型を送信するかを示すためにパラメータシンボルに明示的なキャストをつけてください。以下が例です。

```
SELECT * FROM mytable WHERE x = $1::bigint;
```

デフォルトではパラメータ$1の型はxと同じデータ型に割り当てられますが、これにより強制的にbigintとして扱われます。この方法または型のOIDを数字で指定する方法で、パラメータの型を強制的に決定することがバイナリ書式においてパラメータ値を送る時に強く推奨されます。これは、バイナリ書式はテキスト書式より情報が少なく、そのために、サーバが型の不一致という問題を検出する機会が少なくなるためです。

PQprepare

指定パラメータを持つプリペアド文の作成要求を送信し、その完了を待ちます。

```
PGresult *PQprepare(PGconn *conn,
                    const char *stmtName,
                    const char *query,
                    int nParams,
                    const Oid *paramTypes);
```

PQprepareは、後でPQexecPreparedを使用して実行するプリペアド文を作成します。この機能を使用すると、コマンドを実行の度に解析して計画することなく、繰り返し実行することができます。詳しくはPREPAREを参照してください。PQprepareはプロトコル3.0以降でのみサポートされ、プロトコル2.0を使用している場合は失敗します。

この関数はquery文字列からstmtNameという名前のプリペアド文を作成します。queryは単一のSQLコマンドでなければなりません。stmtNameを""にして、無名の文を作成することができます。もし、無名の文が既に存在していた場合は自動的に置き換えられます。その他の場合、文の名前が現在のセッションで

既に存在するとエラーになります。何らかのパラメータが使用される場合、問い合わせ内では$1、$2などで参照します。nParamsはパラメータ数です。その型については事前にparamTypes[]配列で指定されています。(nParamsがゼロの場合、この配列ポインタはNULLにすることができます。) paramTypes[]は、OIDによりパラメータシンボルに割り当てるデータ型を指定します。paramTypesがNULLの場合、もしくは、配列内の特定要素がゼロの場合、サーバはそのパラメータシンボルに対して、型指定の無いリテラル文字列に対する処理と同等の方法でデータ型を割り当てます。また、問い合わせではnParamsより多くのパラメータシンボルを使用することができます。これらのシンボルに対するデータ型も同様に推測されます。(どのようなデータ型が推測されるかを検出する手法についてはPQdescribePreparedを参照してください。)

PQexec同様、結果は通常PGresultオブジェクトで、その内容でサーバ側の成功や失敗を示します。ヌルという結果はメモリ不足や全くコマンドを送信することができなかったことを示します。こうしたエラーの詳細情報を入手するにはPQerrorMessageを使用してください。

PQexecPreparedで使用するためのプリペアド文は、PREPARE SQL文を実行することでも作成可能です。また、プリペアド文を削除するlibpq関数はありませんが、この目的のためにDEALLOCATESQL文を使用することができます。

PQexecPrepared

指定パラメータによるプリペアド文の実行要求を送信し、結果を待ちます。

```
PGresult *PQexecPrepared(PGconn *conn,
                         const char *stmtName,
                         int nParams,
                         const char * const *paramValues,
                         const int *paramLengths,
                         const int *paramFormats,
                         int resultFormat);
```

PQexecPreparedとPQexecParamsは似ていますが、前者では実行されるコマンドは、問い合わせ文字列を与えるのではなく、事前にプリペアド文を指名することで指定されます。この機能により、繰り返し使用する予定のコマンドを実行する度にではなく、一度だけ解析、計画作成を行うことができます。この文は現在のセッションで事前に準備されていなければなりません。PQexecPreparedは、プロトコル3.0以降の接続でのみサポートされます。プロトコル2.0で使用した場合は失敗します。

パラメータは、問い合わせ文字列ではなく指定されたプリペアド文の名前を与える点を除き、PQexecParamsと同じです。また、paramTypes[]パラメータは存在しません。(プリペアド文のパラメータ型はその作成時点で決定されているため、これは不要です。)

PQdescribePrepared

指定したプリペアド文に関する情報入手要求を送り、入手完了まで待機します。

```
PGresult *PQdescribePrepared(PGconn *conn, const char *stmtName);
```

PQdescribePreparedにより、アプリケーションは事前にプリペアド文に関する情報を入手できます。PQdescribePreparedはプロトコル3.0以降の接続でのみサポートされます。プロトコル2.0で使用すると失敗します。

stmtNameを""またはNULLとすることで、無名の文を参照することができます。これ以外では、存在するプリペアド文の名前でなければなりません。成功すると、PGRES_COMMAND_OKというステータスのPGresultが返されます。PQnparamsおよびPQparamtype関数をこのPGresultに適用して、プリペアド文のパラメータに関する情報を得ることができます。また、PQnfields、PQfname、PQftype関数などを使用して、文の結果列(もしあれば)に関する情報を提供できます。

PQdescribePortal

指定したポータルに関する情報入手要求を送信し、完了まで待機します。

```
PGresult *PQdescribePortal(PGconn *conn, const char *portalName);
```

PQdescribePortalにより、アプリケーションは事前に作成されたポータルの情報を入手することができます。(libpqはポータルへの直接アクセスする方法を提供していませんが、この関数を使用してDECLARE CURSOR SQLコマンドで作成したカーソルの属性を確認することができます。)PQdescribePortalはプロトコル3.0以降の接続でのみサポートされます。プロトコル2.0で使用すると失敗します。

portalNameに""またはNULLを指定して、無名のポータルを参照することができます。これ以外では、既存のポータルの名前でなければなりません。成功すると、PGRES_COMMAND_OKというステータスのPGresultが返されます。PQnfields、PQfname、PQftype関数などをこのPGresultに適用して、ポータルの結果列(もしあれば)に関する情報を得ることができます。

PGresult構造体はサーバから返された結果をカプセル化します。libpqアプリケーションのプログラマは注意してPGresultという抽象化を維持してください。以下のアクセス用関数を使用して、PGresultの内容を取り出してください。将来の変更に影響されますので、PGresult構造体のフィールドを直接参照することは避けてください。

PQresultStatus

コマンドの結果状態を返します。

```
ExecStatusType PQresultStatus(const PGresult *res);
```

PQresultStatusは以下のいずれかの値を返します。

PGRES_EMPTY_QUERY

サーバに送信された文字列が空でした。

PGRES_COMMAND_OK

データを返さないコマンドが正常終了しました。

PGRES_TUPLES_OK

データを返すコマンド(SELECTやSHOWなど)が正常終了しました。

PGRES_COPY_OUT

(サーバからの)コピーアウトデータ転送が始まりました。

PGRES_COPY_IN

(サーバへの)コピーインデータ転送が始まりました。

PGRES_BAD_RESPONSE

サーバが不明な応答を返しました。

PGRES_NONFATAL_ERROR

致命的ではない(注意喚起もしくは警告)エラーが発生しました。

PGRES_FATAL_ERROR

致命的なエラーが発生しました。

PGRES_COPY_BOTH

(サーバからおよびサーバへの)コピーイン/アウトデータ転送が始まりました。現在これはストリーミングレプリケーションのみで使用されます。このためこの状態は通常のアプリケーションでは起こりません。

PGRES_SINGLE_TUPLE

PGresultには現在のコマンドからの結果タプルが1つ含まれます。この状態は問い合わせで単一行モードが選択された場合(33.5参照)のみ起こります。

結果状態がPGRES_TUPLES_OKまたはPGRES_SINGLE_TUPLEであれば、以下で説明する関数を使って問い合わせが返した行を取り出すことができます。ただし、たまたまSELECTコマンドが返す行が0個だったような場合でもPGRES_TUPLES_OKとなることに注意してください。PGRES_COMMAND_OKは、行を決して返さない(RETURNING句の無いINSERTまたはUPDATEなど)コマンド用です。PGRES_EMPTY_QUERYという応答はクライアントソフトウェアの不具合を示しているかもしれません。

PGRES_NONFATAL_ERROR状態の場合、結果はPQexecや他の問い合わせ実行関数によって直接返されません。その代わりに、この種の結果は注意喚起プロセッサ(33.12参照)に渡されます。

PQresStatus

PQresultStatusが返す列挙型から状態コードを説明する文字列定数に変換します。呼び出し元はこの結果を解放してはいけません。

```
char *PQresStatus(ExecStatusType status);
```

PQresultErrorMessage

コマンドに関するエラーメッセージを返します。エラーが何もなければ、空の文字列を返します。

```
char *PQresultErrorMessage(const PGresult *res);
```

エラーがあった場合、返される文字列の最後には改行が含まれます。呼び出し元はこの結果を直接解放してはいけません。関連するPGresultハンドルがPQclearに渡された時にこれは解放されます。

(接続に対する)PQerrorMessageも、PQexecまたはPQgetResult呼び出しの直後なら(結果に対する)PQresultErrorMessageと同じ文字列を返します。しかし、接続に対するエラーメッセージは続いて操作を行うと変化してしまうのに対し、PGresultは自身が破棄されるまでそのエラーメッセージを維持し続けます。このPQresultErrorMessageは個々のPGresultに結び付けられた状態を確認する時に、そしてPQerrorMessageは接続における最後の操作の状態を確認する時に使用してください。

PQresultVerboseErrorMessage

PGresultオブジェクトに関連したエラーメッセージの再フォーマットしたバージョンを返します。

```
char *PQresultVerboseErrorMessage(const PGresult *res,
                                  PGVerbosity verbosity,
                                  PGContextVisibility show_context);
```

状況によっては、クライアントは以前に報告されたエラーのより詳細なバージョンを取得したいと思うかもしれません。PQresultVerboseErrorMessageは、指定のPGresultが生成されたときに、指定した冗長設定がその接続で使われていたならPQresultErrorMessageが生成したであろうメッセージを計算することで、この要請に応えます。PGresultがエラーの結果ではない場合は、「PGresult is not an error result」が代わりに報告されます。返される文字列は行末に改行コードが含まれます。

PGresultからデータを抽出する他の多くの関数と異なり、この関数の結果は新しく割り当てられた文字列です。その文字列が必要なくなったときは、呼び出し側がPQfreemem()を使ってそれを解放しなければなりません。

十分なメモリがないときは、NULLが返されることもありえます。

PQresultErrorField

エラー報告の個々のフィールドを返します。

```
char *PQresultErrorField(const PGresult *res, int fieldcode);
```

fieldcodeはエラーフィールド識別子です。以下に示すシンボルを参照してください。PGresultがエラーではない、もしくは、警告付きの結果である場合や指定したフィールドを含まない場合、NULLが返されます。通常フィールド値には改行が含まれません。フィールド値は関連するPGresultハンドルがPQclearに渡された時に解放されます。

以下のフィールドコードが使用できます。

PG_DIAG_SEVERITY

深刻度。このフィールドの内容は(エラーメッセージの場合)ERROR、FATAL、もしくは、PANIC、(注意喚起メッセージの場合)WARNING、NOTICE、DEBUG、INFO、もしくは、LOGです。これらは、多言語化により翻訳されている可能性があります。常に存在します。

PG_DIAG_SEVERITY_NONLOCALIZED

深刻度。このフィールドの内容は(エラーメッセージの場合)ERROR、FATAL、もしくは、PANIC、(注意喚起メッセージの場合)WARNING、NOTICE、DEBUG、INFO、もしくは、LOGです。これは、内容が多言語化

されないことを除き、PG_DIAG_SEVERITYと同一です。これはPostgreSQLのバージョン9.6以降で生成された報告にのみ存在します。

PG_DIAG_SQLSTATE

エラーのSQLSTATEコードです。SQLSTATEコードは発生したエラーの種類を識別します。フロントエンドアプリケーションにより、特定のデータベースエラーに対して所定の操作（エラー処理など）を行うために使用できます。起こり得るSQLSTATEコードの一覧については付録Aを参照してください。このフィールドは多言語化されず、また、常に存在します。

PG_DIAG_MESSAGE_PRIMARY

可読性を高めた主要エラーメッセージです。(通常は1行です。) 常に存在します。

PG_DIAG_MESSAGE_DETAIL

詳細です。問題に関するより詳細を表す補助的なエラーメッセージです。複数行に跨る可能性があります。

PG_DIAG_MESSAGE_HINT

ヒントです。問題の対応方法についての補助的な提言です。これは、詳細(detail)とは異なり、問題の事象ではなく、(適切でない可能性がありますが)アドバイスを提供することを目的としています。複数行に跨る可能性があります。

PG_DIAG_STATEMENT_POSITION

元の問い合わせ文字列のインデックスとなる、エラーが発生したカーソル位置を示す10進整数を持つ文字列です。先頭文字がインデックス1となり、また、バイトではなく、文字数で数えた位置です。

PG_DIAG_INTERNAL_POSITION

この定義はPG_DIAG_STATEMENT_POSITIONフィールドと同じです。しかし、これは、クライアントが発行したコマンドではなく、カーソル位置が内部生成コマンドを参照する場合に使用されます。このフィールドが存在する時は常にPG_DIAG_INTERNAL_QUERYフィールドが存在します。

PG_DIAG_INTERNAL_QUERY

失敗した内部生成コマンドのテキストです。これは、例えば、PL/pgSQL関数で発行されたSQL問い合わせになります。

PG_DIAG_CONTEXT

エラーが発生した文脈を示すものです。今の所、これは活動中の手続き言語関数や内部生成問い合わせの呼び出しスタックの追跡情報が含まれます。この追跡は行単位で1項目であり、その順番は呼び出し順の反対になります。

PG_DIAG_SCHEMA_NAME

そのエラーが特定のデータベースオブジェクトに付随する場合、もしあれば、そのオブジェクトを含むスキーマ名です。

PG_DIAG_TABLE_NAME

そのエラーが特定のテーブルに付随する場合のテーブル名です。(テーブルのスキーマ名についてはスキーマ名フィールドを参照ください。)

PG_DIAG_COLUMN_NAME

そのエラーが特定のテーブル列に付随する場合の列名です。(テーブルを識別するにはスキーマとテーブル名フィールドを参照ください。)

PG_DIAG_DATATYPE_NAME

そのエラーが特定のデータ型に付随する場合のデータ型名です。(データ型のスキーマ名についてはスキーマ名フィールドを参照ください。)

PG_DIAG_CONSTRAINT_NAME

そのエラーが特定の制約に付随する場合の制約名です。付随するテーブルまたはドメインについては上記にリストされたフィールドを参照ください。(この目的のために、制約は制約構文で作成されていなくてもインデックスは制約として扱われます。)

PG_DIAG_SOURCE_FILE

エラーが報告された場所のソースコードのファイル名です。

PG_DIAG_SOURCE_LINE

エラーが報告された場所のソースコードにおける行番号です。

PG_DIAG_SOURCE_FUNCTION

エラーを報告した、ソースコードにおける関数名です。

> **注記**
>
> スキーマ名、テーブル名、列名、データ型名、および制約名に対するフィールドは限定的なエラー型に対してのみ提供されます。付録Aを参照ください。これらのフィールドのいかなる存在もその他のフィールドの存在を保証すると推測してはなりません。コアエラーの出所は上に記載の相互関係を監視しますが、ユーザ定義関数はこれらフィールドを別目的で使用しているかもしれません。同様の脈絡で、使用しているデータベースで これらのフィールドが同時に存在するオブジェクトを意味すると推測してはなりません。

表示情報の必要に応じた整形はクライアントの責任です。具体的には、必要に応じて長い行を分割しなければなりません。エラーメッセージフィールド内の改行文字は、改行としてではなく段落として分かれたものとして取扱うべきです。

libpqで内部的に生成されたエラーは、深刻度と主要メッセージを持ちますが、通常は他のフィールドを持ちません。3.0より前のプロトコルのサーバで返されるエラーは、深刻度と主要メッセージ、場合によって詳細メッセージを持ちますが、他のフィールドを持ちません。

エラーフィールドはPGresultからのみ利用でき、PGconnからは利用できません。PQerrorFieldという関数はありません。

PQclear

PGresultに割り当てられた記憶領域を解放します。個々の問い合わせ結果は、必要なくなった時にPQclearで解放するべきです。

```
void PQclear(PGresult *res);
```

PGresultオブジェクトは必要な間保持することができます。新しい問い合わせを発行する場合でも、接続を閉じてしまうまではPGresultは消えません。PGresultを解放するには、PQclearを呼び出さなくてはいけません。その操作に失敗してしまうと、アプリケーションのメモリリークを引き起こしてしまいます。

33.3.2. 問い合わせ結果の情報の取り出し

これらの関数を使用して、正常終了した問い合わせ結果を示す(つまり、その状態がPGRES_TUPLES_OKまたはPGRES_SINGLE_TUPLEとなっている)PGresultオブジェクトから情報を抽出することができます。また、成功したDescribe操作から情報を抽出することもできます。Describeの結果はすべて、実際に問い合わせを実行した時に提供されるものと同じ列情報を持ちますが、行はありません。他の状態値を持つオブジェクトでは、これらの関数は、結果が0行0列であるものと同様に動作します。

PQntuples

問い合わせ結果内の行(タプル)数を返します。(PGresultオブジェクトはINT_MAX行に制限されているため、intの結果で十分です。)

```
int PQntuples(const PGresult *res);
```

PQnfields

問い合わせ結果の各行の列(フィールド)の数を返します。

```
int PQnfields(const PGresult *res);
```

PQfname

指定した列番号に対応する列の名前を返します。列番号は0から始まります。呼び出し元はこの結果を直接解放してはいけません。関連するPGresultハンドルがPQclearに渡された時にこれは解放されます。

```
char *PQfname(const PGresult *res,
              int column_number);
```

列番号が範囲外であった場合、NULLが返ります。

PQfnumber

指定した列名に関連する列番号を返します。

```
int PQfnumber(const PGresult *res,
```

```
              const char *column_name);
```

指定した名前に一致する列がなければ、-1が返ります。

指定した名前はSQLコマンドの識別子同様に扱われます。つまり、二重引用符でくくられていない限り、小文字化されます。例えば、以下のSQLで生成された問い合わせ結果を考えます。

```
SELECT 1 AS FOO, 2 AS "BAR";
```

以下により、結果を取り出すことができます。

```
PQfname(res, 0)              foo
PQfname(res, 1)              BAR
PQfnumber(res, "FOO")        0
PQfnumber(res, "foo")        0
PQfnumber(res, "BAR")        -1
PQfnumber(res, "\"BAR\"")    1
```

PQftable

指定した列の抽出元であるテーブルのOIDを返します。列番号は0から始まります。

```
Oid PQftable(const PGresult *res,
             int column_number);
```

列番号が範囲外の場合や指定した列がテーブル列への単純な参照でない場合、3.0より前のプロトコルを使用している場合は、InvalidOidが返されます。pg_classシステムテーブルに問い合わせ、どのテーブルが参照されているのかを正確に求めることができます。

libpqヘッダファイルをインクルードすると、Oid型とInvalidOid定数が定義されます。これらは両方とも何らかの整数型です。

PQftablecol

指定した問い合わせ結果の列を作成した列の(それが属するテーブル内での)列番号を返します。問い合わせ結果の列番号は0から始まりますが、テーブル列には0以外の番号が付けられています。

```
int PQftablecol(const PGresult *res,
                int column_number);
```

列番号が範囲外の場合や指定した列がテーブル列への単純な参照でなかった場合、3.0より前のプロトコルを使用している場合は、ゼロが返されます。

PQfformat

指定した列の書式を示す書式コードを返します。列番号は0から始まります。

```
int PQfformat(const PGresult *res,
```

```
              int column_number);
```

ゼロという書式コードはテキストデータ表現を示し、1という書式コードはバイナリ表現を示します。(他のコードは将来の定義のために予約されています。)

PQftype

指定した列番号に関連したデータ型を返します。返された整数はその型の内部的なOID番号です。列番号は0から始まります。

```
Oid PQftype(const PGresult *res,
            int column_number);
```

pg_typeシステムテーブルに問い合わせて、各種データ型の名前や属性を得ることができます。組み込みデータ型のOIDは、ソースツリー内のsrc/include/catalog/pg_type_d.hファイル内で定義されています。

PQfmod

指定した列番号に関連した列の型修飾子を返します。列番号は0から始まります。

```
int PQfmod(const PGresult *res,
           int column_number);
```

修飾子の値の解釈は型に固有なものです。通常これらは精度やサイズの制限を示します。-1という値は「使用できる情報がない」ことを示します。ほとんどのデータ型は修飾子を使用しません。この場合は常に-1という値になります。

PQfsize

指定した列番号に関連した列のバイト単位のサイズを返します。列番号は0から始まります。

```
int PQfsize(const PGresult *res,
            int column_number);
```

PQfsizeはデータベース行内でその列用に割り当てられる領域を返します。言い替えると、そのデータ型についてのサーバでの内部表現のサイズです。(従って、実際にはクライアントから見るとあまり役には立ちません。) 負の値は可変長データ型を示します。

PQbinaryTuples

PGresultがバイナリデータを持つ場合は1を、テキストデータを持つ場合は0を返します。

```
int PQbinaryTuples(const PGresult *res);
```

この関数は廃れたものです。(COPYを行う接続での使用を除きます。) 単一のPGresultで、ある列はテキストデータを持ち、他の列ではバイナリデータを持つことが可能であるためです。PQfformatの利用が推奨されます。結果のすべての列がバイナリ(書式1)の場合のみPQbinaryTuplesは1を返します。

PQgetvalue

PGresultの1行における単一フィールドの値を返します。行番号と列番号は0から始まります。呼び出し元はこの結果を直接解放してはいけません。関連するPGresultハンドルがPQclearに渡された時に、これは解放されます。

```
char *PQgetvalue(const PGresult *res,
                 int row_number,
                 int column_number);
```

テキスト書式のデータでは、PQgetvalueで返される値はフィールド値のヌル終端の文字列表現となります。バイナリ書式のデータでは、この値はデータ型のtypsend関数とtypreceive関数で決まるバイナリ表現となります。(実際にはこの場合でも値の終わりにゼロというバイトが付与されます。しかし、この値の内部には大抵の場合ヌルが埋め込まれていますので、通常このバイトは有用ではありません。)

フィールド値がNULLの場合、空文字列が返されます。NULL値と空文字列という値とを区別する方法はPQgetisnullを参照してください。

PQgetvalueによって返されるポインタはPGresult構造体の一部の格納領域を指し示します。このポインタが指し示すデータを変更すべきではありません。また、PGresult構造体を解放した後も使用し続ける場合は、データを別の格納領域に明示的にコピーしなければなりません。

PQgetisnull

フィールドがNULL値かどうか検査します。行番号と列番号は0から始まります。

```
int PQgetisnull(const PGresult *res,
                int row_number,
                int column_number);
```

この関数は、フィールドがNULLの場合に1を、フィールドが非NULL値を持つ場合は0を返します。(PQgetvalueでは、NULLフィールドはヌルポインタではなく空文字列を返すことに注意してください。)

PQgetlength

実際のフィールド値の長さをバイト単位で返します。行番号と列番号は0から始まります。

```
int PQgetlength(const PGresult *res,
                int row_number,
                int column_number);
```

これは特定のデータ値についての実際のデータ長です。つまり、PQgetvalueによって指し示されるオブジェクトのサイズです。テキストデータ書式ではstrlen()と同一です。バイナリ書式ではこれは重要な情報です。実際のデータ長を取り出すためにPQfsizeを信用してはなりません。

PQnparams

プリペアド文のパラメータ数を返します。

```
int PQnparams(const PGresult *res);
```

この関数はPQdescribePreparedの結果を確認する時にのみ有用です。他の種類の問い合わせではゼロを返します。

PQparamtype

指定された文パラメータのデータ型を返します。パラメータ番号は0から始まります。

```
Oid PQparamtype(const PGresult *res, int param_number);
```

この関数は、PQdescribePreparedの結果を確認する時にのみ有用です。他の種類の問い合わせではゼロを返します。

PQprint

すべての行と列名(省略可能)を指定した出力ストリームに表示します。

```
void PQprint(FILE *fout,      /* 出力ストリーム */
             const PGresult *res,
             const PQprintOpt *po);

typedef struct
{
    pqbool  header;      /* フィールドヘッダ情報と行数の表示出力 */
    pqbool  align;       /* 位置揃えのためのフィールドへの埋め込み */
    pqbool  standard;    /* 古い、無くなりそうな書式 */
    pqbool  html3;       /* HTML表出力 */
    pqbool  expanded;    /* 拡張テーブル */
    pqbool  pager;       /* 必要に応じたページャの使用 */
    char    *fieldSep;   /* フィールド区切り文字 */
    char    *tableOpt;   /* HTML表要素の属性 */
    char    *caption;    /* HTML 表の表題 */
    char    **fieldName; /* フィールド名を置き換えるNULL終端の配列 */
} PQprintOpt;
```

この関数は以前に問い合わせ結果を表示するためにpsqlで使用されていましたが、今ではもう使用されていません。これはすべてのデータがテキスト書式であるという前提で動作することに注意してください。

33.3.3. 他の結果情報の取り出し

これらの関数はPGresultオブジェクトからその他の情報を取り出すために使用されます。

PQcmdStatus

PGresultを生成したSQLコマンドのコマンド状態タグを返します。

```
char *PQcmdStatus(PGresult *res);
```

これは通常単なるコマンド名ですが、処理行数など追加情報が含まれる場合もあります。呼び出し元はこの戻り値を直接解放してはいけません。関連するPGresultハンドルがPQclearに渡された時にこれは解放されます。

PQcmdTuples

SQLコマンドにより影響を受けた行数を返します。

```
char *PQcmdTuples(PGresult *res);
```

この関数はPGresultを生成したSQLコマンドにより影響を受けた行数を含む文字列を返します。この関数はSELECT、CREATE TABLE AS、INSERT、UPDATE、DELETE、MOVE、FETCH、COPY文の実行、あるいは、INSERT、UPDATE、DELETEを含むプリペアド問い合わせのEXECUTE文の後でのみ使用することができます。PGresultを生成したコマンドが他のコマンドであった場合、PQcmdTuplesは空文字列を返します。呼び出し元はこの戻り値を直接解放してはいけません。関連するPGresultハンドルがPQclearに渡された時にこれは解放されます。

PQoidValue

SQLコマンドが、OIDを持つテーブル内に1行のみを挿入するINSERTだった場合、あるいは、適切なINSERTを持つプリペアド問い合わせのEXECUTEだった場合に、挿入された行のOIDを返します。さもなくばInvalidOidを返します。また、INSERT文の影響を受けたテーブルがOIDを持たなかった場合、この関数はInvalidOidを返します。

```
Oid PQoidValue(const PGresult *res);
```

PQoidStatus

この関数はPQoidValueのため廃止予定になりました。またこれはスレッドセーフではありません。これは挿入された行のOIDを文字列として返します。一方PQoidValueはOID値を返します。

```
char *PQoidStatus(const PGresult *res);
```

33.3.4. SQLコマンドに含めるための文字列のエスケープ処理

PQescapeLiteral

```
char *PQescapeLiteral(PGconn *conn, const char *str, size_t length);
```

PQescapeLiteralは、SQLコマンド内で使用するために文字列をエスケープします。これは、SQLコマンド内のリテラル定数としてデータ値を挿入する時に有用です。特定の文字(引用符やバックスラッシュ)は、SQLパーサによって特殊な解釈がなされないようにエスケープされなければなりません。PQescapeLiteralはこの操作を行います。

PQescapeLiteralはstrパラメータをエスケープしたものをmalloc()で割り当てたメモリ内に返します。その結果が不要になったら、そのメモリをPQfreemem()を使用して解放しなければなりません。ゼロバイ

ト終端は必要なく、lengthに含めて数えてはいけません。(lengthバイトを処理する前にゼロバイト終端が見つかると、PQescapeLiteralはそのゼロで終了します。この動作はstrncpyと似ています。) 返される文字列では、PostgreSQL文字列リテラルパーサで適切に処理することができるように、すべての特殊文字は置換されます。ゼロバイト終端も追加されます。PostgreSQLの文字列リテラルでは前後に必要となる単一引用符も、その結果文字列には含まれています。

エラー時、PQescapeLiteralはNULLを返し、connオブジェクト内に適切なメッセージを残します。

> **ヒント**
>
> 信用できない入力元から受けとった文字列を扱う場合に適切なエスケープ処理を行なうことは非常に重要です。さもなくば、セキュリティ上の危険性が発生します。「SQLインジェクション」攻撃という弱点となり、好ましくないSQLコマンドがデータベースに流れてしまいます。

PQexecParamsまたは同義のルーチン内で別のパラメータとしてデータ値が渡される場合は、エスケープすることは必要でもなければ正しくもないことに注意してください。

PQescapeIdentifier

```
char *PQescapeIdentifier(PGconn *conn, const char *str, size_t length);
```

PQescapeIdentifierは、テーブル、列、関数名などのSQL識別子として使用できるように文字列をエスケープします。これはユーザが提供した識別子に、そのままではSQLパーサで識別子として解釈されない特殊な文字が含まれる可能性がある場合、または、大文字小文字の違いを維持しなければならない状況で識別子に大文字が含まれる可能性がある場合に有用です。

PQescapeIdentifierはstrパラメータをSQL識別子としてエスケープしたものをmalloc()で割り当てたメモリ内に返します。その結果が不要になったら、そのメモリをPQfreemem()を使用して解放しなければなりません。ゼロバイト終端は必要なく、lengthに含めて数えてはいけません。(lengthバイトを処理する前にゼロバイト終端が見つかると、PQescapeIdentifierはそのゼロで終了します。この動作はstrncpyと似ています。) 返される文字列では、SQL識別子として適切に処理することができるように、すべての特殊文字は置換されます。ゼロバイト終端も追加されます。その結果文字列の前後には二重引用符が付与されます。

エラー時、PQescapeIdentifierはNULLを返し、connオブジェクト内に適切なメッセージを残します。

> **ヒント**
>
> 文字列リテラルと同様、SQLインジェクション攻撃を防ぐために、信頼できない入力元から受けとる場合にはSQL識別子をエスケープしなければなりません。

PQescapeStringConn

```
size_t PQescapeStringConn(PGconn *conn,
                          char *to, const char *from, size_t length,
                          int *error);
```

PQescapeStringConnは、PQescapeLiteralとほぼ同様に文字列リテラルをエスケープします。PQescapeLiteralとは異なり、呼び出し元が適切な大きさのバッファを提供することに責任を持ちます。さらにPQescapeStringConnはPostgreSQLの文字リテラルとして囲まれなければならない単一引用符を生成しません。これは、結果をSQLコマンドに挿入するときに付与しなければなりません。fromパラメータはエスケープ対象の文字列の先頭を指すポインタです。lengthパラメータはこの文字列のバイト数を示します。ゼロバイト終端は必要なく、また、lengthではこれを数えてはなりません。(もしlengthバイト処理する前にゼロバイト終端が存在すると、PQescapeStringConnはそのゼロで終了します。この動作はstrncpyと同様です。) toは、最低でもlengthの2倍よりも1バイト多い文字を保持可能なバッファへのポインタにしなければなりません。さもないと、動作は不定になります。toとfrom文字領域が重なる場合の動作も不定です。

errorパラメータがNULLでなければ、*errorには成功の0か、エラーの0以外が設定されます。現時点であり得る唯一のエラー条件は、元文字列に無効なマルチバイト符号が含まれている場合です。出力文字列はエラーであっても生成されますが、サーバが不整合として却下することが想定できます。エラーの際、適切なメッセージはerrorがNULLかどうかにかかわらずconnオブジェクト内に格納されます。

PQescapeStringConnはtoに書き出したバイト数を返します。ただし、文字数にはゼロバイト終端は含まれません。

PQescapeString

PQescapeStringはPQescapeStringConnの推奨されない古いものです。

```
size_t PQescapeString (char *to, const char *from, size_t length);
```

PQescapeStringConnとの唯一の違いは、PQescapeStringはPGconnやerrorパラメータを取らないことです。このため(文字符号化方式のような)接続属性に依存する振舞いを調整できません。その結果間違った結果を返す可能性があります。また、エラー状態を通知する機能がありません。

PQescapeStringは、一度に1つのPostgreSQL接続のみで動作するクライアントプログラムでは安全に利用できます。(この場合知らなければならない「裏側に隠された情報」を知ることができるからです。) 他の場合には、セキュリティ要因でありPQescapeStringConnを利用することで避けなければなりません。

PQescapeByteaConn

bytea型としてSQLコマンド内で使用するバイナリデータをエスケープします。PQescapeStringConnと同様、これは、SQLコマンド文字列にデータを直接含める場合にのみに使用されます。

```
unsigned char *PQescapeByteaConn(PGconn *conn,
                                 const unsigned char *from,
                                 size_t from_length,
                                 size_t *to_length);
```

SQL文内のbyteaリテラルの一部として使用する場合、特定のバイト値はエスケープされなければなりません。PQescapeByteaConnは16進数符号化またはバックスラッシュエスケープ処理を使用してバイトをエスケープします。詳しくは8.4を参照してください。

fromパラメータはエスケープ対象の文字列の先頭バイトを指し示すポインタです。from_lengthパラメータは、このバイナリ列内のバイト数を指定します。(ゼロバイト終端は不要、かつ、数えられません。)

to_lengthパラメータは結果となるエスケープされた文字列の長さを保持する変数へのポインタです。この結果文字列長は、結果内のゼロバイト終端を含みます。

PQescapeByteaConnは、fromパラメータが示すバイナリ文字列をエスケープしたものをmalloc()で確保したメモリ内に返します。その結果が不要になったら、このメモリをPQfreemem()を使用して解放しなければなりません。返される文字列では、PostgreSQLリテラル文字列パーサとbytea入力関数によって適切に処理できるように、すべての特殊な文字が置換されています。ゼロバイト終端も追加されます。PostgreSQLのリテラル文字列をくくる単一引用符は結果文字列には含まれません。

エラー時、ヌルポインタを返し適切なエラーメッセージをconnオブジェクトに格納します。現在、唯一あり得るエラーは結果文字列のメモリ不足です。

PQescapeBytea

PQescapeByteaは、PQescapeByteaConnの推奨されない古いものです。

```
unsigned char *PQescapeBytea(const unsigned char *from,
                             size_t from_length,
                             size_t *to_length);
```

PQescapeByteaのPQescapeByteaConnとの唯一の違いは、PGconnパラメータです。このためPQescapeByteaは、一度に1つのPostgreSQL接続を使用するクライアントプログラムのみで安全に利用することができます。(この場合知らなければならない「裏側に隠された情報」を知ることができるからです。) 複数のデータベース接続を使用するプログラムでは間違った結果を返す可能性があります。(このような場合はPQescapeByteaConnを使用してください。)

PQunescapeBytea

バイナリデータの文字列表現をバイナリデータに変換します。つまり、PQescapeByteaの逆です。これは、byteaデータをテキスト書式で受けとった場合に必要とされます。しかし、バイナリ書式で受けとった場合は不要です。

```
unsigned char *PQunescapeBytea(const unsigned char *from, size_t *to_length);
```

fromパラメータは、例えば、bytea列にPQgetvalueを行なった場合に返される可能性がある、文字列を指し示すポインタです。PQunescapeByteaは、この文字列表現をバイナリ表現に変換します。malloc()で確保したバッファへのポインタを返します。エラー時はNULLです。また、このバッファのサイズをto_lengthに格納します。不要になったら、この結果をPQfreememを使用して解放しなければなりません。

この変換は、PQescapeByteaの逆ではありません。文字列はPQgetvalueから受け取る場合「エスケープされた」ことを予想しないためです。特にこれは、文字列の引用符付けを意識する必要がなく、そのためPGconnパラメータを持つ必要がないことを意味します。

33.4. 非同期コマンドの処理

PQexec関数は普通の同期処理のアプリケーションにおけるコマンドの送信に適したものです。しかし、一部のユーザにとって重要な問題となり得る、数個の問題があります。

- PQexec はコマンドが完了するまで待機します。アプリケーションによっては(例えばユーザインタフェースの調整処理など)他に行うべき作業があります。この場合は応答待ちでブロックさせたくはありません。
- クライアントアプリケーションの実行が結果を待っている間停止されるため、アプリケーションで送信したコマンドをキャンセルさせる指示を行うことは困難です。(シグナルハンドラを使って達成することができますが、他の方法はありません。)
- PQexecが返すことができるPGresult構造体は1つだけです。もし送信した問い合わせ文字列が複数のSQLコマンドを含んでいる場合、PQexecは最後のものだけを除いて、残りすべてのPGresultを破棄してしまいます。
- PQexecは常にコマンドの結果全体を収集し、1つのPGresult内に保管します。アプリケーションにおけるエラー処理を簡単にしますが、多くの行になる結果では非現実的になるかもしれません。

アプリケーションにとってこのような制限が望ましくない場合は、代わりにPQexecを構成する関数PQsendQueryとPQgetResultを使用してください。また、PQsendQueryParamsとPQsendPrepare、PQsendQueryPrepared、PQsendDescribePrepared、PQsendDescribePortalもあり、PQgetResultを使用して、それぞれPQexecParamsとPQprepare、PQexecPrepared、PQdescribePrepared、PQdescribePortalと同等の機能を行うことができます。

PQsendQuery

結果を待つことなく、サーバにコマンドを発行します。コマンドの登録に成功した場合1が、失敗した場合0が返されます。(後者の場合、PQerrorMessageを使用して失敗についてのより多くの情報を取り出してください。)

```
int PQsendQuery(PGconn *conn, const char *command);
```

PQsendQuery呼び出しが成功したら、PQgetResultを繰り返し呼び出して、実行結果を取得します。PQgetResultがヌルポインタを返し、コマンドが完了したことを示すまでは、(同じ接続で)PQsendQueryを再度呼び出すことはできません。

PQsendQueryParams

結果を待つことなく、サーバにコマンドとパラメータとを分けて発行します。

```
int PQsendQueryParams(PGconn *conn,
                      const char *command,
                      int nParams,
                      const Oid *paramTypes,
                      const char * const *paramValues,
                      const int *paramLengths,
                      const int *paramFormats,
                      int resultFormat);
```

これは、問い合わせのパラメータが問い合わせ文字列と分けて指定できる点を除き、PQsendQueryと同じです。この関数のパラメータはPQexecParamsと同様に扱われます。PQexecParams同様、これは2.0プロトコルでは動作しませんし、問い合わせ文字列には1つのコマンドしか指定できません。

PQsendPrepare

指定パラメータを持つプリペアド文の作成要求を送信します。その完了を待ちません。

```
int PQsendPrepare(PGconn *conn,
                  const char *stmtName,
                  const char *query,
                  int nParams,
                  const Oid *paramTypes);
```

これはPQprepareの非同期版です。要求の登録に成功した場合1が、失敗した場合0が返されます。呼び出しの成功の後、サーバがプリペアド文の生成に成功したかを確認するためにはPQgetResultを呼び出してください。この関数のパラメータはPQprepareと同様に扱われます。PQprepare同様、これは2.0プロトコルの接続では動作しません。

PQsendQueryPrepared

結果を待つことなく、指定したパラメータでプリペアド文の実行要求を送信します。

```
int PQsendQueryPrepared(PGconn *conn,
                        const char *stmtName,
                        int nParams,
                        const char * const *paramValues,
                        const int *paramLengths,
                        const int *paramFormats,
                        int resultFormat);
```

これはPQsendQueryParamsと似ていますが、実行されるコマンドは問い合わせ文字列ではなく、事前に準備された文の名前で指定されます。この関数のパラメータはPQexecPreparedと同様に扱われます。PQexecPrepared同様、これは2.0プロトコルでは動作しません。

PQsendDescribePrepared

指定したプリペアド文に関する情報入手要求を送ります。入手完了まで待機しません。

```
int PQsendDescribePrepared(PGconn *conn, const char *stmtName);
```

これはPQdescribePreparedの非同期版です。要求の受付けが可能であれば1が返されます。不可能であれば0が返されます。呼び出しに成功した後、PQgetResultを呼び出して結果を入手してください。この関数のパラメータはPQdescribePreparedと同じように扱われます。PQdescribePrepared同様、2.0プロトコル接続では動作しません。

PQsendDescribePortal

指定したポータルに関する情報入手要求を送信します。完了まで待機しません。

```
int PQsendDescribePortal(PGconn *conn, const char *portalName);
```

これはPQdescribePortalの非同期版です。要求の受付けが可能であれば1が返されます。不可能であれば0が返されます。呼び出しに成功した後、PQgetResultを呼び出して結果を入手してください。この関数のパラメータはPQdescribePortalと同じように扱われます。PQdescribePortal同様、2.0プロトコル接続では動作しません。

PQgetResult

以前に呼び出したPQsendQuery、PQsendQueryParams、PQsendPrepare、PQsendQueryPrepared、PQsendDescribePrepared、PQsendDescribePortalから次の結果を待ち、その結果を返します。コマンドが完了し、これ以上結果がない場合は、ヌルポインタが返されます。

```
PGresult *PQgetResult(PGconn *conn);
```

PQgetResultは、コマンドの完了を示すヌルポインタが返るまで、繰り返し呼び出さなければなりません。(コマンド実行中以外での呼び出しでは、PQgetResultは単にヌルポインタを返します。)PQgetResultの非ヌルの結果はそれぞれ前述と同じPGresultアクセス用関数を使用して処理されなければなりません。各結果オブジェクトに対する処理が終わったら、そのオブジェクトをPQclearを使用して解放することを忘れないでください。コマンドが活動中、かつ、必要な応答データがまだPQconsumeInput で読み込まれていない場合にのみ、PQgetResultがブロックすることに注意してください。

> **注記**
>
> PQresultStatusが致命的なエラーを示した場合であっても、libpqがエラー情報を完全に処理できるようにヌルポインタが返されるまでPQgetResultを呼び出さなければなりません。

PQsendQueryとPQgetResultを使うことでPQexecの問題は1つ解決します。つまり、コマンドが複数のSQLコマンドを含んでいる場合でも、これらのコマンドの結果を個々に得ることができるわけです（これは多重処理を単純な形で実現します。単一のコマンド文字列に含まれる複数の問い合わせの内、後ろのものが処理中でもフロントエンドは先に完了した結果から扱うことができるからです）。

PQsendQueryおよびPQgetResultで得られる、その他のよく望まれる機能は多くの問い合わせ結果を一度に1行受け取ることです。これについては33.5で説明します。

サーバが次のSQLコマンドの処理に入ると、それが完了するまでやはりPQgetResultの呼び出しがフロントエンドをブロックしてしまいます。さらに以下の2つの関数をうまく使用してこれを防ぐことができます。

PQconsumeInput

サーバからの入力が可能になった場合、それを吸い取ります。

```
int PQconsumeInput(PGconn *conn);
```

PQconsumeInput は通常、「エラーなし」を示す1を返しますが、何らかの障害があると0を返します（この場合は、PQerrorMessageを参考にしてください）。この結果は、何らかの入力データが実際に収集されたかどうかを示しているのではないことに注意してください。PQconsumeInput の呼び出し後、アプリケーションはPQisBusy、または必要があればPQnotifiesを呼び出して状態に変化がないか調べることができます。

PQconsumeInput は、結果や通知を扱うようにまだ準備していないアプリケーションからでも呼び出すことができます。この関数は有効なデータを読み込んでバッファに保存し、結果としてselectによる読み込み準備完了の通知をリセットします。従ってアプリケーションはPQconsumeInput を使うとselect()の検査条件をただちに満たすことができますから、あとはゆっくりと結果を調べてやればいいわけです。

PQisBusy

この関数が1を返したのであれば、問い合わせは処理の最中で、PQgetResultも入力を待ったままブロック状態になってしまうでしょう。0が返ったのであれば、PQgetResultを呼び出してもブロックされないことが保証されます。

```
int PQisBusy(PGconn *conn);
```

PQisBusy自身はサーバからデータを読み込む操作をしません。ですから、まず最初にPQconsumeInput を呼び出す必要があります。そうしないとビジー状態がいつまでも続きます。

これら3関数を使用するアプリケーションは通常、select()もしくはpoll()を使用するメインループを持ち、対応しなければならないすべての状態を待機しています。その内の1つの条件は、サーバからの利用可能な入力となるでしょう。これは、select()の見地からは、PQsocketで識別されるファイル記述子上で読み込み可能なデータがあることを意味します。メインループが入力準備完了を検出すると、その入力を読み込むためにPQconsumeInput を呼び出さなければなりません。そして、PQisBusyを、更にPQisBusyが偽(0)を返す場合にPQgetResultも呼び出すことができます。また、PQnotifiesを呼び出して、NOTIFYメッセージ(33.8を参照)を検出することもできます。

また、PQsendQuery/PQgetResultを使用するクライアントは、サーバで処理中のコマンドに対してキャンセルを試行することができます。33.6を参照してください。しかし、PQcancelの戻り値と関係なく、アプリケーションはPQgetResultを使用した通常の結果読み取り手順を続けなければなりません。キャンセル手続きの成功は単に、そのコマンドを通常よりも早めに終わらせるだけです。

上述の関数を使用して、データベースサーバからの入力待ちのためのブロックを行わずに済みます。しかしまだ、サーバへの出力送信を待つためにアプリケーションはブロックする可能性があります。これは比較的あまり発生しませんが、非常に長いSQLコマンドやデータ値が送信される場合に発生することがあります。(しかし、アプリケーションがCOPY IN経由でデータを送信する場合よく発生します。) この発生を防ぎ、完全な非ブロックのデータベース操作を行うためには、さらに以下の関数を使用してください。

PQsetnonblocking

接続の非ブロック状態を設定します。

```
int PQsetnonblocking(PGconn *conn, int arg);
```

argが1の場合、接続状態を非ブロックに設定します。argが0の場合はブロックに設定します。問題がなければ0が、エラー時は-1が返ります。

非ブロック状態ではPQsendQuery、PQputline、PQputnbytes、PQputCopyDataおよびPQendcopyの呼び出しはブロックされませんが、再度呼び出さなければならない場合、エラーが返ります。

PQexecは非ブロックモードにはしたがわないことに注意してください。この関数の呼び出しは、必ずブロック方式で動作します。

PQisnonblocking

データベース接続のブロック状態を返します。

```
int PQisnonblocking(const PGconn *conn);
```

接続が非ブロック状態の場合は1が、ブロック状態の場合は0が返ります。

PQflush

キューに蓄えられたサーバへの出力データの吐き出しを行います。成功時(および送信キューが空の場合)は0が返ります。何らかの原因で失敗した場合は-1が、送信キュー内のデータをすべて送信できなかった場合は1が返ります。(これは接続が非ブロックの場合にのみ発生します。)

```
int PQflush(PGconn *conn);
```

非ブロック接続時にはコマンドやデータを送信した後に、PQflushを呼び出してください。1が返った場合、ソケットの読み込みまたは書き込み準備ができるまで待ってください。書き込み準備ができたら、PQflushを再度呼び出してください。読み込み準備ができたら、PQconsumeInput を呼び出してから、PQflushを再度呼び出してください。これをPQflushが0を返すまで繰り返してください。(例えばNOTICEメッセージのように、こちらがそのデータを読むまで、サーバがデータを送ろうとするのを妨げ、こちらのデータを読もうとしないことがありますので、読み込み準備ができたことを確認してPQconsumeInput で入力をすべて抜き取ることが必要です。) PQflushが0を返した後は、ソケットの読み込み準備が整うまで待ち、上述のように応答を読み取ってください。

33.5. 1行1行問い合わせ結果を受け取る

通常、libpqはSQLコマンドの結果全体を収集し、それを1つのPGresultとしてアプリケーションに返します。これは、多くの行数を返すコマンドでは動作しなくなるかもしれません。こうした場合、アプリケーションはPQsendQueryとPQgetResultを単一行モードで使用することができます。このモードでは、結果行は、サーバから受け取ったかのように、アプリケーションに1度に1行返されます。

単一行モードに入るためには、PQsendQuery(または同系列の関数)の呼び出しに成功した直後にPQsetSingleRowModeを呼び出してください。このモード選択は、現在実行中の問い合わせに対してのみ有効です。その後、33.4の説明通りに、ヌルを返すようになるまでPQgetResultを繰り返し呼び出してください。問い合わせが何らかの行を返す場合、PGRES_TUPLES_OKではなくPGRES_SINGLE_TUPLE状態コードを持つ以外通常の問い合わせ結果と同じように見える、個々のPGresultオブジェクトを返します。最後の行の後、または問い合わせがゼロ行を返す場合は即座に、PGRES_TUPLES_OK状態のゼロ行のオブジェクトが返されます。これはもう行が届かないことを通知するものです。(しかしヌルが返るまでPQgetResultを呼び出さなければならないことに注意してください。) PGresultオブジェクトのすべては、その問い合わせに対する通常のPGresultと同一の行説明データ(列名、型など)を持ちます。各オブジェクトは通常通りPQclearで解放しなければなりません。

PQsetSingleRowMode

現在実行中の問い合わせについて単一行モードを選択します。

```
int PQsetSingleRowMode(PGconn *conn);
```

この関数はPQsendQueryまたはその系列の関数のいずれかの後即座に、PQconsumeInputやPQgetResultなど接続に対する何らかの他の操作を行う前のみに呼び出すことができます。正しい時点で呼び出された場合、この関数は現在の問い合わせに対して単一行モードを有効にし、1を返します。この他の場合、モードは変更されず、関数はゼロを返します。いずれの場合でも、現在の問い合わせが完了した後に通常モードに戻ります。

注意

問い合わせを処理している間、サーバはいくつか行を返した後にエラーになり、問い合わせがアボートする可能性があります。通常のlibpqでは、こうした行を破棄しエラーのみを報告します。しかし単一行モードでは、これらの行はすでにアプリケーションに返されています。このためアプリケーションはPGRES_SINGLE_TUPLE状態のPGresultオブジェクトをいくつか見た後にPGRES_FATAL_ERRORオブジェクトを見るかもしれません。適切な振る舞いのトランザクションのために、最終的に問い合わせが失敗した場合、アプリケーションはこれまで処理した行を破棄するまたは取り消すように設計しなければなりません。

33.6. 処理中の問い合わせのキャンセル

本節で説明する関数を使用して、クライアントアプリケーションはサーバで処理中のコマンドをキャンセルする要求を行うことができます。

PQgetCancel

特定のデータベース接続を通して発行されたコマンドをキャンセルするために必要な情報を持つデータ構造を作成します。

```
PGcancel *PQgetCancel(PGconn *conn);
```

PQgetCancelは、与えられたPGconn接続オブジェクトのPGcancelオブジェクトを作成します。与えられたconnがNULLもしくは無効な接続であった場合、NULLが返されます。PGcancelオブジェクトは不透明な構造体であり、アプリケーションから直接アクセスすることができません。これはPQcancelもしくはPQfreeCancelに渡すことしかできません。

PQfreeCancel

PQgetCancelで作成されたデータ構造を解放します。

```
void PQfreeCancel(PGcancel *cancel);
```

PQfreeCancelは事前にPQgetCancelで作成されたデータオブジェクトを解放します。

PQcancel

サーバに現在のコマンドの廃棄処理を要求します。

```
int PQcancel(PGcancel *cancel, char *errbuf, int errbufsize);
```

キャンセル要求の受け入れが成功すれば1を、そうでなければ0を返します。失敗した場合、errbufにそれを説明するエラーメッセージが収納されます。errbufはerrbufsizeサイズの文字配列でなければなりません。(推奨サイズは256バイトです。)

しかし、要求の受け入れが成功したとしても、その要求の効果が出ることは全く保証していません。もしキャンセル操作が有効であれば、現在のコマンドは間もなく中断され、エラーが結果として返ります。キャンセル操作に失敗した場合(例えばバックエンドがすでにコマンド処理を終了していたため)、目に見える結果は何も出てこなくなります。

errbufがシグナルハンドラ内のローカル変数であれば、PQcancelはシグナルハンドラから起動しても問題ありません。PQcancelの実行中、PGcancelは読み取りのみです。従って、PGconnオブジェクトを操作するスレッドと別のスレッドからこの関数を呼び出すこともできます。

PQrequestCancel

PQrequestCancelはPQcancelの廃止予定の変形版です。

```
int PQrequestCancel(PGconn *conn);
```

サーバに現在のコマンドの廃棄処理を要求します。これはPGconnオブジェクトを直接扱い、また、失敗した場合エラーメッセージはPGconnオブジェクト内に収納されます。(PQerrorMessageにより取り出すことができます。) 機能的には同一ですが、PGconnのエラーメッセージが上書きされることにより、その接続で現在進行中の操作が壊れてしまうため、この方法は複数スレッドプログラムやシグナルハンドラでは問題が起こります。

33.7. 近道インタフェース

PostgreSQLは、サーバへの簡単な関数呼び出しを送信する近道 (fast-path) インタフェースを用意しています。

> **ヒント**
>
> この関数はどちらかというと廃れたものです。同様の性能やそれ以上の機能を、関数呼び出しを定義したプリペアド文を設定することで達成できるからです。そして、その文をパラメータと結果をバイナリ転送するように実行すれば、近道関数呼び出しを置き換えることになります。

PQfn関数は近道インタフェースを使ってサーバ関数の実行を要求します。

```
PGresult *PQfn(PGconn *conn,
               int fnid,
               int *result_buf,
               int *result_len,
```

```
                int result_is_int,
                const PQArgBlock *args,
                int nargs);

typedef struct
{
    int len;
    int isint;
    union
    {
        int *ptr;
        int integer;
    } u;
} PQArgBlock;
```

fnid引数は実行する関数のOIDです。argsとnargsは関数に渡すパラメータを定義します。これらは関数宣言における引数リストに一致しなければなりません。パラメータ構造体のisintが真の場合、u.integerの値はサーバに指定長の整数として送信されます。(これは2もしくは4バイトでなければなりません。) この時、適切なバイト順の交換が行なわれます。isintが偽の場合は、*u.ptrで指定されたバイト数が無処理で送信されます。関数のパラメータデータ型をバイナリ転送で行うために、このデータはサーバで想定する書式である必要があります。(u.ptrをint *型と宣言するのは歴史的なものです。void *と考えた方が良いでしょう。) result_bufは関数の戻り値を格納するバッファを指しています。呼び出し側は戻り値を格納するのに十分な領域を確保しておかなければいけません。(ライブラリ側ではこの検査はしていません!) バイト単位での結果の実データ長はresult_lenが指す整数で返されます。結果が2、4バイト整数だと想定できるならresult_is_intを1に、そうでなければ0を設定します。result_is_intを1にすれば、必要に応じて値のバイト順を入れ換えるようlibpqに指示することになります。そしてクライアントマシン上で正しいint値となるように転送します。4バイト整数は認められた結果の大きさで*result_bufに転送されることに注意してください。result_is_intが0の場合は、バックエンドが送ったバイナリ書式のバイト列を何も修正せずに返します。(この場合、result_bufはvoid *型と考えた方が良いでしょう。)

PQfnは常に有効なPGresult*を返します。結果を使う前にはまず、結果ステータスを調べておくべきでしょう。結果が必要なくなった時点で、PQclearによって、PGresultを解放するのは、呼び出し側の責任です。

このインタフェースを使用した場合、NULL引数やNULL結果、セット値の結果を扱うことができないことに注意してください。

33.8. 非同期通知

PostgreSQLは、LISTENとNOTIFYコマンドを使用した、非同期通知をサポートします。クライアントセッションは、LISTENコマンドを使用して処理対象とする特定の通知チャネルを登録します。(通知監視を取り止めるにはUNLISTENコマンドを使用します。) 任意のセッションでそのチャネル名によるNOTIFYコマンドが実行されると、特定チャネルを監視しているすべてのセッションは非同期に通知を受け取ります。監視者に追加データを通信するために「ペイロード」文字列を渡すことができます。

libpqアプリケーションは、通常のSQLによる問い合わせと同じようにLISTEN、UNLISTENおよびNOTIFYコマンドを発行することができます。NOTIFYメッセージの到着は、続いてPQnotifies.を呼び出せば検出できます。

PQnotifies関数は、サーバから受信した通知メッセージの未処理リストから次の通知を返します。保留中の通知がなくなればヌルポインタを返します。PQnotifiesが通知を返すと、その通知は処理済みとみなされ、通知リストから取り除かれます。

```
PGnotify *PQnotifies(PGconn *conn);

typedef struct pgNotify
{

    char *relname;              /* 通知チャネル名 */
    int  be_pid;                /* 通知元サーバプロセスのプロセスID */
    char *extra;                /* 通知ペイロード文字列 */
} PGnotify;
```

PQnotifiesで返されたPGnotifyオブジェクトの処理が終わったら、PQfreememを使用して確実に解放してください。PGnotifyポインタを解放することは重要です。relnameとextraフィールドは別の割り当てを表していません。(これらのフィールド名は歴史的なものです。特にチャネル名はリレーション名と関係するものである必要はありません。)

例 33.2で非同期通知を使用したサンプルプログラムを示しています。

PQnotifiesは実際にサーバのデータを読み出すわけではありません。これは単に、他のlibpq関数が吸収してしまっていた通知メッセージを返すだけです。libpqの古いリリースでは、NOTIFYメッセージを適切な時点で確実に受け取るには、空の問い合わせでも何でも、とにかく一定時間ごとに問い合わせを送り、そしてPQexecを実行するたびにPQnotifiesを検査するしかありませんでした。今でもこの方法は動作しますが、処理能力の無駄使いをすることになるのでやめておくべきでしょう。

実行すべき問い合わせがない時にNOTIFYメッセージを検査するよい方法は、まずPQconsumeInput を呼び出し、それからPQnotifiesを検査することです。サーバからのデータの到着をselect()で待つことができ、不必要な動作でCPUパワーを消費してしまうことがありません。(select()で使用するファイル記述子番号の取得については、PQsocketを参照してください。) なお、これは問い合わせにPQsendQueryとPQgetResultを使った時でも、またはおなじみのPQexecを使った時でも動作します。しかし通知がコマンドの処理中に届いていないかどうか、PQgetResultあるいはPQexecの実行ごとにPQnotifiesを調べることを忘れないようにしておくべきです。

33.9. COPYコマンド関連関数

PostgreSQLのCOPYコマンドでは、libpqが使っているネットワーク接続に対して読み込み、あるいは書き込みを選ぶことができるようになっています。本節で説明する関数により、アプリケーションはコピーするデータの提供やコピーされるデータの使用が可能になるという利点を持ちます。

全体的な処理として、アプリケーションはまずPQexecもしくは同等な関数経由でCOPY SQLコマンドを発行します。(コマンドでエラーが発生しなければ)この応答は、(指定したコピーの方向に応じて)PGRES_COPY_OUTもしくはPGRES_COPY_INという状態コードを持ったPGresultになります。その後、アプリケーションは本節の関数を使用して、行データを受信、もしくは、送信しなければなりません。データの転送が完了した時、転送に成功したか失敗したかを示す別のPGresultオブジェクトが返されます。その状態は、成功時に

はPGRES_COMMAND_OKになり、何らかの問題が起きていた時には PGRES_FATAL_ERRORになります。この時点で、別のSQLコマンドをPQexec経由で発行することができます。(COPY操作の実行中は、同じ接続を使用して他のSQLコマンドを実行することはできません。)

COPYコマンドが、他にもコマンドを含んだ文字列としてPQexec経由で発行された場合、アプリケーションはCOPY処理を終えた後に、PQgetResult経由で結果の取り出しを続けなければなりません。PQexecコマンド文字列が完了し、その後のコマンドが安全に発行できることが確実になるのは、PQgetResultがNULLを返す時のみです。

本節の関数は、PQexecもしくはPQgetResultからPGRES_COPY_OUTもしくはPGRES_COPY_INという結果状態を得た後のみに実行されなければなりません。

これらの状態値の一つを持つPGresultオブジェクトは、開始したCOPY操作に関する追加データを持ちます。この追加データは、以下の問い合わせ結果を持つ接続で使用される関数を使用して利用することができます。

PQnfields

コピーされる列(フィールド)数を返します。

PQbinaryTuples

0は、コピー全体の書式がテキスト(改行で区切られた行、区切り文字で区切られた列など)であることを示します。1は、コピー全体の書式がバイナリであることを示します。詳細はCOPYを参照してください。

PQfformat

コピー操作対象の列それぞれに関した書式コード(テキストでは0、バイナリでは1)を返します。コピー全体の書式がテキストの場合は、列単位の書式コードは常にゼロです。しかし、バイナリ書式はテキスト列もバイナリ列もサポートすることができます。(しかし、現在のCOPY実装では、バイナリコピーでのみバイナリ列が発生します。そのため、今の所列単位の書式は常に全体の書式と一致します。)

> **注記**
>
> これらの追加データ値はプロトコル3.0を使用した場合にのみ利用可能です。プロトコル2.0を使用する場合は、これらの関数はすべて0を返します。

33.9.1. COPYデータ送信用関数

これらの関数は、COPY FROM STDIN期間にデータを送信するために使用されます。接続がCOPY_IN状態でない時に呼び出された場合、これらは失敗します。

PQputCopyData

COPY_IN状態の間、サーバにデータを送信します。

```
int PQputCopyData(PGconn *conn,
                  const char *buffer,
```

```
                    int nbytes);
```

指定したbufferにあるCOPYデータをnbytes長分、サーバに送信します。データがキューに入れられた場合、この結果は1になります。バッファが一杯でキューに入らなかった場合はゼロになります。(これは、接続が非ブロックモードの場合にのみ起こります。) エラーが発生した場合は-1になります。(戻り値が-1の場合、詳細を取り出すためにはPQerrorMessageを使用してください。戻り値がゼロの場合は書き込み準備が整うまで待ち、再実行してください。)

アプリケーションはCOPYデータストリームを使いやすい大きさのバッファに分けて読み込むことができます。送信時の読み込みバッファの境界には意味的な重要性はありません。データストリームの内容は、COPYコマンドで想定しているデータ書式に一致している必要があります。詳細はCOPYを参照してください。

PQputCopyEnd

COPY_IN状態の間に、サーバにデータ終了指示を送信します。

```
int PQputCopyEnd(PGconn *conn,
                 const char *errormsg);
```

errormsgがNULLの場合は、COPY_IN操作の終了に成功しました。errormsgがNULLでない場合は、COPYは強制的に失敗させられました。errormsgが指し示す文字列はエラーメッセージとして使用されます。(しかし、このエラーメッセージが正しくサーバから返ったものであるとは仮定すべきではありません。サーバは既に別の原因でCOPYに失敗していた可能性があります。また、この強制的な失敗は3.0より前のプロトコルの接続を使用している場合は動作しません。)

終端メッセージが送信された場合は結果は1になります。非ブロックモードでは、終端メッセージがキューに入れられたことしか意味しないかもしれません。(非ブロックモードでデータが送信されたことを確認するには、次に書き込み準備ができるまで待ち、PQflushを呼ぶことを、それが0を返すまでくり返します。) バッファが一杯で終端メッセージがキューに入れられなかった場合はゼロになります。これは、接続が非ブロックモードの場合にのみ起こります。(この場合、書き込み準備ができるまで待ち、再度PQputCopyEndを呼び出してみてください。) ハードエラーが発生した場合は-1になります。このとき、詳細を取得するためにPQerrorMessageを使用できます。

PQputCopyEndの呼び出しに成功した後、PQgetResultを呼び出してCOPYコマンドの最終的な結果状態を取り出してください。通常の方法でこの結果が使用できるようになるまで待機しても構いません。そして、通常の操作に戻ってください。

33.9.2. COPYデータ受信用関数

これらの関数はCOPY TO STDOUT時にデータを受信するために使用されます。COPY_OUT状態以外の接続で呼び出すと、失敗します。

PQgetCopyData

COPY_OUT状態時にサーバからデータを受信します。

```
int PQgetCopyData(PGconn *conn,
```

```
                  char **buffer,
                  int async);
```

COPY期間中、サーバから別の行データの入手を試みます。常に1度に1つの行データが返されます。部分的な行のみが利用可能な場合は返されません。行データの取得に成功することは、そのデータを保持するためのメモリチャンクの割り当てを意味します。bufferパラメータは非NULLでなければなりません。*bufferは割り当てられたメモリへのポインタに、バッファが返されなかった場合はNULLに設定されます。非NULLの結果バッファは、不要になったらPQfreememを使用して解放しなければなりません。

行の取り込みに成功した時、戻り値は行内のデータのバイト数になります。(これは常に0より大きくなります。) 返された文字列は常にヌル終端ですが、おそらくテキストCOPYでのみ有用になるでしょう。ゼロという結果は、COPYが進行中で、行がまだ利用できない状態であることを示します。(asyncが真の場合にのみ発生することがあります。) -1という結果は、COPYが完了したことを示します。-2という結果はエラーが発生したことを示します。(その理由についてはPQerrorMessageを参照してください。)

asyncが真(非0)の場合、PQgetCopyDataは入力待ちのためのブロックを行いません。COPY実行中で完全な行を取り出せない場合PQgetCopyDataは0を返します。(この場合、再試行の前に読み込み準備が整うまで待機してください。PQconsumeInput を呼び出したかどうかは関係ありません。) asyncが偽(0)の場合、PQgetCopyDataはデータが利用できるようになるまで、もしくは、操作が完了するまでブロックします。

PQgetCopyDataが-1を返した後、PQgetResultを呼び出して、COPYコマンドの最終結果状態を取り出してください。通常の方法で結果が利用できるようになるまで待機しても構いません。そして、通常の操作に戻ってください。

33.9.3. 廃れたCOPY用関数

以下の関数はCOPYを取扱う、古めの手法を行います。これらはまだ動作しますが、エラーの取扱いが貧弱であることやデータの終端を検知する方法が不便であることより使用を奨めません。

PQgetline

改行で終端する文字列(サーバから送信されたもの)を長さlengthのバッファ用文字列に読み込みます。

```
int PQgetline(PGconn *conn,
              char *buffer,
              int length);
```

この関数はバッファにlength-1個までの文字をコピーし、終端の改行を1バイトのゼロに置き換えます。PQgetlineは、入力の終端ではEOFを、行全体が読み込まれれば0を返します。そしてまだ終端の改行が読み込まれていないうちにバッファがいっぱいになってしまった場合は1を返します。

アプリケーションは新しく読み込んだ行が、\.という2文字であるかどうか確認しなければいけません。この2文字は、COPYコマンドの結果をサーバが送信し終えたことを示すものです。アプリケーションには、仮にlength-1文字より長い行を受け取るようなことがあっても、間違いなく\.行を認識するような配慮が必要です (また例えば長いデータの行の終端を、最終行と取り違えないようにもしてください)。

PQgetlineAsync

COPYデータ行（サーバから送信されたもの）を、ブロッキングなしでバッファに読み込みます。

```
int PQgetlineAsync(PGconn *conn,
                   char *buffer,
                   int bufsize);
```

PQgetlineと似ていますが、COPYのデータを非同期的に、つまりブロッキングなしで読み出さなければならないアプリケーションで使用することができます。COPYコマンドを発行し、そしてPGRES_COPY_OUT応答を受け取ったら、アプリケーションはデータ終了の合図を受け取るまでPQconsumeInputとPQgetlineAsyncを呼び出します。

PQgetlineと違い、この関数はデータ終了の検出に対して責任を持ちます。

PQgetlineAsyncの個々の呼び出しでは、libpqの入力バッファ内で完全な行データが利用できる場合にデータを返します。さもなければ、行の残りが届くまでデータは返されません。この関数は、コピーデータの終端を示す符号を認識すると-1を、また何もデータがなければ0を、そしてデータを返す場合はそのバイト数を正の値で返します。もし-1が返されたら、呼び出し側は次にPQendcopyを呼び出さなければいけません。それから通常の処理に戻ります。

返されるデータは行データの境界を越えて拡張されることはありません。可能であれば行全体を一度に返します。しかし呼び出し側が準備したバッファが少なすぎ、サーバから送られてくる行を保持しておくことができない場合には、分割された行データを返します。テキストデータでは、これは最後の1バイトが\nかどうかを確認すれば検出できます。（バイナリCOPYの場合に同様の検出を行うためには、実際にCOPYデータの書式を解析しなければなりません。）なお、返される文字列はヌル終端ではありません。（ヌル終端を後から付け加えるのであれば、実際に確保するバッファサイズ-1をbufsizeとして渡すようにしてください。）

PQputline

サーバにヌル終端の文字列を送信します。問題なければ0を返します。文字列の送信ができなかった場合はEOFを返します。

```
int PQputline(PGconn *conn,
              const char *string);
```

PQputlineの呼び出しによって送信されるCOPYデータストリームは、PQgetlineAsyncで返される書式と同じ書式を持ちます。ただし、アプリケーションは、PQputline毎に正確に1つのデータ行を送信するように強制されていません。呼び出し毎に行の一部や複数の行を送信しても問題ありません。

> **注記**
>
> PostgreSQLプロトコル3.0より前では、アプリケーションは、サーバに対してCOPYデータの送信を完了したことを通知するために、最終の行として\.という2文字を明示的に送信する必要がありました。これはまだ動作します。しかし、これは廃れたものとして、\.の特殊な意味は将来のリリースで無くなることが予想されます。実際のデータの送信完了後にPQendcopyを呼び出すことが重要です。

PQputnbytes

ヌル終端ではない文字列をサーバに送信します。問題なければ0を返します。文字列の送信ができなかった場合はEOFを返します。

```
int PQputnbytes(PGconn *conn,
                const char *buffer,
                int nbytes);
```

これはまさにPQputlineと同様です。ただし、直接送信バイト数を指定するため、ヌル終端である必要がありません。バイナリデータを送信する時はこのプロシージャを使用してください。

PQendcopy

サーバと同期します。

```
int PQendcopy(PGconn *conn);
```

この関数はサーバがコピーを完了するのを待ちます。この関数は、PQputlineを使ったサーバへの文字列送信が完了した時点、あるいはPQgetlineを使ったサーバからの文字列受信が完了した時点のいずれでも呼び出さなければなりません。これを発行しないと、サーバはクライアントとの「同期がずれた」状態になってしまいます。この関数から戻った時点で、サーバは次のSQLコマンドを受ける準備が整います。正常に終了した場合、返り値は0です。さもなくば、非ゼロです。(戻り値が非ゼロの場合、PQerrorMessageを使用して詳細を取り出してください。)

PQgetResultを使う場合、アプリケーションはPQgetlineを繰り返し呼び出してPGRES_COPY_OUTに応答し、終端行を見つけたら続いてPQendcopyを呼び出さなければなりません。それから、PQgetResultがヌルポインタを返すまで、PQgetResultのループに戻らなければなりません。同じようにPGRES_COPY_INは連続したPQputlineで処理し、それからPQendcopyで締めくくった後にPQgetResultのループに戻ります。このようにすることで、一連のSQLコマンド群に含めたCOPYコマンドを確実に、また正しく実行できるはずです。

比較的古いアプリケーションでは、COPYをPQexecで実行し、PQendcopyの実行でトランザクションは完了する、と想定していることがよくあります。これはコマンド文字列中のSQLがCOPYだけであった時にのみ正しく動作します。

33.10. 制御関数

これらの関数はlibpqの動作の各種詳細を制御します。

PQclientEncoding

クライアント符号化方式を返します。

```
int PQclientEncoding(const PGconn *conn);
```

これがEUC_JPなどのシンボル文字列ではなく符号化方式IDを返すことに注意してください。成功しなかった場合には、-1が返ります。符号化方式IDを符号化方式名に変換するためには以下を使用してください。

```
char *pg_encoding_to_char(int encoding_id);
```

PQsetClientEncoding

クライアント符号化方式を設定します。

```
int PQsetClientEncoding(PGconn *conn, const char *encoding);
```

connはサーバへの接続、encodingは使用したい符号化方式です。この関数は符号化方式の設定に成功すると、ゼロを返します。さもなくば-1を返します。この接続における現在の符号化方式はPQclientEncodingを使用して決定することができます。

PQsetErrorVerbosity

PQerrorMessageとPQresultErrorMessageで返されるメッセージの冗長度を決定します。

```
typedef enum
{
    PQERRORS_TERSE,
    PQERRORS_DEFAULT,
    PQERRORS_VERBOSE,
    PQERRORS_SQLSTATE
} PGVerbosity;

PGVerbosity PQsetErrorVerbosity(PGconn *conn, PGVerbosity verbosity);
```

PQsetErrorVerbosityは冗長度モードを設定し、接続における以前の状態を返します。*TERSE*モードでは、返されるメッセージには深刻度、主テキスト、位置のみが含まれます。これは通常単一行に収まります。*DEFAULT*モードでは、上に加え、詳細、ヒント、文脈フィールドが含まれるメッセージが生成されます(これは複数行に跨るかもしれません。) *VERBOSE*モードでは、すべての利用可能なフィールドが含まれます。*SQLSTATE*モードでは、エラーの深刻度と、利用可能であればSQLSTATEエラーコードだけが含まれます(利用できなければ、出力は*TERSE*モードのようになります)。

冗長度の変更は、既に存在するPGresultオブジェクト内から取り出せるメッセージには影響を与えません。その後に作成されたオブジェクトにのみ影響を与えます。(ただし、以前のエラーを異なる冗長さで表示したい場合はPQresultVerboseErrorMessageを参照してください。)

PQsetErrorContextVisibility

PQerrorMessageおよびPQresultErrorMessageから返されるメッセージ内のCONTEXTフィールドの扱いについて決定します。

```
typedef enum
{
    PQSHOW_CONTEXT_NEVER,
    PQSHOW_CONTEXT_ERRORS,
    PQSHOW_CONTEXT_ALWAYS
} PGContextVisibility;
```

```
PGContextVisibility PQsetErrorContextVisibility(PGconn *conn, PGContextVisibility
 show_context);
```

PQsetErrorContextVisibilityはコンテキストの表示モードを設定し、その接続での以前の設定を返します。このモードはメッセージにCONTEXTフィールドが含まれるかどうかを制御します。*NEVER*モードでは、決してCONTEXTを含みませんが、*ALWAYS*ではCONTEXTが利用可能であれば常に含まれます。*ERRORS*モード（デフォルト）では、CONTEXTはエラーメッセージには含まれますが、注意や警告では含まれません。（しかしながら、冗長設定が*TERSE*や*SQLSTATE*の場合は、コンテキストの表示モードに関わらずCONTEXTフィールドは省略されます。）

このモードを変更しても、既存のPGresultから取得可能なメッセージには影響を与えず、その後で作成されるものにのみ影響します。（ただし、以前のエラーについて異なる表示モードで表示したい場合は、PQresultVerboseErrorMessageを参照してください。）

PQtrace

クライアント／サーバ間の通信トレースを有効にし、デバッグ用のファイルストリームに書き出します。

```
void PQtrace(PGconn *conn, FILE *stream);
```

注記

Windowsにおいて、libpqライブラリとアプリケーションを異なるフラグでコンパイルすると、この関数呼び出しでFILEポインタの内部表現の違いによりアプリケーションはクラッシュするでしょう。特に、このライブラリを使用するアプリケーションでは、マルチスレッド/シングルスレッド、リリース/デバッグ、静的リンク/動的リンクに関して、ライブラリと同じフラグを使わなければなりません。

PQuntrace

PQtraceによって起動されたトレース処理を無効にします。

```
void PQuntrace(PGconn *conn);
```

33.11. 雑多な関数

よくあることですが、うまく分類できない関数がいくつか存在します。

PQfreemem

libpqが割り当てたメモリを解放します。

```
void PQfreemem(void *ptr);
```

具体的にはPQescapeByteaConn、PQescapeBytea、PQunescapeByteaおよびPQnotifiesによりlibpqが割り当てたメモリを解放します。Microsoft Windowsにおいてfree()ではなく、この関数を使用することが特に重要です。DLLにおけるメモリ割り当てとアプリケーションにおけるその解放が、DLLとアプリケーションとでマルチスレッド/シングルスレッド、リリース用/デバッグ用、静的/動的フラグが同じ場合でのみ動作するためです。Microsoft Windowsプラットフォーム以外では、この関数は標準ライブラリのfree()関数と同じです。

PQconninfoFree

PQconndefaultsもしくはPQconninfoParseが割り当てたデータ構造を解放します。

```
void PQconninfoFree(PQconninfoOption *connOptions);
```

単純なPQfreememは、配列が補助文字列への参照を含んでいることから、このためには作業しません。

PQencryptPasswordConn

PostgreSQLパスワードの暗号化された形式を準備します。

```
char *PQencryptPasswordConn(PGconn *conn, const char *passwd, const char *user, const char
 *algorithm);
```

この関数は、ALTER USER joe PASSWORD 'pwd'のようなコマンドを送信したいクライアントアプリケーションで使用されることを意図したものです。こうしたコマンドでは、コマンドログが活動の監視などで晒されてしまうため、元々の平文テキストでパスワードを送信しないことが推奨されています。その代わりに、この関数を使用して送信前にパスワードを暗号化形式に変換してください。

passwdとuser引数は、関数が使用する平文のパスワードとそのSQL上のユーザ名です。algorithmは、パスワードを暗号化するために使用する暗号化アルゴリズムを指定します。現在サポートされているアルゴリズムは、md5とscram-sha-256です。(古いサーババージョンとの互換性のために、md5の別名として、onとoffも受け付けます。) scram-sha-256のサポートは、PostgreSQLバージョン10で導入されたので、古いサーババージョンでは正しく動作しないことに注意してください。algorithmがNULLなら、この関数はサーバに問合せて現在のpassword_encryption設定を返します。これは、ブロックする可能性があり、また現在のトランザクションがアボートしているか、あるいは他の問合せを実行中でビジーなら失敗します。サーバのデフォルトアルゴリズムを使用したいが、ブロックは避けたい、という場合は、PQencryptPasswordConnを呼び出す前にpassword_encryptionを自分で調べ、その値をalgorithmに渡してください。

戻り値はmallocで割り当てられた文字列です。呼び出し元は、その文字列にエスケープしなければならない特殊な文字列が含まれていないことを仮定することができます。処理が終わった時にPQfreememを使用して結果を解放してください。エラーの場合にNULLが返され、接続オブジェクトに対応するメッセージが格納されます。

PQencryptPassword

md5暗号化形式のPostgreSQLパスワードを準備します。

```
char *PQencryptPassword(const char *passwd, const char *user);
```

PQencryptPasswordは、古くて非推奨のバージョンのPQencryptPasswordConnです。違いは、PQencryptPasswordはコネクションオブジェクトを必要とせず、md5が常に暗号化アルゴリズムに使用されることです。

PQmakeEmptyPGresult

与えられたステータスで空のPGresultオブジェクトを構築します。

```
PGresult *PQmakeEmptyPGresult(PGconn *conn, ExecStatusType status);
```

これは空のPGresultオブジェクトを割り当てて、初期化するlibpqの内部関数です。メモリが割り当てられなかった場合、この関数はNULLを返します。一部のアプリケーションで結果オブジェクト(特にエラーステータスを伴ったオブジェクト)それ自身を生成することが便利であることが分かりましたので、外部公開されました。connが非ヌルで、statusがエラーを示唆している場合、特定された接続の現在のエラーメッセージはPGresultにコピーされます。同時に、connが非ヌルの場合、接続で登録された任意のイベントプロシージャはPGresultにコピーされます。(それらはPGEVT_RESULTCREATE呼び出しを受けませんが、PQfireResultCreateEventsを理解します。) libpq自身で返されたPGresultと同様に、最終的にはこのオブジェクトに対してPQclearを呼び出さなければならないことに注意してください。

PQfireResultCreateEvents

PGresultオブジェクトに登録されたそれぞれのイベントプロシージャに対し、PGEVT_RESULTCREATEイベント(33.13を参照)を発行します。イベントプロシージャが成功の場合は非ゼロ、失敗の場合はゼロを返します。

```
int PQfireResultCreateEvents(PGconn *conn, PGresult *res);
```

conn引数はイベントプロシージャに渡されますが、直接には使用されません。イベントプロシージャが使用しない場合はNULLで構いません。

このオブジェクトに対し、PGEVT_RESULTCREATEもしくはPGEVT_RESULTCOPYイベントを過去に受け取ったイベントプロシージャは再び発行されません。

この関数がPQmakeEmptyPGresultと分離されている主たる理由は、多くの場合イベントプロシージャを呼び出す前にPGresultを作成し、データを挿入するのが適切であることによります。

PQcopyResult

PGresultオブジェクトのコピーを作ります。コピーは元の結果にいかなる方法でもリンクされず、コピーが不要になった時にPQclearを呼び出されなければなりません。関数が失敗するとNULLが返されます。

```
PGresult *PQcopyResult(const PGresult *src, int flags);
```

これは正確なコピーの作成を目的としたものではありません。返された結果は常にPGRES_TUPLES_OK状態の中に置かれ、元の結果におけるエラーメッセージはまったくコピーされません。(しかしコマンド状態文字列をコピーします。) flags引数はその他にコピーするものがないかを決定します。それはいくつかのフラグのビット単位のORです。PG_COPYRES_ATTRSは元の結果の属性(列定義)のコピーを指定します。PG_COPYRES_TUPLESは元の結果のタプルのコピーを指定します。(これは属性もコピーされることを意味しています。) PG_COPYRES_NOTICEHOOKSは元の結果の警告フックのコピーを指定します。

PG_COPYRES_EVENTSは元の結果イベントのコピーを指定します。(しかし、元の結果に関連したインスタンスデータはまったくコピーされません。)

PQsetResultAttrs

PGresultオブジェクトの属性を設定します。

```
int PQsetResultAttrs(PGresult *res, int numAttributes, PGresAttDesc *attDescs);
```

提供されたattDescsは結果にコピーされます。もしattDescsポインタがNULL、またはnumAttributesが1未満の場合、要求は無視され、関数は成功します。resが既に属性を所有している場合、関数は失敗に終わります。関数が失敗すると、戻り値はゼロです。関数が成功すると戻り値は非ゼロになります。

PQsetvalue

PGresultオブジェクトのタプルフィールド値を設定します。

```
int PQsetvalue(PGresult *res, int tup_num, int field_num, char *value, int len);
```

必要に応じて関数は自動的に結果の内部タプル配列を肥大化させます。しかし、tup_num引数はPQntuplesと同じか、もしくは小さくなければなりません。その意味は、この関数は一回にタプル配列を1タプル大きくさせるだけだからです。とは言っても、存在するいかなるタプルの任意のフィールドも、順序を問わず変更できます。もしfield_numに値が既に存在すれば、書き換えられます。lenが-1、またはvalueがNULLであれば、フィールドの値はSQLのNULLに設定されます。valueは結果のプライベート格納領域にコピーされるため、関数が返った後ではもう必要がなくなります。関数が失敗すると、戻り値はゼロです。関数が成功すると戻り値は非ゼロになります。

PQresultAlloc

PGresultオブジェクトに補助ストレージを割り当てます。

```
void *PQresultAlloc(PGresult *res, size_t nBytes);
```

resが消去された時、この関数で割り付けられたメモリは解放されます。関数が失敗すると戻り値はNULLです。mallocと同じように、どのような種類のデータでも結果は適切に整列されることが保証されています。

PQresultMemorySize

PGresultオブジェクトのために割り当てられたバイト数を取り出します。

```
size_t PQresultMemorySize(const PGresult *res);
```

この値はPGresultオブジェクトに関連するmalloc要求すべての和、すなわちPQclearで解放される空間全体です。この情報はメモリ消費を管理するのに有用でしょう。

PQlibVersion

使用中のlibpqのバージョンを返します。

```
int PQlibVersion(void);
```

この関数の結果を使用して、現在読み込まれているバージョンのlibpqで特定の機能が利用可能かどうかを実行時に決定することができます。例えばこの関数を使用して、PQconnectdbでどの接続オプションが利用できるかを確認することができます。

返却値の形式は、メジャーバージョン番号に10000を掛け、マイナーバージョン番号を加えたものです。例えば、バージョン10.1では100001を返し、バージョン11.0では110000を返します。

バージョン10よりも前では、PostgreSQLでは、最初の2つの部分がメジャーバージョンを表す、3つの部分からなるバージョン番号が使われていました。これらのバージョンでは、PQlibVersionはそれぞれの部分に2桁の数字を使います。たとえば、バージョン9.1.5では90105が返され、バージョン9.2.0では90200が返されます。

ですから、機能の互換性を見極めるのが目的なら、アプリケーションはPQlibVersionの結果を10000ではなく、100で割り、論理的なメジャーバージョンを求めるべきです。すべてのリリースで、最後の2桁だけがマイナーリリースで異なります。(バグ修正リリースです。)

> **注記**
>
> この関数はPostgreSQLバージョン9.1で追加されました。このため以前のバージョンにおいて要求される機能を検知するために使用することができません。この関数の呼び出しがバージョン9.1以降とのリンク依存性を作成するためです。

33.12. 警告処理

問い合わせ実行関数では、サーバにより生成された通知と警告メッセージは、問い合わせの失敗を意味していないので返されません。その代わり、それらは通知処理関数に渡され、ハンドラから返った後も実行は通常通り継続します。デフォルトの通知処理関数はstderrにメッセージを出力しますが、アプリケーションは自身の処理関数を提供することでこの動作を書き換えることができます。

歴史的理由で、通知レシーバと通知プロセッサと呼ばれる2階層の通知処理が存在します。デフォルトの動作は、通知レシーバが通知を書式化し、出力のため通知プロセッサに文字列を渡します。しかし、独自の通知レシーバを提供することを選んだアプリケーションでは、通常、通知プロセッサ層を無視し、すべての作業を単に通知レシーバで行います。

関数PQsetNoticeReceiverは接続オブジェクトに対し現在の通知レシーバを設定もしくは確認します。同様に、PQsetNoticeProcessorは現在の通知プロセッサの設定もしくは確認を行います。

```
typedef void (*PQnoticeReceiver) (void *arg, const PGresult *res);

PQnoticeReceiver
PQsetNoticeReceiver(PGconn *conn,
                    PQnoticeReceiver proc,
                    void *arg);

typedef void (*PQnoticeProcessor) (void *arg, const char *message);
```

```
PQnoticeProcessor
PQsetNoticeProcessor(PGconn *conn,
                     PQnoticeProcessor proc,
                     void *arg);
```

各関数は、以前の通知レシーバもしくは通知プロセッサ用の関数へのポインタを返し、新しい値を設定します。関数ポインタにヌルを渡した場合、何も変更されず、現在のポインタが返されるだけです。

サーバから注意/警告メッセージを受け取ると、あるいは、libpq内部で注意/警告メッセージが生成されると、通知レシーバ関数が呼び出されます。PGRES_NONFATAL_ERROR PGresultという形でメッセージが渡されます。(これにより、レシーバはPQresultErrorFieldを使用して個々のフィールドを取り出すことや、PQresultErrorMessageあるいはPQresultVerboseErrorMessageを使用して事前に整形された完全なメッセージを取得することができます。) PQsetNoticeReceiverに渡されたvoidポインタと同じものも渡されます。(このポインタを使用して、必要に応じてアプリケーション特有の状態にアクセスすることができます。)

デフォルトの通知レシーバは単に(PQresultErrorMessageを使用して)メッセージを取り出し、それを通知プロセッサに渡すだけです。

通知プロセッサは、テキスト形式で与えられた注意/警告メッセージの取扱いに責任を持ちます。メッセージは(最後の改行を含む)文字列テキストで渡され、更に、PQsetNoticeProcessorに渡したものと同じvoidポインタも渡されます。(このポインタを使用して、必要に応じてアプリケーション特有の状態にアクセスすることができます。)

デフォルトの通知プロセッサは以下のような単純なものです。

```
static void
defaultNoticeProcessor(void *arg, const char *message)
{
    fprintf(stderr, "%s", message);
}
```

一旦通知レシーバや通知プロセッサを設定したら、PGconnオブジェクトか、それから生成されたPGresultオブジェクトが存在している間は、その関数が呼び出される可能性があると考えておくべきです。PGresultの生成時には、PGconnの現在の警告処理用のポインタが、PQgetvalueのような関数で使用可能であるように、PGresultへコピーされます。

33.13. イベントシステム

libpqのイベントシステムは、PGconnおよびPGresultオブジェクトの作成と削除のような関心を引くlibpqイベントについて登録されたイベントハンドラに通知を行うため設計されています。主たる使用状況は、アプリケーションがそれ自身のデータをPGconnまたはPGresultと提携させ、データが適切な時間に解放されることを保証するものです。

それぞれの登録されたイベントハンドラは、libpqからは曖昧としたvoid *ポインタとしてだけ知られる2つのデータの断片と提携します。イベントハンドラがPGconnで登録された時にアプリケーションが提供する通過地点ポインタがあります。通過地点ポインタはPGconnやそれから生成されたすべての(複数の)PGresultが有効な間決して変わることはありません。したがって使用された場合、長

期間生存しているデータを指し示します。さらに、インスタンスデータポインタがあって、それはすべてのPGconnとPGresultでNULLから開始します。ポインタは、PQinstanceData、PQsetInstanceData、PQresultInstanceDataおよびPQsetResultInstanceData関数を使って操作することができます。通過地点ポインタとは異なり、PGconnのインスタンスデータはそれから作成されたPGresultにより自動的に継承されません。libpqは通過地点とインスタンスデータポインタが（もしあったとしても）何を指し示すのか判らず、決して解放しようとは試みません。それはイベントハンドラの責任です。

33.13.1. イベントの種類

PGEventId列挙はイベントシステムにより処理されるイベントの種類に名前をつけます。その値はすべてPGEVTで始まる名前を持っています。それぞれのイベントの種類に対し、イベントハンドラに渡されるパラメータを運ぶ関連したイベント情報構造体があります。イベントの種類を以下に示します。

PGEVT_REGISTER

登録イベントはPQregisterEventProcが呼ばれたとき発生します。イベントプロシージャが必要とするかもしれない任意のinstanceDataを初期化するために、これは理想的な時間です。接続毎、イベントハンドラ毎でたった1つの登録イベントが発行されます。イベントプロシージャが失敗すると、登録は中止されます。

```
typedef struct
{
    PGconn *conn;
} PGEventRegister;
```

PGEVT_REGISTERイベントが受け取られると、evtInfoポインタはPGEventRegister *にキャストされなければなりません。この構造体はCONNECTION_OK状態ではなくてはならないPGconnを含んでいます。そしてそれは、効果のあるPGconnを取得した直後、PQregisterEventProcを呼び出せば、保証されます。失敗コードを返すとき、PGEVT_CONNDESTROYイベントが送られないので、すべての消去が実行されなければなりません。

PGEVT_CONNRESET

接続初期化イベントはPQresetまたはPQresetPollの完了時点で発行されます。どちらの場合も、初期化が成功したときのみ発行されます。イベントプロシージャが失敗すると、接続初期化全体が失敗します。PGconnはCONNECTION_BAD状態になり、PQresetPollはPGRES_POLLING_FAILEDを返します。

```
typedef struct
{
    PGconn *conn;
} PGEventConnReset;
```

PGEVT_CONNRESETイベントが受け取られた時、evtInfoポインタはPGEventConnReset *にキャストされなければなりません。含まれたPGconnは単に初期化されますが、すべてのイベントデータは変更されずに残ります。このイベントはすべての関連したinstanceDataの初期化・再読み込み・再問い合わせに使用されなければなりません。イベントプロシージャがPGEVT_CONNRESET処理に失敗したとしても、接続が閉じられた時PGEVT_CONNDESTROYイベントを依然として受け付けることに注意してください。

PGEVT_CONNDESTROY

接続破棄イベントはPQfinishに対応して発行されます。libpqはこのメモリを管理する機能がありませんので、そのイベントデータを的確に消去するのはイベントプロシージャの責任です。消去の失敗はメモリーリークに通じます。

```
typedef struct
{
    PGconn *conn;
} PGEventConnDestroy;
```

PGEVT_CONNDESTROYイベントが受け取られた時、evtInfoポインタはPGEventConnDestroy *にキャストされなければなりません。このイベントはPQfinishが他のすべての消去を行う前に発行されます。イベントプロシージャの戻り値は、PQfinishから失敗を示唆する方法がないので無視されます。同時に、イベントプロシージャの失敗が不要なメモリ消去処理を中止してはなりません。

PGEVT_RESULTCREATE

結果作成イベントは、PQgetResultを含み、結果を生成する任意の問い合わせ実行関数に対応して発行されます。このイベントは結果が成功裏に作成されたときのみ発行されます。

```
typedef struct
{
    PGconn *conn;
    PGresult *result;
} PGEventResultCreate;
```

PGEVT_RESULTCREATEイベントが受け取られた時、evtInfoポインタはPGEventResultCreate *にキャストされなければなりません。connは結果を生成するために使われた接続です。これは、結果と関連しなければならないすべてのinstanceDataを初期化するために、理想的な場所です。イベントプロシージャが失敗すると、結果は消去され、失敗が伝播します。イベントプロシージャはそれ自身の結果オブジェクトをPQclearしようと試みてはいけません。失敗コードを返す時、PGEVT_RESULTDESTROYイベントは送られないのですべての消去が行われなくてはなりません。

PGEVT_RESULTCOPY

結果コピーイベントはPQcopyResultの応答として発行されます。このイベントはコピーが完了した後にのみ発行されます。元の結果に対するPGEVT_RESULTCREATEもしくはPGEVT_RESULTCOPYイベントを成功裏に処理したイベントプロシージャのみ、PGEVT_RESULTCOPYイベントを受け取ります。

```
typedef struct
{
    const PGresult *src;
    PGresult *dest;
} PGEventResultCopy;
```

PGEVT_RESULTCOPYイベントが受け取られた時、evtInfoポインタはPGEventResultCopy *にキャストされなければなりません。src結果はコピーされるものであり、一方でdest結果はコピー先です。このイベ

ントはinstanceDataのディープコピーを提供するために使用されます。PQcopyResultではこれを行うことができないためです。もしイベントプロシージャが失敗すると、コピー操作全体は失敗になり、dest結果は消去されます。失敗コードを返す時、PGEVT_RESULTDESTROYイベントがコピー先の結果に対し送られないため、すべての消去を行われなければなりません。

PGEVT_RESULTDESTROY

結果破棄イベントはPQclearに対応して発行されます。libpqはこのメモリを管理する機能がありませんので、そのイベントデータを的確に消去するのはイベントプロシージャの責任です。消去の失敗はメモリーリークに通じます。

```
typedef struct
{
    PGresult *result;
} PGEventResultDestroy;
```

PGEVT_RESULTDESTROYが受け取られた時、evtInfoポインタはPGEventResultDestroy *にキャストされなければなりません。このイベントはPQclearがその他の消去を行う以前に起動されなければなりません。イベントプロシージャの戻り値は、PQclearから失敗を示唆する方法がないので無視されます。同時に、イベントプロシージャの失敗が不要なメモリ消去処理を中止してはなりません。

33.13.2. イベントコールバックプロシージャ

PGEventProc

PGEventProcはイベントプロシージャへのポインタに対するtypedefです。つまり、libpqからイベントを受け取るユーザコールバック関数です。イベントプロシージャのシグネチャは以下でなければなりません。

```
int eventproc(PGEventId evtId, void *evtInfo, void *passThrough)
```

evtIdパラメータはどのPGEVTイベントが発生したかを示します。evtInfoポインタは、イベントに対する追加情報を入手するため適切な構造体型にキャストされなければなりません。passThroughパラメータは、イベントプロシージャが登録された時、PQregisterEventProcに提供されるポインタです。関数は成功した場合非ゼロを、失敗した場合ゼロを返さなければなりません。

特定のイベントプロシージャは任意のPGconnにおいて一回だけ登録できます。これは、プロシージャのアドレスが関連するインスタンスデータを特定する検索キーとして用いられるからです。

> **注意**
>
> Windowsにおいて、関数は2つの異なるアドレスを持つことができます。外部から可視のDLLと内部から可視のDLLです。libpqのイベントプロシージャ関数ではこれらのアドレスのうちの1つだけが使用されることに注意してください。さもないと、混乱が起きます。正常に機能するコードを書く最も単純な規則は、イベントプロシージャがstaticとして宣言されることを確実にすることです。もし、プロシージャのアドレスがそれ自身のファイルの外部から有効とならなければならない場合、アドレスを返すため別の関数を公開します。

33.13.3. イベントサポート関数

PQregisterEventProc

libpqでイベントコールバックプロシージャを登録します。

```
int PQregisterEventProc(PGconn *conn, PGEventProc proc,
                        const char *name, void *passThrough);
```

そのイベントを取得したいそれぞれのPGconnで1回イベントプロシージャは登録されなければなりません。一つの接続に登録できるイベントプロシージャの数には、メモリ以外の制限はありません。関数は成功した場合非ゼロ、失敗の場合ゼロを返します。

libpqイベントが発行されたときproc引数が呼ばれます。そのメモリアドレスはinstanceDataを検索するのにも使用されます。name引数はエラーメッセージ内でイベントプロシージャを参照するために使用されます。この値はNULLもしくは空文字列であってはなりません。このname文字列はPGconnにコピーされますので、渡されたものは長寿命である必要がありません。passThroughポインタはイベントが発生した時はいつでもprocに渡されます。この引数はNULLであっても構いません。

PQsetInstanceData

procプロシージャに対するconn接続のinstanceDataをdataに設定します。成功の場合非ゼロ、失敗の場合ゼロが返ります。(connでprocが正しく登録されていない場合のみ失敗する可能性があります。)

```
int PQsetInstanceData(PGconn *conn, PGEventProc proc, void *data);
```

PQinstanceData

procプロシージャに関連したconn接続のinstanceData、または存在しなければNULLを返します。

```
void *PQinstanceData(const PGconn *conn, PGEventProc proc);
```

PQresultSetInstanceData

procに対する結果のinstanceDataをdataに設定します。成功の場合非ゼロ、失敗の場合ゼロが返ります。(結果でproc正しく登録されていない場合のみ失敗する可能性があります。)

```
int PQresultSetInstanceData(PGresult *res, PGEventProc proc, void *data);
```

dataで示された領域は、PQresultAllocを使って割り当てたのでない限り、PQresultMemorySizeでは考慮されないことに注意してください。(結果を破棄する時に、領域を明示的に解放する必要がなくなりますので、PQresultAllocを使って割り当てるのがお勧めです。)

PQresultInstanceData

procに関連した結果のinstanceData、または存在しなければNULLを返します。

```
void *PQresultInstanceData(const PGresult *res, PGEventProc proc);
```

33.13.4. イベント事例

以下にlibpq接続と結果に関連したプライベートデータを管理する例の大枠を示します。

```
/* libpqイベントに必要なヘッダ（覚書：libpq-fe.hのインクルード）*/
#include <libpq-events.h>


/* instanceData */
typedef struct
{
    int n;
    char *str;
} mydata;

/* PGEventProc */
static int myEventProc(PGEventId evtId, void *evtInfo, void *passThrough);

int
main(void)
{
    mydata *data;
    PGresult *res;
    PGconn *conn =
        PQconnectdb("dbname=postgres options=-csearch_path=");

    if (PQstatus(conn) != CONNECTION_OK)
    {
        fprintf(stderr, "Connection to database failed: %s",
                PQerrorMessage(conn));
        PQfinish(conn);
        return 1;
    }


    /* イベントを受け取るべきすべての接続で１回呼ばれる
     * myEventProcにPGEVT_REGISTERを送る
     */
    if (!PQregisterEventProc(conn, myEventProc, "mydata_proc", NULL))
    {
        fprintf(stderr, "Cannot register PGEventProc\n");
        PQfinish(conn);
        return 1;
    }
```

```
    /* conn instanceDataが有効 */
    data = PQinstanceData(conn, myEventProc);


    /* myEventProcにPGEVT_RESULTCREATEを送る */
    res = PQexec(conn, "SELECT 1 + 1");


    /* 結果 instanceDataが有効 */
    data = PQresultInstanceData(res, myEventProc);


    /* PG_COPYRES_EVENTSが使われた場合、
      PGEVT_RESULTCOPYをmyEventProcに送る */
    res_copy = PQcopyResult(res, PG_COPYRES_TUPLES | PG_COPYRES_EVENTS);


    /* PQcopyResult呼び出しの過程でPG_COPYRES_EVENTSが使用された場合、

     * 結果 instanceDataが有効
     */
    data = PQresultInstanceData(res_copy, myEventProc);


    /* 双方のclearがPGEVT_RESULTDESTROYをmyEventProcに送る */
    PQclear(res);
    PQclear(res_copy);


    /* PGEVT_CONNDESTROYをmyEventProcに送る */
    PQfinish(conn);

    return 0;
}

static int
myEventProc(PGEventId evtId, void *evtInfo, void *passThrough)
{
    switch (evtId)
    {
        case PGEVT_REGISTER:
        {
            PGEventRegister *e = (PGEventRegister *)evtInfo;
            mydata *data = get_mydata(e->conn);
```

```
        /* アプリ特有のデータを接続に関連付ける */
        PQsetInstanceData(e->conn, myEventProc, data);
        break;
    }

    case PGEVT_CONNRESET:
    {
        PGEventConnReset *e = (PGEventConnReset *)evtInfo;
        mydata *data = PQinstanceData(e->conn, myEventProc);

        if (data)
          memset(data, 0, sizeof(mydata));
        break;
    }

    case PGEVT_CONNDESTROY:
    {
        PGEventConnDestroy *e = (PGEventConnDestroy *)evtInfo;
        mydata *data = PQinstanceData(e->conn, myEventProc);


        /* connが破棄されたのでインスタンスデータを解放 */
        if (data)
          free_mydata(data);
        break;
    }

    case PGEVT_RESULTCREATE:
    {
        PGEventResultCreate *e = (PGEventResultCreate *)evtInfo;
        mydata *conn_data = PQinstanceData(e->conn, myEventProc);
        mydata *res_data = dup_mydata(conn_data);


        /* アプリ特有のデータを結果と（connから複写して）関連付ける */
        PQsetResultInstanceData(e->result, myEventProc, res_data);
        break;
    }

    case PGEVT_RESULTCOPY:
    {
        PGEventResultCopy *e = (PGEventResultCopy *)evtInfo;
        mydata *src_data = PQresultInstanceData(e->src, myEventProc);
        mydata *dest_data = dup_mydata(src_data);
```

```
            /* アプリ特有のデータを結果と（結果から複写して）関連付ける */
            PQsetResultInstanceData(e->dest, myEventProc, dest_data);
            break;
        }

        case PGEVT_RESULTDESTROY:
        {
            PGEventResultDestroy *e = (PGEventResultDestroy *)evtInfo;
            mydata *data = PQresultInstanceData(e->result, myEventProc);


            /* 結果が破棄されたためインスタンスデータを解放 */
            if (data)
              free_mydata(data);
            break;
        }


        /* 未知のイベント識別子。単にtrueを返す */
        default:
            break;
    }


    return true; /* イベント処理成功 */
}
```

33.14. 環境変数

以下の環境変数を使用して、呼び出し側のプログラムで直接値を指定しなかった場合の接続パラメータのデフォルト値を選ぶことができます。この値は、PQconnectdb、PQsetdbLoginおよびPQsetdbで使用されます。例えば、簡単なクライアントアプリケーションでは、データベース接続情報を直接プログラムに記述しない方が便利です。

- PGHOSTはhost接続パラメータと同様に動作します。
- PGHOSTADDRはhostaddr接続パラメータと同様に動作します。PGHOSTの代わりに設定して、または、PGHOSTに追加して、DNS検索に要するオーバヘッドをなくすことができます。
- PGPORTはport接続パラメータと同様に動作します。
- PGDATABASEはdbname接続パラメータと同様に動作します。
- PGUSERはuser接続パラメータと同様に動作します。
- PGPASSWORDはpassword接続パラメータと同様に動作します。この環境変数は、一部のオペレーティングシステムではroot以外のユーザがpsコマンド経由で環境変数を見ることができるなど、セキュリティ上の

理由から現在では推奨されていません。代わりにパスワードファイル(33.15を参照してください)を使用することを検討してください。

- PGPASSFILEはpassfile接続パラメータと同様に動作します。
- PGCHANNELBINDINGはchannel_binding接続パラメータと同様に動作します。
- PGSERVICEはservice接続パラメータと同様に動作します。
- PGSERVICEFILEはユーザごとの接続サービスファイルを指定します。設定されていない場合、デフォルトは~/.pg_service.conf(33.16参照)となります。
- PGOPTIONSはoptions接続パラメータと同様に動作します。
- PGAPPNAMEはapplication_name接続パラメータと同様に動作します。
- PGSSLMODEはsslmode接続パラメータと同様に動作します。
- PGREQUIRESSLはrequiressl接続パラメータと同様に動作します。この環境変数はPGSSLMODE変数があるため、廃止予定となっています。両方の変数を設定すると、PGREQUIRESSLの設定は無視されます。
- PGSSLCOMPRESSIONはsslcompression接続パラメータと同様に動作します。
- PGSSLCERTはsslcert接続パラメータと同様に動作します。
- PGSSLKEYはsslkey接続パラメータと同様に動作します。
- PGSSLROOTCERTはsslrootcert接続パラメータと同様に動作します。
- PGSSLCRLはsslcrl接続パラメータと同様に動作します。
- PGREQUIREPEERはrequirepeer接続パラメータと同様に動作します。
- PGSSLMINPROTOCOLVERSIONはssl_min_protocol_version接続パラメータと同様に動作します。
- PGSSLMAXPROTOCOLVERSIONはssl_min_protocol_version接続パラメータと同様に動作します。
- PGGSSENCMODEはgssencmode接続パラメータと同様に動作します。
- PGKRBSRVNAMEはkrbsrvname接続パラメータと同様に動作します。
- PGGSSLIBはgsslib接続パラメータと同様に動作します。
- PGCONNECT_TIMEOUTはconnect_timeout接続パラメータと同様に動作します。
- PGCLIENTENCODINGはclient_encoding接続パラメータと同様に動作します。
- PGTARGETSESSIONATTRSはtarget_session_attrs接続パラメータと同様に動作します。

以下の環境変数を使用して、PostgreSQLセッション毎のデフォルト動作を指定することができます。(また、ユーザ毎、もしくは、データベース毎を単位としたデフォルト動作の設定方法についてはALTER ROLEおよびALTER DATABASEコマンドを参照してください。)

- PGDATESTYLEはデフォルトの日付/時刻表現形式を設定します。(SET datestyle TO ...と等価です。)

- PGTZはデフォルトの時間帯を設定します。(SET timezone TO ...と等価です。)
- PGGEQOは遺伝的問い合わせオプティマイザのデフォルトのモードを設定します。(SET geqo TO ...と等価です。)

これらの環境変数の正確な値については、SET SQLコマンドを参照してください。

以下の環境変数は、libpqの内部動作を決定します。これらはコンパイル時のデフォルトを上書きします。

- PGSYSCONFDIRはpg_service.confファイルがあるディレクトリを設定します。また今後のバージョンでは他のシステム全体の設定ファイルとなるかもしれません。
- PGLOCALEDIRはメッセージのローカライズ用のlocaleファイルがあるディレクトリを設定します。

33.15. パスワードファイル

ユーザのホームディレクトリの.pgpassは、接続にパスワードが必要な場合(かつ、他に指定されたパスワードが無かった場合)に使用するパスワードを格納するファイルです。Microsoft Windowsでは、このファイルの名前は%APPDATA%\postgresql\pgpass.conf(ここで%APPDATA%はユーザのプロファイル内のアプリケーションデータディレクトリ)です。他に、接続パラメータpassfileを利用するか、環境変数PGPASSFILEで、パスワードファイルを指定できます。

このファイル内の行の書式は次の通りです。

```
hostname:port:database:username:password
```

(このファイルでは、上のような行をコピーし、その先頭に#をつけて忘れないようにコメントとして残すことができます。) 先頭の4フィールドはそれぞれリテラル値にすることも、あるいはすべてに一致する*を使用することもできます。最初に現在の接続パラメータと一致した行のパスワードフィールドが使用されます。(従って、ワイルドカードを使用する場合は、始めの方により具体的な項目を入力してください。) 項目内に:または\を含める必要があれば、\でこれらの文字をエスケープする必要があります。ホスト名フィールドは、host接続パラメータか、もし指定されていれば、hostaddrパラメータと一致します。どちらも指定されていなければ、ホスト名localhostが検索されます。接続がUnixドメインソケット接続で、hostパラメータがlibpqのデフォルトソケットディレクトリパスに一致した場合も、ホスト名localhostが検索されます。スタンバイサーバでは、replicationという名称のデータベースは、マスタサーバとの間でなされるストリーミングレプリケーション用の接続に一致します。同一のクラスタ内のすべてのデータベースに対するパスワードは同じものですので、データベースフィールドの有用性は限定的なものです。

Unixシステムにおいて、パスワードファイルの権限はグループ、他者へのアクセスをすべて拒否しなければなりません。これはchmod 0600 ~/.pgpassといったコマンドによって行います。権限をこれよりも緩くすると、このファイルは無視されます。Microsoft Windowsにおいては、このファイルが安全なディレクトリに格納されていることを前提としていますので、特別に行われる権限の検査はありません。

33.16. 接続サービスファイル

接続サービスファイルにより、libpq接続パラメータをひとつのサービス名に関連付けることができます。サービス名は、libpq接続によって指定され、関連付けられた設定が利用されます。これは、接続パラメータを

libpqアプリケーションの再コンパイルをせずに修正できるというものです。サービス名はPGSERVICE環境変数を利用することで使用できます。

この接続サービスファイルは、ユーザごとに~/.pg_service.confというサービスファイルとすること、または、PGSERVICEFILE環境変数で指定される場所にすることができます。また、システム全体についてのファイルとして`pg_config --sysconfdir`/pg_service.confとすること、PGSYSCONFDIR環境変数で指定されたディレクトリに置くことができます。ユーザ用、システム用のファイルで同名のサービス定義が存在する場合、ユーザ用のものが優先されます。

このファイルは「INIファイル」書式を使用します。セクション名がサービス名となり、パラメータが接続パラメータです。33.1.2のリストを参照してください。以下に例を示します。

```
# comment
[mydb]
host=somehost
port=5433
user=admin
```

例となるファイルがshare/pg_service.conf.sampleにあります。

33.17. 接続パラメータのLDAP検索

libpqがLDAPサポート(configure時の--with-ldapオプション)付きでコンパイルされている場合、中央サーバからLDAPを通してhostやdbnameなどの接続オプションを取り出すことができます。この利点は、データベースの接続パラメータが変わった場合に、すべてのクライアントマシンで接続情報を更新しなくても済む点です。

LDAP接続パラメータ検索は、pg_service.confという接続サービスファイル(33.16を参照)を使用します。pg_service.conf内のldap://から始まる行は、LDAP URLとして認識され、LDAP問い合わせが実行されることを示します。その結果は、keyword = valueという組み合わせのリストでなければなりません。これらが接続用オプションの設定に使用されます。このURLはRFC 1959に従ったもので、以下のような形式でなければなりません。

```
ldap://[hostname[:port]]/search_base?attribute?search_scope?filter
```

ここで、hostnameのデフォルトはlocalhost、portのデフォルトは389です。

pg_service.confの処理はLDAP検索が成功した時に終わります。しかし、もしLDAPサーバへのアクセスができなかった場合は継続します。これはアクセスに失敗した時に、異なるLDAPサーバを指し示す他のLDAP行や以前からのkeyword = valueの組み合わせ、デフォルトの接続オプションを参照する予備機能を提供します。この場合にエラーメッセージを受け取りたい場合は、LDAP URL行の後に文法的に不正な行を記載してください。

LDIFファイルとして作成されたLDAP項目の例を以下に示します。

```
version:1
dn:cn=mydatabase,dc=mycompany,dc=com
```

```
changetype:add
objectclass:top
objectclass:device
cn:mydatabase
description:host=dbserver.mycompany.com
description:port=5439
description:dbname=mydb
description:user=mydb_user
description:sslmode=require
```

これは、以下のようなLDAP URLから得られます。

```
ldap://ldap.mycompany.com/dc=mycompany,dc=com?description?one?(cn=mydatabase)
```

また、LDAP検索と通常のサービスファイル項目とを混在させることもできます。pg_service.confの一節について完全な例を以下に示します。

```
# only host and port are stored in LDAP, specify dbname and user explicitly
[customerdb]
dbname=customer
user=appuser
ldap://ldap.acme.com/cn=dbserver,cn=hosts?pgconnectinfo?base?(objectclass=*)
```

33.18. SSLサポート

PostgreSQLは、セキュリティを高めるためにクライアントサーバ間の通信を暗号化するSSL接続の使用を元来サポートしています。サーバ側のSSL機能についての詳細は18.9を参照してください。

libpqはシステム全体に対するOpenSSL設定ファイルを読み込みます。デフォルトでは、ファイル名はopenssl.cnfで、openssl version -dで報告されるディレクトリに格納されています。このデフォルトはOPENSSL_CONF環境変数に希望する設定ファイル名を設定することで変更することができます。

33.18.1. サーバ証明書のクライアント検証

デフォルトではPostgreSQLはサーバ証明書の検証をまったく行いません。これは、(例えば、DNSレコードを変更したり、もしくはサーバのIPアドレスを乗っ取ったりして)クライアントに知られずにサーバの身元をなりすませることを意味します。なりすましを防止するには、クライアントは、トラストチェーン(chain of trust)を通じて、サーバの身元を検証できなければなりません。トラストチェーンは、ルート(自己署名)認証局(CA)証明書をあるコンピュータに設置し、そのルート証明書によって署名されたリーフ証明書を他のコンピュータに設置することによって確立されます。また、ルート証明書によって署名された「中間」証明書を使って、リーフ証明書に署名することによっても可能です。

クライアントがサーバの身元を検証するためには、ルート証明書をクライアントに設置し、そのルート証明書によって署名されたリーフ証明書をサーバに設置します。サーバがクライアントの身元を検証するためには、ルート証明書をサーバに設置し、そのルート証明書によって署名されたリーフ証明書をクライアントに設置し

ます。一つ以上の中間証明書（通常リーフ証明書とともに格納されます）を使って、リーフ証明書をルート証明書につなげることもできます。

トラストチェーンがひとたび確立されれば、クライアントがサーバから送信されたリーフ証明書を検証する二つの方法があります。パラメータsslmodeがverify-caに設定されている場合、libpqはクライアントに格納されたルート証明書までの証明書連鎖を検査することで、サーバが信用に足るかを検証します。sslmodeがverify-fullに設定されていると、libpqは同時にサーバホスト名が証明書のそれと一致するかを検証します。SSL接続はサーバ証明書が検証されない場合失敗します。安全性に慎重を期するほとんどのサーバ環境ではverify-fullを推奨します。

verify-fullモードでは、ホスト名を証明書のサブジェクト別名(Subject Alternative Name)属性と、あるいはdNSNameタイプのサブジェクト別名がないときはコモンネーム属性とマッチさせます。証明書の名前属性がアスタリスク（*）で始まると、それはワイルドカードとして取り扱われ、ドット（.）を除くすべての文字とマッチします。これは、証明書がサブドメインとマッチしないことを意味します。もし接続がホスト名ではなくIPアドレスを使用するのであれば、（いかなるDNS検索もせず）IPアドレスがマッチさせられます。

サーバ証明書の検証を可能にするには、1つ以上のルート証明書を、ユーザのホームディレクトリの~/.postgresql/root.crtファイルに置かなければなりません。（Microsoft Windowsの場合、このファイルの名前は%APPDATA%\postgresql\root.crtです。）サーバより送信された証明書連鎖から、クライアントに格納されたルート証明書にリンクするために（中間証明書が）必要なら、中間証明書もそのファイルに追加する必要があります。

~/.postgresql/root.crlファイル（Microsoft Windowsでは%APPDATA%\postgresql\root.crl）が存在する場合、証明書失効リスト（CRL）の項目もまた検査されます。

ルート証明書ファイルとCRLの格納場所を接続パラメータsslrootcertとsslcrl、もしくは環境変数PGSSLROOTCERTとPGSSLCRLで変更することができます。

> **注記**
>
> より古いバージョンのPostgreSQLとの後方互換性のために、ルートCAファイルが存在する場合、sslmode=requireの動作はverify-caの場合と同じになっています。つまり、サーバ証明書がCAに対して検証されます。この動作に依存することは勧めません。また証明書の検証を必要とするアプリケーションは常にverify-caまたはverify-fullを使用すべきです。

33.18.2. クライアント証明書

サーバが、クライアントのリーフ証明書を要求することによってクライアントの身元を検証しようとする場合、libpqはユーザのホームディレクトリにある~/.postgresql/postgresql.crtファイルに格納された証明書を送信します。証明書は、サーバが信頼するルート証明書につながらなければなりません。対応する~/.postgresql/postgresql.key秘密キーファイルも存在しなければなりません。秘密キーファイルは他者やグループからのアクセスを許可してはいけません。chmod 0600 ~/.postgresql/postgresql.keyコマンドでこれを実現してください。Microsoft Windowsでは、このファイルの名前はそれぞれ%APPDATA%\postgresql\postgresql.crtと%APPDATA%\postgresql\postgresql.keyであり、このディレクトリは安全であると想定されますので、特別な権限検査は行われません。証明書とキーファイルの格納場所はsslcertおよびsslkey接続パラメータ、またはPGSSLCERTおよびPGSSLKEY環境変数で上書きされます。

postgresql.crt中の最初の証明書は、クライアント証明書でなければなりません。クライアントの秘密鍵と一致していなければならないからです。オプションで、ファイルに「中間」証明書を追加することができます。そうすることによって、サーバ上に中間証明書（ssl_ca_file）の格納が不要になります。

証明書とキーはPEMまたはASN.1 DER形式です。

キーは平文テキストで、あるいは、OpenSSLで対応しているAES-128など任意のアルゴリズムを使ってパスフレーズで暗号化して、格納できます。キーが暗号化されて格納された場合、パスフレーズはsslpassword接続オプションで供給してもよいです。暗号化されたキーが供給されて、かつ、sslpasswordが無いか空欄の場合、TTYが利用可能であればパスワードはOpenSSLによりEnter PEM pass phrase:プロンプトで対話的に入力が要求されます。アプリケーションはクライアント証明書のプロンプトとsslpasswordパラメータの操作を、自身のキーパスワードコールバックを供給することで置き換えできます。PQsetSSLKeyPassHook_OpenSSLを参照してください。

証明書の作成手順については、18.9.5をご覧ください。

33.18.3. 異なるモードで提供される保護

sslmodeパラメータ値を変更することで、異なったレベルの保護を提供します。SSLは以下の3種類の攻撃に対する保護を提供することができます。

盗聴

> クライアント・サーバ間のネットワークトラフィックを第三者が監視することができれば、（ユーザ名とパスワードを含め）双方の接続情報と通過するデータを読み取ることができます。SSLはこれを防止するために暗号を使用します。

中間者攻撃（MITM）

> データがクライアント・サーバ間で渡されている時に、第三者がそのデータを変更できれば、サーバを装うことができ、従ってたとえ暗号化されていてもデータを理解し変更することができます。第三者はそこで、この攻撃を検出不可能にする接続情報とデータを元のサーバに送ることができます。これを行う共通した媒介はDNSポイズニングとアドレス乗っ取りを含み、それに従ってクライアントは意図したサーバではなく異なったサーバに誘導されます。同時に、このことを成し遂げるいくつかの異なった攻撃も存在します。SSLはクライアントに対しサーバを認証することで、この防止に証明書検証を使用します。

なりすまし

> 第三者が認定されたクライアントを装うことができれば、それはアクセスしてはならないデータに簡単にアクセス可能になります。典型的にこれは心もとないパスワード管理から生じます。SSLは有効な証明書の所持者のみサーバにアクセスできることを確実にすることで、この防止策としてクライアント証明書を使用します。

SSLで信頼できるとされる接続では、SSLの使用を接続確立前にクライアントとサーバの双方において設定されなければなりません。サーバのみに構成されると、クライアントはサーバが高度なセキュリティを必要とすることが判る以前に、（例えばパスワードのような）機密事項を扱う情報を結局送ることになります。libpqにおいて、sslmodeパラメータをverify-fullまたはverify-caに設定し、そして対象を検証するためルート証明書をシステムに提供することで、安全な接続を確実に行うことができます。これは暗号化されたweb閲覧に対するhttps URLの使用とよく似ています。

一度サーバが認証されると、クライアントは機密事項を扱うデータを送ることができます。この意味は、これまでクライアントは認証に証明書が使われているかどうかを知る必要がなく、サーバ構成においてのみこのことを指定しても安全だと言うことです。

すべてのSSLオプションでは暗号化の形式と鍵交換といったオーバヘッドがかかります。このため性能と安全性との間で決定されるべきトレードオフがあります。表 33.1は異なるsslmode値が防御する危険性と、安全性とオーバヘッドに対する声明を示したものです。

表33.1 SSLモードの説明

sslmode	盗聴防止	MITM防止	声明
disable	いいえ	いいえ	セキュリティはどうでもよく、暗号化の負荷を払いたくない
allow	たぶん	いいえ	セキュリティはどうでもよいが、サーバがそれを強く要求するのであれば暗号化のオーバヘッドを払ってもよい
prefer	たぶん	いいえ	セキュリティはどうでもよいが、サーバがそれをサポートするのであれば暗号化のオーバヘッドを払ってもよい
require	はい	いいえ	データを暗号化して欲しい。そしてオーバヘッドも受け入れる。意図したサーバに常に接続することをネットワークが確実にしてくれると信用する
verify-ca	はい	CAの方針に依存	データを暗号化して欲しい。そしてオーバヘッドも受け入れる。信頼するサーバに確実に接続したい
verify-full	はい	はい	データを暗号化して欲しい。そしてオーバヘッドも受け入れる。信頼するサーバに接続すること、そのサーバが指定したものであることを確実にしたい

verify-caとverify-fullの差異はルートCAの規定に依存します。公的なCAが使用されるとき、verify-caはそのCAで他の誰かが登録したかもしれないサーバへの接続を許可します。この場合、verify-fullが常に使用されなければなりません。独自CAが使用されるとき、または自己署名証明書であったとしてもverify-caは十分な防御策を提供します。

sslmodeのデフォルト値はpreferです。表で示したように、これはセキュリティの視点では意味がなく、可能であれば性能上のオーバヘッドを保証するだけです。これは後方互換性を提供するためのみにデフォルトとなっているもので、安全性確保の観点からは推奨されません。

33.18.4. SSLクライアントファイル使用方法

表 33.2にクライアントにおけるSSL設定に関連するファイルをまとめます。

表33.2 libpq/クライアントにおけるSSLファイルの使用方法

ファイル	内容	効果
~/.postgresql/postgresql.crt	クライアント証明書	サーバにより要求されます

ファイル	内容	効果
~/.postgresql/postgresql.key	クライアントの秘密キー	所有者により送信されるクライアント証明書を証明します。証明書の所有者が信頼できることを意味していません。
~/.postgresql/root.crt	信頼できる認証局	サーバ証明書が信頼できる認証局により署名されたか検査します。
~/.postgresql/root.crl	認証局により失効された証明書	サーバ証明書はこのリストにあってはいけません

33.18.5. SSLライブラリの初期化

使用するアプリケーションがlibsslとlibcryptoの両方またはいずれか一方のライブラリを初期化し、libpqがSSLサポート付きで構築された場合、libsslとlibcryptoの両方またはいずれか一方のライブラリはアプリケーションによって初期化されたことをlibpqに伝えるためPQinitOpenSSLを呼び出さなければなりません。これにより、libpqはこれらのライブラリを初期化しなくなります。

PQinitOpenSSL

アプリケーションがどのセキュリティライブラリを初期化するか選択することができます。

```
void PQinitOpenSSL(int do_ssl, int do_crypto);
```

do_sslが非ゼロの時、libpqは最初のデータベース接続を開始する以前にOpenSSLライブラリを初期化します。do_cryptoが非ゼロの時、libcryptoライブラリが初期化されます。デフォルトでは(PQinitOpenSSLが呼ばれない場合)、両方のライブラリが初期化されます。SSLサポートがコンパイルされていない場合、この関数は存在しますが何もしません。

使用するアプリケーションがOpenSSLまたはその基礎をなすlibcryptoライブラリのいずれかを使用し、そして初期化するのであれば、最初のデータベース接続開始以前に、適切なパラメータをゼロにしてこの関数を呼び出さなければなりません。同時に、データベース接続開始前に初期化を行ったことの確認をしてください。

PQinitSSL

アプリケーションがどのセキュリティライブラリを初期化するか選択することができます。

```
void PQinitSSL(int do_ssl);
```

この関数はPQinitOpenSSL(do_ssl, do_ssl)と等価です。OpenSSLおよびlibcryptoの両方を初期化する、もしくは両方ともしないアプリケーションにとっては(この関数で)十分です。

PostgreSQL 8.0以降、PQinitSSLは含まれていますが、PQinitOpenSSLはPostgreSQL 8.4で追加されました。従って、旧バージョンのlibpqで動かす必要があるアプリケーションではPQinitSSLの方が好ましいかもしれません。

33.19. スレッド化プログラムの振舞い

デフォルトでlibpqは再入可能、かつ、スレッドセーフです。アプリケーションコードをコンパイルする時にコンパイラの特殊なコマンドラインオプションを使う必要があるかもしれません。スレッドを有効にしたアプリケーションの構築方法についての情報は、使用するシステムの文書を参照してください。また、PTHREAD_CFLAGSとPTHREAD_LIBSに関してsrc/Makefile.globalも一読してください。以下の関数により、libpqのスレッドセーフ状態を確認することができます。

PQisthreadsafe

libpqライブラリのスレッドセーフ状態を返します。

```
int PQisthreadsafe();
```

libpqがスレッドセーフの場合1が、さもなくば0が返ります。

スレッドに関する1つの制限として、異なるスレッドから同時に同一のPGconnオブジェクトを操作することはできません。具体的には、異なるスレッドから同一接続オブジェクトを介してコマンドを同時に発行することができません。(コマンドの同時実行が必要な場合、接続を複数使用してください。)

PGresultオブジェクトは生成後、読み込み専用であり、そのためスレッド間で自由に渡すことができます。しかし33.11や33.13で説明するPGresultを変更する関数のいずれかを使用している場合、同一のPGresultに対する同時操作を防ぐことも、作成者の責任です。

非推奨の関数、PQrequestCancelやPQoidStatusはスレッドセーフではありませんので、マルチスレッドプログラムでは使用してはなりません。PQrequestCancelはPQcancelに、PQoidStatusはPQoidValueに置き換えることができます。

(libpqの内部に加えて)アプリケーション中でKerberosを利用している場合、Kerberos関数はスレッドセーフではありませんのでKerberos呼び出しの前後をロックする必要があるでしょう。libpqとアプリケーション間のロック処理を協調させる方法としてlibpqのソースコードのPQregisterThreadLock関数を参照してください。

33.20. libpqプログラムの構築

libpqを使用するプログラムの構築(つまり、コンパイルとリンク)を行うためには、以下をすべて実施する必要があります。

- libpq-fe.hヘッダファイルをインクルードします。

```
#include <libpq-fe.h>
```

これを忘れると、通常コンパイラから以下のようなエラーメッセージが発生します。

```
foo.c: In function `main':
foo.c:34: `PGconn' undeclared (first use in this function)
foo.c:35: `PGresult' undeclared (first use in this function)
foo.c:54: `CONNECTION_BAD' undeclared (first use in this function)
foo.c:68: `PGRES_COMMAND_OK' undeclared (first use in this function)
```

```
foo.c:95: `PGRES_TUPLES_OK' undeclared (first use in this function)
```

- コンパイラに-Idirectoryオプションを付与することで、コンパイラにPostgreSQLヘッダファイルをインストールしたディレクトリを通知します。（デフォルトでこのディレクトリを検索するコンパイラもあります。その場合はこのオプションを省くことができます。）例えば、以下のようなコンパイルコマンドになります。

  ```
  cc -c -I/usr/local/pgsql/include testprog.c
  ```

  Makefileを使用しているのであれば、CPPFLAGS変数にこのオプションを追加してください。

  ```
  CPPFLAGS += -I/usr/local/pgsql/include
  ```

  他のユーザがそのプログラムをコンパイルする可能性がある場合は、上のようにディレクトリの場所を直接書き込むべきではありません。その代わりにpg_configユーティリティを実行して、各システムにおけるヘッダファイルの在処を検索させることができます。

  ```
  $ pg_config --includedir
  /usr/local/include
  ```

  もしも、pkg-configがインストールされている場合、代わりとして以下を実行します。

  ```
  $ pkg-config --cflags libpq
  -I/usr/local/include
  ```

  これは既にパスの最前部で-Iが含まれていることに注意してください。

  正確なオプションを指定できなかった結果、コンパイラは以下のようなエラーメッセージを生成します。

  ```
  testlibpq.c:8:22: libpq-fe.h: No such file or directory
  ```

- 最終的なプログラムのリンク時、-lpqオプションを指定して、libpqライブラリを組み込んでください。同時に-Ldirectoryオプションを指定して、コンパイラにlibpqライブラリの在処を通知してください。（繰り返しますが、コンパイラはデフォルトでいくつかのディレクトリを検索します。）移植性を最大にするために、-lpqオプションの前に-Lを記述してください。以下に例を示します。

  ```
  cc -o testprog testprog1.o testprog2.o -L/usr/local/pgsql/lib -lpq
  ```

  同様にpg_configを使用してライブラリのあるディレクトリを見つけることもできます。

  ```
  $ pg_config --libdir
  /usr/local/pgsql/lib
  ```

  さもなくば、この場合もやはりpkg-configを使用します。

  ```
  $ pkg-config --libs libpq
  -L/usr/local/pgsql/lib -lpq
  ```

重ねて、これはパスのみならず全てのオプションを表示することに注意してください。

この部分で問題があった場合のエラーメッセージは以下のようなものになります。

```
testlibpq.o: In function `main':
testlibpq.o(.text+0x60): undefined reference to `PQsetdbLogin'
testlibpq.o(.text+0x71): undefined reference to `PQstatus'
testlibpq.o(.text+0xa4): undefined reference to `PQerrorMessage'
```

これは-lpqの付け忘れを示します。

```
/usr/bin/ld: cannot find -lpq
```

これは-Lの付け忘れ、あるいは、ディレクトリ指定の間違いを示します。

33.21. サンプルプログラム

以下を含むサンプルプログラムが、ソースコード配布物内のsrc/test/examplesディレクトリにあります。

例33.1 libpq サンプルプログラム 1

```
/*
 * src/test/examples/testlibpq.c
 *
 *
 * testlibpq.c
 *

 *              C言語PostgreSQLフロントエンドライブラリlibpqの試験
 */
#include <stdio.h>
#include <stdlib.h>
#include "libpq-fe.h"

static void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

int
main(int argc, char **argv)
{
    const char *conninfo;
```

```
PGconn     *conn;
PGresult   *res;
int         nFields;
int         i,
            j;

/*

 * ユーザがコマンドラインでパラメータを提供した場合、
  conninfo文字列として使用する。
 * 提供されない場合はデフォルトでdbname=postgresを使用する。
 * その他の接続パラメータについては環境変数やデフォルトを使用する。
 */
if (argc > 1)
    conninfo = argv[1];
else
    conninfo = "dbname = postgres";


/* データベースとの接続を確立する */
conn = PQconnectdb(conninfo);


/* バックエンドとの接続確立に成功したかを確認する */
if (PQstatus(conn) != CONNECTION_OK)
{
    fprintf(stderr, "Connection to database failed: %s",
            PQerrorMessage(conn));
    exit_nicely(conn);
}


/* 悪意のユーザによる乗っ取りを防ぐように常に安全なサーチパスを設定 */
res = PQexec(conn,
             "SELECT pg_catalog.set_config('search_path', '', false)");
if (PQresultStatus(res) != PGRES_TUPLES_OK)
{
    fprintf(stderr, "SET failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}

/*

 * メモリリークを避けるため、
  必要なくなったときにはいつでもPGresultを
```

```
 * PQclearすべき
 */
PQclear(res);

/*

 * この試験ケースではカーソルを使用する。
 * そのため、
  トランザクションブロック内で実行する必要がある。
 * すべてを単一の"select * from pg_database"というPQexec()で行うこと
 * も可能だが、
  例としては簡単過ぎる。
 */


/* トランザクションブロックを開始する。 */
res = PQexec(conn, "BEGIN");
if (PQresultStatus(res) != PGRES_COMMAND_OK)
{
    fprintf(stderr, "BEGIN command failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}
PQclear(res);

/*

 * データベースのシステムカタログpg_databaseから行を取り出す。
 */
res = PQexec(conn, "DECLARE myportal CURSOR FOR select * from pg_database");
if (PQresultStatus(res) != PGRES_COMMAND_OK)
{
    fprintf(stderr, "DECLARE CURSOR failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}
PQclear(res);

res = PQexec(conn, "FETCH ALL in myportal");
if (PQresultStatus(res) != PGRES_TUPLES_OK)
{
    fprintf(stderr, "FETCH ALL failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}
```

```
    /* まず属性名を表示する。 */
    nFields = PQnfields(res);
    for (i = 0; i < nFields; i++)
        printf("%-15s", PQfname(res, i));
    printf("\n\n");


    /* そして行を表示する。 */
    for (i = 0; i < PQntuples(res); i++)
    {
        for (j = 0; j < nFields; j++)
            printf("%-15s", PQgetvalue(res, i, j));
        printf("\n");
    }

    PQclear(res);


    /* ポータルを閉ざす。ここではエラーチェックは省略した… */
    res = PQexec(conn, "CLOSE myportal");
    PQclear(res);


    /* トランザクションを終了する */
    res = PQexec(conn, "END");
    PQclear(res);


    /* データベースとの接続を閉じ、後始末を行う。 */
    PQfinish(conn);

    return 0;
}
```

例33.2 libpq サンプルプログラム 2

```
/*
 * src/test/examples/testlibpq2.c
 *
 *
 * testlibpq2.c

 *              非同期通知インタフェースの試験
 *
```

```
 * このプログラムを起動し、
      別ウィンドウからpsqlを使用して以下を実行してください。
 *   NOTIFY TBL2;
 * 4回繰り返すとこのプログラムは終了します。
 *

 * もう少し凝りたければ、
      以下を実施してください。
 * 以下のコマンド(src/test/examples/testlibpq2.sqlで提供)でデータベースを作成します。
 *
 *   CREATE SCHEMA TESTLIBPQ2;
 *   SET search_path = TESTLIBPQ2;
 *   CREATE TABLE TBL1 (i int4);
 *   CREATE TABLE TBL2 (i int4);
 *   CREATE RULE r1 AS ON INSERT TO TBL1 DO
 *     (INSERT INTO TBL2 VALUES (new.i); NOTIFY TBL2);
 *
 * Start this program, then from psql do this four times:
 *
 *   INSERT INTO TESTLIBPQ2.TBL1 VALUES (10);
 */

#ifdef WIN32
#include <windows.h>
#endif
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/time.h>
#include <sys/types.h>
#ifdef HAVE_SYS_SELECT_H
#include <sys/select.h>
#endif

#include "libpq-fe.h"

static void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

int
main(int argc, char **argv)
```

```
{
    const char *conninfo;
    PGconn     *conn;
    PGresult   *res;
    PGnotify   *notify;
    int         nnotifies;

    /*

     * ユーザがコマンドラインでパラメータを提供した場合、
      conninfo文字列として使用する。
     * 提供されない場合はデフォルトでdbname=postgresを使用する。
     * その他の接続パラメータについては環境変数やデフォルトを使用する。
     */
    if (argc > 1)
        conninfo = argv[1];
    else
        conninfo = "dbname = postgres";


    /* データベースとの接続を確立する。 */
    conn = PQconnectdb(conninfo);


    /* バックエンドとの接続確立に成功したかを確認する */
    if (PQstatus(conn) != CONNECTION_OK)
    {
        fprintf(stderr, "Connection to database failed: %s",
                PQerrorMessage(conn));
        exit_nicely(conn);
    }


    /* 悪意のユーザによる乗っ取りを防ぐように常に安全なサーチパスを設定 */
    res = PQexec(conn,
                 "SELECT pg_catalog.set_config('search_path', '', false)");
    if (PQresultStatus(res) != PGRES_TUPLES_OK)
    {
        fprintf(stderr, "SET failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }

    /*
     * Should PQclear PGresult whenever it is no longer needed to avoid memory
     * leaks
```

```
     */
    PQclear(res);

    /*

     * LISTENコマンドを発行して、
      INSERTルールによる通知を有効にする。
     */
    res = PQexec(conn, "LISTEN TBL2");
    if (PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "LISTEN command failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }
    PQclear(res);


    /* 4回通知を受けたら終了する。 */
    nnotifies = 0;
    while (nnotifies < 4)
    {
        /*

         * その接続で何かが起こるまで待機する。ここでは入力待ちのために
         * select(2)を使用する。poll()や類似機能を使用することも可能
         * である。
         */
        int         sock;
        fd_set      input_mask;

        sock = PQsocket(conn);

        if (sock < 0)

            break;              /* 発生してはならない。 */

        FD_ZERO(&input_mask);
        FD_SET(sock, &input_mask);

        if (select(sock + 1, &input_mask, NULL, NULL, NULL) < 0)
        {
            fprintf(stderr, "select() failed: %s\n", strerror(errno));
            exit_nicely(conn);
        }
```

```
        /* ここで入力を確認する。 */
        PQconsumeInput(conn);
        while ((notify = PQnotifies(conn)) != NULL)
        {
            fprintf(stderr,
                    "ASYNC NOTIFY of '%s' received from backend PID %d\n",
                    notify->relname, notify->be_pid);
            PQfreemem(notify);
            nnotifies++;
            PQconsumeInput(conn);
        }
    }

    fprintf(stderr, "Done.\n");


    /* データベースとの接続を閉じ、後始末を行う。 */
    PQfinish(conn);

    return 0;
}
```

例33.3 libpq サンプルプログラム 3

```
/*
 * src/test/examples/testlibpq3.c
 *
 *
 * testlibpq3.c

 *      行以外のパラメータとバイナリI/Oの試験。
 *


 * 実行前に、
      以下のコマンド(src/test/examples/testlibpq3.sqlで提供)を使用して
 * データベースを作成してください。
 *
 * CREATE SCHEMA testlibpq3;
 * SET search_path = testlibpq3;
 * SET standard_conforming_strings = ON;
 * CREATE TABLE test1 (i int4, t text, b bytea);
 * INSERT INTO test1 values (1, 'joe''s place', '\000\001\002\003\004');
 * INSERT INTO test1 values (2, 'ho there', '\004\003\002\001\000');
 *
```

```
 * 以下の出力が想定されます。
 *
 * tuple 0: got
 *  i = (4 bytes) 1
 *  t = (11 bytes) 'joe's place'
 *  b = (5 bytes) \000\001\002\003\004
 *
 * tuple 0: got
 *  i = (4 bytes) 2
 *  t = (8 bytes) 'ho there'
 *  b = (5 bytes) \004\003\002\001\000
 */

#ifdef WIN32
#include <windows.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <sys/types.h>
#include "libpq-fe.h"


/* ntohl/htonl用 */
#include <netinet/in.h>
#include <arpa/inet.h>


static void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

/*

 * この関数は上のコメントで定義したテーブルからバイナリフォーマットでフェッチした
 * クエリ結果を表示します。
 * main() 関数が2度使うので、
      結果を分割します。
 */
static void
show_binary_results(PGresult *res)
```

```
{
    int         i,
                j;
    int         i_fnum,
                t_fnum,
                b_fnum;


    /* 結果中の列オーダーの仮定を嫌うので PQfnumber を利用する */
    /* PQfnumber  */
    i_fnum = PQfnumber(res, "i");
    t_fnum = PQfnumber(res, "t");
    b_fnum = PQfnumber(res, "b");

    for (i = 0; i < PQntuples(res); i++)
    {
        char       *iptr;
        char       *tptr;
        char       *bptr;
        int         blen;
        int         ival;


        /* 列の値を取得(NULLを出来る限り無視) */
        iptr = PQgetvalue(res, i, i_fnum);
        tptr = PQgetvalue(res, i, t_fnum);
        bptr = PQgetvalue(res, i, b_fnum);

        /*

         * INT4のバイナリ表現はネットワークバイトオーダーによる。
         * よって、
      ローカルバイトオーダーに合わせた方が良い。
         */
        ival = ntohl(*((uint32_t *) iptr));

        /*

         * TEXT型のバイナリ表現も同様にテキスト。
         * 更にlibpqはその最後にゼロバイトを付与するので、

         * C言語の文字列として単純に扱うことができる。
         *


         * BYTEA のバイト表現はバイトの集まりである。
         * null 埋め込みを含むのでフィールド長に注意を払わなければいけない。
```

```
         */
        blen = PQgetlength(res, i, b_fnum);

        printf("tuple %d: got\n", i);
        printf(" i = (%d bytes) %d\n",
               PQgetlength(res, i, i_fnum), ival);
        printf(" t = (%d bytes) '%s'\n",
               PQgetlength(res, i, t_fnum), tptr);
        printf(" b = (%d bytes) ", blen);
        for (j = 0; j < blen; j++)
            printf("\\%03o", bptr[j]);
        printf("\n\n");
    }
}

int
main(int argc, char **argv)
{
    const char *conninfo;
    PGconn     *conn;
    PGresult   *res;
    const char *paramValues[1];
    int         paramLengths[1];
    int         paramFormats[1];
    uint32_t    binaryIntVal;

    /*

     * ユーザがコマンドラインでパラメータを提供した場合、
      conninfo文字列として使用する。
     * 提供されない場合はデフォルトでdbname=postgresを使用する。
     * その他の接続パラメータについては環境変数やデフォルトを使用する。
     */
    if (argc > 1)
        conninfo = argv[1];
    else
        conninfo = "dbname = postgres";


    /* データベースとの接続を確立する */
    conn = PQconnectdb(conninfo);


    /* バックエンドとの接続確立に成功したかを確認する */
    if (PQstatus(conn) != CONNECTION_OK)
    {
```

```
    fprintf(stderr, "Connection to database failed: %s",
            PQerrorMessage(conn));
    exit_nicely(conn);
}


/* 悪意のユーザによる乗っ取りを防ぐように常に安全なサーチパスを設定 */
res = PQexec(conn, "SET search_path = testlibpq3");
if (PQresultStatus(res) != PGRES_COMMAND_OK)
{
    fprintf(stderr, "SET failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}
PQclear(res);

/*

 * このプログラムのポイントは、
  行外パラメータを持つPQexecParams()の使用方法、

 * および、
  データのバイナリ転送を示すことである。
 *

 * この最初の例はパラメータをテキストとして渡す。
 * しかし結果はバイナリフォーマットで受ける。
 * 行外パラメータを使うことで、
  データがテキストであっても引用符付けや
 * エスケープ処理といった多くの長たらしいゴミをなくすことができる。
 * パラメータ値内部の引用符に対して特殊な処理を行う必要がないことに注目して
 * ほしい。
 */


/* 以下が行外パラメータの値である。 */
paramValues[0] = "joe's place";

res = PQexecParams(conn,
                   "SELECT * FROM test1 WHERE t = $1",

                   1,           /* パラメータは1つ。 */
                   NULL,        /* バックエンドにパラメータの型を推測させる。 */
                   paramValues,
                   NULL,        /* テキストのため、
  パラメータ長は不要。 */
```

```
                    NULL,       /* デフォルトですべてのパラメータはテキスト。 */
                    1);         /* バイナリ結果を要求。 */

if (PQresultStatus(res) != PGRES_TUPLES_OK)
{
    fprintf(stderr, "SELECT failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}

show_binary_results(res);

PQclear(res);

/*

 * 2つ目の例は、
  バイナリフォームの中で整数値パラメータを渡す。
 * そして再びバイナリフォームで結果を受け取る。
 *

 * バックエンドにパラメータタイプを推測させていると PQexecParams に伝えるが、

 * クエリテキストの中にパラメータシンボルを入れることによって 強制的に決定する。
 * これはバイナリパラメータに送るときに安全で良い大きさである。
 */


/* 整数値 "2" をネットワークバイトオーダーに変換 */
binaryIntVal = htonl((uint32_t) 2);


/* PQexecParams 用にパラメータ配列をセットする */
paramValues[0] = (char *) &binaryIntVal;
paramLengths[0] = sizeof(binaryIntVal);

paramFormats[0] = 1;        /* バイナリ */

res = PQexecParams(conn,
                    "SELECT * FROM test1 WHERE i = $1::int4",

                    1,      /* パラメータは1つ */
                    NULL,   /* バックエンドにパラメータの型を推測させる。 */
                    paramValues,
                    paramLengths,
                    paramFormats,
```

```
                      1);       /* バイナリ結果を要求。 */

    if (PQresultStatus(res) != PGRES_TUPLES_OK)
    {
        fprintf(stderr, "SELECT failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }

    show_binary_results(res);

    PQclear(res);


    /* データベースとの接続を閉じ、後始末を行う。 */
    PQfinish(conn);

    return 0;
}
```

第34章 ラージオブジェクト

PostgreSQLにはラージオブジェクト機能があります。これは、特殊なラージオブジェクト構造に格納されたユーザデータに対してストリーム様式のアクセスを提供します。全体をまるごと簡単に操作するには巨大過ぎるデータ値を操作する場合、ストリーミングアクセスが有用です。

本章では、PostgreSQLラージオブジェクトデータに関する、実装、プログラミング、問い合わせ言語インタフェースについて説明します。libpq Cライブラリを例として本章で使用していますが、ほとんどのPostgreSQL固有のプログラミングインタフェースは同等の機能を持っています。他のインタフェースでは、巨大な値を汎用的にサポートできるように、ラージオブジェクトインタフェースを内部で使用しているかもしれません。ここでは説明しません。

34.1. はじめに

すべてのラージオブジェクトは`pg_largeobject`というひとつのシステムテーブル内に格納されます。各ラージオブジェクトはまた`pg_largeobject_metadata`システムテーブルの中に項目を持ちます。ラージオブジェクトを、ファイル標準操作に似た読み取り/書き出しAPIを使用して、作成、変更、削除することができます。

PostgreSQLではまた、単一のデータベースページよりも大きな値を自動的にテーブルごとに存在する二次格納領域に格納する「TOAST」という格納システムをサポートします。これによりラージオブジェクトの一部は不要になりました。ラージオブジェクト機能に残る利点の1つは、そのサイズが4テラバイトまで可能であるという点です。TOASTではフィールドは1ギガバイトまでしか扱えません。また、ラージオブジェクトの部分読み取り、部分更新は効率的に行うことができます。一方TOAST化されたフィールドに対する操作のほとんどは、そのフィールド全体を単位として読み取り、または書き出しがなされます。

34.2. 実装機能

ラージオブジェクトの実装では、ラージオブジェクトを「チャンク」に分割し、チャンクをデータベース内の行に格納しています。B-treeインデックスは読み書き用のランダムアクセスに際して、正確なチャンク番号の高速検索を保証しています。

ラージオブジェクト用のチャンクは継続性を持ちません。例えば、アプリケーションが新しくラージオブジェクトを開き、1000000オフセットにシークし、数バイトそこに書き出した場合、これは1000000バイトほどの格納領域が割り当てられることにはなりません。データバイトの範囲に対応するチャンクのみが実際に書き出されます。しかし読み取り操作は最後に存在するチャンクの前にある未割り当ての領域すべてとしてゼロを読み取ります。これはUnixファイルシステムの「スパース割り当て」ファイルの一般動作に一致します。

PostgreSQL 9.0からラージオブジェクトは所有者およびアクセス権限を持ちます。これはGRANTおよびREVOKEを使用して管理可能です。ラージオブジェクトの読み取りにはSELECT権限が必要です。また書き出し、切り詰めのためにはUPDATE権限が必要です。ラージオブジェクトの所有者(またはデータベーススーパーユーザ)のみがラージオブジェクトの削除、コメント付け、所有者の変更が可能です。過去のリリースとの互換性に関するこの動作を調整するためには`lo_compat_privileges`実行時パラメータを参照してください。

34.3. クライアントインタフェース

本節では、PostgreSQLのlibpqクライアントインタフェースライブラリで提供されるラージオブジェクトへのアクセス手段について説明します。PostgreSQLラージオブジェクトインタフェースは、Unixファイルシステムインタフェースに因んで設計されており、open、read、write、lseekなど同様のインタフェースを有しています。

ラージオブジェクトファイル記述子はトランザクションの間でしか有効でありませんので、これらの関数を使用したラージオブジェクトの操作はすべてSQLトランザクションブロック内で行われなければなりません。

これらの関数のいずれか1つの実行時にエラーが発生した場合、関数は他ではあり得ない値、通常は0または-1を返します。エラーを説明するメッセージは接続オブジェクト内に格納され、PQerrorMessageを用いて取り出すことができます。

これらの関数を使用するクライアントアプリケーションは、libpq/libpq-fs.hヘッダファイルをインクルードし、libpqライブラリとリンクしなければなりません。

34.3.1. ラージオブジェクトの作成

```
Oid lo_creat(PGconn *conn, int mode);
```

この関数はラージオブジェクトを新規に作成します。戻り値は新規ラージオブジェクトに割り当てられたOIDで、失敗時にはInvalidOid(0)が返されます。PostgreSQL 8.1では、modeは使用されず、無視されます。しかし、以前のリリースとの後方互換性を保持するために、これをINV_READ、INV_WRITE、INV_READ | INV_WRITEに設定することが最善です。(これらの定数シンボルはlibpq/libpq-fs.hヘッダファイルで定義されています。)

以下に例を示します。

```
inv_oid = lo_creat(conn, INV_READ|INV_WRITE);
```

```
Oid lo_create(PGconn *conn, Oid lobjId);
```

この関数もラージオブジェクトを新規に作成します。割り当てられるOIDをlobjIdで指定することができます。こうした場合、そのOIDが他のラージオブジェクトですでに使用されていた場合、失敗します。lobjIdがInvalidOid(0)の場合、lo_createは未使用のOIDを割り当てます。(これはlo_creatと同じ動作です。) 戻り値は新規ラージオブジェクトに割り当てられたOIDで、失敗時にはInvalidOid(0)が返されます。

lo_createはPostgreSQL 8.1から導入されました。この関数を古いバージョンで実行させると失敗し、InvalidOidが返されます。

例を示します。

```
inv_oid = lo_create(conn, desired_oid);
```

34.3.2. ラージオブジェクトのインポート

オペレーティングシステム上のファイルをラージオブジェクトとしてインポートするには、以下の関数を呼び出します。

```
Oid lo_import(PGconn *conn, const char *filename);
```

filenameには、ラージオブジェクトとしてインポートするオペレーティングシステム上のファイルのパス名を指定します。戻り値は、新規ラージオブジェクトに割り当てられたOIDです。失敗時はInvalidOid(0)が返されます。このファイルがサーバではなく、クライアントインタフェースライブラリから読み取られることに注意してください。ですから、このファイルはクライアントのファイルシステム上に存在し、クライアントアプリケーションから読み取り可能でなければなりません。

```
Oid lo_import_with_oid(PGconn *conn, const char *filename, Oid lobjId);
```

この関数も新規のラージオブジェクトをインポートします。割り当てられるOIDをlobjIdで指定することができます。こうした場合、そのOIDが他のラージオブジェクトですでに使用されていた場合、失敗します。lobjIdがInvalidOid(0)の場合、lo_import_with_oidは未使用のOIDを割り当てます(これはlo_importと同じ動作です)。戻り値は新規ラージオブジェクトに割り当てられたOIDで、失敗時にはInvalidOid(0)が返されます。

lo_import_with_oidはPostgreSQL 8.4から導入され、8.1から導入されたlo_createを内部で使用しています。この関数を8.0以前のバージョンで実行させると失敗し、InvalidOidが返されます。

34.3.3. ラージオブジェクトのエクスポート

ラージオブジェクトをオペレーティングシステム上のファイルにエクスポートするには、以下の関数を呼び出します。

```
int lo_export(PGconn *conn, Oid lobjId, const char *filename);
```

lobjId引数には、エクスポートさせるラージオブジェクトのOIDを指定し、filename引数には、オペレーティングシステム上のファイルのパス名を指定します。このファイルはサーバではなく、クライアントインタフェースライブラリによって書き込まれることに注意してください。成功時には1、失敗時には-1が返されます。

34.3.4. 既存のラージオブジェクトのオープン

読み取りまたは書き込みのために既存のラージオブジェクトを開く場合は、以下の関数を呼び出します。

```
int lo_open(PGconn *conn, Oid lobjId, int mode);
```

lobjId引数には開きたいラージオブジェクトのOIDを指定します。modeの各ビットは、そのオブジェクトを読み取りのみ(INV_READ)、書き込みのみ(INV_WRITE)、またはその両方できるように開くのかを制御するものです。(これらの定数シンボルはlibpq/libpq-fs.hヘッダファイルで定義されています。) lo_openは、

lo_read、lo_write、lo_lseek、lo_lseek64、lo_tell、lo_tell64、lo_truncate、lo_truncate64、lo_closeで使用する(非負の)ラージオブジェクト記述子を返します。この記述子は現在のトランザクション期間のみで有効です。失敗時には-1が返されます。

現時点では、サーバはINV_WRITEモードとINV_READ | INV_WRITEモードとを区別しません。どちらの場合でも記述子から読み取り可能です。しかし、これらのモードとINV_READだけのモードとの間には大きな違いがあります。INV_READモードでは記述子に書き込むことができません。そして、読み込んだデータは、このトランザクションや他のトランザクションで後で書き込んだかどうかは関係なく、lo_openを実行した時に有効だったトランザクションスナップショットの時点のラージオブジェクトの内容を反映したものになります。INV_WRITEを付けて開いた記述子から読み取ると、現在のトランザクションによる書き込みや他のトランザクションがコミットした書き込みすべてを反映したデータが返されます。これは、通常のSELECT SQLコマンドにおけるREPEATABLE READトランザクションの動作とREAD COMMITTEDトランザクションの動作の違いに似ています。

ラージオブジェクトにSELECT権限が与えられていなかったり、INV_WRITEが指定されていて、かつUPDATE権限が与えられていないと、lo_openは失敗します。(PostgreSQL 11よりも前では、こうした権限チェックはディスクリプタを使って最初に読み出し、あるいは書き込みの呼び出しを実際に行う際に実施されていました。) この権限チェックは、lo_compat_privileges実行時パラメータで無効にすることができます。

以下に例を示します。

```
inv_fd = lo_open(conn, inv_oid, INV_READ|INV_WRITE);
```

34.3.5. ラージオブジェクトへのデータの書き込み

```
int lo_write(PGconn *conn, int fd, const char *buf, size_t len);
```

lenバイトを、buf(lenサイズでなければなりません)からfdラージオブジェクト記述子に書き込みます。fd引数は事前に実行したlo_openの戻り値でなければいけません。実際に書き込まれたバイト数が返されます(現在の実装ではエラーが発生しない限りlenと常に等しくなります)。エラーイベントが発生した場合は、-1を返します。

lenパラメータはsize_tとして宣言されていますが、この関数はINT_MAXより大きな値を拒絶します。実際には、多くても数メガバイトのチャンクでデータを転送することが最善です。

34.3.6. ラージオブジェクトからのデータの読み込み

```
int lo_read(PGconn *conn, int fd, char *buf, size_t len);
```

len長のバイトを、fdラージオブジェクト記述子からbuf(lenサイズでなければなりません)に読み込みます。fd引数は事前に実行したlo_openの戻り値でなければいけません。実際に読み込まれたバイト数が返されます。ラージオブジェクトの最後に先に達した場合はlenより小さな値になります。エラーイベントが発生した場合は、-1値を返します。

lenパラメータはsize_tとして宣言されていますが、この関数はINT_MAXより大きな値を拒絶します。実際には、多くても数メガバイトをチャンク内にデータを転送することが最善です。

34.3.7. ラージオブジェクトのシーク

ラージオブジェクト記述子に関連付けされている、現在の読み取りまたは書き込みを行う位置を変更するには、以下の関数を呼び出します。

```
int lo_lseek(PGconn *conn, int fd, int offset, int whence);
```

この関数はfdで識別されるラージオブジェクト識別子の現在の位置を指すポインタを、offsetで指定した新しい位置に変更します。whenceに指定可能な値は、SEEK_SET（オブジェクトの先頭位置からシーク）、SEEK_CUR（現在位置からシーク）、SEEK_END（オブジェクトの末尾位置からシーク）のいずれかです。戻り値は新しい位置ポインタで、エラー時に-1が返されます。

2GBを超えるサイズのラージオブジェクトを取り扱う場合は代わりに以下を使用してください。

```
pg_int64 lo_lseek64(PGconn *conn, int fd, pg_int64 offset, int whence);
```

この関数はlo_lseekと同じ動作をしますが、offsetとして2GBを超える値を受付け、2GBより大きな結果を出力します。lo_lseekは2GBを超える新しい位置ポインタが指定された場合に失敗することに注意してください。

lo_lseek64はPostgreSQL 9.3にて追加されました。この関数をより古いバージョンのサーバに対して実行した場合には失敗し、-1が返ります。

34.3.8. ラージオブジェクトのシーク位置の入手

ラージオブジェクト記述子の現在の読み取り、書き込み位置を入手するには、以下の関数を呼び出します。

```
int lo_tell(PGconn *conn, int fd);
```

エラーが発生した場合は-1が返されます。

サイズが2GBを超える可能性があるラージオブジェクトを取り扱う場合は代わりに以下を使用します。

```
pg_int64 lo_tell64(PGconn *conn, int fd);
```

この関数はlo_tellと同じ動作をしますが、2GBより大きな結果を出力します。lo_tellは2GBを超える新しい位置での読み書きに失敗します。

lo_tell64はPostgreSQL 9.3にて追加されました。この関数をより古いバージョンのサーバに対して実行した場合には失敗し、-1が返ります。

34.3.9. ラージオブジェクトを切り詰める

ラージオブジェクトを指定した長さに切り詰めるには、以下を呼び出します。

```
int lo_truncate(PGcon *conn, int fd, size_t len);
```

この関数はラージオブジェクト記述子fdをlen長に切り詰めます。fd引数は前もってlo_openが返したものでなければなりません。lenが現在のラージオブジェクト長より大きければ、ラージオブジェクトは指定された長さまでヌルバイト('\0')で拡張されます。成功時lo_truncateはゼロを返します。失敗時の戻り値は-1です。

fdディスクリプタの読み取り/書き出し位置は変わりません。

lenパラメータはsize_tとして宣言されていますが、lo_truncateはINT_MAXより大きな値を拒絶します。

2GBを超える可能性があるラージオブジェクトを取り扱う場合は代わりに以下を使用します。

```
int lo_truncate64(PGcon *conn, int fd, pg_int64 len);
```

この関数はlo_truncateと同じ動作をしますが、2GBを超えるlenを受け付けることができます。

lo_truncateはPostgreSQL 8.3で新規に導入されました。この関数を古いバージョンのサーバに対して実行した場合は失敗し、-1が返されます。

lo_truncate64はPostgreSQL 9.3にて追加されました。この関数をより古いバージョンのサーバに対して実行した場合には失敗し、-1が返ります。

34.3.10. ラージオブジェクト記述子を閉じる

以下を呼び出すことでラージオブジェクト記述子を閉ざすことができます。

```
int lo_close(PGconn *conn, int fd);
```

ここで、fdはlo_openの戻り値であるラージオブジェクト記述子です。成功すると、lo_closeは0を返します。失敗すると、-1を返します。

開いたままのラージオブジェクト記述子は全てトランザクションの終了時に自動的に閉ざされます。

34.3.11. ラージオブジェクトの削除

データベースからラージオブジェクトを削除するには、以下の関数を呼び出します。

```
int lo_unlink(PGconn *conn, Oid lobjId);
```

lobjId引数は削除するラージオブジェクトのOIDを指定します。成功時に1を、失敗時に-1を返します。

34.4. サーバ側の関数

SQLからラージオブジェクトを操作するのに適応したサーバ側の関数を表 34.1に列挙します。

表34.1 SQL向けラージオブジェクト関数

関数 説明 例
lo_from_bytea (loid oid, data bytea) → oid

関数 説明 例
ラージオブジェクトを作成してそこにdataを格納する。loidが0であれば、システムが空いているOIDを選び、そうでなければそのOIDが使われる(すでにそのOIDを持つラージオブジェクトがあればエラーになる)。成功すれば、そのラージオブジェクトのOIDが返される。 lo_from_bytea(0, '\xffffff00') → 24528
lo_put (loid oid, offset bigint, data bytea) → void ラージオブジェクト内の与えられたオフセットからdataを書き込む。必要であれば、ラージオブジェクトは拡張される。 lo_put(24528, 1, '\xaa') →
lo_get (loid oid [, offset bigint, length integer]) → bytea そこからラージオブジェクトの内容または部分文字列を取り出す。 lo_get(24528, 0, 3) → \xffaaff

これまで説明したクライアント側の関数それぞれに対応する、追加のサーバ側の関数があります。実際、ほとんどのクライアント側の関数は対応するサーバ側の関数に対する単なるインタフェースです。SQLコマンドからの呼び出しが便利な関数は、lo_creat、lo_create、lo_unlink、lo_import、lo_exportです。これらの使用例を示します。

```
CREATE TABLE image (
    name            text,
    raster          oid
);


SELECT lo_creat(-1);        -- 新しい空のラージオブジェクトのOIDを返します


SELECT lo_create(43213);    -- OID 43213でラージオブジェクトの生成を試行します


SELECT lo_unlink(173454);  -- OID 173454のラージオブジェクトを削除します

INSERT INTO image (name, raster)
    VALUES ('beautiful image', lo_import('/etc/motd'));


INSERT INTO image (name, raster)  -- 上と同じですが使用するOIDを指定します
    VALUES ('beautiful image', lo_import('/etc/motd', 68583));

SELECT lo_export(image.raster, '/tmp/motd') FROM image
    WHERE name = 'beautiful image';
```

サーバ側のlo_importおよびlo_export関数の動作はクライアント側の関数とかなり異なります。この2つの関数はサーバのファイルシステム上のファイルの読み書きを、データベースを所有するユーザの権限で行います。したがって、デフォルトではこれらの使用はスーパーユーザに限定されています。対照的に、クライ

アント側のインポート関数とエクスポート関数はクライアントのファイルシステム上のファイルをクライアントプログラムの権限で読み書きします。このクライアント側の関数は、対象となるラージオブジェクトの読み出し、書き込み権限を除き、データベース権限を必要としません。

> **注意**
>
> サーバサイドlo_importとlo_export関数に対してGRANTを非スーパーユーザに適用することは可能ですが、その結果が意味することについて慎重な考慮が必要です。そうした権限を持つ悪意のあるユーザは、(たとえば、サーバ設定ファイルを書き換えることによって)容易にその権限を拡張してスーパーユーザになることができるでしょう。あるいは、そのようにしてデータベーススーパーユーザ権限を取得することなく、サーバのファイルシステムを攻撃することができるでしょう。したがって、そうした権限を持つロールへのアクセスは、スーパーユーザロールへのアクセスとまったく同様に、注意深く防御されなければなりません。にもかかわらず、サーバサイドのlo_importあるいはlo_exportを定形業務に使う必要があるなら、完全なスーパーユーザ権限よりは、そうした権限を持つロールを使う方が安全です。偶発的な間違いから来る被害のリスクを減らすのに役立つからです。

またlo_readおよびlo_writeの機能はサーバサイドの呼び出しを介しても利用することができます。しかしサーバサイドの関数名はクライアント側のインタフェースとは異なり、アンダースコアが含まれません。loreadおよびlowriteとしてこれらの関数を呼び出さなければなりません。

34.5. サンプルプログラム

例 34.1は、libpqを使ったラージオブジェクトインタフェースの使い方を示すサンプルプログラムです。プログラムの一部はコメントアウトされていますが、読者にわかりやすいようにそのまま残してあります。このプログラムは、ソース配布物内のsrc/test/examplesにあります。

例34.1 Libpqを使用したラージオブジェクトのサンプルプログラム

```
/*-------------------------------------------------------------------
 *
 * testlo.c



 *    libpqによるラージオブジェクトを使用する試験
 *
 * Portions Copyright (c) 1996-2020, PostgreSQL Global Development Group
 * Portions Copyright (c) 1994, Regents of the University of California
 *
 *
 * IDENTIFICATION
 *    src/test/examples/testlo.c
 *
 *-------------------------------------------------------------------
```

```
 */
#include <stdio.h>
#include <stdlib.h>

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

#include "libpq-fe.h"
#include "libpq/libpq-fs.h"

#define BUFSIZE         1024

/*
 * importFile -



 *    "in_filename"ファイルをラージオブジェクト"lobjOid"としてデータベースにインポートする。
 *
 */
static Oid
importFile(PGconn *conn, char *filename)
{
    Oid         lobjId;
    int         lobj_fd;
    char        buf[BUFSIZE];
    int         nbytes,
                tmp;
    int         fd;

    /*



     * 読み込むファイルを開く
     */
    fd = open(filename, O_RDONLY, 0666);
    if (fd < 0)



    {                           /* エラー時 */
        fprintf(stderr, "cannot open unix file\"%s\"\n", filename);
    }
```

```
    /*



     * ラージオブジェクトを作成する
     */
    lobjId = lo_creat(conn, INV_READ | INV_WRITE);
    if (lobjId == 0)
        fprintf(stderr, "cannot create large object");

    lobj_fd = lo_open(conn, lobjId, INV_WRITE);

    /*



     * Unixファイルから読み込み、
      転置ファイルへ書き出す
     */
    while ((nbytes = read(fd, buf, BUFSIZE)) > 0)
    {
        tmp = lo_write(conn, lobj_fd, buf, nbytes);
        if (tmp < nbytes)
            fprintf(stderr, "error while reading \"%s\"", filename);
    }

    close(fd);
    lo_close(conn, lobj_fd);

    return lobjId;
}

static void
pickout(PGconn *conn, Oid lobjId, int start, int len)
{
    int         lobj_fd;
    char       *buf;
    int         nbytes;
    int         nread;

    lobj_fd = lo_open(conn, lobjId, INV_READ);
    if (lobj_fd < 0)
        fprintf(stderr, "cannot open large object %u", lobjId);

    lo_lseek(conn, lobj_fd, start, SEEK_SET);
```

```
    buf = malloc(len + 1);

    nread = 0;
    while (len - nread > 0)
    {
        nbytes = lo_read(conn, lobj_fd, buf, len - nread);
        buf[nbytes] = '\0';
        fprintf(stderr, ">>> %s", buf);
        nread += nbytes;
        if (nbytes <= 0)
            break;              /* no more data? */
    }
    free(buf);
    fprintf(stderr, "\n");
    lo_close(conn, lobj_fd);
}

static void
overwrite(PGconn *conn, Oid lobjId, int start, int len)
{
    int         lobj_fd;
    char       *buf;
    int         nbytes;
    int         nwritten;
    int         i;

    lobj_fd = lo_open(conn, lobjId, INV_WRITE);
    if (lobj_fd < 0)
        fprintf(stderr, "cannot open large object %u", lobjId);

    lo_lseek(conn, lobj_fd, start, SEEK_SET);
    buf = malloc(len + 1);

    for (i = 0; i < len; i++)
        buf[i] = 'X';
    buf[i] = '\0';

    nwritten = 0;
    while (len - nwritten > 0)
    {
        nbytes = lo_write(conn, lobj_fd, buf + nwritten, len - nwritten);
        nwritten += nbytes;
        if (nbytes <= 0)
        {
            fprintf(stderr, "\nWRITE FAILED!\n");
            break;
```

```
        }
    }
    free(buf);
    fprintf(stderr, "\n");
    lo_close(conn, lobj_fd);
}


/*
 * exportFile -



 *    ラージオブジェクト"lobjOid"を"out_filename"ファイルにエクスポートする。
 *
 */
static void
exportFile(PGconn *conn, Oid lobjId, char *filename)
{
    int         lobj_fd;
    char        buf[BUFSIZE];
    int         nbytes,
                tmp;
    int         fd;

    /*



     * ラージオブジェクトを作成する
     */
    lobj_fd = lo_open(conn, lobjId, INV_READ);
    if (lobj_fd < 0)
        fprintf(stderr, "cannot open large object %u", lobjId);

    /*



     * 書き込むファイルを開く
     */
    fd = open(filename, O_CREAT | O_WRONLY | O_TRUNC, 0666);
    if (fd < 0)
```

```
    {                           /* エラー時 */
        fprintf(stderr, "cannot open unix file\"%s\"",
                filename);
    }

    /*



     * 転置ファイルから読み込み、
      Unixファイルへ書き出す。
     */
    while ((nbytes = lo_read(conn, lobj_fd, buf, BUFSIZE)) > 0)
    {
        tmp = write(fd, buf, nbytes);
        if (tmp < nbytes)
        {
            fprintf(stderr, "error while writing \"%s\"",
                    filename);
        }
    }

    lo_close(conn, lobj_fd);
    close(fd);

    return;
}

static void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

int
main(int argc, char **argv)
{
    char       *in_filename,
               *out_filename;
    char       *database;
    Oid         lobjOid;
    PGconn     *conn;
    PGresult   *res;

    if (argc != 4)
```

```
{
    fprintf(stderr, "Usage: %s database_name in_filename out_filename\n",
            argv[0]);
    exit(1);
}

database = argv[1];
in_filename = argv[2];
out_filename = argv[3];

/*




 * 接続を設定する
 */
conn = PQsetdb(NULL, NULL, NULL, NULL, database);




/* バックエンドとの接続が成功したかどうか確認する */
if (PQstatus(conn) != CONNECTION_OK)
{
    fprintf(stderr, "Connection to database failed: %s",
            PQerrorMessage(conn));
    exit_nicely(conn);
}




/* 常に安全なサーチパスを設定する。そのため、悪意のあるユーザは操作できない。 */
res = PQexec(conn,
             "SELECT pg_catalog.set_config('search_path', '', false)");
if (PQresultStatus(res) != PGRES_TUPLES_OK)
{
    fprintf(stderr, "SET failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}
PQclear(res);

res = PQexec(conn, "begin");
PQclear(res);
printf("importing file \"%s\" ...\n", in_filename);
```

```
/*  lobjOid = importFile(conn, in_filename); */
    lobjOid = lo_import(conn, in_filename);
    if (lobjOid == 0)
        fprintf(stderr, "%s\n", PQerrorMessage(conn));
    else
    {
        printf("\tas large object %u.\n", lobjOid);

        printf("picking out bytes 1000-2000 of the large object\n");
        pickout(conn, lobjOid, 1000, 1000);

        printf("overwriting bytes 1000-2000 of the large object with X's\n");
        overwrite(conn, lobjOid, 1000, 1000);

        printf("exporting large object to file \"%s\" ...\n", out_filename);
/*      exportFile(conn, lobjOid, out_filename); */
        if (lo_export(conn, lobjOid, out_filename) < 0)
            fprintf(stderr, "%s\n", PQerrorMessage(conn));
    }

    res = PQexec(conn, "end");
    PQclear(res);
    PQfinish(conn);
    return 0;
}
```

第35章 ECPG — C言語による埋め込みSQL

本章では、PostgreSQLの埋め込みSQLパッケージについて説明します。このパッケージはCとC++言語で作成されました。作者はLinus Tolke(<linus@epact.se>)とMichael Meskes(<meskes@postgresql.org>)です。元々これはCで動作するように作成されました。C++でも動作しますが、C++の構文すべてはまだ認識できません。

本書は完全なものではありません。しかし、このインタフェースは標準化されているので、SQLに関するその他の資料から追加情報を入手できます。

35.1. 概念

埋め込みSQLプログラムは通常のプログラミング言語(ここではC)で記述されたコードで、特別にマークされたセクション内のSQLコマンドとともに使用されます。このプログラムを構築するには、まずソースコード(*.pgc)を埋め込みSQLプリプロセッサに渡します。ソースコードは、プリプロセッサによって通常のCプログラム(*.c)に変換され、その後Cコンパイラによって処理されます。(コンパイルとリンクの詳細については35.10を参照してください)変換されたECPGアプリケーションは、libpqライブラリにある関数を埋め込みSQLライブラリ(ecpglib)を介して呼び出し、通常のフロントエンド・バックエンドプロトコルを使ってPostgreSQLサーバと通信します。

CコードからSQLコマンドを扱う場合は、埋め込みSQLの方が他の手法よりも有効です。まず、埋め込みSQLはCプログラムの変数との面倒な双方間の情報移行を処理してくれます。さらに、プログラム内のSQLコードは構築時に正確な構文になっているかどうか検査されます。また、C言語での埋め込みSQLは標準SQLで既に定義されており、他の様々なSQLデータベースシステムでサポートされています。PostgreSQLの実装は可能な限りこの標準に準拠するよう設計されています。また通常の場合、他のSQLデータベース用に作成された埋め込みSQLプログラムを比較的簡単にPostgreSQLへ移植することができます。

先に述べた通り、埋め込みSQLインタフェース用のプログラムは、通常のCプログラムに、データベース関連処理を行うための特別なコードを加えたものです。この特別なコードは、常に、次のような形式になっています。

```
EXEC SQL ...;
```

このSQL文は、構文上でC言語の文の置き換えとなります。SQL文によりますが、グローバルレベル、または関数内で記述することができます。埋め込みSQL文における大文字小文字の区別の有無は、Cコードではなく、通常のSQLコードの規則に従います。また、SQL標準の部分にある入れ子のC形式のコメントを許します。しかし、プログラムのCの部分では、入れ子のコメントを受け付けないというC標準に従います。

以下の節で、すべての埋め込みSQL文について説明します。

35.2. データベース接続の管理

この節では、データベース接続の開始、終了、および切り替え方について解説します。

35.2.1. データベースサーバへの接続

以下のSQL文を使用して、データベースへ接続します。

```
EXEC SQL CONNECT TO target [AS connection-name] [USER user-name];
```

targetは以下の方法で指定されます。

- dbname[@hostname][:port]
- tcp:postgresql://hostname[:port][/dbname][?options]
- unix:postgresql://hostname[:port][/dbname][?options]
- 上の形式のいずれかを含むSQL文字列リテラル。
- 上の形式のいずれかを含む文字変数への参照。
- DEFAULT

接続対象をリテラル（つまり、変数を参照しない形）で指定し、その値を引用符でくくらなかった場合、大文字小文字の区別に関して通常のSQLの規則が適用されます。また、この場合、必要に応じて個々のパラメータを二重引用符で別々にくくることもできます。実際には、おそらく（単一引用符でくくられた）文字列リテラルもしくは変数の参照を使用した方がエラーをより防止することができます。DEFAULT接続対象は、デフォルトデータベース、デフォルトのユーザ名で接続を初期化します。この場合は、ユーザ名と接続名を分けて指定することができません。

ユーザ名を指定するには、別の方法もあります。

- username
- username/password
- username IDENTIFIED BY password
- username USING password

これまで同様、usernameとpasswordは、SQL識別子、SQL文字列リテラル、文字型変数への参照を取ることができます。

接続対象にoptionsを含めるのなら、keyword=value指定をアンパサンド(&)で区切って構成します。許されるキーワードは、libpqが認識するものと同じです(33.1.2を参照してください)。keywordやvalueの前の空白は無視されますが、中や後の空白は無視されません。valueの中に&を書く方法はないことに注意してください。

1つのプログラム内で複数の接続を処理する場合には、connection-nameを使用します。プログラムで1つしか接続を使わない場合は省略して構いません。最も最近に開かれた接続が現在の接続になり、SQL文を実行しようとする時にデフォルトでこの接続が使用されます（本章の後で説明します）。

信用できないユーザが安全なスキーマ使用パターンを採用していないデータベースにアクセスできる場合、各セッションをsearch_pathから一般のユーザが書き込み可能なスキーマを取り除くことから

始めます。例えば、options=-c search_path=をoptionsに追加したり、接続後にEXEC SQL SELECT pg_catalog.set_config('search_path', '', false);を発行したりします。この配慮はECPGに特有のものではありません。任意のSQLコマンドを実行するインタフェースすべてに当てはまります。

以下にCONNECT文について、数例を示します。

```
EXEC SQL CONNECT TO mydb@sql.mydomain.com;

EXEC SQL CONNECT TO unix:postgresql://sql.mydomain.com/mydb AS myconnection USER john;

EXEC SQL BEGIN DECLARE SECTION;
const char *target = "mydb@sql.mydomain.com";
const char *user = "john";
const char *passwd = "secret";
EXEC SQL END DECLARE SECTION;
 ...
EXEC SQL CONNECT TO :target USER :user USING :passwd;

/* もしくは EXEC SQL CONNECT TO :target USER :user/:passwd; */
```

最後の形式では、文字変数参照として上を参照する変数を使用しています。後の節で、接頭辞にコロンを持つ場合のSQL文内でのC変数の使用方法について説明します。

接続対象の書式は標準SQLでは規定されていないことに注意してください。そのため、移植可能なアプリケーションを開発したいのであれば、上の例の最後の方法を基にして、接続対象文字列をどこかにカプセル化してください。

35.2.2. 接続の選択

前節で示したSQL文は現在の接続、つまり、最も最近に開いた接続上で実行されます。複数の接続を管理する必要があるアプリケーションでは、これを処理する2つの方法があります。

1つ目の選択肢は、各SQL文で明示的に接続を選択することです。以下に例を示します。

```
EXEC SQL AT connection-name SELECT ...;
```

アプリケーションが複数の接続を不特定な順番で使用する必要がある場合、この選択肢は特に適しています。

アプリケーションの実行に複数スレッドを使用する場合、スレッド間で接続を同時に共有できません。接続へのアクセスを(ミューテクスを使用して)明示的に制御するか、または各スレッド用の接続を使用するかを行わなければなりません。

2番目の選択肢は、現在の接続を切り替えるSQL文を実行することです。以下のSQL文です。

```
EXEC SQL SET CONNECTION connection-name;
```

多くのSQL文を同一接続に対して使用する場合、この選択肢は特に便利です。

以下に複数のデータベースコネクションを管理しているプログラムの例を示します。

```
#include <stdio.h>

EXEC SQL BEGIN DECLARE SECTION;
    char dbname[1024];
EXEC SQL END DECLARE SECTION;

int
main()
{
    EXEC SQL CONNECT TO testdb1 AS con1 USER testuser;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;
    EXEC SQL CONNECT TO testdb2 AS con2 USER testuser;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;
    EXEC SQL CONNECT TO testdb3 AS con3 USER testuser;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;


    /* この問い合わせは最後に開いたデータベース"testdb3"で実行される。 */
    EXEC SQL SELECT current_database() INTO :dbname;
    printf("current=%s (should be testdb3)\n", dbname);


    /* "testdb2"で問い合わせを実行するには"AT"を使う */
    EXEC SQL AT con2 SELECT current_database() INTO :dbname;
    printf("current=%s (should be testdb2)\n", dbname);


    /* 現在の接続を"testdb1"に切り替える。 */
    EXEC SQL SET CONNECTION con1;

    EXEC SQL SELECT current_database() INTO :dbname;
    printf("current=%s (should be testdb1)\n", dbname);

    EXEC SQL DISCONNECT ALL;
    return 0;
}
```

この例は、次のような出力を生成します。

```
current=testdb3 (should be testdb3)
current=testdb2 (should be testdb2)
current=testdb1 (should be testdb1)
```

35.2.3. 接続を閉じる

接続を閉じるには以下のSQL文を使用します。

```
EXEC SQL DISCONNECT [connection];
```

connectionは以下の方法で指定されます。

- connection-name
- DEFAULT
- CURRENT
- ALL

接続名の指定がなければ、現在の接続が閉じられます。

アプリケーションでは、過去に開いたすべての接続を明示的に閉じることを推奨します。

35.3. SQLコマンドの実行

すべてのSQLコマンドは、埋め込みSQLアプリケーション内で実行できます。以下に例をいくつか示します。

35.3.1. SQL文の実行

テーブルの作成:

```
EXEC SQL CREATE TABLE foo (number integer, ascii char(16));
EXEC SQL CREATE UNIQUE INDEX num1 ON foo(number);
EXEC SQL COMMIT;
```

行の挿入:

```
EXEC SQL INSERT INTO foo (number, ascii) VALUES (9999, 'doodad');
EXEC SQL COMMIT;
```

行の削除:

```
EXEC SQL DELETE FROM foo WHERE number = 9999;
EXEC SQL COMMIT;
```

行の更新:

```
EXEC SQL UPDATE foo
```

```
    SET ascii = 'foobar'
    WHERE number = 9999;
EXEC SQL COMMIT;
```

単一の行を返すSELECT文は、同様に EXEC SQL を用いて直接実行することができます。複数の行を扱うためには、アプリケーションはカーソルを使わなければなりません; 35.3.2 を参照してください。(特殊なケースでは、アプリケーションは複数の行をホスト変数の配列に一度に読み込むことができます; 35.4.4.3.1 を参照してください)

単一行の検索:

```
EXEC SQL SELECT foo INTO :FooBar FROM table1 WHERE ascii = 'doodad';
```

同様に、設定パラメータは SHOW コマンドによって取得することができます:

```
EXEC SQL SHOW search_path INTO :var;
```

:somethingという形のトークンはホスト変数です。つまり、Cプログラム内の変数を参照するものです。これについては35.4で説明します。

35.3.2. カーソルの使用

複数行の結果セットを受け取るためには、アプリケーションはカーソルを定義し、必要に応じてレコードを一行ずつ取り込む必要があります。カーソルを使った処理は、カーソルの宣言、カーソルのオープン、カーソルからのFETCH、カーソルのクローズという流れになります。

カーソルを用いたSELECT:

```
EXEC SQL DECLARE foo_bar CURSOR FOR
    SELECT number, ascii FROM foo
    ORDER BY ascii;
EXEC SQL OPEN foo_bar;
EXEC SQL FETCH foo_bar INTO :FooBar, DooDad;
...
EXEC SQL CLOSE foo_bar;
EXEC SQL COMMIT;
```

カーソルの宣言の詳細については DECLARE を、FETCH コマンドの詳細については FETCH を参照してください。

> **注記**
>
> ECPGの DECLARE コマンド自身は、PostgreSQLバックエンドに送られるSQL文を実行しません。OPEN コマンドが実行された段階で、バックエンド内部で(DECLAREコマンドで宣言された)カーソルが開かれます。

35.3.3. トランザクションの管理

デフォルトモードでは、SQL文はEXEC SQL COMMITが発行されることによってのみコミットされます。埋め込みSQLインタフェースでも、ecpgコマンド（ecpgを参照）の-tコマンドラインオプション、あるいは EXEC SQL SET AUTOCOMMIT TO ON文によって（psqlのデフォルトの振舞いに似た）トランザクションの自動コミットをサポートしています。自動コミットモードでは、問い合わせが明示的なトランザクションブロックの内部にある場合を除き、すべての問い合わせが自動的にコミットされます。自動コミットモードは、EXEC SQL SET AUTOCOMMIT TO OFFを使用して明示的に無効にすることができます。

以下のトランザクション管理コマンドを使用することができます:

EXEC SQL COMMIT

実行中のトランザクションのコミット。

EXEC SQL ROLLBACK

実行中のトランザクションのロールバック。

EXEC SQL PREPARE TRANSACTION transaction_id

2相コミット用に現在のトランザクションをプリペアします。

EXEC SQL COMMIT PREPARED transaction_id

プリペアド状態のトランザクションをコミットします。

EXEC SQL ROLLBACK PREPARED transaction_id

プリペアド状態のトランザクションをロールバックします。

EXEC SQL SET AUTOCOMMIT TO ON

自動コミットモードの有効化。

EXEC SQL SET AUTOCOMMIT TO OFF

自動コミットモードの無効化。デフォルト状態。

35.3.4. プリペアド文

SQL文に渡す値がコンパイル時に決まらない場合、または同じSQL文を何度も実行する場合、プリペアド文が便利です。

SQL文はPREPAREコマンドを使ってプリペアします。まだ決まっていない値については、プレースホルダ「?」を使います:

```
EXEC SQL PREPARE stmt1 FROM "SELECT oid, datname FROM pg_database WHERE oid = ?";
```

SQL文が一行のみの結果を返却する場合には、アプリケーションはSQL文をPREPAREした後、USINGを用いてプレースホルダに実際の値を与えてEXECUTEを実行することができます。

```
EXEC SQL EXECUTE stmt1 INTO :dboid, :dbname USING 1;
```

SQL文が複数の行を返却する場合には、アプリケーションはプリペアド文の宣言に対応したカーソルを利用することができます。入力パラメータを設定するために、カーソルはUSINGとともに開かれなければなりません:

```
EXEC SQL PREPARE stmt1 FROM "SELECT oid,datname FROM pg_database WHERE oid > ?";
EXEC SQL DECLARE foo_bar CURSOR FOR stmt1;


/* 結果集合の最後に到達したら、whileループから抜ける */
EXEC SQL WHENEVER NOT FOUND DO BREAK;

EXEC SQL OPEN foo_bar USING 100;
...
while (1)
{
    EXEC SQL FETCH NEXT FROM foo_bar INTO :dboid, :dbname;
    ...
}
EXEC SQL CLOSE foo_bar;
```

プリペアド文をこれ以上必要としなくなったら、解放処理をしなければなりません:

```
EXEC SQL DEALLOCATE PREPARE name;
```

PREPARE についての詳細は PREPARE を参照してください。また、プレースホルダと入力パラメータの利用についての詳細は 35.5 を参照してください。

35.4. ホスト変数の使用

35.3では、埋め込みSQLプログラムでどのようにSQL文を実行するのかについて説明しました。このSQL文の中には固定値しか使用しないものや、ユーザが指定する値をSQL文の中に挿入する手段を提供しないもの、問い合わせが返す値をプログラムで処理する手段を提供しないものがありました。この種のSQL文は実際のアプリケーションでは役に立ちません。本節では、*ホスト変数*という単純な機構を使用した、Cプログラムと埋め込みSQL文との間でデータをやり取りする方法を詳細に説明します。埋め込みSQLプログラムでは、SQL文を*ホスト言語*となるCプログラムコードにおける*ゲスト*とみなします。したがって、Cプログラムの変数は*ホスト変数*と呼ばれます。

PostgreSQLバックエンドとECPGアプリケーションの間で値をやり取りするその他の方法は、35.7 で説明されているSQLデスクリプタを使う方法です。

35.4.1. 概要

埋め込みSQLにおけるCプログラムとSQL文との間でのデータのやり取りは特に単純です。値に適切な引用符を付与するといった、様々な複雑な処理を伴う、プログラムにデータを文中に貼り付けさせるという方法はなく、単にSQL文の中に、先頭にコロンを付けたC変数名を書くだけです。以下に例を示します。

```
EXEC SQL INSERT INTO sometable VALUES (:v1, 'foo', :v2);
```

このSQL文は、v1とv2という2つのC変数を参照し、また、通常のSQL文字列リテラルも使用しています。これは、使用できるデータの種類は1つだけという制限がないことを表しています。

SQL文内にCの変数を挿入するこの様式は、SQL文で値式が想定されている所であればどこでも動作します。

35.4.2. 宣言セクション

例えば問い合わせ内のパラメータとして、プログラムからデータベースへデータを渡す、もしくは、データベースからプログラムへデータを渡すためには、このようなデータを含むように意図されたC変数を、埋め込みSQLプリプロセッサが管理できるように、特殊な印のついたセクションで宣言する必要があります。

このセクションは以下で始まります。

```
EXEC SQL BEGIN DECLARE SECTION;
```

そして、以下で終わります。

```
EXEC SQL END DECLARE SECTION;
```

この行の間は、以下のような通常のC変数宣言でなければなりません。

```
int   x = 4;
char  foo[16], bar[16];
```

見てわかるとおり、省略可能ですが、変数に初期値を代入することができます。変数のスコープはプログラム内の宣言セクションの場所により決まります。また、以下のような暗黙的に宣言セクションを生成する構文を使って変数を宣言することもできます。

```
EXEC SQL int i = 4;
```

プログラム内に複数の宣言セクションを持たせることができます。

また、宣言は普通のC変数としてそのまま出力ファイルに出力されます。ですので、これらを再度宣言する必要はありません。通常、SQLコマンドで使用する予定がない変数はこの特別なセクションの外側で宣言されます。

構造体や共用体の定義もまた、DECLAREセクションの内側で表す必要があります。さもないと、プリプロセッサはその定義が不明であるために、これらの型を扱うことができません。

35.4.3. クエリ実行結果の受け取り

ここまでで、プログラムで生成したデータをSQLコマンドに渡すことができるようになりました。しかし、どのように問い合わせの結果を取り出すのでしょうか？この目的のために、埋め込みSQLでは、通常のSELECTとFETCHを派生した、特殊なコマンドを提供しています。これらのコマンドは特別なINTO句を持ち、ここで返された値をどのホスト変数に格納すればよいかを指定します。SELECT は単一行を返却する問い合わせに使用され、FETCH は複数の行を返却する問い合わせにおいてカーソルとともに使用されます。

以下にサンプルを示します。

```
/*

 * 以下のテーブルを前提とする
 * CREATE TABLE test1 (a int, b varchar(50));
 */

EXEC SQL BEGIN DECLARE SECTION;
int v1;
VARCHAR v2;
EXEC SQL END DECLARE SECTION;

 ...

EXEC SQL SELECT a, b INTO :v1, :v2 FROM test;
```

INTO句が選択リストとFROM句の間に現れます。選択リスト内の要素数とINTO直後のリスト（目的リストとも呼ばれます）の要素数は等しくなければなりません。

以下にFETCHコマンドの使用例を示します。

```
EXEC SQL BEGIN DECLARE SECTION;
int v1;
VARCHAR v2;
EXEC SQL END DECLARE SECTION;

 ...

EXEC SQL DECLARE foo CURSOR FOR SELECT a, b FROM test;

 ...

do
{
```

```
    ...
    EXEC SQL FETCH NEXT FROM foo INTO :v1, :v2;
    ...
} while (...);
```

ここでは、INTO句が通常のすべての句の後ろに現れています。

35.4.4. データ型の対応

ECPGアプリケーションがPostgreSQLバックエンドとCアプリケーションの間で値をやり取りする際、例えばサーバからクエリの結果を受け取る、または入力パラメータとともにSQL文を実行する場合、それらの値はPostgreSQLのデータ型とホスト言語の変数の型（具体的にはC言語のデータ型）の間で変換される必要があります。ECPGの重要な点のひとつは、ほとんどの場合においてECPGがこれらを自動的に扱うということです。

この点において、2つのデータ型があります: いくつかの単純なPostgreSQLのデータ型、integer や text などは、アプリケーションから直接読んだり書いたりすることができます。その他のPostgreSQLのデータ型、timestamp や numeric などは、特別なライブラリ関数によってしかアクセスすることができません; 35.4.4.2 を参照してください。

表 35.1には、PostgreSQLのどのデータ型がC言語のデータ型に対応するかが示されています。与えられたPostgreSQLのデータ型へ値を書き込みまたは読み込みしたい場合には、対応するC言語のデータ型の変数を宣言セクションにおいて宣言しなければなりません。

表35.1 PostgreSQLデータ型とC言語変数型の対応

PostgreSQLデータ型	ホスト変数型
smallint	short
integer	int
bigint	long long int
decimal	decimal[a]
numeric	numeric[a]
real	float
double precision	double
smallserial	short
serial	int
bigserial	long long int
oid	unsigned int
character(n), varchar(n), text	char[n+1], VARCHAR[n+1]
name	char[NAMEDATALEN]
timestamp	timestamp[a]
interval	interval[a]

PostgreSQLデータ型	ホスト変数型
date	date[a]
boolean	bool[b]
bytea	char *, bytea[n]

[a]この型は特別なライブラリ関数を通してのみアクセスできます; 35.4.4.2 を参照。

[b]ネイティブでなければ ecpglib.h で宣言。

35.4.4.1. 文字列の処理

varcharやtextのような文字列のデータ型を扱うため、ホスト変数を宣言するための2つの方法があります。

ひとつは char の配列 char[] を使うことで、C言語において文字列データを扱うもっとも一般的な方法です。

```
EXEC SQL BEGIN DECLARE SECTION;
    char str[50];
EXEC SQL END DECLARE SECTION;
```

文字列の長さについて、自分自身で気を付けておく必要があります。上記のホスト変数を49文字以上の文字列を返すクエリのターゲット変数として使った場合、バッファオーバーフローが発生します。

その他の方法は、ECPGによって提供される特殊なデータ型 VARCHAR を使う方法です。VARCHAR の配列の定義は、すべての変数が名前の付いた struct に変換されます。以下のような宣言は:

```
VARCHAR var[180];
```

次のように変換されます:

```
struct varchar_var { int len; char arr[180]; } var;
```

メンバー変数 arr は終端のゼロの1バイトを含む文字列を保持します。よって、文字列を VARCHAR ホスト変数に格納する場合には、ホスト変数はゼロ終端を含んだ長さで宣言されなければなりません。メンバー変数 len は arr に格納された文字列のゼロ終端を含まない長さを保持します。ホスト変数をクエリの入力として使用する際、strlen(arr) と len が違った場合には短いものが使用されます。

VARCHAR は大文字でも小文字でも記述することができますが、混在して記述することはできません。

char と VARCHAR ホスト変数は、他のSQLのデータ型の値を文字列表現として保持することもできます。

35.4.4.2. 特殊なデータ型へのアクセス

ECPGには、PostgreSQLサーバからのいくつかの特殊なデータ型とやりとりするための特殊なデータ型があります。特に、numeric, decimal, date, timestamp, interval 型へのサポートを実装しています。これらの

データ型は複雑な内部構造を持つため、ホスト変数のプリミティブ型(int, long long int, または char[])に対応させることはできません。アプリケーションは特別な型としてホスト変数を宣言し、pgtypesライブラリ内の関数を使ってアクセスすることで、これらの型を扱います。35.6 で詳細を解説されるpgtypesライブラリは、例えばタイムスタンプにインターバルを加算する際にクエリをSQLサーバに送らずに済ますような、これらの型を扱うための基本的な関数を含んでいます。

以降のサブセクションは、これらの特殊なデータ型を説明します。pgtypesライブラリ関数についての詳細は35.6を参照してください。

35.4.4.2.1. timestamp, date

以下は、timestamp 変数をECPGホストアプリケーションで扱う典型的なパターンです。

最初に、プログラムは timestamp 型のためのヘッダファイルをインクルードする必要があります:

```
#include <pgtypes_timestamp.h>
```

次に、宣言セクションで timestamp 型のホスト変数を宣言します:

```
EXEC SQL BEGIN DECLARE SECTION;
timestamp ts;
EXEC SQL END DECLARE SECTION;
```

そして、ホスト変数へ値を読み込んだら、pgtypesライブラリ関数を使って処理をします。以降の例では、 timestamp の値は PGTYPEStimestamp_to_asc() 関数によって text (ASCII) 形式に変換されます:

```
EXEC SQL SELECT now()::timestamp INTO :ts;

printf("ts = %s\n", PGTYPEStimestamp_to_asc(ts));
```

この例は、以下のような結果を表示します。

```
ts = 2010-06-27 18:03:56.949343
```

また、DATE型も同じ方法で扱うことができます。プログラムは pgtypes_date.h をインクルードし、ホスト変数を date 型として宣言し、PGTYPESdate_to_asc() 関数によって DATE の値を text 形式に変換します。pgtypesライブラリ関数についての詳細は、35.6 を参照してください。

35.4.4.2.2. interval

interval 型の扱い方は timestamp や date 型と似ています。但し、interval 型の値のために明示的にメモリを確保する必要があります。言い換えると、この変数のためのメモリ領域はスタックではなくヒープ上に確保されます。

以下にプログラム例を示します:

```
#include <stdio.h>
#include <stdlib.h>
#include <pgtypes_interval.h>

int
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;
    interval *in;
EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO testdb;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;

    in = PGTYPESinterval_new();
    EXEC SQL SELECT '1 min'::interval INTO :in;
    printf("interval = %s\n", PGTYPESinterval_to_asc(in));
    PGTYPESinterval_free(in);

    EXEC SQL COMMIT;
    EXEC SQL DISCONNECT ALL;
    return 0;
}
```

35.4.4.2.3. numeric, decimal

numeric と decimal 型の扱い方は interval 型と似ています: ポインタ宣言を必要とし、ヒープメモリを確保する必要があり、pgtypesライブラリ関数を使って変数にアクセスします。pgtypesライブラリ関数の詳細については、35.6 を参照してください。

decimal 型に対する専用の関数は提供されていません。アプリケーションは処理を行うために pgtypesライブラリ関数を使って numeric 変数に変換する必要があります。

以下に numeric および decimal 型の変数の処理の例を示します。

```
#include <stdio.h>
#include <stdlib.h>
#include <pgtypes_numeric.h>

EXEC SQL WHENEVER SQLERROR STOP;

int
main(void)
```

```
{
EXEC SQL BEGIN DECLARE SECTION;
    numeric *num;
    numeric *num2;
    decimal *dec;
EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO testdb;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;

    num = PGTYPESnumeric_new();
    dec = PGTYPESdecimal_new();

    EXEC SQL SELECT 12.345::numeric(4,2), 23.456::decimal(4,2) INTO :num, :dec;

    printf("numeric = %s\n", PGTYPESnumeric_to_asc(num, 0));
    printf("numeric = %s\n", PGTYPESnumeric_to_asc(num, 1));
    printf("numeric = %s\n", PGTYPESnumeric_to_asc(num, 2));


    /* decimalの値を表示するためdecimalをnumericに変換する。 */
    num2 = PGTYPESnumeric_new();
    PGTYPESnumeric_from_decimal(dec, num2);

    printf("decimal = %s\n", PGTYPESnumeric_to_asc(num2, 0));
    printf("decimal = %s\n", PGTYPESnumeric_to_asc(num2, 1));
    printf("decimal = %s\n", PGTYPESnumeric_to_asc(num2, 2));

    PGTYPESnumeric_free(num2);
    PGTYPESdecimal_free(dec);
    PGTYPESnumeric_free(num);

    EXEC SQL COMMIT;
    EXEC SQL DISCONNECT ALL;
    return 0;
}
```

35.4.4.2.4. bytea

bytea型の扱いは、VARCHARと似ています。bytea型の配列の定義は、すべての変数が名前の付いたstructに変換されます。以下のような宣言は:

```
bytea var[180];
```

次のように変換されます:

```
struct bytea_var { int len; char arr[180]; } var;
```

メンバ変数arrはバイナリフォーマットデータを保持します VARCHARとは異なり、'\0'をデータの一部として扱うこともできます。データは、ecpglibによりhex書式から、またはhex書式に変換されて、送信または受信されます。

> **注記**
>
> bytea変数は、bytea_outputがhexに設定されている場合にのみ使うことができます。

35.4.4.3. 非プリミティブ型のホスト変数

ホスト変数として、配列、typedef、構造体およびポインタも使うことができます。

35.4.4.3.1. 配列

ホスト変数としての配列の使い方には二通りの利用方法があります。一つ目の使い方は、35.4.4.1 で説明されたように char[] または VARCHAR[] の何らかのテキスト文字列を保持するための方法です。二つ目の使い方は、カーソルを用いずに複数行を返却するクエリ結果を受け取るために使う方法です。配列を使わない場合、複数行からなるクエリの実行結果を処理するには、カーソルと FETCH コマンドを使用する必要があります。しかし、配列のホスト変数を使うと、複数行を一括して受け取ることができます。配列の長さはすべての行を受け入れられるように定義されなければなりません。でなければバッファーオーバーフローが発生するでしょう。

以下の例は pg_database システムテーブルをスキャンし、利用可能なデータベースのすべてのOIDとデータベース名を表示します:

```
int
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;
    int dbid[8];
    char dbname[8][16];
    int i;
EXEC SQL END DECLARE SECTION;

    memset(dbname, 0, sizeof(char)* 16 * 8);
    memset(dbid, 0, sizeof(int) * 8);

    EXEC SQL CONNECT TO testdb;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;
```

```
    /* 複数行を一度に配列へと取り出す。 */
    EXEC SQL SELECT oid,datname INTO :dbid, :dbname FROM pg_database;

    for (i = 0; i < 8; i++)
        printf("oid=%d, dbname=%s\n", dbid[i], dbname[i]);

    EXEC SQL COMMIT;
    EXEC SQL DISCONNECT ALL;
    return 0;
}
```

この例は、以下の結果を表示します。(実際の値はローカルな環境に依存します)

```
oid=1, dbname=template1
oid=11510, dbname=template0
oid=11511, dbname=postgres
oid=313780, dbname=testdb
oid=0, dbname=
oid=0, dbname=
oid=0, dbname=
```

35.4.4.3.2. 構造体

メンバー変数の名前がクエリ結果のカラム名に合致する構造体は、複数のカラムを一括して受け取るために利用することができます。構造体は複数のカラムの値を単一のホスト変数で扱うことを可能にします。

以下の例は、pg_databaseシステムテーブルおよびpg_database_size()関数を使って、利用可能なデータベースのOID、名前、サイズを取得します。この例では、メンバー変数の名前がSELECT結果の各カラムに合致する構造体dbinfo_tが、複数のホスト変数に格納することなくFETCH文の一行の結果を受け取るために使用されています。

```
EXEC SQL BEGIN DECLARE SECTION;
    typedef struct
    {
       int oid;
       char datname[65];
       long long int size;
    } dbinfo_t;

    dbinfo_t dbval;
EXEC SQL END DECLARE SECTION;

    memset(&dbval, 0, sizeof(dbinfo_t));
```

```
    EXEC SQL DECLARE cur1 CURSOR FOR SELECT oid, datname, pg_database_size(oid) AS size FROM
 pg_database;
    EXEC SQL OPEN cur1;


    /* 結果集合の最後に到達したら、
      whileループから抜ける */
    EXEC SQL WHENEVER NOT FOUND DO BREAK;

    while (1)
    {

        /* 複数列を1つの構造体に取り込む。 */
        EXEC SQL FETCH FROM cur1 INTO :dbval;


        /* 構造体のメンバを表示する。 */
        printf("oid=%d, datname=%s, size=%lld\n", dbval.oid, dbval.datname, dbval.size);
    }

    EXEC SQL CLOSE cur1;
```

この例は、次の結果を示します（実際の値はローカルな環境に依存します）

```
oid=1, datname=template1, size=4324580
oid=11510, datname=template0, size=4243460
oid=11511, datname=postgres, size=4324580
oid=313780, datname=testdb, size=8183012
```

構造体のホスト変数は、多数のカラムを構造体のフィールドとして「吸収」します。追加のカラムは他のホスト変数に割り当てることができます。例えば、上記のプログラムは構造体に含まれない size 変数を使って以下のように書き換えることができます。

```
EXEC SQL BEGIN DECLARE SECTION;
    typedef struct
    {
       int oid;
       char datname[65];
    } dbinfo_t;

    dbinfo_t dbval;
    long long int size;
EXEC SQL END DECLARE SECTION;

    memset(&dbval, 0, sizeof(dbinfo_t));
```

```
    EXEC SQL DECLARE cur1 CURSOR FOR SELECT oid, datname, pg_database_size(oid) AS size FROM
 pg_database;
    EXEC SQL OPEN cur1;


    /* 結果集合の最後に到達したら、
      whileループから抜ける */
    EXEC SQL WHENEVER NOT FOUND DO BREAK;

    while (1)
    {

        /* 複数列を1つの構造体に取り込む。 */
        EXEC SQL FETCH FROM cur1 INTO :dbval, :size;


        /* 構造体のメンバを表示する。 */
        printf("oid=%d, datname=%s, size=%lld\n", dbval.oid, dbval.datname, size);
    }

    EXEC SQL CLOSE cur1;
```

35.4.4.3.3. typedef

新しい型と既存の型を対応付けるためには typedef キーワードを使ってください。

```
EXEC SQL BEGIN DECLARE SECTION;
    typedef char mychartype[40];
    typedef long serial_t;
EXEC SQL END DECLARE SECTION;
```

また、同様に以下を使うこともできます:

```
EXEC SQL TYPE serial_t IS long;
```

この宣言は、宣言セクションの一部である必要はありません。

35.4.4.3.4. ポインタ

ほとんどの一般的な型のポインタを宣言することができます。但し、自動メモリ確保を使わずにクエリのターゲット変数として使うことはできません。自動メモリ確保については 35.7 を参照してください。

```
EXEC SQL BEGIN DECLARE SECTION;
```

```
    int   *intp;
    char **charp;
EXEC SQL END DECLARE SECTION;
```

35.4.5. 非プリミティブSQLデータ型の扱い方

本節では、非スカラー型およびユーザ定義のSQLデータ型をECPGアプリケーションで扱う方法を示します。この内容は、前の説で説明した非プリミティブ型のホスト変数の扱い方とは別のものです。

35.4.5.1. 配列

SQLの多次元配列は、ECPGにおいては直接的にはサポートされていません。SQLの1次元配列をC言語の配列のホスト変数に対応させることはできますし、その逆もできます。しかし、文の作成時にはecpgがその列の型を知らないので、C言語の配列を対応するSQLの配列に入力できるか確かめられません。SQL文の出力を処理する時には、ecpgは必要な情報を持っていますので、どちらも配列であるか確かめます。

もし、クエリが配列の 要素 に対して個別にアクセスした場合、ECPGにおける配列の利用を避けることができます。その際、要素に対応させることができる型のホスト変数を利用しなければなりません。例えば、カラムの型が integer の配列の場合、int 型のホスト変数を使用することができます。同様に、要素の型が varchar または text の場合、char[] ないし VARCHAR[] 型のホスト変数を使用することができます。

以下に例を示します。次のようなテーブルを仮定します:

```
CREATE TABLE t3 (
    ii integer[]
);

testdb=> SELECT * FROM t3;
     ii
-------------
 {1,2,3,4,5}
(1 row)
```

以下のプログラム例は、配列の4番目の要素を取得し、それを int 型のホスト変数に保存します:

```
EXEC SQL BEGIN DECLARE SECTION;
int ii;
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE cur1 CURSOR FOR SELECT ii[4] FROM t3;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;
```

```
while (1)
{
    EXEC SQL FETCH FROM cur1 INTO :ii ;
    printf("ii=%d\n", ii);
}

EXEC SQL CLOSE cur1;
```

この例は以下のような結果を示します:

```
ii=4
```

複数の配列の要素を、配列型のホスト変数の複数の要素にマッピングするためには、配列型のカラムの各要素とホスト変数配列の各要素は、以下の例のように別々に管理されなければなりません:

```
EXEC SQL BEGIN DECLARE SECTION;
int ii_a[8];
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE cur1 CURSOR FOR SELECT ii[1], ii[2], ii[3], ii[4] FROM t3;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    EXEC SQL FETCH FROM cur1 INTO :ii_a[0], :ii_a[1], :ii_a[2], :ii_a[3];
    ...
}
```

繰り返しになりますが、以下の例は

```
EXEC SQL BEGIN DECLARE SECTION;
int ii_a[8];
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE cur1 CURSOR FOR SELECT ii FROM t3;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{

    /* 間違い */
    EXEC SQL FETCH FROM cur1 INTO :ii_a;
```

```
    ...
}
```

この場合は正しく動作しません。なぜなら、配列型のカラムをホストの配列変数に直接対応させることはできないからです。

もうひとつの回避策は、配列をホスト変数の char[] または VARCHAR[] 型に文字列表現として保存することです。この表現方法についての詳細は 8.15.2 を参照してください。このことは、配列にはホストプログラム内で自然な形ではアクセスできないことを意味しています（文字列表現を解析する追加処理が無ければ）。

35.4.5.2. 複合型

複合型はECPGでは直接はサポートされていませんが、簡単な回避方法が利用可能です。利用可能なワークアラウンドは、先に配列において説明されたものと似ています: 各属性に個別にアクセスするか、外部の文字列表現を使います。

以降の例のため、以下の型とテーブルを仮定します:

```
CREATE TYPE comp_t AS (intval integer, textval varchar(32));
CREATE TABLE t4 (compval comp_t);
INSERT INTO t4 VALUES ( (256, 'PostgreSQL') );
```

もっとも分かりやすい解決法は、各属性に個別にアクセスすることです。以下のプログラムは、comp_t型の各要素を個別に選択することによってサンプルのテーブルからデータを受け取ります:

```
EXEC SQL BEGIN DECLARE SECTION;
int intval;
varchar textval[33];
EXEC SQL END DECLARE SECTION;


/* SELECTリストに複合型の列の各要素を書く。 */
EXEC SQL DECLARE cur1 CURSOR FOR SELECT (compval).intval, (compval).textval FROM t4;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{

    /* 複合型の列の各要素をホスト変数に取り出す。 */
    EXEC SQL FETCH FROM cur1 INTO :intval, :textval;

    printf("intval=%d, textval=%s\n", intval, textval.arr);
}
```

```
EXEC SQL CLOSE cur1;
```

この例を拡張して、FETCH コマンドの値を格納するホスト変数を一つの構造体にまとめることができます。構造体の形のホスト変数の詳細については 35.4.4.3.2 を参照してください。構造体に変更するために、この例は以下のように変更することができます。二つのホスト変数 intval と textval を comp_t 構造体のメンバー変数とし、構造体を FETCH コマンドで指定します。

```
EXEC SQL BEGIN DECLARE SECTION;
typedef struct
{
    int intval;
    varchar textval[33];
} comp_t;

comp_t compval;
EXEC SQL END DECLARE SECTION;


/* SELECTリストに複合型の列の各要素を書く。 */
EXEC SQL DECLARE cur1 CURSOR FOR SELECT (compval).intval, (compval).textval FROM t4;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{

    /* SELECTリストの値をすべて1つの構造体に取り込む。 */
    EXEC SQL FETCH FROM cur1 INTO :compval;

    printf("intval=%d, textval=%s\n", compval.intval, compval.textval.arr);
}

EXEC SQL CLOSE cur1;
```

構造体が FETCH コマンドで使われていますが、属性名は SELECT 句において各々が指定されています。これは、複合型の値のすべての属性を示す * を用いることで拡張することができます。

```
...
EXEC SQL DECLARE cur1 CURSOR FOR SELECT (compval).* FROM t4;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
```

```
    /* SELECTリストの値をすべて1つの構造体に取り込む。 */
    EXEC SQL FETCH FROM cur1 INTO :compval;

    printf("intval=%d, textval=%s\n", compval.intval, compval.textval.arr);
}
...
```

この方法であれば、ECPGが複合型そのものを理解できないとしても、複合型はほぼシームレスに構造体に対応させることができます。

最後に、char[] または VARCHAR[] 型のホスト変数に外部の文字列表現として複合型の値を格納することもできます。しかし、この方法ではホストプログラムから値のフィールドにアクセスするのは簡単ではありません。

35.4.5.3. ユーザ定義の基本型

新しいユーザ定義の基本型は、ECPGでは直接的にはサポートされていません。外部の文字列表現、char[]またはVARCHAR[] 型のホスト変数を使うことができ、この解決法は多くの型について確かに適切かつ十分です。

以下に37.13に含まれるcomplex型を使った例を示します。この型の外部文字列表現は(%f,%f)で、37.13のcomplex_in()関数およびcomplex_out()関数で定義されています。以下の例は、カラムaとbに、complex型の値(1,1)および(3,3)を挿入し、その後、それらをテーブルからSELECTします。

```
EXEC SQL BEGIN DECLARE SECTION;
    varchar a[64];
    varchar b[64];
EXEC SQL END DECLARE SECTION;

    EXEC SQL INSERT INTO test_complex VALUES ('(1,1)', '(3,3)');

    EXEC SQL DECLARE cur1 CURSOR FOR SELECT a, b FROM test_complex;
    EXEC SQL OPEN cur1;

    EXEC SQL WHENEVER NOT FOUND DO BREAK;

    while (1)
    {
        EXEC SQL FETCH FROM cur1 INTO :a, :b;
        printf("a=%s, b=%s\n", a.arr, b.arr);
    }

    EXEC SQL CLOSE cur1;
```

この例は、以下の結果を示します。

```
a=(1,1), b=(3,3)
```

その他の回避方法は、ユーザ定義型をECPGにおいて直接的に使うことを避けることであり、ユーザ定義型とECPGが扱えるプリミティブ型を変換する関数またはキャストを作成することです。ただし、型のキャスト、特に暗黙のものは型システムにおいて慎重に導入されなければなりません。

例を示します。

```
CREATE FUNCTION create_complex(r double, i double) RETURNS complex
LANGUAGE SQL
IMMUTABLE
AS $$ SELECT $1 * complex '(1,0)' + $2 * complex '(0,1)' $$;
```

この定義の後、以下の例は

```
EXEC SQL BEGIN DECLARE SECTION;
double a, b, c, d;
EXEC SQL END DECLARE SECTION;

a = 1;
b = 2;
c = 3;
d = 4;

EXEC SQL INSERT INTO test_complex VALUES (create_complex(:a, :b), create_complex(:c, :d));
```

以下と同じ効果をもたらします。

```
EXEC SQL INSERT INTO test_complex VALUES ('(1,2)', '(3,4)');
```

35.4.6. 指示子

上の例ではNULL値を扱いません。実際、取り出し例では、もしデータベースからNULL値が取り出された場合にはエラーが発生します。データベースへNULL値を渡す、または、データベースからNULL値を取り出すためには、第二のホスト変数指定をデータを格納するホスト変数それぞれに追加しなければなりません。第二のホスト変数は指示子と呼ばれ、データがNULLかどうかを表すフラグが含まれます。NULLの場合、実際のホスト変数の値は無視されます。以下に、NULL値の取り出しを正しく扱う例を示します。

```
EXEC SQL BEGIN DECLARE SECTION;
VARCHAR val;
int val_ind;
EXEC SQL END DECLARE SECTION:

 ...
```

```
EXEC SQL SELECT b INTO :val :val_ind FROM test1;
```

値がNULLでなければ、指示子変数val_indは0となります。値がNULLならば負の値となります。

指示子は他の機能を持ちます。指示子の値が正ならば、値がNULLではありませんが、ホスト変数に格納する際に一部切り詰められたことを示します。

プリプロセッサ ecpg に引数 -r no_indicator が渡された場合、「no-indicator」モードで動作します。no-indicator モードでは、指示子変数が指定されなかった場合、(入力および出力において)文字列型に対して空の文字列としてnull値が、整数型に対してはもっとも小さな値が割り当てられます(例えば、int の場合 INT_MIN です)。

35.5. 動的SQL

多くの場合、アプリケーションが実行しなければならないSQL文は、アプリケーションを作成する段階で決まります。しかし、中には、SQL文が実行時に構成されることや外部ソースで提供されることがあります。このような場合、SQL文を直接Cソースコードに埋め込むことはできません。しかし、文字列変数として提供される任意のSQL文を呼び出すことができる機能が存在します。

35.5.1. 結果セットを伴わないSQL文の実行

任意のSQL文を実行するもっとも簡単な方法は、EXECUTE IMMEDIATE コマンドを使用することです。例を示します:

```
EXEC SQL BEGIN DECLARE SECTION;
const char *stmt = "CREATE TABLE test1 (...);";
EXEC SQL END DECLARE SECTION;

EXEC SQL EXECUTE IMMEDIATE :stmt;
```

EXECUTE IMMEDIATEは結果セットを返却しないSQL文(例えば、DDL、INSERT、UPDATE、DELETE)に使用することができます。結果を受け取るSQL文(例えば、SELECT)をこの方法で実行することはできません。次の節で、その実行方法を説明します。

35.5.2. 入力パラメータを伴うSQL文の実行

任意のSQL文を実行するより強力な方法は、一度プリペアをし、その後でプリペアド文を実行したいところで実行することです。また、SQL文を汎用化した形でプリペアし、パラメータを置き換えることで特定のSQL文を実行させることも可能です。SQL文をプリペアする時、後でパラメータとして置き換えたいところには疑問符を記述してください。以下に例を示します。

```
EXEC SQL BEGIN DECLARE SECTION;
const char *stmt = "INSERT INTO test1 VALUES(?, ?);";
EXEC SQL END DECLARE SECTION;
```

```
EXEC SQL PREPARE mystmt FROM :stmt;
 ...
EXEC SQL EXECUTE mystmt USING 42, 'foobar';
```

プリペアド文が必要なくなった時、割当てを解除しなければなりません。

```
EXEC SQL DEALLOCATE PREPARE name;
```

35.5.3. 結果セットを返却するSQL文の実行

単一行を編訳するSQL文を実行するには、EXECUTE を使うことができます。結果を保存するには、INTO 句を追加します。

```
EXEC SQL BEGIN DECLARE SECTION;
const char *stmt = "SELECT a, b, c FROM test1 WHERE a > ?";
int v1, v2;
VARCHAR v3[50];
EXEC SQL END DECLARE SECTION;

EXEC SQL PREPARE mystmt FROM :stmt;
 ...
EXEC SQL EXECUTE mystmt INTO :v1, :v2, :v3 USING 37;
```

EXECUTEコマンドはINTO句、USING句、この両方を持つことも、どちらも持たないこともできます。

クエリが2行以上の結果を返すことが想定される場合、以下の例のようにカーソルを使う必要があります。（カーソルの詳細については 35.3.2 を参照してください）

```
EXEC SQL BEGIN DECLARE SECTION;
char dbaname[128];
char datname[128];
char *stmt = "SELECT u.usename as dbaname, d.datname "
             "  FROM pg_database d, pg_user u "
             "  WHERE d.datdba = u.usesysid";
EXEC SQL END DECLARE SECTION;

EXEC SQL CONNECT TO testdb AS con1 USER testuser;
EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;

EXEC SQL PREPARE stmt1 FROM :stmt;

EXEC SQL DECLARE cursor1 CURSOR FOR stmt1;
EXEC SQL OPEN cursor1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;
```

```
while (1)
{
    EXEC SQL FETCH cursor1 INTO :dbaname,:datname;
    printf("dbaname=%s, datname=%s\n", dbaname, datname);
}

EXEC SQL CLOSE cursor1;

EXEC SQL COMMIT;
EXEC SQL DISCONNECT ALL;
```

35.6. pgtypes ライブラリ

pgtypesライブラリはPostgreSQLデータベースの型をCプログラムで使用できるようにC言語に対応させます。また、これらの型を使用したCの基本的な計算を行う関数も提供します。この計算には、PostgreSQLサーバを使用しません。以下の例を参照してください。

```
EXEC SQL BEGIN DECLARE SECTION;
   date date1;
   timestamp ts1, tsout;
   interval iv1;
   char *out;
EXEC SQL END DECLARE SECTION;

PGTYPESdate_today(&date1);
EXEC SQL SELECT started, duration INTO :ts1, :iv1 FROM datetbl WHERE d=:date1;
PGTYPEStimestamp_add_interval(&ts1, &iv1, &tsout);
out = PGTYPEStimestamp_to_asc(&tsout);
printf("Started + duration: %s\n", out);
PGTYPESchar_free(out);
```

35.6.1. 文字列

PGTYPESnumeric_to_ascのような一部の関数は新たに割り当てられた文字列へのポインタを返します。この結果はfreeの代わりにPGTYPESchar_freeで解放することが必要です。(これはWindows上でのみ重要です。Windowsではメモリの割り当てと解放は同じライブラリで実施されることが必要な場合があります。)

35.6.2. numeric 型

numeric 型では任意の精度での計算機能を提供します。PostgreSQLサーバにおける対応する型については8.1を参照してください。任意の精度を持つために、この変数は動的に拡張、縮小できなければなりませ

ん。これが、PGTYPESnumeric_newやPGTYPESnumeric_free関数では、ヒープ領域上にのみしか numeric 変数を作成できない理由です。decimal 型も似ていますが精度が限定されていますので、ヒープ領域以外にスタック領域上でも作成可能です。

以下の関数は numeric 型で使用することができます:

PGTYPESnumeric_new

新規割当ての numeric 型へのポインタを要求します。

```
numeric *PGTYPESnumeric_new(void);
```

PGTYPESnumeric_free

numeric 型を解放し、そのメモリをすべて解放します。

```
void PGTYPESnumeric_free(numeric *var);
```

PGTYPESnumeric_from_asc

文字列表記から numeric 型に変換します。

```
numeric *PGTYPESnumeric_from_asc(char *str, char **endptr);
```

有効な書式の例を示します。-2、.794、+3.44、592.49E07、-32.84e-4。値への変換に成功した場合、有効なポインタが返されます。失敗した場合は NULL ポインタが返されます。現在ECPGは文字列全体を解析しますので、現時点では*endptr内に最初の無効な文字のアドレスを格納することをサポートしません。このためendptrを安全に NULL にすることができます。

PGTYPESnumeric_to_asc

numeric 型numの文字列表現を持つ、mallocで割り当てられた文字列へのポインタを返します。

```
char *PGTYPESnumeric_to_asc(numeric *num, int dscale);
```

numeric の値は、必要に応じて四捨五入され、dscale 桁の十進数で出力されます。結果はPGTYPESchar_free()で解放しなければなりません。

PGTYPESnumeric_add

2つの numeric 変数を加算し、3番目の numeric 変数に格納します。

```
int PGTYPESnumeric_add(numeric *var1, numeric *var2, numeric *result);
```

この関数は変数var1と変数var2を加算し、その結果をresultに格納します。成功時0を、エラー時-1を返します。

PGTYPESnumeric_sub

2つの numeric 型変数を減算し、3番目の numeric 型変数に結果を格納します。

```
int PGTYPESnumeric_sub(numeric *var1, numeric *var2, numeric *result);
```

この関数は変数var1から変数var2を差し引きます。演算結果を変数resultに格納します。成功時0を、エラー時-1を返します。

PGTYPESnumeric_mul

2つの numeric 型変数を乗算し、3番目の numeric 型変数に結果を格納します。

```
int PGTYPESnumeric_mul(numeric *var1, numeric *var2, numeric *result);
```

この関数は変数var1と変数var2を掛け合わせます。演算結果を変数resultに格納します。成功時0を、エラー時-1を返します。

PGTYPESnumeric_div

2つの numeric 型変数で除算し、3番目の numeric 型変数に結果を格納します。

```
int PGTYPESnumeric_div(numeric *var1, numeric *var2, numeric *result);
```

この関数は変数var1を変数var2で割ります。演算結果を変数resultに格納します。成功時0を、エラー時-1を返します。

PGTYPESnumeric_cmp

2つのnumeric型変数を比較します。

```
int PGTYPESnumeric_cmp(numeric *var1, numeric *var2)
```

この関数は2つのnumeric型変数を比較します。エラーの場合INT_MAXが返ります。成功時、この関数は以下のいずれかを返します:

- var1がvar2より大きければ1。
- var1がvar2より小さければ-1。
- var1がvar2と等しければ0。

PGTYPESnumeric_from_int

int型変数をnumeric型変数に変換します。

```
int PGTYPESnumeric_from_int(signed int int_val, numeric *var);
```

この関数はsigned int型の変数を受付け、numeric型変数var内に格納します。成功時0、失敗時-1が返ります。

PGTYPESnumeric_from_long

long int型変数をnumeric型変数に変換します。

```
int PGTYPESnumeric_from_long(signed long int long_val, numeric *var);
```

この関数は符号付long int型の変数を受付け、numeric 型変数var内に格納します。成功時0、失敗時-1が返ります。

PGTYPESnumeric_copy

numeric 型変数を他の numeric 型変数にコピーします。

```
int PGTYPESnumeric_copy(numeric *src, numeric *dst);
```

この関数は、srcが指し示す変数の値をdstが指し示す変数にコピーします。成功時0、失敗時-1が返ります。

PGTYPESnumeric_from_double

double型の変数を numeric 型変数に変換します。

```
int  PGTYPESnumeric_from_double(double d, numeric *dst);
```

この関数はdouble型の変数を受付け、変換結果をdstが指し示す変数内に格納します。成功時0、失敗時-1が返ります。

PGTYPESnumeric_to_double

numeric 型変数をdouble型に変換します。

```
int PGTYPESnumeric_to_double(numeric *nv, double *dp)
```

この関数はnvが指し示す numeric 型変数の値をdpが指し示すdouble型変数に変換します。成功時0、オーバーフローを含むエラーが発生した時-1が返ります。オーバーフローが発生した場合はさらに、グローバル変数errnoはPGTYPES_NUM_OVERFLOWに設定されます。

PGTYPESnumeric_to_int

numeric 型変数をint型に変換します。

```
int PGTYPESnumeric_to_int(numeric *nv, int *ip);
```

この関数はnvが指し示す numeric 型変数の値をipが指し示す整数型変数に変換します。成功時0、オーバーフローを含むエラーが発生した時-1が返ります。オーバーフローが発生した場合はさらに、グローバル変数errnoはPGTYPES_NUM_OVERFLOWに設定されます。

PGTYPESnumeric_to_long

numeric 型変数をlong型に変換します。

```
int PGTYPESnumeric_to_long(numeric *nv, long *lp);
```

この関数はnvが指し示す numeric 型変数の値をlpが指し示すlong変数に変換します。成功時0、オーバーフローを含むエラーが発生した時-1が返ります。オーバーフローが発生した場合はさらに、グローバル変数errnoはPGTYPES_NUM_OVERFLOWに設定されます。

PGTYPESnumeric_to_decimal

numeric 型変数を decimal 型に変換します。

```
int PGTYPESnumeric_to_decimal(numeric *src, decimal *dst);
```

この関数はsrcが指し示す numeric 型変数の値をdstが指し示す decimal 型変数に変換します。成功時0、オーバーフローを含むエラーが発生した時-1が返ります。オーバーフローが発生した場合はさらに、グローバル変数errnoはPGTYPES_NUM_OVERFLOWに設定されます。

PGTYPESnumeric_from_decimal

decimal 型変数を numeric 型に変換します。

```
int PGTYPESnumeric_from_decimal(decimal *src, numeric *dst);
```

この関数はsrcが指し示す decimal 型変数の値をdstが指し示す numeric 型変数に変換します。成功時0、エラーが発生した時-1が返ります。decimal 型は制限付の numeric 型として実装されていますので、この変換ではオーバーフローは起きません。

35.6.3. 日付型

Cの日付型を使用して、プログラムからSQLの日付型を取り扱うことができます。PostgreSQLサーバにおける対応する型については8.5を参照してください。

日付型を操作するために以下の関数を使用することができます:

PGTYPESdate_from_timestamp

タイムスタンプから日付部分を取り出します。

```
date PGTYPESdate_from_timestamp(timestamp dt);
```

この関数は唯一の引数としてタイムスタンプを受付け、そこから日付部分を取り出します。

PGTYPESdate_from_asc

テキスト表現から日付型に変換します。

```
date PGTYPESdate_from_asc(char *str, char **endptr);
```

この関数はCのchar*型文字列strとCのchar*型文字列endptrへのポインタを受付けます。現在ECPGは文字列全体を解析しますので、現時点では*endptrに最初の無効な文字のアドレスを格納することをサポートしません。このためendptrを安全にNULLにすることができます。

この関数が常にMDY書式の日付を前提としている点に注意してください。現在ECPGにはこれを変更するための変数がありません。

表 35.2に許される入力書式を示します。

表35.2 有効なPGTYPESdate_from_ascの入力書式

入力	結果
January 8, 1999	January 8, 1999
1999-01-08	January 8, 1999
1/8/1999	January 8, 1999
1/18/1999	January 18, 1999
01/02/03	February 1, 2003
1999-Jan-08	January 8, 1999
Jan-08-1999	January 8, 1999
08-Jan-1999	January 8, 1999
99-Jan-08	January 8, 1999
08-Jan-99	January 8, 1999
08-Jan-06	January 8, 2006
Jan-08-99	January 8, 1999
19990108	ISO 8601; January 8, 1999
990108	ISO 8601; January 8, 1999
1999.008	年と年内日数
J2451187	Jユリウス日
January 8, 99 BC	紀元前99年

PGTYPESdate_to_asc

日付型変数のテキスト表現を返します。

```
char *PGTYPESdate_to_asc(date dDate);
```

この関数は唯一の引数として日付型dDateを受付けます。この関数は1999-01-18、つまりYYYY-MM-DD書式で日付を出力します。結果はPGTYPESchar_free()で解放しなければなりません。

PGTYPESdate_julmdy

日付型の変数から、日、月、年の値を取り出します。

```
void PGTYPESdate_julmdy(date d, int *mdy);
```

この関数は日付型のdと、3つの整数値を持つ配列mdyへのポインタを受付けます。この変数名はその並びを表し、mdy[0]には月数、mdy[1]には日数が、mdy[2]には年が入ります。

PGTYPESdate_mdyjul

日付の日、月、年を指定した3つの整数からなる配列から日付値を作成します。

```
void PGTYPESdate_mdyjul(int *mdy, date *jdate);
```

この関数は、1番目の引数として3つの整数からなる配列(mdy)、2番目の引数として処理結果を格納する日付型の変数へのポインタを受付けます。

PGTYPESdate_dayofweek

日付値から週内日数を表す数を返します。

```
int PGTYPESdate_dayofweek(date d);
```

この関数は唯一の引数としてdate型変数dを受付け、その日付の週内日数を表す整数を返します。

- 0 - 日曜
- 1 - 月曜
- 2 - 火曜
- 3 - 水曜
- 4 - 木曜
- 5 - 金曜
- 6 - 土曜

PGTYPESdate_today

現在の日付を取得します。

```
void PGTYPESdate_today(date *d);
```

この関数は現在の日付に設定されるdate型変数(d)を指し示すポインタを受付けます。

PGTYPESdate_fmt_asc

date型変数を書式マスクを使用したテキスト表現に変換します。

```
int PGTYPESdate_fmt_asc(date dDate, char *fmtstring, char *outbuf);
```

この関数は変換対象のdate型(dDate)、書式マスク(fmtstring)、日付のテキスト表現を格納するための文字列(outbuf)を受付けます。

成功時に0、エラーが発生した場合は負の値が返ります。

以下のリテラルを使用して、フィールドを指定することができます。

- dd - 月内の日数。
- mm - 年内の月数。
- yy - 二桁表記の年数
- yyyy - 四桁表記の年数
- ddd - 曜日の名前（省略形）
- mmm - 月の名前（省略形）

他の文字はすべて出力文字列にそのままコピーされます。

表 35.3にいくつかの可能性のある書式を示します。この関数の使用方法に関するアイディアを提供しています。出力行はすべて同じ日付、1959年11月23日に基づいています。

表35.3 有効なPGTYPESdate_fmt_ascの入力書式

書式	結果
mmddyy	112359
ddmmyy	231159
yymmdd	591123
yy/mm/dd	59/11/23
yy mm dd	59 11 23
yy.mm.dd	59.11.23
.mm.yyyy.dd.	.11.1959.23.
mmm. dd, yyyy	Nov. 23, 1959
mmm dd yyyy	Nov 23 1959
yyyy dd mm	1959 23 11
ddd, mmm. dd, yyyy	Mon, Nov. 23, 1959
(ddd) mmm. dd, yyyy	(Mon) Nov. 23, 1959

PGTYPESdate_defmt_asc

書式マスクを使用してCのchar*文字列からdate型の値に変換します。

```
int PGTYPESdate_defmt_asc(date *d, char *fmt, char *str);
```

この関数は、処理結果を格納するための日付型へのポインタ(d)、日付を解析するための書式マスク(fmt)、日付のテキスト表現を含むCのchar*文字列(str)を受付けます。テキスト表現は書式マスクに合った表現であることが仮定されています。しかし、文字列と書式マスクを1:1に対応付けする必要はありません。この関数は並んだ順番に解析し、年の位置を表すyyまたはyyyyを、月の位置を表すmmを、日の位置を表すddを検索します。

表 35.4はいくつかの可能性のある書式を示します。これはこの関数の使用方法に関するアイディアを提供します。

表35.4 有効なrdefmtdateの入力書式

書式	文字列	結果
ddmmyy	21-2-54	1954-02-21
ddmmyy	2-12-54	1954-12-02
ddmmyy	20111954	1954-11-20
ddmmyy	130464	1964-04-13
mmm.dd.yyyy	MAR-12-1967	1967-03-12
yy/mm/dd	1954, February 3rd	1954-02-03
mmm.dd.yyyy	041269	1969-04-12
yy/mm/dd	In the year 2525, in the month of July, mankind will be alive on the 28th day	2525-07-28
dd-mm-yy	I said on the 28th of July in the year 2525	2525-07-28
mmm.dd.yyyy	9/14/58	1958-09-14
yy/mm/dd	47/03/29	1947-03-29
mmm.dd.yyyy	oct 28 1975	1975-10-28
mmddyy	Nov 14th, 1985	1985-11-14

35.6.4. timestamp型

Cのタイムスタンプ型を使用してプログラムからSQLのタイムスタンプ型データを扱うことができます。PostgreSQLにおける対応する型については8.5を参照してください。

以下の関数を使用してタイムスタンプ型を扱うことができます:

PGTYPEStimestamp_from_asc

テキスト表現のタイムスタンプをタイムスタンプ型変数に変換します。

```
timestamp PGTYPEStimestamp_from_asc(char *str, char **endptr);
```

この関数は変換対象の文字列(str)とC char*へのポインタ(endptr)を受付けます。現在ECPGは文字列全体を解析しますので、現時点では*endptrに最初の無効な文字の場所を格納をすることサポートしません。このためendptrを安全に NULL にすることができます。

この関数は成功時変換後のタイムスタンプを返します。エラー時、PGTYPESInvalidTimestampが返され、errnoがPGTYPES_TS_BAD_TIMESTAMPに設定されます。この値についての重要な注意書きについてPGTYPESInvalidTimestampを参照してください。

通常、入力文字列には許される日付指定の任意の組み合わせ、空白文字、許される時間指定を含むことができます。時間帯はECPGでサポートされていない点に注意してください。変換することはできます

が、例えばPostgreSQLサーバが行うような計算を行うことはできません。時間帯指定は警告無しに無視されます。

表 35.5に入力文字列の例をいくつか示します。

表35.5 有効なPGTYPEStimestamp_from_ascの入力書式

入力	結果
1999-01-08 04:05:06	1999-01-08 04:05:06
January 8 04:05:06 1999 PST	1999-01-08 04:05:06
1999-Jan-08 04:05:06.789-8	1999-01-08 04:05:06.789 (時間帯指定は無視されます。)
J2451187 04:05-08:00	1999-01-08 04:05:00 (時間帯指定は無視されます。)

PGTYPEStimestamp_to_asc

date型をC char*文字列に変換します。

```
char *PGTYPEStimestamp_to_asc(timestamp tstamp);
```

この関数はtimestamp型のtstampを唯一の引数として受付け、timestamp型のテキスト表現を含む割り当てられた文字列を返します。結果はPGTYPESchar_free()で解放しなければなりません。

PGTYPEStimestamp_current

現在のタイムスタンプを取り出します。

```
void PGTYPEStimestamp_current(timestamp *ts);
```

この関数は現在のタイムスタンプを取り出し、tsが指し示すtimestamp型変数に格納します。

PGTYPEStimestamp_fmt_asc

書式マスクを使用してtimestamp型変数をC char*に変換します。

```
int PGTYPEStimestamp_fmt_asc(timestamp *ts, char *output, int str_len, char *fmtstr);
```

この関数は、最初の引数として変換対象のtimestamp型 (ts)を、出力バッファのポインタ(output)、出力バッファで割当て可能な最大長 (str_len)、変換に使用する書式マスク(fmtstr)を受付けます。

成功するとこの関数は0を返します。エラーが発生した場合は負の値が返ります。

書式マスクには以下の書式指定を使用することができます。書式指定はlibcのstrftime関数で使用されるものと同じです。書式指定以外は出力バッファにコピーされます。

- %A - 各言語の曜日名称に置換されます。
- %a - 各言語の曜日略称に置換されます。
- %B - 各言語の月名称に置換されます。

- %b - 各言語の月略称に置換されます。
- %C - 年を100で割った10進数に置換されます。1桁の場合は先頭に0が付与されます。
- %c - 各言語の日付時刻表現に置換されます。
- %D - %m/%d/%yと同じです。
- %d - 月内の10進日数(01–31)に置換されます。
- %E* %O* - POSIXロケール拡張です。%Ec %EC %Ex %EX %Ey %EY %Od %Oe %OH %OI %Om %OM %OS %Ou %OU %OV %Ow %OW %Oy という並びは別の表現を提供するものと仮定されています。

  さらに、%OBは、（日に関する仕様がない単体で使用される）別の月名を表すものとして実装されています。
- %e - 月内10進日数(1–31)に置換されます。1桁の場合は前に空白が付けられます。
- %F - %Y-%m-%dと同じです。
- %G - 世紀付の10進数として年に置換されます。この年は週の部分がより多く含まれます。（月曜が週の最初の日です。）
- %g -%G同様に年に置換されますが、世紀の部分を除く10進数(00–99)になります。
- %H - 10進の時間(24時間単位)に置換されます(00–23)。
- %h - %bと同じです。
- %I - 10進の時間（12時間単位）に置換されます(01–12)。
- %j - 10進の年内日数に置換されます(001–366)。
- %k - 10進の時間(24時間単位)に置換されます(0–23)。1桁の場合は先頭に空白が付けられます。
- %l - 10進の時間（12時間単位）に置換されます(1–12)。1桁の場合は先頭に空白が付けられます。
- %M - 10進の分数に置換されます(00–59)。
- %m -10進の月数に置換されます(01–12)。
- %n - 改行に置換されます。
- %O* - %E*と同じです。
- %p - 各言語の「午前」または「午後」に適切に置換されます。
- %R - %H:%Mと同じです。
- %r - %I:%M:%S %pと同じです。
- %S - 10進の秒数に置換されます(00–60)。
- %s - エポック、UTCからの秒数に置換されます。

- %T - %H:%M:%Sと同じです。
- %t - タブに置換されます。
- %U - 10進の週番号（日曜が週の先頭です）に置換されます(00–53)。
- %u - 10進の週番号（月曜が週の先頭です）に置換されます(1–7)。
- %V - 10進の年内の週番号（月曜が週の先頭です）に置換されます(01–53)。新しい年で、1月1日を含む週が4日以上存在する場合、その週が1となります。さもなくば、この週は前年の週となり、次の週が1となります。
- %v - %e-%b-%Yと同じです。
- %W - 10進の年内の週番号（月曜が週の先頭です）に置換されます(00–53)。
- %w - 10進の週内日数（日曜が週の先頭です）に置換されます(0–6)。
- %X - 各言語の時間表現に置換されます。
- %x - 各言語の日付表現に置換されます。
- %Y - 10進の世紀付年に置換されます。
- %y - 10進の世紀なし年に置換されます(00–99)。
- %Z - 時間帯名称に置換されます。
- %z - UTCからの時間帯オフセットに置換されます。UTCより東では正符号が先頭に付き、西では負符号が付きます。それぞれ2桁の時間と分がその後に続きますが、その区切りはありません。（この形式はRFC 822の日付ヘッダでよく使用されます。）
- %+ - 各言語の日付時刻表現に置換されます。
- %-* - GNU libc拡張です。数値出力を行う際に何も文字を詰めません。
- $_* - GNU libcの拡張です。明示的に空白文字を使用して文字を詰めます。
- %0* - GNU libcの拡張です。明示的に0を使用して文字を詰めます。
- %% - %に置換されます。

PGTYPESTimestamp_sub

タイムスタンプの減算を行い、その結果をinterval型の変数に格納します。

```
int PGTYPESTimestamp_sub(timestamp *ts1, timestamp *ts2, interval *iv);
```

この関数はts1が指し示すタイムスタンプ型変数からts2が指し示すタイムスタンプ型変数を差し引き、ivが指し示すinterval型変数に結果を格納します。

成功すると、この関数は0を返し、エラーが発生した場合は負の値を返します。

PGTYPEStimestamp_defmt_asc

書式マスクを使用して、テキスト表現からtimestamp値へ変換します。

```
int PGTYPEStimestamp_defmt_asc(char *str, char *fmt, timestamp *d);
```

この関数はstr変数内に格納されたタイムスタンプのテキスト表現、fmt変数内に格納された使用される書式マスクを受付けます。結果はdが指し示す変数内に格納されます。

書式マスクfmtが NULL ならば、この関数はデフォルトの書式マスク%Y-%m-%d %H:%M:%Sを使用するようになります。

これはPGTYPEStimestamp_fmt_asc関数の逆です。使用できる書式マスク項目についてはその文書を参照してください。

PGTYPEStimestamp_add_interval

timestamp型変数にinterval型変数を加算します。

```
int PGTYPEStimestamp_add_interval(timestamp *tin, interval *span, timestamp *tout);
```

この関数はtimestamp型変数tinへのポインタとinterval型変数spanへのポインタを受付けます。これは、interval値をtimestamp値に加算し、その結果のtimestamp値をtoutが指し示す変数に格納します。

成功するとこの関数は0を返します。エラーが発生した場合は負の値を返します。

PGTYPEStimestamp_sub_interval

timestamp型変数からinterval型変数の値を引きます。

```
int PGTYPEStimestamp_sub_interval(timestamp *tin, interval *span, timestamp *tout);
```

この関数はtinが指し示すtimestamp型変数からspanが指し示すinterval型変数を引きます。結果はtoutが指し示す変数に保存されます。

成功するとこの関数は0を、エラーが発生した場合は負の値を返します。

35.6.5. interval型

Cにおけるinterval型を用いることにより、プログラムからSQLのinterval型のデータを扱うことができます。PostgreSQLサーバにおける対応する型については8.5を参照してください。

以下の関数を使用して、interval型を扱うことができます。

PGTYPESinterval_new

新しく割り当てたinterval型変数へのポインタを返します。

```
interval *PGTYPESinterval_new(void);
```

PGTYPESinterval_free

以前に割り当てられたinterval型変数のメモリを解放します。

```
void PGTYPESinterval_free(interval *intvl);
```

PGTYPESinterval_from_asc

テキスト表現からinterval型に変換します。

```
interval *PGTYPESinterval_from_asc(char *str, char **endptr);
```

この関数は入力文字列strを変換し、割当てられたinterval型へのポインタを返します。現在ECPGは文字列全体を解析しますので、現時点では*endptrに最初の無効な文字のアドレスを格納することをサポートしません。このためendptrを安全に NULL にすることができます。

PGTYPESinterval_to_asc

interval型変数をテキスト表現に変換します。

```
char *PGTYPESinterval_to_asc(interval *span);
```

この関数はspanが指し示すinterval型変数をC char*に変換します。出力は@ 1 day 12 hours 59 mins 10 secsのようになります。結果はPGTYPESchar_free()で解放しなければなりません。

PGTYPESinterval_copy

interval型変数をコピーします。

```
int PGTYPESinterval_copy(interval *intvlsrc, interval *intvldest);
```

この関数は、intvlsrcが指し示すinterval型変数を intvldestが指し示す変数にコピーします。事前に格納先の変数用のメモリを割り当てる必要があることに注意してください。

35.6.6. decimal型

decimal型はnumeric型に似ています。しかし、その最大精度は30有効桁に制限されています。ヒープ上にしか作成できないnumeric型と比べ、decimal型はスタックまたはヒープ上に作成することができます。(このためにはPGTYPESdecimal_newおよびPGTYPESdecimal_free関数を使用します。) 35.15で説明するInformix互換モードではdecimal型を扱う関数がより多く存在します。

以下の関数を使用してdecimal型を扱うことができます。これらはlibcompatライブラリに含まれるものだけではありません。

PGTYPESdecimal_new

新しく割り当てられたdecimal型変数へのポインタを要求します。

```
decimal *PGTYPESdecimal_new(void);
```

PGTYPESdecimal_free

decimal型を解放し、そのメモリをすべて解放します。

```
void PGTYPESdecimal_free(decimal *var);
```

35.6.7. pgtypeslibのerrno値

PGTYPES_NUM_BAD_NUMERIC

引数はnumeric型変数（またはnumeric型変数へのポインタ）を含んでいるはずですが、実際のメモリ上の表現は無効でした。

PGTYPES_NUM_OVERFLOW

オーバーフローが発生しました。numeric型はほぼ任意の精度を扱うことができますので、numeric型変数から他の型への変換ではオーバーフローが発生する可能性があります。

PGTYPES_NUM_UNDERFLOW

アンダーフローが発生しました。numeric型はほぼ任意の精度を扱うことができますので、numeric型変数から他の型への変換ではアンダーフローが発生する可能性があります。

PGTYPES_NUM_DIVIDE_ZERO

ゼロ除算をしようとしました。

PGTYPES_DATE_BAD_DATE

PGTYPESdate_from_asc関数に無効な日付文字列が渡されました。

PGTYPES_DATE_ERR_EARGS

PGTYPESdate_defmt_asc関数に無効な引数が渡されました。

PGTYPES_DATE_ERR_ENOSHORTDATE

PGTYPESdate_defmt_asc関数により入力文字列内に無効なトークンが見つかりました。

PGTYPES_INTVL_BAD_INTERVAL

PGTYPESinterval_from_asc関数に無効な内部文字列が渡されました。もしくはPGTYPESinterval_to_asc関数に無効な内部値が渡されました。

PGTYPES_DATE_ERR_ENOTDMY

PGTYPESdate_defmt_asc関数内の日/月/年の代入において不整合がありました。

PGTYPES_DATE_BAD_DAY

PGTYPESdate_defmt_asc関数により無効な月内日数が見つかりました。

PGTYPES_DATE_BAD_MONTH

PGTYPESdate_defmt_asc関数によって無効な月値が見つかりました。

PGTYPES_TS_BAD_TIMESTAMP

PGTYPEStimestamp_from_asc関数に無効なタイムスタンプ文字列が渡されました。もしくはPGTYPEStimestamp_to_asc関数に無効なtimestamp値が渡されました。

PGTYPES_TS_ERR_EINFTIME

コンテキスト内で扱うことができない、無限なタイムスタンプ値がありました。

35.6.8. pgtypeslibの特殊な定数

PGTYPESInvalidTimestamp

無効なタイムスタンプを表すtimestamp型の値です。これは解析エラーの場合にPGTYPEStimestamp_from_asc関数によって返されます。timestampデータ型の内部表現のため、PGTYPESInvalidTimestampはまた同時に有効なタイムスタンプでもあります。これは1899-12-31 23:59:59に設定されます。エラーを検知するためには、PGTYPEStimestamp_from_ascを呼び出す度にその後、PGTYPESInvalidTimestampを試験するだけではなく、errno != 0も試験してください。

35.7. 記述子領域の使用

SQL記述子領域はSELECT、FETCH、DESCRIBE文の結果を処理する、より洗練された手法です。SQL記述子領域は1行のデータをメタデータ項目と一緒に1つのデータ構造体としてグループ化します。特に動的SQL文を実行する場合は結果列の性質が前もってわかりませんので、メタデータが有用です。PostgreSQLは記述子領域を使用するための2つの方法、名前付きSQL記述子領域とC構造化SQLDA、を提供します。

35.7.1. 名前付きSQL記述子領域

名前付きSQL記述子領域は、記述子全体に関する情報を持つヘッダと、基本的に結果行内の1つの列を記述する、1つ以上の項目記述子領域から構成されます。

SQL記述子領域を使用可能にするためには、それを以下のように割り当てなければなりません。

```
EXEC SQL ALLOCATE DESCRIPTOR identifier;
```

この識別子は記述子領域の「変数名」として使用されます。記述子が不要になったら、以下のように解放してください。

```
EXEC SQL DEALLOCATE DESCRIPTOR identifier;
```

記述子領域を使用するには、INTO句内の格納対象として、ホスト変数を列挙するのではなく、記述子領域を指定してください。

```
EXEC SQL FETCH NEXT FROM mycursor INTO SQL DESCRIPTOR mydesc;
```

結果セットが空の場合であっても、記述子領域には問い合わせのメタデータ、つまりフィールド名、が含まれます。

まだ実行されていないプリペアド問い合わせでは、結果セットのメタデータを入手するためにDESCRIBEを使用することができます。

```
EXEC SQL BEGIN DECLARE SECTION;
char *sql_stmt = "SELECT * FROM table1";
EXEC SQL END DECLARE SECTION;

EXEC SQL PREPARE stmt1 FROM :sql_stmt;
EXEC SQL DESCRIBE stmt1 INTO SQL DESCRIPTOR mydesc;
```

PostgreSQL 9.0より前では、SQLキーワードは省略可能でした。このためDESCRIPTORおよびSQL DESCRIPTORは名前付きSQL記述子領域を生成しました。これは強制事項になり、SQLキーワードを省略すると、SQLDA記述子領域を生成します。35.7.2を参照してください。

DESCRIBEおよびFETCH文では、INTOおよびUSINGキーワードを同じように使用することができます。これらは結果セットと記述子領域内のメタデータを生成します。

さて、どうやって記述子領域からデータを取り出すのでしょうか。この記述子領域を名前付きフィールドを持つ構造体とみなすことができます。ヘッダからフィールド値を取り出し、それをホスト変数に格納するには、以下のコマンドを使用します。

```
EXEC SQL GET DESCRIPTOR name :hostvar = field;
```

今のところ、COUNTというヘッダフィールドが1つだけ定義されています。これは、記述子領域に存在する項目数を表すものです（つまり、結果内に含まれる列数です）。このホスト変数は整数型でなければなりません。項目記述子領域からフィールドを取り出すには、以下のコマンドを使用します。

```
EXEC SQL GET DESCRIPTOR name VALUE num :hostvar = field;
```

numはリテラル整数、もしくは整数を持つホスト変数を取ることができます。取り得るフィールドは以下の通りです。

CARDINALITY (整数)

結果セット内の行数です。

DATA

実際のデータ項目です（したがってこのフィールドのデータ型は問い合わせに依存します）。

DATETIME_INTERVAL_CODE (整数)

TYPEが9の場合、DATETIME_INTERVAL_CODEは、DATEでは1、TIMEでは2、TIMESTAMPでは3、TIME WITH TIME ZONEでは4、TIMESTAMP WITH TIME ZONEでは5という値を取ります。

DATETIME_INTERVAL_PRECISION (整数)

未実装です。

INDICATOR (整数)

(NULL値や値の切り詰めを示す)指示子です。

KEY_MEMBER (整数)

実装されていません。

LENGTH (整数)

データの文字列の長さです。

NAME (文字列)

列名です。

NULLABLE (整数)

実装されていません。

OCTET_LENGTH (整数)

データの文字表現のバイト長です。

PRECISION (整数)

(numeric型用の)精度です。

RETURNED_LENGTH (整数)

データの文字数です。

RETURNED_OCTET_LENGTH (整数)

データの文字表現のバイト長です。

SCALE (整数)

(numeric型用の)桁です。

TYPE (整数)

列のデータ型の数値コードです。

EXECUTE、DECLAREおよびOPEN文では、INTOおよびUSINGの効果は異なります。また、問い合わせやカーソル用の入力パラメータを提供するために記述子領域は手作業で構築することができます。USING SQL

DESCRIPTOR nameは入力パラメータとパラメータ付きの問い合わせに渡す方法です。名前付きSQL記述子領域を構築するSQL文は以下の通りです。

```
EXEC SQL SET DESCRIPTOR name VALUE num field = :hostvar;
```

PostgreSQLは、1つのFETCH文内の1レコードを複数取り出し、ホスト変数に格納することをサポートします。この場合ホスト変数は配列であると仮定されます。

```
EXEC SQL BEGIN DECLARE SECTION;
int id[5];
EXEC SQL END DECLARE SECTION;

EXEC SQL FETCH 5 FROM mycursor INTO SQL DESCRIPTOR mydesc;

EXEC SQL GET DESCRIPTOR mydesc VALUE 1 :id = DATA;
```

35.7.2. SQLDA記述子領域

SQLDA記述子領域は、問い合わせの結果セットとメタデータを取り出すために使用可能なC言語の構造体です。1つの構造体には結果セットの1レコードが格納されます。

```
EXEC SQL include sqlda.h;
sqlda_t         *mysqlda;

EXEC SQL FETCH 3 FROM mycursor INTO DESCRIPTOR mysqlda;
```

SQLキーワードが省略されていることに注意してください。35.7.1のINTOおよびUSINGの使用状況に関する段落はここで多少追加して適用します。DESCRIBE文では、INTOが使用されている場合DESCRIPTORキーワードは完全に省略可能です。

```
EXEC SQL DESCRIBE prepared_statement INTO mysqlda;
```

SQLDAを使用するプログラムの一般的な流れは以下の通りです。

1. 問い合わせをプリペアし、そのカーソルを宣言します。
2. 結果セット用のSQLDAを宣言します。
3. 入力パラメータ用のSQLDAを宣言し、初期化（メモリ割り当て、パラメータの設定）します。
4. 入力用SQLDAでカーソルを開きます。
5. カーソルから行を取り出し、出力用SQLDAに格納します。
6. 出力用SQLDAから値をホスト変数に（必要に応じて変換を行い）読み取ります。

7. カーソルを閉じます。

8. 入力用SQLDAに割り当てられたメモリ領域を解放します。

35.7.2.1. SQLDAのデータ構造

SQLDAはsqlda_t、sqlvar_t、struct sqlnameという3つの種類のデータ構造を使用します。

> **ヒント**
>
> PostgreSQLのSQLDAはIBM DB2ユニバーサルデータベースのものと似たデータ構造を持ちます。このため、DB2のSQLDAに関する技術情報の一部はPostgreSQLのSQLDAの理解のより良い助けになるでしょう。

35.7.2.1.1. sqlda_t構造体

sqlda_t構造体は実際のSQLDAの型です。これは1つのレコードを保持します。そして2つ以上のsqlda_t構造体をdesc_nextフィールド内においてポインタを使ってリンクリスト内でつなげることができます。こうして行の順序付き集合を表現します。このため、2つ以上の行を取り出す時、アプリケーションは各sqlda_tノードのdesc_nextポインタを追うことでそれらを読み取ることができます。

sqlda_tの定義は以下の通りです。

```
struct sqlda_struct
{
    char            sqldaid[8];
    long            sqldabc;
    short           sqln;
    short           sqld;
    struct sqlda_struct *desc_next;
    struct sqlvar_struct sqlvar[1];
};

typedef struct sqlda_struct sqlda_t;
```

フィールドの意味は以下の通りです。

sqldaid

ここには"SQLDA "文字列リテラルが含まれます。

sqldabc

ここにはバイト単位の割り当てられた領域のサイズが含まれます。

sqln

USINGキーワードを使用してOPEN、DECLARE、EXECUTE文に渡される場合、ここにはパラメータ付き問い合わせの入力パラメータ数が含まれます。SELECT、EXECUTE、FETCH文の出力として使用される場合、この値はsqld文と同じです。

sqld

ここには結果セットのフィールド数が含まれます。

desc_next

問い合わせが複数のレコードを返す場合、複数結び付いたSQLDA構造体が返されます。desc_nextにリスト内の次の項目を指し示すポインタが保持されます。

sqlvar

これは結果セット内の列の配列です。

35.7.2.1.2. sqlvar_t構造体

sqlvar_t構造体は列の値と型や長さなどのメタデータを保持します。この型の定義は以下の通りです。

```
struct sqlvar_struct
{
    short          sqltype;
    short          sqllen;
    char          *sqldata;
    short         *sqlind;
    struct sqlname sqlname;
};

typedef struct sqlvar_struct sqlvar_t;
```

フィールドの意味は以下の通りです。

sqltype

ここにはフィールドの型識別子が含まれます。値についてはecpgtype.hのenum ECPGttypeを参照してください。

sqllen

フィールドのバイナリ長が含まれます。例えばECPGt_intでは4バイトです。

sqldata

データそのものを指し示します。データ書式は35.4.4で説明します。

sqlind

データのNULL指示子を指し示します。0は非NULLを、-1はNULLを意味します。

sqlname

フィールドの名前です。

35.7.2.1.3. struct sqlname構造体

struct sqlname構造体は列名を保持します。sqlvar_t構造体のメンバとして使用されます。構造体の定義は以下の通りです。

```
#define NAMEDATALEN 64

struct sqlname
{
        short           length;
        char            data[NAMEDATALEN];
};
```

フィールドの意味は以下の通りです。

length

フィールド名の長さが含まれます。

data

実際のフィールド名が含まれます。

35.7.2.2. SQLDAを使用した結果セットの取り出し

SQLDAを通して問い合わせの結果を取り出す一般的な手順は以下に示します。

1. 結果セットを受けとるためのsqlda_t構造体を宣言します。
2. 宣言したSQLDAを指定した問い合わせを処理するためにFETCH/EXECUTE/DESCRIBEを実行します。
3. sqlda_t構造体のメンバsqlnを検索することにより結果セット内のレコード数を検査します。
4. sqlda_t構造体のメンバsqlvar[0]、sqlvar[1]などから各列の値を入手します。
5. sqlda_t構造体のメンバdesc_nextポインタを追い、次の行（sqlda_t構造体）に進みます。
6. 必要なだけ上を繰り返します。

以下にSQLDAを通して結果セットを取り出す例を示します。

まず、結果セットを受け取るsqlda_t構造体を宣言します。

```
sqlda_t *sqlda1;
```

次にコマンド内にSQLDAを指定します。以下はFETCHコマンドの例です。

```
EXEC SQL FETCH NEXT FROM cur1 INTO DESCRIPTOR sqlda1;
```

行を取り出すためにリンクリストを追うループを実行します。

```
sqlda_t *cur_sqlda;

for (cur_sqlda = sqlda1;
     cur_sqlda != NULL;
     cur_sqlda = cur_sqlda->desc_next)
{
    ...
}
```

ループの内側では、行の列データ（sqlvar_t構造体）それぞれを取り出す別のループを実行します。

```
for (i = 0; i < cur_sqlda->sqld; i++)
{
    sqlvar_t v = cur_sqlda->sqlvar[i];
    char *sqldata = v.sqldata;
    short sqllen  = v.sqllen;
    ...
}
```

列の値を入手するために、sqlvar_t構造体のメンバsqltypeの値を検査します。そして、列の型に応じて、sqlvarフィールドからホスト変数にデータをコピーするための適切な方法に切り替えます。

```
char var_buf[1024];

switch (v.sqltype)
{
    case ECPGt_char:
        memset(&var_buf, 0, sizeof(var_buf));
        memcpy(&var_buf, sqldata, (sizeof(var_buf) <= sqllen ? sizeof(var_buf) - 1 : sqllen));
        break;


    case ECPGt_int: /* 整数 */
        memcpy(&intval, sqldata, sqllen);
        snprintf(var_buf, sizeof(var_buf), "%d", intval);
        break;

    ...
```

```
}
```

35.7.2.3. SQLDAを使用した問い合わせパラメータ渡し

プリペアド問い合わせに入力パラメータを渡すためにSQLDAを使用する、一般的な手順は以下の通りです。

1. プリペアド問い合わせ（プリペアド文）を作成します。
2. 入力用SQLDAとしてsqlda_t構造体を宣言します。
3. 入力用SQLDA用にメモリ領域を（sqlda_t構造体として）割り当てます。
4. 割り当てたメモリに入力値を設定（コピー）します。
5. 入力用SQLDAを指定してカーソルを開きます。

以下に例を示します。

まずプリペアド文を作成します。

```
EXEC SQL BEGIN DECLARE SECTION;
char query[1024] = "SELECT d.oid, * FROM pg_database d, pg_stat_database s WHERE d.oid = s.datid
 AND (d.datname = ? OR d.oid = ?)";
EXEC SQL END DECLARE SECTION;

EXEC SQL PREPARE stmt1 FROM :query;
```

次にSQLDA用にメモリを割り当て、sqlda_t構造体のメンバ変数sqlnに入力パラメータ数を設定します。プリペアド問い合わせで2つ以上の入力パラメータが必要な場合、アプリケーションは(パラメータ数 - 1) * sizeof(sqlvar_t)で計算される追加のメモリ空間を割り当てなければなりません。ここで示す例では2つの入力パラメータ用にメモリ空間を割り当てます。

```
sqlda_t *sqlda2;

sqlda2 = (sqlda_t *) malloc(sizeof(sqlda_t) + sizeof(sqlvar_t));
memset(sqlda2, 0, sizeof(sqlda_t) + sizeof(sqlvar_t));



sqlda2->sqln = 2; /* 入力変数の個数 */
```

メモリを割り当てた後、sqlvar[]配列にパラメータ値を格納します。（これは、SQLDAが結果セットを受け取る時に列値を取り出すために使用した配列と同じです。）この例では、入力パラメータは文字列型を持つ"postgres"と整数型を持つ1です。

```
sqlda2->sqlvar[0].sqltype = ECPGt_char;
sqlda2->sqlvar[0].sqldata = "postgres";
sqlda2->sqlvar[0].sqllen  = 8;
```

```
int intval = 1;
sqlda2->sqlvar[1].sqltype = ECPGt_int;
sqlda2->sqlvar[1].sqldata = (char *) &intval;
sqlda2->sqlvar[1].sqllen  = sizeof(intval);
```

ここまでで設定したSQLDAを指定するカーソルを開くことで、入力パラメータはプリペアド文に渡されます。

```
EXEC SQL OPEN cur1 USING DESCRIPTOR sqlda2;
```

最後に、問い合わせ結果を受け取るために使用するSQLDAとは異なり、入力用SQLDAの使用後、割り当てたメモリ空間を明示的に解放しなければなりません。

```
free(sqlda2);
```

35.7.2.4. SQLDAを使用するサンプルアプリケーション

以下に、システムカタログから入力パラメータにより指定されたデータベースの統計情報にアクセスし取り出す方法を示す、プログラム例を示します。

このアプリケーションは、pg_databaseとpg_stat_databaseシステムテーブルをデータベースOIDで結合し、2つの入力パラメータ(データベースpostgresとOID1)により取り出されるデータベース統計情報を読み取り、表示します。

まず、入力用のSQLDAと出力用のSQLDAを宣言します。

```
EXEC SQL include sqlda.h;



sqlda_t *sqlda1; /* 出力記述子 */
sqlda_t *sqlda2; /* 入力記述子 */
```

次に、データベースに接続し、プリペアド文を作成し、プリペアド文用のカーソルを宣言します。

```
int
main(void)
{
    EXEC SQL BEGIN DECLARE SECTION;
    char query[1024] = "SELECT d.oid,* FROM pg_database d, pg_stat_database s WHERE d.oid=s.datid
 AND ( d.datname=? OR d.oid=? )";
    EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO testdb AS con1 USER testuser;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;

    EXEC SQL PREPARE stmt1 FROM :query;
    EXEC SQL DECLARE cur1 CURSOR FOR stmt1;
```

次に、入力パラメータのために入力用SQLDA内にいくつかの値を格納します。入力用SQLDAのためのメモリを割り当て、入力パラメータの個数をsqlnに設定します。型、値、値の長さをsqlvar構造体内のsqltype、sqldata、sqllenに格納します。

```
    /* 入力パラメータ用のSQLDA構造体を作成する。 */
    sqlda2 = (sqlda_t *) malloc(sizeof(sqlda_t) + sizeof(sqlvar_t));
    memset(sqlda2, 0, sizeof(sqlda_t) + sizeof(sqlvar_t));

    sqlda2->sqln = 2; /* 入力変数の数 */

    sqlda2->sqlvar[0].sqltype = ECPGt_char;
    sqlda2->sqlvar[0].sqldata = "postgres";
    sqlda2->sqlvar[0].sqllen  = 8;

    intval = 1;
    sqlda2->sqlvar[1].sqltype = ECPGt_int;
    sqlda2->sqlvar[1].sqldata = (char *)&intval;
    sqlda2->sqlvar[1].sqllen  = sizeof(intval);
```

入力用SQLDAを設定し終えた後、入力用SQLDAを付けたカーソルを開きます。

```
    /* 入力パラメータ付きでカーソルを開く。 */
    EXEC SQL OPEN cur1 USING DESCRIPTOR sqlda2;
```

開いたカーソルから出力用SQLDA内に行を取り込みます。(一般的に結果セット内の行をすべて取り込むためには、ループ内でFETCHを繰り返し呼び出さなければなりません。)

```
while (1)
    {
        sqlda_t *cur_sqlda;



        /* 記述子をカーソルに割り当てる */
        EXEC SQL FETCH NEXT FROM cur1 INTO DESCRIPTOR sqlda1;
```

次に、sqlda_t構造体のリンクリストを追うことで、SQLDAから取り込んだレコードを取り出します。

```
for (cur_sqlda = sqlda1 ;
         cur_sqlda != NULL ;
         cur_sqlda = cur_sqlda->desc_next)
    {
        ...
```

最初のレコードから各列を読み取ります。列数はsqldに、最初の列の実データはsqlvar[0]に格納されています。どちらもsqlda_t構造体のメンバです。

```
        /* 1行の列をすべて表示する。 */
        for (i = 0; i < sqlda1->sqld; i++)
        {
            sqlvar_t v = sqlda1->sqlvar[i];
            char *sqldata = v.sqldata;
            short sqllen  = v.sqllen;

            strncpy(name_buf, v.sqlname.data, v.sqlname.length);
            name_buf[v.sqlname.length] = '\0';
```

ここで、列データがv変数内に格納されました。列の型についてv.sqltypeを検索しながら、すべてのデータをホスト変数にコピーします。

```
switch (v.sqltype) {
                int intval;
                double doubleval;
                unsigned long long int longlongval;

                case ECPGt_char:
                    memset(&var_buf, 0, sizeof(var_buf));
                    memcpy(&var_buf, sqldata, (sizeof(var_buf) <= sqllen ? sizeof(var_buf)-1 :
 sqllen));
                    break;


                case ECPGt_int: /* 整数 */
                    memcpy(&intval, sqldata, sqllen);
                    snprintf(var_buf, sizeof(var_buf), "%d", intval);
                    break;

                ...

                default:
                    ...
            }

            printf("%s = %s (type: %d)\n", name_buf, var_buf, v.sqltype);
        }
```

すべてのレコードを処理した後カーソルを閉じ、データベースとの接続を切断します。

```
EXEC SQL CLOSE cur1;
    EXEC SQL COMMIT;

    EXEC SQL DISCONNECT ALL;
```

プログラム全体を例 35.1に示します。

例35.1 SQLDAプログラムの例

```
#include <stdlib.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

EXEC SQL include sqlda.h;


sqlda_t *sqlda1; /* 出力記述子 */
sqlda_t *sqlda2; /* 入力記述子 */

EXEC SQL WHENEVER NOT FOUND DO BREAK;
EXEC SQL WHENEVER SQLERROR STOP;

int
main(void)
{
    EXEC SQL BEGIN DECLARE SECTION;
    char query[1024] = "SELECT d.oid,* FROM pg_database d, pg_stat_database s WHERE d.oid=s.datid
 AND ( d.datname=? OR d.oid=? )";

    int intval;
    unsigned long long int longlongval;
    EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO uptimedb AS con1 USER uptime;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;

    EXEC SQL PREPARE stmt1 FROM :query;
    EXEC SQL DECLARE cur1 CURSOR FOR stmt1;


    /* 入力パラメータ用のSQLDA構造体を作成する */
    sqlda2 = (sqlda_t *)malloc(sizeof(sqlda_t) + sizeof(sqlvar_t));
    memset(sqlda2, 0, sizeof(sqlda_t) + sizeof(sqlvar_t));

    sqlda2->sqln = 2; /* 入力変数の数 */

    sqlda2->sqlvar[0].sqltype = ECPGt_char;
    sqlda2->sqlvar[0].sqldata = "postgres";
    sqlda2->sqlvar[0].sqllen  = 8;
```

```
    intval = 1;
    sqlda2->sqlvar[1].sqltype = ECPGt_int;
    sqlda2->sqlvar[1].sqldata = (char *) &intval;
    sqlda2->sqlvar[1].sqllen  = sizeof(intval);


    /* 入力パラメータ付きでカーソルを開く。 */
    EXEC SQL OPEN cur1 USING DESCRIPTOR sqlda2;

    while (1)
    {
        sqlda_t *cur_sqlda;


        /* 記述子をカーソルに割り当てる */
        EXEC SQL FETCH NEXT FROM cur1 INTO DESCRIPTOR sqlda1;

        for (cur_sqlda = sqlda1 ;
             cur_sqlda != NULL ;
             cur_sqlda = cur_sqlda->desc_next)
        {
            int i;
            char name_buf[1024];
            char var_buf[1024];


            /* 1行の列をすべて表示する。 */
            for (i=0 ; i<cur_sqlda->sqld ; i++)
            {
                sqlvar_t v = cur_sqlda->sqlvar[i];
                char *sqldata = v.sqldata;
                short sqllen  = v.sqllen;

                strncpy(name_buf, v.sqlname.data, v.sqlname.length);
                name_buf[v.sqlname.length] = '\0';

                switch (v.sqltype)
                {
                    case ECPGt_char:
                        memset(&var_buf, 0, sizeof(var_buf));
                        memcpy(&var_buf, sqldata, (sizeof(var_buf)<=sqllen ? sizeof(var_buf)-1 :
sqllen) );
                        break;


                    case ECPGt_int: /* 整数 */
```

```
                        memcpy(&intval, sqldata, sqllen);
                        snprintf(var_buf, sizeof(var_buf), "%d", intval);
                        break;

                    case ECPGt_long_long: /* bigint */
                        memcpy(&longlongval, sqldata, sqllen);
                        snprintf(var_buf, sizeof(var_buf), "%lld", longlongval);
                        break;

                    default:
                    {
                        int i;
                        memset(var_buf, 0, sizeof(var_buf));
                        for (i = 0; i < sqllen; i++)
                        {
                            char tmpbuf[16];
                            snprintf(tmpbuf, sizeof(tmpbuf), "%02x ", (unsigned char)
 sqldata[i]);
                            strncat(var_buf, tmpbuf, sizeof(var_buf));
                        }
                    }
                        break;
                }

                printf("%s = %s (type: %d)\n", name_buf, var_buf, v.sqltype);
            }

            printf("\n");
        }
    }

    EXEC SQL CLOSE cur1;
    EXEC SQL COMMIT;

    EXEC SQL DISCONNECT ALL;

    return 0;
}
```

この例の出力は以下のようなものになるはずです（一部の数値は変動します）。

```
oid = 1 (type: 1)
datname = template1 (type: 1)
datdba = 10 (type: 1)
encoding = 0 (type: 5)
datistemplate = t (type: 1)
```

```
datallowconn = t (type: 1)
datconnlimit = -1 (type: 5)
datlastsysoid = 11510 (type: 1)
datfrozenxid = 379 (type: 1)
dattablespace = 1663 (type: 1)
datconfig =  (type: 1)
datacl = {=c/uptime,uptime=CTc/uptime} (type: 1)
datid = 1 (type: 1)
datname = template1 (type: 1)
numbackends = 0 (type: 5)
xact_commit = 113606 (type: 9)
xact_rollback = 0 (type: 9)
blks_read = 130 (type: 9)
blks_hit = 7341714 (type: 9)
tup_returned = 38262679 (type: 9)
tup_fetched = 1836281 (type: 9)
tup_inserted = 0 (type: 9)
tup_updated = 0 (type: 9)
tup_deleted = 0 (type: 9)

oid = 11511 (type: 1)
datname = postgres (type: 1)
datdba = 10 (type: 1)
encoding = 0 (type: 5)
datistemplate = f (type: 1)
datallowconn = t (type: 1)
datconnlimit = -1 (type: 5)
datlastsysoid = 11510 (type: 1)
datfrozenxid = 379 (type: 1)
dattablespace = 1663 (type: 1)
datconfig =  (type: 1)
datacl =  (type: 1)
datid = 11511 (type: 1)
datname = postgres (type: 1)
numbackends = 0 (type: 5)
xact_commit = 221069 (type: 9)
xact_rollback = 18 (type: 9)
blks_read = 1176 (type: 9)
blks_hit = 13943750 (type: 9)
tup_returned = 77410091 (type: 9)
tup_fetched = 3253694 (type: 9)
tup_inserted = 0 (type: 9)
tup_updated = 0 (type: 9)
tup_deleted = 0 (type: 9)
```

35.8. エラー処理

本節では、埋め込みSQLプログラムにおいて、例外条件や警告をどのように扱うことができるかについて説明します。このために、共に使用できる2つの機能があります。

- WHENEVERコマンドを使用して、警告条件、エラー条件を扱うようにコールバックを設定することができます。
- エラーまたは警告に関する詳細情報はsqlca変数から入手することができます。

35.8.1. コールバックの設定

エラーや警告を受け取る簡単な手法の1つは、特定の条件が発生する度に特定の動作を実行するように設定することです。一般的には以下のようになります。

```
EXEC SQL WHENEVER condition action;
```

conditionは以下のいずれかを取ることができます。

SQLERROR

SQL文の実行中にエラーが発生する度に、指定した動作が呼び出されます。

SQLWARNING

SQL文の実行中に警告が発生する度に、指定した動作が呼び出されます。

NOT FOUND

SQL文が0行を受け取る、もしくは0行に影響する時、指定した動作が呼び出されます。(この条件はエラーではありませんが、これを特別に扱いたい場合があります。)

actionは以下のいずれかを取ることができます。

CONTINUE

これは、実際のところ、その条件が無視されることを意味します。これがデフォルトです。

GOTO label
GO TO label

指定したラベルに移動します (Cのgoto文を使用します)。

SQLPRINT

標準エラーにメッセージを出力します。これは、単純なプログラムやプロトタイプ作成時に役に立ちます。メッセージの詳細は設定できません。

STOP

プログラムを終了させるexit(1)を呼び出します。

DO BREAK

Cのbreak文を実行します。これはループ内、もしくはswitch文内でのみ使用しなければなりません。

DO CONTINUE

Cのcontinue文を実行します。これはループ文の中でのみ実行すべきものです。実行した場合、制御の流れがループの先頭に戻ります。

CALL name (args)
DO name (args)

指定した引数で、指定したC関数を呼び出します。(この使用法は通常のPostgreSQL構文でのCALLおよびDOとは意味が異なります。)

標準SQLではCONTINUEとGOTO(とGO TO)のみを提供しています。

簡単なプログラムで使用してみたくなるような例を以下に示します。警告が発生した場合に簡単なメッセージを表示し、エラーが発生した場合にプログラムを中断します。

```
EXEC SQL WHENEVER SQLWARNING SQLPRINT;
EXEC SQL WHENEVER SQLERROR STOP;
```

EXEC SQL WHENEVER文はCの構文ではなく、SQLプリプロセッサのディレクティブです。設定したエラーもしくは警告動作は、最初のEXEC SQL WHENEVERと条件を発生させたSQL文の間で、同一条件に異なる動作が設定されない限り、ハンドラを設定した箇所より後にある、すべての埋め込みSQL文に適用されます。Cプログラムの制御フローは関係しません。ですので、以下の2つのCプログラムの抜粋はどちらも望み通りの動作を行いません。

```
/*

 * 間違い
 */
int main(int argc, char *argv[])
{
    ...
    if (verbose) {
        EXEC SQL WHENEVER SQLWARNING SQLPRINT;
    }
    ...
    EXEC SQL SELECT ...;
    ...
}
```

```
/*

 * 間違い
 */
```

```
int main(int argc, char *argv[])
{
    ...
    set_error_handler();
    ...
    EXEC SQL SELECT ...;
    ...
}

static void set_error_handler(void)
{
    EXEC SQL WHENEVER SQLERROR STOP;
}
```

35.8.2. sqlca

より強力にエラーを扱うために、埋め込みSQLインタフェースは以下の構造体を持つsqlca（SQL通信領域）という名前のグローバル変数を提供します。

```
struct
{
    char sqlcaid[8];
    long sqlabc;
    long sqlcode;
    struct
    {
        int sqlerrml;
        char sqlerrmc[SQLERRMC_LEN];
    } sqlerrm;
    char sqlerrp[8];
    long sqlerrd[6];
    char sqlwarn[8];
    char sqlstate[5];
} sqlca;
```

（マルチスレッド化されたプログラムでは、各スレッドは自動的にsqlcaのコピーを独自に持ちます。これは標準Cのerrnoグローバル変数の扱いと同様に動作します。）

sqlcaは警告とエラーの両方を対象としています。1つのSQL文の実行時に複数の警告やエラーが発生した場合、sqlcaは最後のものに関した情報のみを含みます。

直前のSQL文でエラーがなければ、sqlca.sqlcodeは0に、sqlca.sqlstateは"00000"になります。
警告やエラーが発生した場合は、sqlca.sqlcodeは負の値に、sqlca.sqlstateは"00000"以外になります。正のsqlca.sqlcodeは、直前の問い合わせが0行を返したなどの無害な条件を示します。
sqlca.sqlcodeとsqlca.sqlstateは2つの異なるエラーコードスキームです。後で詳細に説明します。

直前のSQL文が成功すると、sqlca.sqlerrd[1]は処理された行のOIDが、もしあれば、格納されます。また、もしそのコマンドで適切ならば、sqlca.sqlerrd[2]は処理された、もしくは返された行数が格納されます。

エラーもしくは警告の場合、sqlca.sqlerrm.sqlerrmcには、そのエラーを説明する文字列が格納されます。sqlca.sqlerrm.sqlerrmlフィールドにはsqlca.sqlerrm.sqlerrmcに格納されたエラーメッセージ長が格納されます（strlen()の結果です。おそらくCプログラマは必要としないでしょう）。一部のメッセージは固定長のsqlerrmc配列には長過ぎることに注意してください。この場合は切り詰められます。

警告の場合、sqlca.sqlwarn[2]はWに設定されます（他のすべての場合では、これはW以外の何かに設定されます）。sqlca.sqlwarn[1]がWに設定された場合、ホスト変数に代入する際に値が切り詰められています。他の要素が警告を示すように設定された場合、sqlca.sqlwarn[0]はWに設定されます。

今のところ、sqlcaid、sqlabc、sqlerrpならびにsqlerrdとsqlwarnの上記以外の要素は有用な情報を持ちません。

sqlcaは標準SQLでは定義されていません。しかし、複数の他のSQLデータベースシステムで実装されています。その定義は基本部分は似ていますが、移植性を持つアプリケーションを作成する場合は実装の違いを注意して調査しなければなりません。

ここでWHENEVERとsqlcaを組み合わせて使用して、エラーが発生した時にsqlcaの内容を表示する、1つの例を示します。これはおそらく、より「ユーザ向け」のエラー処理を組み込む前の、アプリケーションのデバッグまたはプロトタイプで有用です。

```
EXEC SQL WHENEVER SQLERROR CALL print_sqlca();

void
print_sqlca()
{
    fprintf(stderr, "==== sqlca ====\n");
    fprintf(stderr, "sqlcode: %ld\n", sqlca.sqlcode);
    fprintf(stderr, "sqlerrm.sqlerrml: %d\n", sqlca.sqlerrm.sqlerrml);
    fprintf(stderr, "sqlerrm.sqlerrmc: %s\n", sqlca.sqlerrm.sqlerrmc);
    fprintf(stderr, "sqlerrd: %ld %ld %ld %ld %ld %ld\n",
 sqlca.sqlerrd[0],sqlca.sqlerrd[1],sqlca.sqlerrd[2],

 sqlca.sqlerrd[3],sqlca.sqlerrd[4],sqlca.sqlerrd[5]);
    fprintf(stderr, "sqlwarn: %d %d %d %d %d %d %d %d\n", sqlca.sqlwarn[0], sqlca.sqlwarn[1],
 sqlca.sqlwarn[2],
                                                          sqlca.sqlwarn[3], sqlca.sqlwarn[4],
 sqlca.sqlwarn[5],
                                                          sqlca.sqlwarn[6], sqlca.sqlwarn[7]);
    fprintf(stderr, "sqlstate: %5s\n", sqlca.sqlstate);
    fprintf(stderr, "===============\n");
}
```

結果は以下のようになります（ここでのエラーはテーブル名の誤記述によるものです。）。

```
==== sqlca ====
```

```
sqlcode: -400
sqlerrm.sqlerrml: 49
sqlerrm.sqlerrmc: relation "pg_databasep" does not exist on line 38
sqlerrd: 0 0 0 0 0 0
sqlwarn: 0 0 0 0 0 0 0 0
sqlstate: 42P01
===============
```

35.8.3. SQLSTATE対SQLCODE

sqlca.sqlstateとsqlca.sqlcodeはエラーコードを提供する異なる2つの機構です。共に標準SQLから派生されたものですが、SQLCODEはSQL-92版では廃れたものとされ、以降の版から削除されました。したがって、新規アプリケーションではSQLSTATEを使用することを強く勧めます。

SQLSTATEは5要素の文字配列です。この5文字は、各種のエラー条件、警告条件のコードを表現する数字、大文字から構成されます。SQLSTATEは階層を持った機構です。最初の2文字は条件を汎化したクラスを示し、残り3文字は汎化クラスの副クラスを示します。成功状態は00000というコードで示されます。SQLSTATEコードのほとんどは標準SQLで定義されています。PostgreSQLサーバは本質的にSQLSTATEエラーコードをサポートしています。したがって、すべてのアプリケーションでこのエラーコードを使用することで、高度な一貫性を達成することができます。詳細については付録Aを参照してください。

廃止されたエラーコードの機構であるSQLCODEは単なる整数です。0という値は成功を意味し、正の値は追加情報を持った成功を、負の値はエラーを示します。標準SQLでは、直前のコマンドが0行を返す、もしくは0行に影響したことを示す+100という正の値のみを定義しています。負の値は規定されていません。したがって、この機構では低い移植性しか達成できず、また、コード体系も階層を持っていません。歴史的に、PostgreSQLの埋め込みSQLプロセッサには、いくつかの特殊なSQLCODEの値が専用に割り当てられていました。以下に、その数値とそのシンボル名の一覧を示します。これらは他のSQL実装への移植性がないことを忘れないでください。アプリケーションのSQLSTATE機構への移行を簡易化するために、対応するSQLSTATEも示しています。しかし、2つのしくみの間の関係は1対1ではなく1対多です（実際は多対多です）。ですので、場合ごとに付録Aに示したグローバルな各SQLSTATEを参照しなければなりません。

以下は割り当て済みのSQLCODEです。

0 (ECPG_NO_ERROR)

エラーがないことを示す。(SQLSTATE 00000)

100 (ECPG_NOT_FOUND)

これは、最後に実行したコマンドが取り出した、または、処理した行がゼロ行であったこと、あるいは、カーソルの最後であることを示す、害のない条件です。(SQLSTATE 02000)

以下のように、カーソルをループ内で処理する時、ループを中断する時を検知する方法として、このコードを使用することができます。

```
while (1)
{
```

```
    EXEC SQL FETCH ... ;
    if (sqlca.sqlcode == ECPG_NOT_FOUND)
        break;
}
```

しかし、WHENEVER NOT FOUND DO BREAKはこれを内部で効率的に行います。このため、通常、外部で明示的に記述する利点はありません。

-12 (ECPG_OUT_OF_MEMORY)

仮想メモリ不足を示します。この数値は-ENOMEMとして定義します。(SQLSTATE YE001)

-200 (ECPG_UNSUPPORTED)

ライブラリが把握していない何かをプリプロセッサが生成したことを示します。おそらく、互換性がないプリプロセッサとライブラリのバージョンを使用しています。(SQLSTATE YE002)

-201 (ECPG_TOO_MANY_ARGUMENTS)

コマンドの想定より多くのホスト変数が指定されたことを意味します。(SQLSTATE 07001もしくは07002)

-202 (ECPG_TOO_FEW_ARGUMENTS)

コマンドの想定よりも少ないホスト変数が指定されたことを意味します。(SQLSTATE 07001もしくは07002)

-203 (ECPG_TOO_MANY_MATCHES)

問い合わせが複数行を返したけれども、SQL文では1つの結果の格納の準備だけしかしていなかったことを意味します (例えば、指定された変数が配列ではなかった)。(SQLSTATE 21000)

-204 (ECPG_INT_FORMAT)

ホスト変数の型がintですが、データベース内のデータ型が異なり、その値をintとして解釈させることができませんでした。ライブラリはこの変換にstrtol()を使用します。(SQLSTATE 42804)

-205 (ECPG_UINT_FORMAT)

ホスト変数の型がunsigned intですが、データベース内のデータ型が異なり、その値をunsigned intとして解釈させることができませんでした。ライブラリはこの変換にstrtoul()を使用します。(SQLSTATE 42804)

-206 (ECPG_FLOAT_FORMAT)

ホスト変数の型がfloatですが、データベース内のデータ型が異なり、その値をfloatとして解釈させることができませんでした。ライブラリはこの変換にstrtod()を使用します。(SQLSTATE 42804)

-207 (ECPG_NUMERIC_FORMAT)

ホスト変数の型がnumericですが、データベース内のデータ型が異なり、その値をnumericとして解釈させることができませんでした。(SQLSTATE 42804)

-208 (ECPG_INTERVAL_FORMAT)

ホスト変数の型がintervalであり、データベース内のデータが他の型であり、interval値として解釈することができない値を含みます。(SQLSTATE 42804)

-209 (ECPG_DATE_FORMAT)

ホスト変数の型がdateであり、データベース内のデータが他の型であり、date値として解釈することができない値を含みます。(SQLSTATE 42804)

-210 (ECPG_TIMESTAMP_FORMAT)

ホスト変数の型がtimestampであり、データベース内のデータが他の型であり、timestamp値として解釈することができない値を含みます。(SQLSTATE 42804)

-211 (ECPG_CONVERT_BOOL)

これは、ホスト変数の型がboolですが、データベース内のデータが't'でも'f'でもなかったことを意味します。(SQLSTATE 42804)

-212 (ECPG_EMPTY)

PostgreSQLサーバに送信されたSQL文が空でした (通常埋め込みSQLプログラムでは発生しません。ですので、これは内部エラーを示しているかもしれません)。(SQLSTATE YE002)

-213 (ECPG_MISSING_INDICATOR)

NULL値が返されましたが、NULL用の指示子変数が与えられていませんでした。(SQLSTATE 22002)

-214 (ECPG_NO_ARRAY)

配列が必要な箇所に普通の変数が使用されていました。(SQLSTATE 42804)

-215 (ECPG_DATA_NOT_ARRAY)

配列値が必要な箇所にデータベースが普通の変数を返しました。(SQLSTATE 42804)

-216 (ECPG_ARRAY_INSERT)

値を配列に挿入できません。(SQLSTATE 42804)

-220 (ECPG_NO_CONN)

存在しない接続にプログラムがアクセスしようとしました。(SQLSTATE 08003)

-221 (ECPG_NOT_CONN)

存在するが開いていない接続にプログラムがアクセスしようとしました (これは内部エラーです)。(SQLSTATE YE002)

-230 (ECPG_INVALID_STMT)

使用しようとしたSQL文がプリペアされていませんでした。(SQLSTATE 26000)

-239 (ECPG_INFORMIX_DUPLICATE_KEY)

重複キーエラー。一意性制約違反（Informix互換モード）。(SQLSTATE 23505)

-240 (ECPG_UNKNOWN_DESCRIPTOR)

指定した記述子が見つかりませんでした。使用しようとしたSQL文はプリペアされていませんでした。（SQLSTATE 33000）

-241 (ECPG_INVALID_DESCRIPTOR_INDEX)

記述子のインデックスが範囲外でした。（SQLSTATE 07009）

-242 (ECPG_UNKNOWN_DESCRIPTOR_ITEM)

無効な記述子項目が要求されました。（これは内部エラーです。）（SQLSTATE YE002）

-243 (ECPG_VAR_NOT_NUMERIC)

動的なSQL文の実行時にデータベースが数値を返しましたが、ホスト変数が数値でありませんでした。（SQLSTATE 07006）

-244 (ECPG_VAR_NOT_CHAR)

動的なSQL文の実行時にデータベースが数値以外を返しましたが、ホスト変数が数値でした。（SQLSTATE 07006）

-284 (ECPG_INFORMIX_SUBSELECT_NOT_ONE)

副問い合わせの結果が単一行ではありません（Informix互換モード）。(SQLSTATE 21000)

-400 (ECPG_PGSQL)

PostgreSQLサーバで何らかのエラーが発生しました。このメッセージはPostgreSQLサーバからのエラーメッセージを含みます。

-401 (ECPG_TRANS)

PostgreSQLサーバがトランザクションのコミットやロールバックを始めることができないことを通知しました。（SQLSTATE 08007）

-402 (ECPG_CONNECT)

データベースへの接続試行に失敗しました。(SQLSTATE 08001)

-403 (ECPG_DUPLICATE_KEY)

重複キーエラー。一意性制約違反。(SQLSTATE 23505)

-404 (ECPG_SUBSELECT_NOT_ONE)

副問い合わせの結果が単一行ではありません。(SQLSTATE 21000)

-602 (ECPG_WARNING_UNKNOWN_PORTAL)

無効なカーソル名が指定されました。(SQLSTATE 34000)

-603 (ECPG_WARNING_IN_TRANSACTION)

トランザクションが進行中です。(SQLSTATE 25001)

-604 (ECPG_WARNING_NO_TRANSACTION)

活動中（進行中）のトランザクションがありません。(SQLSTATE 25P01)

-605 (ECPG_WARNING_PORTAL_EXISTS)

既存のカーソル名が指定されました。(SQLSTATE 42P03)

35.9. プリプロセッサ指示子

ecpgプリプロセッサがファイルを解析および処理する方法を変更することができる、プリプロセッサ指示子が複数あります。

35.9.1. ファイルのインクルード

埋め込みSQLプログラムに外部ファイルをインクルードするには、以下を使用します。

```
EXEC SQL INCLUDE filename;
EXEC SQL INCLUDE <filename>;
EXEC SQL INCLUDE "filename";
```

埋め込みSQLプリプロセッサは、filename.hという名前のファイルを探し、その前処理を行い、最終的にC出力の中に含めます。このようにして、ヘッダファイル内の埋め込みSQL文が正しく扱われます。

ecpgプリプロセッサは以下の順番で複数のディレクトリからファイルを検索します。

- カレントディレクトリ
- /usr/local/include
- ビルド時に設定されたPostgreSQLのインクルードディレクトリ（例えば、/usr/local/pgsql/include）
- /usr/include

しかしEXEC SQL INCLUDE "filename"が使われる場合、現在のディレクトリのみが検索されます。

各ディレクトリの中で、プリプロセッサはまず指定されたファイル名を探します。見つからなければ（指定されたファイル名がこの接尾辞を持っていない限り）ファイル名に.hを付けて再検索します。

EXEC SQL INCLUDEは以下とは異なることに注意してください。

```
#include <filename.h>
```

このファイルにはSQLコマンド用前処理が行われないためです。当然ながら、他のヘッダファイルをインクルードするCの#includeディレクティブを使用することができます。

注記

通常のSQLの大文字小文字の区別規則に従うEXEC SQL INCLUDEコマンドの一部であったとしても、インクルードファイルの名前は大文字小文字が区別されます。

35.9.2. defineおよびundef指示子

Cで既知の#define指示子と同様、埋め込みSQLでも似たような概念を持ちます。

```
EXEC SQL DEFINE name;
EXEC SQL DEFINE name value;
```

このため、以下のように名前を定義することができます。

```
EXEC SQL DEFINE HAVE_FEATURE;
```

また、定数を定義することもできます。

```
EXEC SQL DEFINE MYNUMBER 12;
EXEC SQL DEFINE MYSTRING 'abc';
```

事前の定義を削除するにはundefを使用します。

```
EXEC SQL UNDEF MYNUMBER;
```

当然、Cの#defineや#undefを埋め込みSQLプログラムで使用することは可能です。違いは宣言した値がどこで評価されるかです。EXEC SQL DEFINEを使用する場合、ecpgプリプロセッサがその定義を評価し、その値を置換します。例えば、

```
EXEC SQL DEFINE MYNUMBER 12;
...
EXEC SQL UPDATE Tbl SET col = MYNUMBER;
```

と記載した場合、ecpgによる置換がすでに行われていますので、CコンパイラではMYNUMBERという名前や識別子を参照することはありません。埋め込みSQL問い合わせで使用する予定の定数に#defineを使用することはできませんので注意してください。この場合、埋め込みSQLプリプロセッサがこの宣言を参照することができないためです。

35.9.3. ifdef、ifndef、elif、else、endif指示子

以下の指示子を使用して、コンパイルするコード部分を選択することができます。

`EXEC SQL ifdef name;`

nameを検査し、そのnameがEXEC SQL define nameで定義されていた場合に後続の行を処理します。

`EXEC SQL ifndef name;`

nameを検査し、そのnameがEXEC SQL define nameで定義されていない場合に後続の行を処理します。

`EXEC SQL elif name;`

EXEC SQL ifdef nameまたはEXEC SQL ifndef name指示子の後で省略可能な代替セクションを開始します。elifセクションはいくつでも現れることがあり得ます。elifに続く行は、nameが定義されていて、かつ、同じifdef/ifndef...endif構文の前のセクションが処理されていない場合に、処理されます。

`EXEC SQL else;`

EXEC SQL ifdef nameまたはEXEC SQL ifndef name指示子の後で最後の代替セクションを開始します。同じifdef/ifndef...endif構文の前のセクションが処理されていない場合に、後続の行が処理されます。

`EXEC SQL endif;`

ifdef/ifndef...endif構文を終了します。後続の行は普通に処理されます。

ifdef/ifndef...endif構文は127段階まで入れ子にできます。

この例は3つのSET TIMEZONEコマンドのうちちょうど1つをコンパイルします。

```
EXEC SQL ifdef TZVAR;
EXEC SQL SET TIMEZONE TO TZVAR;
EXEC SQL elif TZNAME;
EXEC SQL SET TIMEZONE TO TZNAME;
EXEC SQL else;
EXEC SQL SET TIMEZONE TO 'GMT';
EXEC SQL endif;
```

35.10. 埋め込みSQLプログラムの処理

ここまでで、埋め込みSQL Cプログラムの作成方法は理解できたと思います。ここからはそのコンパイル方法についてお話しします。コンパイルの前に、そのファイルを埋め込みSQL Cプリプロセッサに通します。これは、使用するSQL文を特別な関数呼び出しに変換します。コンパイル後、必要な関数を持つ特別なライブラリとリンクしなければなりません。これらの関数は引数から情報を取り出し、libpqを使用してそのSQLを実行し、出力用に指定された引数にその結果を格納します。

プリプロセッサプログラムはecpgという名前で、通常PostgreSQLのインストレーションに含まれています。通常、埋め込みSQLプログラムの拡張子は.pgcとします。prog1.pgcという名前のプログラムファイルがある場合、単純に以下を呼び出すことで前処理を行うことができます。

```
ecpg prog1.pgc
```

これはprog1.cという名前のファイルを作成します。入力ファイルがこの提案パターンに従った名前でない場合、-o オプションを使用して明示的に出力ファイルを指定することができます。

前処理後のファイルは普通にコンパイルできます。以下に例を示します。

```
cc -c prog1.c
```

生成されたCソースファイルはPostgreSQLインストレーションに付随するヘッダファイルをインクルードします。ですので、デフォルトで検索されない場所にPostgreSQLをインストールした場合は、コンパイル用のコマンドラインに-I/usr/local/pgsql/includeのようなオプションを追加しなければなりません。

埋め込みSQLプログラムをリンクするためには、以下のように、libecpgライブラリを含めなければなりません。

```
cc -o myprog prog1.o prog2.o ... -lecpg
```

繰り返しになりますが、コマンドラインに-L/usr/local/pgsql/libといったオプションを追加する必要があるかもしれません。

インストール先のパスを取得するために、パッケージ名libecpgでpg_configまたはpkg-configを使うことができます。

大規模プロジェクトの構築処理をmakeを使用して管理している場合、以下の暗黙規則をMakefileに含めておくと便利です。

```
ECPG = ecpg

%.c: %.pgc
        $(ECPG) $<
```

ecpgコマンドの完全な構文はecpgに説明があります。

デフォルトではecpgはスレッドセーフです。しかしクライアントコードのコンパイル時に他のスレッド関連のコマンドラインオプションを使用する必要があるかもしれません。

35.11. ライブラリ関数

libecpgライブラリには基本的に、埋め込みSQLコマンドで表現される機能を実装するために使用する「隠された」関数が含まれています。しかし、直接呼び出すことができる便利な関数もあります。これによりコードが移植不可能になることに注意してください。

- ECPGdebug(int on, FILE *stream)は第1引数が0以外で渡された場合、デバッグログを有効にします。デバッグログはstreamに出力されます。このログには、すべての入力変数が挿入されたすべてのSQL文と、PostgreSQLサーバが返した結果が含まれます。SQL文のエラーを見つける時に非常に役に立ちます。

  > **注記**
  >
  > Windowsでは、ecpgライブラリとアプリケーションが異なるフラグでコンパイルされると、この関数の呼び出しは、FILEポインタの内部表現が異なるため、アプリケーションをクラッシュさせる可能性があります。特に、そのライブラリを使用するすべてのライブラリとすべてのアプリケーションに対して、multithreaded/single-threaded、release/debug、およびstatic/dynamicフラグは同じでなければなりません。

- ECPGget_PGconn(const char *connection_name)は、指定された名前で識別されるライブラリデータベース接続ハンドルを返します。connection_nameの設定がNULLの場合、現在の接続ハンドルが返されます。接続ハンドルを識別できない場合、関数はNULLを返します。必要ならば返される接続ハンドルを使用して、任意のlibpqの他の関数を呼び出すことができます。

  > **注記**
  >
  > libpq関数を直接使用してecpgからデータベース接続ハンドルを操作することは推奨されません。

- ECPGtransactionStatus(const char *connection_name)は、connection_nameで識別される指定接続の現在のトランザクション状態を返します。返される状態コードの詳細については33.2とlibpqのPQtransactionStatusを参照してください。

- ECPGstatus(int lineno, const char* connection_name)はデータベースに接続している場合は真を、さもなくば偽を返します。単一の接続を使用している場合はconnection_nameをNULLとすることができます。

35.12. ラージオブジェクト

ラージオブジェクトはECPGで直接サポートされていません。しかしECPGアプリケーションは、ECPGget_PGconn()関数を呼び出して必要なPGconnを入手して、libpqラージオブジェクト関数を介してラージオブジェクトを操作することができます。(しかしECPGget_PGconn()関数の使用とPGconnを直接触ることは非常に注意して行わなければなりません。理想を言えば他のECPGデータベースアクセス呼び出しと混在させないようにしてください。)

ECPGget_PGconn()に関しては35.11を参照してください。ラージオブジェクト関数インタフェースについては第34章を参照してください。

ラージオブジェクト関数をトランザクションブロック内で呼び出さなければなりません。このため自動コミットが無効な場合、BEGINコマンドを明示的に発行しなければなりません。

例 35.2では、ECPGアプリケーション内でラージオブジェクトの作成、書き出し、読み取りを行う方法を示すプログラム例を示します。

例35.2 ラージオブジェクトにアクセスするECPGプログラム

```
#include <stdio.h>
#include <stdlib.h>
#include <libpq-fe.h>
#include <libpq/libpq-fs.h>

EXEC SQL WHENEVER SQLERROR STOP;

int
main(void)
{
    PGconn     *conn;
    Oid         loid;
    int         fd;
    char        buf[256];
    int         buflen = 256;
    char        buf2[256];
    int         rc;

    memset(buf, 1, buflen);

    EXEC SQL CONNECT TO testdb AS con1;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;

    conn = ECPGget_PGconn("con1");
    printf("conn = %p\n", conn);


    /* 作成 */
    loid = lo_create(conn, 0);
    if (loid &lt; 0)
        printf("lo_create() failed: %s", PQerrorMessage(conn));

    printf("loid = %d\n", loid);


    /* 書き出しテスト */
    fd = lo_open(conn, loid, INV_READ|INV_WRITE);
    if (fd &lt; 0)
        printf("lo_open() failed: %s", PQerrorMessage(conn));

    printf("fd = %d\n", fd);

    rc = lo_write(conn, fd, buf, buflen);
    if (rc &lt; 0)
```

```
        printf("lo_write() failed\n");

    rc = lo_close(conn, fd);
    if (rc &lt; 0)
        printf("lo_close() failed: %s", PQerrorMessage(conn));


    /* 読み取りテスト */
    fd = lo_open(conn, loid, INV_READ);
    if (fd &lt; 0)
        printf("lo_open() failed: %s", PQerrorMessage(conn));

    printf("fd = %d\n", fd);

    rc = lo_read(conn, fd, buf2, buflen);
    if (rc &lt; 0)
        printf("lo_read() failed\n");

    rc = lo_close(conn, fd);
    if (rc &lt; 0)
        printf("lo_close() failed: %s", PQerrorMessage(conn));


    /* 確認 */
    rc = memcmp(buf, buf2, buflen);
    printf("memcmp() = %d\n", rc);


    /* 後始末 */
    rc = lo_unlink(conn, loid);
    if (rc &lt; 0)
        printf("lo_unlink() failed: %s", PQerrorMessage(conn));

    EXEC SQL COMMIT;
    EXEC SQL DISCONNECT ALL;
    return 0;
}
```

35.13. C++アプリケーション

ECPGはC++アプリケーションを多少の制限がありますがサポートします。本節ではいくつかの注意を説明します。

ecpgプリプロセッサはC（またはCのようなもの）と埋め込みSQLコマンドで記述された入力ファイルを取り、埋め込みSQLコマンドをC言語の小塊に変換し、最終的に.cファイルを作成します。ecpgが生成するC言語の

小塊で使用されるライブラリ関数のヘッダファイル定義は、C++で使用される場合extern "C" { ... }で囲まれます。このためC++でも継ぎ目なく動作するはずです。

しかし一般的には、ecpgプリプロセッサはCのみを理解しています。C++言語の特殊な構文や予約語を取り扱いません。このため、C++に特化した複雑な機能を使用するC++アプリケーションコードの中に記述された埋め込みSQLコードの一部は、正しく前処理することに失敗する、または想定通りに動作しないかもしれません。

C++アプリケーションで埋め込みSQLコードを使用する安全な方法は、ECPGの呼び出しをCモジュール内に隠蔽し、残りのC++コードとまとめてリンクすることです。C++アプリケーションコードがデータベースにアクセスするためにはそのCモジュールを呼び出します。35.13.2を参照してください。

35.13.1. ホスト変数のスコープ

ecpgプリプロセッサはCにおける変数のスコープを理解しています。C言語では、変数のスコープはコードブロックに基づきますので、どちらかといえば単純です。しかしC++では クラスメンバ変数は宣言場所とは異なるコードブロック内で参照されます。このためecpgプリプロセッサはクラスメンバ変数のスコープを理解していません。

例えば、以下の場合、ecpgプリプロセッサはtestメソッド内のdbname変数の定義を見つけることができません。このためエラーになります。

```
class TestCpp
{
    EXEC SQL BEGIN DECLARE SECTION;
    char dbname[1024];
    EXEC SQL END DECLARE SECTION;

  public:
    TestCpp();
    void test();
    ~TestCpp();
};

TestCpp::TestCpp()
{
    EXEC SQL CONNECT TO testdb1;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;
}

void Test::test()
{
    EXEC SQL SELECT current_database() INTO :dbname;
    printf("current_database = %s\n", dbname);
}

TestCpp::~TestCpp()
{
```

```
    EXEC SQL DISCONNECT ALL;
}
```

このコードは以下のようなエラーになります。

```
ecpg test_cpp.pgc
test_cpp.pgc:28: ERROR: variable "dbname" is not declared
```

このスコープ問題を回避するためには、testメソッドを中間格納領域としてローカル変数を使用するように変更することができます。しかしこの手法は悪い回避策でしかありません。コードを醜くしますし性能も劣化させます。

```
void TestCpp::test()
{
    EXEC SQL BEGIN DECLARE SECTION;
    char tmp[1024];
    EXEC SQL END DECLARE SECTION;

    EXEC SQL SELECT current_database() INTO :tmp;
    strlcpy(dbname, tmp, sizeof(tmp));

    printf("current_database = %s\n", dbname);
}
```

35.13.2. 外部のCモジュールを用いたC++アプリケーションの開発

C++におけるecpgの技術的な制限を理解しているのであれば、ECPG機能を使用するC++アプリケーションを実現するためには、リンク段階でCオブジェクトとC++オブジェクトをリンクする方が、C++コード内で埋め込みSQLコマンドを直接記述することより優れているという結論に至るでしょう。本節では、簡単な例を用いて、C++アプリケーションコードから埋め込みSQLコマンドを分離する方法について説明します。この例では、アプリケーションはC++で実装し、PostgreSQLサーバに接続するためにCおよびECPGを使用します。

Cファイル（*.pgc）、ヘッダファイル、C++ファイルという3種類のファイルを作成しなければなりません。

test_mod.pgc

C内に埋め込まれたサブルーチンモジュールです。プリプロセッサによりtest_mod.cに変換されます。

```
#include "test_mod.h"
#include <stdio.h>

void
db_connect()
{
```

```
    EXEC SQL CONNECT TO testdb1;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;
}

void
db_test()
{
    EXEC SQL BEGIN DECLARE SECTION;
    char dbname[1024];
    EXEC SQL END DECLARE SECTION;

    EXEC SQL SELECT current_database() INTO :dbname;
    printf("current_database = %s\n", dbname);
}

void
db_disconnect()
{
    EXEC SQL DISCONNECT ALL;
}
```

test_mod.h

Cモジュール（test_mod.pgc）内の関数宣言を持つヘッダファイルです。test_cpp.cppでインクルードされます。このファイルは、C++モジュールからリンクされますので、宣言を囲むextern "C"ブロックを持たなければなりません。

```
#ifdef __cplusplus
extern "C" {
#endif

void db_connect();
void db_test();
void db_disconnect();

#ifdef __cplusplus
}
#endif
```

test_cpp.cpp

mainルーチンとこの例でのC++クラスを含む、アプリケーションの主要コードです。

```
#include "test_mod.h"

class TestCpp
{
```

```
  public:
    TestCpp();
    void test();
    ~TestCpp();
};

TestCpp::TestCpp()
{
    db_connect();
}

void
TestCpp::test()
{
    db_test();
}

TestCpp::~TestCpp()
{
    db_disconnect();
}

int
main(void)
{
    TestCpp *t = new TestCpp();

    t->test();
    return 0;
}
```

アプリケーションを構築するためには、以下の処理を行います。ecpgを実行してtest_mod.pgcをtest_mod.cに変換します。そしてCコンパイラを用いてtest_mod.cをコンパイルしtest_mod.oを生成します。

```
ecpg -o test_mod.c test_mod.pgc
cc -c test_mod.c -o test_mod.o
```

次にC++コンパイラを用いてtest_cpp.cppをコンパイルしtest_cpp.oを生成します。

```
c++ -c test_cpp.cpp -o test_cpp.o
```

最後に、C++コンパイラドライバを用いてtest_cpp.oおよびtest_mod.oというオブジェクトファイルを実行形式ファイルにリンクします。

```
c++ test_cpp.o test_mod.o -lecpg -o test_cpp
```

35.14. 埋め込みSQLコマンド

本節では、埋め込みSQL固有のSQLコマンドをすべて説明します。また、言及がない限り、埋め込みSQLでも使用することができる、SQLコマンドに列挙されたSQLコマンドを参照してください。

ALLOCATE DESCRIPTOR

ALLOCATE DESCRIPTOR — SQL記述子領域を割り当てます。

概要

```
ALLOCATE DESCRIPTOR name
```

説明

ALLOCATE DESCRIPTORは、PostgreSQLサーバとホストプログラムとの間のデータ交換のために使用することができる、新しい名前付きSQL記述子領域を割り当てます。

記述子領域は、使用した後でDEALLOCATE DESCRIPTORコマンドを使用して解放しなければなりません。

パラメータ

name

SQL記述子の名前です。大文字小文字を区別します。これはSQL識別子またはホスト変数になることができます。

例

```
EXEC SQL ALLOCATE DESCRIPTOR mydesc;
```

互換性

ALLOCATE DESCRIPTORは標準SQLで規定されています。

関連項目

DEALLOCATE DESCRIPTOR, GET DESCRIPTOR, SET DESCRIPTOR

CONNECT

CONNECT — データベース接続を確立します。

概要

```
CONNECT TO connection_target [ AS connection_name ] [ USER connection_user ]
CONNECT TO DEFAULT
CONNECT connection_user
DATABASE connection_target
```

説明

CONNECTコマンドはクライアントとPostgreSQLサーバとの間の接続を確立します。

パラメータ

connection_target

connection_target 以下の複数の形式の1つを使用して、接続する対象サーバを指定します。

[database_name] [@host] [:port]

TCP/IPを介した接続。

unix:postgresql://host [:port] / [database_name] [?connection_option]

Unixドメインソケットを介した接続。

tcp:postgresql://host [:port] / [database_name] [?connection_option]

TCP/IPを介した接続。

SQL文字列定数

上記形式のいずれかで記述された値を持ちます。

ホスト変数

上記形式のいずれかで記述された値を持つchar[]またはVARCHAR[]型のホスト変数。

connection_name

他のコマンドで参照することができる、このデータベース接続の識別子です。省略可能です。これはSQL識別子またはホスト変数とすることができます。

connection_user

データベース接続用のユーザ名です。

このパラメータは、user_name/password、user_name IDENTIFIED BY password、user_name USING passwordのいずれかの形式を使用して、ユーザ名とパスワードを指定することができます。

ユーザ名とパスワードは、SQL識別子、文字列定数、ホスト変数とすることができます。

DEFAULT

libpqで定義された、デフォルトの接続パラメータすべてを使用します。

例

以下に接続パラメータを指定する複数の種類を示します。

```
EXEC SQL CONNECT TO "connectdb" AS main;
EXEC SQL CONNECT TO "connectdb" AS second;
EXEC SQL CONNECT TO "unix:postgresql://200.46.204.71/connectdb" AS main USER connectuser;
EXEC SQL CONNECT TO "unix:postgresql://localhost/connectdb" AS main USER connectuser;
EXEC SQL CONNECT TO 'connectdb' AS main;
EXEC SQL CONNECT TO 'unix:postgresql://localhost/connectdb' AS main USER :user;
EXEC SQL CONNECT TO :db AS :id;
EXEC SQL CONNECT TO :db USER connectuser USING :pw;
EXEC SQL CONNECT TO @localhost AS main USER connectdb;
EXEC SQL CONNECT TO REGRESSDB1 as main;
EXEC SQL CONNECT TO AS main USER connectdb;
EXEC SQL CONNECT TO connectdb AS :id;
EXEC SQL CONNECT TO connectdb AS main USER connectuser/connectdb;
EXEC SQL CONNECT TO connectdb AS main;
EXEC SQL CONNECT TO connectdb@localhost AS main;
EXEC SQL CONNECT TO tcp:postgresql://localhost/ USER connectdb;
EXEC SQL CONNECT TO tcp:postgresql://localhost/connectdb USER connectuser IDENTIFIED BY connectpw;
EXEC SQL CONNECT TO tcp:postgresql://localhost:20/connectdb USER connectuser IDENTIFIED BY
 connectpw;
EXEC SQL CONNECT TO unix:postgresql://localhost/ AS main USER connectdb;
EXEC SQL CONNECT TO unix:postgresql://localhost/connectdb AS main USER connectuser;
EXEC SQL CONNECT TO unix:postgresql://localhost/connectdb USER connectuser IDENTIFIED BY
 "connectpw";
EXEC SQL CONNECT TO unix:postgresql://localhost/connectdb USER connectuser USING "connectpw";
EXEC SQL CONNECT TO unix:postgresql://localhost/connectdb?connect_timeout=14 USER connectuser;
```

以下にホスト変数を使用して接続パラメータを指定する方法を示すプログラム例を示します。

```
int
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;

    char *dbname     = "testdb";    /* データベース名 */
```

```
    char *user       = "testuser";  /* 接続ユーザ名 */
    char *connection = "tcp:postgresql://localhost:5432/testdb";
                                    /* 接続文字列 */
    char ver[256];                  /* バージョン文字列を保持するバッファ */
EXEC SQL END DECLARE SECTION;

    ECPGdebug(1, stderr);

    EXEC SQL CONNECT TO :dbname USER :user;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;
    EXEC SQL SELECT version() INTO :ver;
    EXEC SQL DISCONNECT;

    printf("version: %s\n", ver);

    EXEC SQL CONNECT TO :connection USER :user;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;
    EXEC SQL SELECT version() INTO :ver;
    EXEC SQL DISCONNECT;

    printf("version: %s\n", ver);

    return 0;
}
```

互換性

CONNECTは標準SQLで規定されていますが、接続パラメータの書式は実装に特化しています。

関連項目

DISCONNECT, SET CONNECTION

DEALLOCATE DESCRIPTOR

DEALLOCATE DESCRIPTOR — SQL記述子領域の割り当てを解除します。

概要

```
DEALLOCATE DESCRIPTOR name
```

説明

DEALLOCATE DESCRIPTORは名前付きSQL記述子領域の割り当てを解除します。

パラメータ

name

割り当てを解除する記述子の名前です。大文字小文字を区別します。これはSQL識別子またはホスト変数にすることができます。

例

```
EXEC SQL DEALLOCATE DESCRIPTOR mydesc;
```

互換性

DEALLOCATE DESCRIPTORは標準SQLで規定されています。

関連項目

ALLOCATE DESCRIPTOR, GET DESCRIPTOR, SET DESCRIPTOR

DECLARE

DECLARE — カーソルを定義します。

概要

```
DECLARE cursor_name [ BINARY ] [ INSENSITIVE ] [ [ NO ] SCROLL ] CURSOR [ { WITH | WITHOUT }
 HOLD ] FOR prepared_name
DECLARE cursor_name [ BINARY ] [ INSENSITIVE ] [ [ NO ] SCROLL ] CURSOR [ { WITH | WITHOUT }
 HOLD ] FOR query
```

説明

DECLAREは、プリペアド文の結果セット全体を繰り返し処理するカーソルを宣言します。このコマンドは直接的なDECLARESQLコマンドとは多少異なる意味を持ちます。こちらは問い合わせを実行し、取り出し用の結果セットの準備を行いますが、埋め込みSQLコマンドでは、問い合わせの結果セット全体を繰り返す「ループ変数」の名前を宣言するだけです。実際の実行はOPENコマンドでカーソルが開いた時に起こります。

パラメータ

`cursor_name`

カーソル名です。大文字小文字を区別します。これはSQL識別子またはホスト変数とすることができます。

`prepared_name`

プリペアド問い合わせの名前です。SQL識別子またはホスト変数のいずれかです。

`query`

このカーソルで返される行を供給するSELECTまたはVALUESコマンドです。

カーソルオプションの意味についてはDECLAREを参照してください。

例

以下に問い合わせ用のカーソルを宣言する例を示します。

```
EXEC SQL DECLARE C CURSOR FOR SELECT * FROM My_Table;
EXEC SQL DECLARE C CURSOR FOR SELECT Item1 FROM T;
EXEC SQL DECLARE cur1 CURSOR FOR SELECT version();
```

プリペアド文用のカーソルを宣言する例を示します。

```
EXEC SQL PREPARE stmt1 AS SELECT version();
```

```
EXEC SQL DECLARE cur1 CURSOR FOR stmt1;
```

互換性

DECLAREは標準SQLで規定されています。

関連項目

OPEN, CLOSE, DECLARE

DESCRIBE

DESCRIBE — プリペアド文または結果セットに関する情報を入手します。

概要

```
DESCRIBE [ OUTPUT ] prepared_name USING [ SQL ] DESCRIPTOR descriptor_name
DESCRIBE [ OUTPUT ] prepared_name INTO [ SQL ] DESCRIPTOR descriptor_name
DESCRIBE [ OUTPUT ] prepared_name INTO sqlda_name
```

説明

DESCRIBEは、実際に行を取り込むことなく、プリペアド文に含まれる結果列に関するメタデータ情報を取り出します。

パラメータ

`prepared_name`

プリペアド文の名前です。これはSQL識別子またはホスト変数とすることができます。

`descriptor_name`

記述子の名前です。大文字小文字を区別します。これはSQL識別子またはホスト変数とすることができます。

`sqlda_name`

SQLDA変数の名前です。

例

```
EXEC SQL ALLOCATE DESCRIPTOR mydesc;
EXEC SQL PREPARE stmt1 FROM :sql_stmt;
EXEC SQL DESCRIBE stmt1 INTO SQL DESCRIPTOR mydesc;
EXEC SQL GET DESCRIPTOR mydesc VALUE 1 :charvar = NAME;
EXEC SQL DEALLOCATE DESCRIPTOR mydesc;
```

互換性

DESCRIBEは標準SQLで規定されています。

関連項目

ALLOCATE DESCRIPTOR, GET DESCRIPTOR

DISCONNECT

DISCONNECT — データベース接続を終了します。

概要

```
DISCONNECT connection_name
DISCONNECT [ CURRENT ]
DISCONNECT DEFAULT
DISCONNECT ALL
```

説明

DISCONNECTはデータベースとの接続（またはすべての接続）を閉ざします。

パラメータ

connection_name

CONNECTコマンドで確立したデータベース接続の名前です。

CURRENT

直前に開いた接続またはSET CONNECTIONコマンドで設定された接続のいずれかである、「現在の」接続を閉ざします。これはDISCONNECTに引数が与えられなかった場合のデフォルトです。

DEFAULT

デフォルトの接続を閉ざします。

ALL

開いているすべての接続を閉じます。

例

```
int
main(void)
{
    EXEC SQL CONNECT TO testdb AS DEFAULT USER testuser;
    EXEC SQL CONNECT TO testdb AS con1 USER testuser;
    EXEC SQL CONNECT TO testdb AS con2 USER testuser;
    EXEC SQL CONNECT TO testdb AS con3 USER testuser;
```

```
    EXEC SQL DISCONNECT CURRENT;  /* con3を閉じる       */
    EXEC SQL DISCONNECT DEFAULT;  /* DEFAULTを閉じる    */
    EXEC SQL DISCONNECT ALL;      /* con2とcon1を閉じる */

    return 0;
}
```

互換性

DISCONNECTは標準SQLで規定されています。

関連項目

CONNECT, SET CONNECTION

EXECUTE IMMEDIATE

EXECUTE IMMEDIATE — SQL文を動的にプリペアし、実行します。

概要

```
EXECUTE IMMEDIATE string
```

説明

EXECUTE IMMEDIATEは動的に指定されたSQL文を、結果行を受け取ることなく、即座にプリペアし実行します。

パラメータ

string

実行するSQL文を含むC文字列リテラルまたはホスト変数です。

例

以下に、EXECUTE IMMEDIATEとcommandホスト変数を使用してINSERTを実行する例を示します。

```
sprintf(command, "INSERT INTO test (name, amount, letter) VALUES ('db: ''r1''', 1, 'f')");
EXEC SQL EXECUTE IMMEDIATE :command;
```

互換性

EXECUTE IMMEDIATEは標準SQLで規定されています。

GET DESCRIPTOR

GET DESCRIPTOR — SQL記述子領域から情報を入手します。

概要

```
GET DESCRIPTOR descriptor_name :cvariable = descriptor_header_item [, ... ]
GET DESCRIPTOR descriptor_name VALUE column_number :cvariable = descriptor_item [, ... ]
```

説明

GET DESCRIPTORはSQL記述子領域から問い合わせ結果セットに関する情報を取り出し、それをホスト変数に格納します。記述子領域は通常、このコマンドを使用してホスト言語変数に情報を転送する前に、FETCHまたはSELECTを用いて値が投入されます。

このコマンドには2つの構文があります。1番目の構文では、そのまま結果セットに適用されている記述子の「ヘッダ」項目を取り出します。行数が1つの例です。列番号を追加のパラメータとして必要とする2番目の構文では特定の列に関する情報を取り出します。例えば、列名と列の実際の値です。

パラメータ

`descriptor_name`

記述子の名前です。

`descriptor_header_item`

どのヘッダ情報を取り出すかを識別するトークンです。結果セット内の列数を入手するCOUNTのみが現在サポートされています。

`column_number`

情報を取り出す列の番号です。1から数えます。

`descriptor_item`

どの列に関する情報を取り出すかを識別するトークンです。サポートされる項目のリストについては35.7.1を参照してください。

`cvariable`

記述子領域から取り出したデータを受け取るホスト変数です。

例

この例は結果セット内の列数を取り出します。

```
EXEC SQL GET DESCRIPTOR d :d_count = COUNT;
```

この例は最初の列のデータ長を取り出します。

```
EXEC SQL GET DESCRIPTOR d VALUE 1 :d_returned_octet_length = RETURNED_OCTET_LENGTH;
```

この例は、2番目の列のデータ本体を文字列として取り出します。

```
EXEC SQL GET DESCRIPTOR d VALUE 2 :d_data = DATA;
```

以下は、SELECT current_database();を実行し、列数、列のデータ長、列のデータを表示する手続き全体を示す例です。

```
int
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;
    int  d_count;
    char d_data[1024];
    int  d_returned_octet_length;
EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO testdb AS con1 USER testuser;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;
    EXEC SQL ALLOCATE DESCRIPTOR d;


    /* カーソルを宣言して開いて、
      記述子をそのカーソルに割り当てる */
    EXEC SQL DECLARE cur CURSOR FOR SELECT current_database();
    EXEC SQL OPEN cur;
    EXEC SQL FETCH NEXT FROM cur INTO SQL DESCRIPTOR d;


    /* 全列数を得る */
    EXEC SQL GET DESCRIPTOR d :d_count = COUNT;
    printf("d_count                 = %d\n", d_count);


    /* 返された列の長さを得る */
    EXEC SQL GET DESCRIPTOR d VALUE 1 :d_returned_octet_length = RETURNED_OCTET_LENGTH;
    printf("d_returned_octet_length = %d\n", d_returned_octet_length);


    /* 返された列を文字列として取得する */
    EXEC SQL GET DESCRIPTOR d VALUE 1 :d_data = DATA;
```

```
    printf("d_data                  = %s\n", d_data);


    /* 閉じる */
    EXEC SQL CLOSE cur;
    EXEC SQL COMMIT;

    EXEC SQL DEALLOCATE DESCRIPTOR d;
    EXEC SQL DISCONNECT ALL;

    return 0;
}
```

この例を実行すると、結果は以下のようになります。

```
d_count                 = 1
d_returned_octet_length = 6
d_data                  = testdb
```

互換性

GET DESCRIPTORは標準SQLで規定されています。

関連項目

ALLOCATE DESCRIPTOR, SET DESCRIPTOR

OPEN

OPEN — 動的カーソルを開きます。

概要

```
OPEN cursor_name
OPEN cursor_name USING value [, ... ]
OPEN cursor_name USING SQL DESCRIPTOR descriptor_name
```

説明

OPENはカーソルを開き、省略することができますが、実際の値をカーソル定義内のプレースホルダにバインドします。カーソルは事前にDECLAREコマンドを用いて宣言されていなければなりません。OPENの実行により問い合わせがサーバ上で実行を開始されます。

パラメータ

`cursor_name`

開くカーソルの名前です。これはSQL識別子またはホスト変数とすることができます。

`value`

カーソル内のプレースホルダにバインドされる値です。これは、SQL定数、ホスト変数、指示子を持つホスト変数とすることができます。

`descriptor_name`

カーソル内のプレースホルダにバインドされる値を含む記述子の名前です。これはSQL識別子またはホスト変数とすることができます。

例

```
EXEC SQL OPEN a;
EXEC SQL OPEN d USING 1, 'test';
EXEC SQL OPEN c1 USING SQL DESCRIPTOR mydesc;
EXEC SQL OPEN :curname1;
```

互換性

OPENは標準SQLで規定されています。

関連項目

DECLARE, CLOSE

PREPARE

PREPARE — 実行のためにSQL文をプリペアします。

概要

```
PREPARE name FROM string
```

説明

PREPAREは実行用に文字列として動的に指定されたSQL文をプリペアします。これは、埋め込みプログラム内でも使用することができる、直接的なPREPARE SQL文とは異なります。EXECUTEコマンドを使用して、どちらの種類のプリペアド文を実行することができます。

パラメータ

prepared_name

プリペアド問い合わせ用の識別子です。

string

Cリテラル文字列、または、プリペア可能な文であるSELECT/INSERT/UPDATE/DELETEの1つを含むホスト変数、のいずれかです。

例

```
char *stmt = "SELECT * FROM test1 WHERE a = ? AND b = ?";

EXEC SQL ALLOCATE DESCRIPTOR outdesc;
EXEC SQL PREPARE foo FROM :stmt;

EXEC SQL EXECUTE foo USING SQL DESCRIPTOR indesc INTO SQL DESCRIPTOR outdesc;
```

互換性

PREPAREは標準SQLで規定されています。

関連項目

EXECUTE

SET AUTOCOMMIT

SET AUTOCOMMIT — 現在のセッションの自動コミット動作を設定します。

概要

```
SET AUTOCOMMIT { = | TO } { ON | OFF }
```

説明

SET AUTOCOMMITは現在のデータベースセッションの自動コミット動作を設定します。デフォルトでは埋め込みSQLプログラムは自動コミットモードではありません。このためCOMMITコマンドを必要なところで明示的に発行しなければなりません。このコマンドはセッションを、個々のSQL文それぞれが暗黙的にコミットされる、自動コミットモードに変更することができます。

互換性

SET AUTOCOMMITはPostgreSQL ECPGの拡張です。

SET CONNECTION

SET CONNECTION — データベース接続を選択します。

概要

```
SET CONNECTION [ TO | = ] connection_name
```

説明

SET CONNECTIONは、上書きされない限りすべてのコマンドが使用する、「現在の」データベース接続を設定します。

パラメータ

connection_name

CONNECTコマンドで確立したデータベース接続の名前です。

DEFAULT

接続をデフォルトの接続に設定します。

例

```
EXEC SQL SET CONNECTION TO con2;
EXEC SQL SET CONNECTION = con1;
```

互換性

SET CONNECTIONは標準SQLで規定されています。

関連項目

CONNECT, DISCONNECT

SET DESCRIPTOR

SET DESCRIPTOR — SQL記述子領域に情報を設定します。

概要

```
SET DESCRIPTOR descriptor_name descriptor_header_item = value [, ... ]
SET DESCRIPTOR descriptor_name VALUE number descriptor_item = value [, ...]
```

説明

SET DESCRIPTORはSQL記述子領域に値を投入します。その後、通常、記述子領域はプリペアド問い合わせ実行においてパラメータをバインドするために使用されます。

このコマンドは2つの構文があります。最初の構文は、特定のデータと独立した、記述子の「ヘッダ」に適用します。2番目の構文は、番号で識別される特定のデータに値を割り当てます。

パラメータ

`descriptor_name`

記述子の名前です。

`descriptor_header_item`

設定するヘッダ情報項目を識別するトークンです。記述子項目数を設定するCOUNTのみが現在サポートされています。

`number`

設定する記述子項目の番号です。番号は1から数えます。

`descriptor_item`

記述子内のどの項目の情報を設定するかを識別するトークンです。サポートされる項目のリストについては35.7.1を参照してください。

`value`

記述子項目に格納する値です。これはSQL定数またはホスト変数とすることができます。

例

```
EXEC SQL SET DESCRIPTOR indesc COUNT = 1;
EXEC SQL SET DESCRIPTOR indesc VALUE 1 DATA = 2;
EXEC SQL SET DESCRIPTOR indesc VALUE 1 DATA = :val1;
```

```
EXEC SQL SET DESCRIPTOR indesc VALUE 2 INDICATOR = :val1, DATA = 'some string';
EXEC SQL SET DESCRIPTOR indesc VALUE 2 INDICATOR = :val2null, DATA = :val2;
```

互換性

SET DESCRIPTORは標準SQLで規定されています。

関連項目

ALLOCATE DESCRIPTOR, GET DESCRIPTOR

TYPE

TYPE — 新しいデータ型を定義します。

概要

```
TYPE type_name IS ctype
```

説明

TYPEコマンドは新しいCの型を定義します。これは宣言セクションにtypedefを記述することと同じです。

ecpgが-cオプション付きで実行された場合にのみこのコマンドは認識されます。

パラメータ

`type_name`

新しい型の名前です。これは有効なCの型名でなければなりません。

`ctype`

Cの型指定です。

例

```
EXEC SQL TYPE customer IS
    struct
    {
        varchar name[50];
        int     phone;
    };

EXEC SQL TYPE cust_ind IS
    struct ind
    {
        short   name_ind;
        short   phone_ind;
    };
```

```
EXEC SQL TYPE c IS char reference;
EXEC SQL TYPE ind IS union { int integer; short smallint; };
EXEC SQL TYPE intarray IS int[AMOUNT];
EXEC SQL TYPE str IS varchar[BUFFERSIZ];
EXEC SQL TYPE string IS char[11];
```

以下にEXEC SQL TYPEを使用するプログラム例を示します。

```
EXEC SQL WHENEVER SQLERROR SQLPRINT;

EXEC SQL TYPE tt IS
    struct
    {
        varchar v[256];
        int     i;
    };

EXEC SQL TYPE tt_ind IS
    struct ind {
        short   v_ind;
        short   i_ind;
    };

int
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;
    tt t;
    tt_ind t_ind;
EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO testdb AS con1;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;

    EXEC SQL SELECT current_database(), 256 INTO :t:t_ind LIMIT 1;

    printf("t.v = %s\n", t.v.arr);
    printf("t.i = %d\n", t.i);

    printf("t_ind.v_ind = %d\n", t_ind.v_ind);
    printf("t_ind.i_ind = %d\n", t_ind.i_ind);

    EXEC SQL DISCONNECT con1;

    return 0;
}
```

このプログラムの出力は以下のようになります。

```
t.v = testdb
t.i = 256
t_ind.v_ind = 0
t_ind.i_ind = 0
```

互換性

TYPEコマンドはPostgreSQLの拡張です。

VAR

VAR — 変数を定義します。

概要

```
VAR varname IS ctype
```

説明

VARコマンドは新しいCデータ型にホスト変数を割り当てます。ホスト変数は宣言セクションで前もって宣言されていなければなりません。

パラメータ

`varname`

Cの変数名です。

`ctype`

Cの型指定です。

例

```
Exec sql begin declare section;
short a;
exec sql end declare section;
EXEC SQL VAR a IS int;
```

互換性

VARコマンドはPostgreSQLの拡張です。

WHENEVER

WHENEVER — SQL文により特定の分類の条件が発生する時に行う動作を指定します。

概要

```
WHENEVER { NOT FOUND | SQLERROR | SQLWARNING } action
```

説明

SQL実行の結果において特殊な状態（行がない、SQL警告またはSQLエラー）で呼び出される動作を定義します。

パラメータ

パラメータの説明については35.8.1を参照してください。

例

```
EXEC SQL WHENEVER NOT FOUND CONTINUE;
EXEC SQL WHENEVER NOT FOUND DO BREAK;
EXEC SQL WHENEVER NOT FOUND DO CONTINUE;
EXEC SQL WHENEVER SQLWARNING SQLPRINT;
EXEC SQL WHENEVER SQLWARNING DO warn();
EXEC SQL WHENEVER SQLERROR sqlprint;
EXEC SQL WHENEVER SQLERROR CALL print2();
EXEC SQL WHENEVER SQLERROR DO handle_error("select");
EXEC SQL WHENEVER SQLERROR DO sqlnotice(NULL, NONO);
EXEC SQL WHENEVER SQLERROR DO sqlprint();
EXEC SQL WHENEVER SQLERROR GOTO error_label;
EXEC SQL WHENEVER SQLERROR STOP;
```

以下は、結果セットを通したループ処理を扱うためにWHENEVER NOT FOUND BREAKを使用する典型的なアプリケーションです。

```
int
main(void)
{
    EXEC SQL CONNECT TO testdb AS con1;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;
    EXEC SQL ALLOCATE DESCRIPTOR d;
```

```
    EXEC SQL DECLARE cur CURSOR FOR SELECT current_database(), 'hoge', 256;
    EXEC SQL OPEN cur;


    /* 結果集合の最後に到達したら、whileループから抜ける */
    EXEC SQL WHENEVER NOT FOUND DO BREAK;

    while (1)
    {
        EXEC SQL FETCH NEXT FROM cur INTO SQL DESCRIPTOR d;
        ...
    }

    EXEC SQL CLOSE cur;
    EXEC SQL COMMIT;

    EXEC SQL DEALLOCATE DESCRIPTOR d;
    EXEC SQL DISCONNECT ALL;

    return 0;
}
```

互換性

WHENEVERは標準SQLで規定されていますが、ほとんどの動作はPostgreSQLの拡張です。

35.15. Informix互換モード

ecpgを*Informix互換モード*というモードで動作させることができます。このモードが有効ならば、Informix E/SQL用のInformixプリプロセッサであるかのように動作します。一般的にいうと、これにより埋め込みSQLコマンドを導入する際にEXEC SQLプリミティブの代わりにドル記号を使用することができます。

```
$int j = 3;
$CONNECT TO :dbname;
$CREATE TABLE test(i INT PRIMARY KEY, j INT);
$INSERT INTO test(i, j) VALUES (7, :j);
$COMMIT;
```

注記

$とその後に続くinclude、define、ifdefなどのプリプロセッサ指示子の間に空白文字を含めてはなりません。こうしないと、プリプロセッサはトークンをホスト変数として解析します。

INFORMIX、INFORMIX_SEという2つの互換モードがあります。

互換モードを使用するプログラムをリンクする際、ECPGに同梱されるlibcompatとリンクすることを忘れないでください。

以前に説明した構文上の飾りの他に、Informix互換モードでは、入力、出力、データ変換関数、E/SQLからECPGで既知の埋め込みSQL文変換に関する関数もいくつか移植しています。

Informix互換モードはECPGのpgtypeslibライブラリと密接に関係しています。pgtypeslibはSQLデータ型とCホストプログラム内のデータ型を対応付けし、ほとんどのInformix互換モードで追加された関数を使用してこれらのCホストプログラム型を操作することができます。しかし、互換範囲は制限されています。これはInformixの動作を真似ることはしません。これを使用して、多少は同じ名前で同じ基本動作を行う関数を操作、提供できますが、Informixを使用しているのであれば、完全な置き換えにはなりません。さらに一部のデータ型は異なります。例えば、PostgreSQLの日付時刻やinterval型ではYEAR TO MINUTEのような範囲を持ちませんので、これらはECPGではサポートできないことがわかります。

35.15.1. 追加の型

右側を切り詰めた文字列データを格納するInformixの特別な"string"仮想型はtypedefを使用せずともInformixモードでサポートされるようになりました。実際Informixモードでは、ECPGはtypedef sometype string;を含むソースファイルの処理を拒絶します。

```
EXEC SQL BEGIN DECLARE SECTION;

string userid; /* この変数は切り詰められたデータを含むことになる */
EXEC SQL END DECLARE SECTION;

EXEC SQL FETCH MYCUR INTO :userid;
```

35.15.2. 追加または存在しない埋め込みSQL文

CLOSE DATABASE

このSQL文は現在の接続を閉じます。実際、これはECPGのDISCONNECT CURRENTと同義です。

```
$CLOSE DATABASE;                /* 現在の接続を閉じる */
EXEC SQL CLOSE DATABASE;
```

FREE cursor_name

InformixのESQL/Cと比べECPGの動作方法に違いがあります(つまり純粋に文法の変換がどの段階で行われ、背後の実行時ライブラリにどの段階で依存するか)ので、ECPGにはFREE cursor_name文はありません。このためECPGにおいて、DECLARE CURSORがカーソル名を使用する実行時ライブラリ内の関数呼び出しに変換されません。これはECPG実行時ライブラリ内ではSQLカーソルの実行状況を保持しておらず、PostgreSQLサーバ内のみで保持していることを意味します。

FREE statement_name

FREE statement_nameはDEALLOCATE PREPARE statement_nameの類義語です。

35.15.3. Informix互換SQLDA記述子領域

Informix互換モードは35.7.2の説明と異なる構造体をサポートします。以下を参照してください。

```
struct sqlvar_compat
{
    short   sqltype;
    int     sqllen;
    char   *sqldata;
    short  *sqlind;
    char   *sqlname;
    char   *sqlformat;
    short   sqlitype;
    short   sqlilen;
    char   *sqlidata;
    int     sqlxid;
    char   *sqltypename;
    short   sqltypelen;
    short   sqlownerlen;
    short   sqlsourcetype;
    char   *sqlownername;
    int     sqlsourceid;
    char   *sqlilongdata;
    int     sqlflags;
    void   *sqlreserved;
};

struct sqlda_compat
{
    short  sqld;
    struct sqlvar_compat *sqlvar;
    char   desc_name[19];
    short  desc_occ;
    struct sqlda_compat *desc_next;
    void  *reserved;
};

typedef struct sqlvar_compat    sqlvar_t;
typedef struct sqlda_compat     sqlda_t;
```

大域的な属性を以下に示します。

sqld

SQLDA記述子内のフィールド数です。

sqlvar

フィールド単位の属性へのポインタです。

desc_name

未使用です。ゼロバイトで埋められます。

desc_occ

割り当てられた構造体のサイズです。

desc_next

結果セットに複数のレコードが含まれる場合、次のSQLDA構造体へのポインタです。

reserved

未使用のポインタでNULLが含まれます。Informix互換のために保持されます。

フィールド毎の属性を以下に示します。これらはsqlvar配列内に格納されます。

sqltype

フィールドの型です。定数はsqltypes.h内に記載されています。

sqllen

フィールドデータ長です。

sqldata

フィールドデータへのポインタです。このポインタはchar *型です。指し示されるデータはバイナリ書式です。以下に例を示します。

```
int intval;

switch (sqldata->sqlvar[i].sqltype)
{
    case SQLINTEGER:
        intval = *(int *)sqldata->sqlvar[i].sqldata;
        break;
  ...
}
```

sqlind

NULL指示子へのポインタです。DESCRIBEまたはFETCHで返される場合、常に有効なポインタです。EXECUTE ... USING sqlda;への入力として使用される場合、NULLポインタ値はこのフィールドの値が

非NULLであることを意味します。さもなくば、有効なポインタとsqlitypeは適切に設定されなければなりません。以下に例を示します。

```
if (*(int2 *)sqldata->sqlvar[i].sqlind != 0)
    printf("value is NULL\n");
```

sqlname

フィールド名です。ゼロ終端の文字列です。

sqlformat

Informixでは予約されています。このフィールドのPQfformatの値です。

sqlitype

NULL指示子データの型です。サーバからデータが返される場合は常にSQLSMINTです。パラメータ付き問い合わせでSQLDAが使用される時、データは集合型にしたがって扱われます。

sqlilen

NULL指示子データの長さです。

sqlxid

フィールドの拡張型で、PQftypeの結果です。

sqltypename
sqltypelen
sqlownerlen
sqlsourcetype
sqlownername
sqlsourceid
sqlflags
sqlreserved

未使用です。

sqlilongdata

sqllenが32キロバイトより大きい場合sqldataと同じです。

以下に例を示します。

```
EXEC SQL INCLUDE sqlda.h;


    sqlda_t        *sqlda; /* これは埋め込まれたDECLARE SECTIONの中にある必要はない */

    EXEC SQL BEGIN DECLARE SECTION;
    char *prep_stmt = "select * from table1";
    int i;
```

```
    EXEC SQL END DECLARE SECTION;

    ...

    EXEC SQL PREPARE mystmt FROM :prep_stmt;

    EXEC SQL DESCRIBE mystmt INTO sqlda;

    printf("# of fields: %d\n", sqlda->sqld);
    for (i = 0; i < sqlda->sqld; i++)
      printf("field %d: \"%s\"\n", sqlda->sqlvar[i]->sqlname);

    EXEC SQL DECLARE mycursor CURSOR FOR mystmt;
    EXEC SQL OPEN mycursor;
    EXEC SQL WHENEVER NOT FOUND GOTO out;

    while (1)
    {
      EXEC SQL FETCH mycursor USING sqlda;
    }

    EXEC SQL CLOSE mycursor;


    free(sqlda); /* 主な構造体はすべてfree()される、
                  * sqldaとsqlda->sqlvarは1つの割り当て領域内にある */
```

より詳細についてはsqlda.hヘッダファイルとsrc/interfaces/ecpg/test/compat_informix/sqlda.pgcリグレッションテストを参照してください。

35.15.4. 追加関数

decadd

2つのdecimal型変数を加算します。

```
int decadd(decimal *arg1, decimal *arg2, decimal *sum);
```

この関数は、decimal型の最初の演算項目(arg1)へのポインタ、decimal型の2番目の演算項目(arg2)へのポインタ、加算結果を格納するdecimal型値(sum)へのポインタを受付けます。成功すると、この関数は0を返します。オーバーフローが発生した場合はECPG_INFORMIX_NUM_OVERFLOWが、アンダーフローの場合はECPG_INFORMIX_NUM_UNDERFLOWが返ります。この他の失敗が発生した場合は-1が返り、errnoにはpgtypeslibにおける対応するerrno番号が設定されます。

deccmp

2つのdecimal型変数を比較します。

```
int deccmp(decimal *arg1, decimal *arg2);
```

この関数は、最初のdecimal値(arg1)へのポインタ、2番目のdecimal値(arg2)へのポインタを受付け、どちらが大きいかを示すint値を返します。

- arg1が指し示す値がarg2が指し示す値より大きければ1。
- arg1が指し示す値がarg2が指し示す値より小さければ-1。
- arg1が指し示す値とarg2が指し示す値が同じならば0。

deccopy

decimal値をコピーします。

```
void deccopy(decimal *src, decimal *target);
```

この関数は、最初の引数としてコピー元のdecimal値(src)へのポインタ、2番目の引数としてdecimal型のコピー先構造体(target)へのポインタを受付けます。

deccvasc

ASCII表現からdecimal型に値を変換します。

```
int deccvasc(char *cp, int len, decimal *np);
```

この関数は、変換対象の文字列表現を持つ文字列(cp)へのポインタとその文字列長lenを受付けます。npはこの操作結果を格納するdecimal型の値へのポインタです。

有効な書式の例は以下の通りです。-2、.794、+3.44、592.49E07、-32.84e-4。

この関数は成功時0を返します。オーバーフローやアンダーフローが発生した場合はECPG_INFORMIX_NUM_OVERFLOWやECPG_INFORMIX_NUM_UNDERFLOWが返されます。ASCII表現の解析ができなかった場合はECPG_INFORMIX_BAD_NUMERICが、指数部分の解析に問題がある場合はECPG_INFORMIX_BAD_EXPONENTが返されます。

deccvdbl

double型の値をdecimal型の値に変換します。

```
int deccvdbl(double dbl, decimal *np);
```

この関数は、最初の引数として変換対象のdouble型の変数(dbl)を受付けます。2番目の引数(np)として、この関数は操作結果を格納するdecimal型変数へのポインタを受付けます。

この関数は成功時に0を返します。変換が失敗した場合は負の値が返ります。

deccvint

int型の値をdecimal型の値に変換します。

```
int deccvint(int in, decimal *np);
```

この関数は最初の引数として、変換対象のint型変数(in)を受付けます。2番目の引数(np)として、この関数は変換結果を格納するdecimal型変数へのポインタを受付けます。

この関数は成功時に0を返します。変換が失敗した場合は負の値が返ります。

deccvlong

long型の値をdecimal型の値に変換します。

```
int deccvlong(long lng, decimal *np);
```

この関数は最初の引数として、変換対象のlong型変数(lng)を受付けます。2番目の引数(np)として、この関数は変換結果を格納するdecimal型変数へのポインタを受付けます。

この関数は成功時に0を返します。変換が失敗した場合は負の値が返ります。

decdiv

2つのdecimal型変数の除算を行います。

```
int decdiv(decimal *n1, decimal *n2, decimal *result);
```

この関数は、1番目の演算項目(n1)と2番目の演算項目(n2)となる変数のポインタを受付け、n1/n2を計算します。resultは、操作結果を格納する変数へのポインタです。

成功時0が返され、除算の失敗時には負の値が返されます。オーバーフローやアンダーフローが発生した場合、この関数はそれぞれECPG_INFORMIX_NUM_OVERFLOW、ECPG_INFORMIX_NUM_UNDERFLOWを返します。0割りが発生した場合はECPG_INFORMIX_DIVIDE_ZEROが返されます。

decmul

2つのdecimal型変数を乗算します。

```
int decmul(decimal *n1, decimal *n2, decimal *result);
```

この関数は、1番目の演算項目(n1)と2番目の演算項目(n2)となる変数のポインタを受付け、n1*n2を計算します。resultは、操作結果を格納する変数へのポインタです。

成功時0が返され、乗算の失敗時には負の値が返されます。オーバーフローやアンダーフローが発生した場合、この関数はそれぞれECPG_INFORMIX_NUM_OVERFLOW、ECPG_INFORMIX_NUM_UNDERFLOWを返します。

decsub

10進数型値同士の引算を行います。

```
int decsub(decimal *n1, decimal *n2, decimal *result);
```

この関数は、1番目の演算項目(n1)と2番目の演算項目(n2)となる変数のポインタを受付け、n1-n2を計算します。resultは、操作結果を格納する変数へのポインタです。

成功時0が返され、減算の失敗時には負の値が返されます。オーバーフローやアンダーフローが発生した場合、この関数はそれぞれECPG_INFORMIX_NUM_OVERFLOW、ECPG_INFORMIX_NUM_UNDERFLOWを返します。

dectoasc

decimal型変数をC char* 文字列のASCII表現に変換します。

```
int dectoasc(decimal *np, char *cp, int len, int right)
```

この関数はdecimal型変数(np)のポインタを受け付け、テキスト表現に変換します。cpは変換結果を格納するためのバッファです。rightパラメータは、decimal小数点の右側の何桁を出力するかを指定します。結果はこの10進桁数で丸められます。rightを-1にすることで、すべての有効な桁数が出力されるようになります。lenで示す出力バッファ長が、最後のNULL文字を含むテキスト表現を格納するのには不十分であった場合、結果には*という1文字が格納され、-1が返されます。

この関数は、cpバッファが小さすぎる場合に-1を返します。メモリ不足の場合はECPG_INFORMIX_OUT_OF_MEMORYを返します。

dectodbl

decimal型変数をdoubleに変換します。

```
int dectodbl(decimal *np, double *dblp);
```

この関数は変換対象のdecimal型変数(np)のポインタと処理結果を格納するdouble変数(dblp)へのポインタを受け付けます。

成功時0が、変換失敗時負の値が返されます。

dectoint

decimal型変数を整数型に変換します。

```
int dectoint(decimal *np, int *ip);
```

この関数は変換対象のdecimal型変数(np)のポインタと処理結果を格納するint型変数(ip)へのポインタを受け付けます。

成功時0が、変換失敗時負の値が返されます。オーバーフローが発生した場合はECPG_INFORMIX_NUM_OVERFLOWが返されます。

このECPGの実装はInformixの実装と異なることに注意してください。Informixでは、整数範囲に-32767から32767までという制限をしていますが、ECPGでの制限はアーキテクチャに依存(-INT_MAX .. INT_MAX)します。

dectolong

decimal型変数をlong型に変換します。

```
int dectolong(decimal *np, long *lngp);
```

この関数は変換対象のdecimal型変数(np)のポインタと処理結果を格納するlong変数(lngp)へのポインタを受け付けます。

成功時0が、変換失敗時負の値が返されます。オーバーフローが発生した場合はECPG_INFORMIX_NUM_OVERFLOWが返されます。

このECPGの実装はInformixの実装と異なることに注意してください。Informixでは、整数範囲に-2,147,483,647から2,147,483,647までという制限をしていますが、ECPGでの制限はアーキテクチャに依存(-LONG_MAX .. LONG_MAX)します。

rdatestr

date型をC char*文字列に変換します。

```
int rdatestr(date d, char *str);
```

この関数は2つの引数を受付けます。最初の引数は変換対象のdate型(d)、2番目は変換後の文字列へのポインタです。出力書式は常にyyyy-mm-ddですので、少なくとも11文字（NULL終端を含む）を結果文字列に割り当てなければなりません。

この関数は成功時0を、エラー時負の値を返します。

このECPGの実装はInformixの実装と異なることに注意してください。Informixでは、環境変数により書式を変更できますが、ECPGでは出力書式を変更することはできません。

rstrdate

date型のテキスト表現を解析します。

```
int rstrdate(char *str, date *d);
```

この関数は、変換対象のdate型のテキスト表現(str)とdate型変数のポインタ(d)を受付けます。この関数では書式マスクを指定することができません。Informixのデフォルトの書式マスクであるmm/dd/yyyyを使用します。内部的には、この関数はrdefmtdateを使用して実装しています。したがってrstrdateは速くありません。もし選択肢があるのであれば、書式マスクを明示的に指定することができるrdefmtdateを選択すべきです。

この関数はrdefmtdateと同様の値を返します。

rtoday

現在の日付を（date型で）入手します。

```
void rtoday(date *d);
```

この関数はdate型変数(d)へのポインタを受付け、そこに現在の日付を格納します。

内部的には、この関数はPGTYPESdate_today関数を使用します。

rjulmdy

date型変数から、日、月、年の値を取り出します。

```
int rjulmdy(date d, short mdy[3]);
```

この関数は日付d、3つのshort integer型の値からなる配列mdyへのポインタを受付けます。この変数名はその並びを表し、mdy[0]には月数、mdy[1]には日数が、mdy[2]には年が入ります。

現在この関数は常に0を返します。

内部的にはこの関数はPGTYPESdate_julmdy関数を使用します。

rdefmtdate

書式マスクを使用して、文字列をdate型の値に変換します。

```
int rdefmtdate(date *d, char *fmt, char *str);
```

この関数は、処理結果を格納するためのdate型へのポインタ(d)、日付を解析するための書式マスク(fmt)、dateのテキスト表現を含むCのchar*文字列(str)を受付けます。テキスト表現は書式マスクに合った表現であることが仮定されています。しかし、文字列と書式マスクを1:1に対応付けする必要はありません。この関数は並んだ順番に解析し、年の位置を表すyyまたはyyyyを、月の位置を表すmmを、日の位置を表すddを検索します。

この関数は以下の値を返します。

- 0 - 関数が正常に終了しました。
- ECPG_INFORMIX_ENOSHORTDATE - 日付に、日、月、年を区切る文字がありませんでした。この場合、入力文字列は6バイト、8バイトのいずれかでなければなりませんが、そうではありませんでした。
- ECPG_INFORMIX_ENOTDMY - 書式文字列が正しく年月日の順番を示していません。
- ECPG_INFORMIX_BAD_DAY - 入力文字列に有効な日が含まれていません。
- ECPG_INFORMIX_BAD_MONTH - 入力文字列に有効な月が含まれていません。
- ECPG_INFORMIX_BAD_YEAR - 入力文字列に有効な年が含まれていません。

内部的には、この関数はPGTYPESdate_defmt_asc関数を使用して実装しています。この関数の説明には、入力例の表がありますので、こちらも参照してください。

rfmtdate

書式マスクを使用してdate型変数をテキスト表現に変換します。

```
int rfmtdate(date d, char *fmt, char *str);
```

この関数は変換対象の日付(d)、書式マスク(fmt)、日付のテキスト表現を格納する文字列(str)を受付けます。

成功時0、エラーが発生した場合は負の値が返されます。

内部的にはこの関数はPGTYPESdate_fmt_asc関数を使用します。例が記載されていますので、こちらも参照してください。

rmdyjul

日付の日、月、年を表す3つのshort integer型からなる配列から日付型の値を作成します。

```
int rmdyjul(short mdy[3], date *d);
```

この関数は3つのshort integer型からなる配列(mdy)と処理結果を格納するdate型変数へのポインタを受付けます。

現在この関数は常に0を返します。

内部的にはこの関数はPGTYPESdate_mdyjul関数を使用して実装しています。

rdayofweek

日付型の値の週内日数を示す値を返します。

```
int rdayofweek(date d);
```

この関数はdate型変数dをその唯一の引数として受付け、その日付の週内日数を示す整数を返します。

- 0 - 日曜
- 1 - 月曜
- 2 - 火曜
- 3 - 水曜
- 4 - 木曜
- 5 - 金曜
- 6 - 土曜

内部的にはこの関数は PGTYPESdate_dayofweek関数を使用して実装しています。

dtcurrent

現在のタイムスタンプを取り出します。

```
void dtcurrent(timestamp *ts);
```

この関数は現在のタイムスタンプを受け取り、tsが指し示すタイムスタンプ型変数に格納します。

dtcvasc

テキスト表現からtimestamp型変数にタイムスタンプを解析します。

```
int dtcvasc(char *str, timestamp *ts);
```

この関数は対象の文字列(str)と処理結果を格納するtimestamp型変数(ts)へのポインタを受付けます。

この関数は成功時0を返し、エラー時負の値を返します。

内部的にはこの関数はPGTYPEStimestamp_from_asc関数を使用します。入力例の表がありますので、こちらも参照してください。

dtcvfmtasc

書式マスクを使用してタイムスタンプのテキスト表現をtimestamp型変数に変換します。

```
dtcvfmtasc(char *inbuf, char *fmtstr, timestamp *dtvalue)
```

この関数は、対象とする文字列(inbuf)、使用する書式マスク(fmtstr)、処理結果を格納するtimestamp変数(dtvalue)へのポインタを受付けます。

この関数はPGTYPEStimestamp_defmt_asc関数を使用して実装されています。使用可能な書式指定のリストがありますので、こちらも参照してください。

この関数は成功時に0を、エラー時負の値を返します。

dtsub

timestamp型同士で減算を行い、interval型変数を返します。

```
int dtsub(timestamp *ts1, timestamp *ts2, interval *iv);
```

この関数はts1が指し示すtimestamp型変数からts2が指し示すtimestamp型変数を引きます。結果はivが指し示すinterval型変数に格納されます。

成功時この関数は0を返し、エラー時負の値を返します。

dttoasc

timestamp型変数をC char*文字列に変換します。

```
int dttoasc(timestamp *ts, char *output);
```

この関数は対象のtimestamp型変数(ts)へのポインタ、処理結果を格納する文字列(output)を受付けます。これはtsを標準SQLに従うテキスト表現(YYYY-MM-DD HH:MM:SSとして定義)に変換します。

成功時この関数は0を返し、エラー時負の値を返します。

dttofmtasc

書式マスクを使用してtimestamp型変数をC char*に変換します。

```
int dttofmtasc(timestamp *ts, char *output, int str_len, char *fmtstr);
```

この関数は、最初の引数として変換対象のタイムスタンプ(ts)を、出力バッファのポインタ(output)、出力バッファで割当て可能な最大長 (str_len)、変換に使用する書式マスク(fmtstr)を受付けます。

成功時この関数は0を返します。エラーが発生した場合は負の値を返します。

内部的に、この関数はPGTYPEStimestamp_fmt_asc関数を使用します。使用できる書式マスクに関する情報がありますので、こちらも参照してください。

intoasc

interval型変数をC char*文字列に変換します。

```
int intoasc(interval *i, char *str);
```

この関数は、変換対象のinterval型変数(i)へのポインタ、処理結果を格納する文字列(str)を受付けます。これはiを標準SQLに従うテキスト表現(YYYY-MM-DD HH:MM:SSとして定義)に変換します。

成功時、この関数は0を返します。エラーが発生した場合は負の値を返します。

rfmtlong

long integer値を書式マスクを使用してテキスト表現に変換します。

```
int rfmtlong(long lng_val, char *fmt, char *outbuf);
```

この関数は、long型の値lng_val、書式マスクfmt、出力バッファoutbufへのポインタを受付けます。これはlong型の値を書式マスクに従ってテキスト表現に変換します。

書式マスクは以下の書式指定文字を組み合わせることができます。

- * (アスタリスク) - この位置が空白ならばアスタリスクで埋めます。
- & (アンパサンド) - この位置が空白ならば0で埋めます。
- # - 先頭のゼロを空白に変換します。
- < - 文字列内で数値を左そろえします。
- , (カンマ) - 4桁以上の数値をカンマで区切った3桁にグループ化します。
- . (ピリオド) - この文字は数値から小数部分を区別します。
- - (マイナス) - 数値が負の場合、マイナス記号を付けます。
- + (プラス) - 数値が正の場合プラス記号を付けます。
- (- これは負の値の先頭のマイナス記号を置き換えます。マイナス記号は現れません。
-) - この文字はマイナス記号を置き換え、負の値の最後に出力します。
- $ - 通貨記号

rupshift

文字列を大文字に変換します。

```
void rupshift(char *str);
```

この関数は文字列へのポインタを受付け、すべての小文字を大文字に変換します。

byleng

文字列内の文字数を返します。ただし、末尾の空白は数えません。

```
int byleng(char *str, int len);
```

この関数は最初の引数として、固定長の文字列(str)を、2番目の引数としてその文字列長 (len)想定しています。これは、文字列から末尾の空白を取り除いた、有効文字の数を返します。

ldchar

固定長の文字列をNULL終端の文字列に複製します。

```
void ldchar(char *src, int len, char *dest);
```

この関数はコピー対象の固定長の文字列(src)、文字列長(len)、格納先メモリ(dest)へのポインタを受付けます。destが指し示す文字列には少なくともlen+1バイトを割り当てなければならない点に注意してください。この関数は多くてもlenバイトを新しい場所にコピーします。(元の文字列が末尾に空白文字を持つ場合に少なくなります。) そして、NULL終端を付与します。

rgetmsg

```
int rgetmsg(int msgnum, char *s, int maxsize);
```

この関数は存在しますが、現在実装されていません。

rtypalign

```
int rtypalign(int offset, int type);
```

この関数は存在しますが、現在実装されていません。

rtypmsize

```
int rtypmsize(int type, int len);
```

この関数は存在しますが、現在実装されていません。

rtypwidth

```
int rtypwidth(int sqltype, int sqllen);
```

この関数は存在しますが、現在実装されていません。

rsetnull

変数にNULLを設定します。

```
int rsetnull(int t, char *ptr);
```

この関数は、変数の種類を示す整数とC char*にキャストした変数自体へのポインタを受付けます。

以下の種類が存在します。

- CCHARTYPE - charまたは char*型の変数用
- CSHORTTYPE - short int型の変数用
- CINTTYPE - int型の変数用
- CBOOLTYPE - boolean型の変数用
- CFLOATTYPE - float型の変数用
- CLONGTYPE - long型の変数用
- CDOUBLETYPE - double型の変数用
- CDECIMALTYPE - decimal型の変数用
- CDATETYPE - date型の変数用
- CDTIMETYPE - timestamp型の変数用

以下にこの関数の呼び出し例を示します。

```
$char c[] = "abc       ";
$short s = 17;
$int i = -74874;

rsetnull(CCHARTYPE, (char *) c);
rsetnull(CSHORTTYPE, (char *) &s);
rsetnull(CINTTYPE, (char *) &i);
```

risnull

変数がNULLか検査します。

```
int risnull(int t, char *ptr);
```

この関数は検査する変数の種類(t)、変数(ptr)へのポインタを受付けます。後者はchar*にキャストする必要があることに注意してください。取り得る変数種類については rsetnull関数を参照してください。

この関数の使用方法の例を示します。

```
$char c[] = "abc       ";
$short s = 17;
$int i = -74874;
```

```
risnull(CCHARTYPE, (char *) c);
risnull(CSHORTTYPE, (char *) &s);
risnull(CINTTYPE, (char *) &i);
```

35.15.5. 追加の定数

ここで示す定数はすべてエラーを示すものであり、負の値を表すように定義されていることに注意してください。また、他の定数の説明では、現在の実装で定数が表す数値がわかります。しかし、この数値に依存してはなりません。しかし、これらのすべてが負の値であることに依存することは可能です。

`ECPG_INFORMIX_NUM_OVERFLOW`

計算時にオーバーフローが発生した場合、関数はこの値を返します。内部的には-1200（Informixの定義）と定義されています。

`ECPG_INFORMIX_NUM_UNDERFLOW`

計算時にアンダーフローが発生した場合、関数はこの値を返します。内部的には-1201（Informixの定義）と定義されています。

`ECPG_INFORMIX_DIVIDE_ZERO`

計算時にゼロ除算が発生した場合、関数はこの値を返します。内部的には-1202（Informixの定義）と定義されています。

`ECPG_INFORMIX_BAD_YEAR`

日付の解析時に年の値が不正であった場合、関数はこの値を返します。内部的には-1204（Informixの定義）と定義されています。

`ECPG_INFORMIX_BAD_MONTH`

日付の解析時に月の値が不正であった場合、関数はこの値を返します。内部的には-1205（Informixの定義）と定義されています。

`ECPG_INFORMIX_BAD_DAY`

日付の解析時に日の値が不正であった場合、関数はこの値を返します。内部的には-1206（Informixの定義）と定義されています。

`ECPG_INFORMIX_ENOSHORTDATE`

解析処理が短縮日付表現を必要としているが、正しい長さの日付文字列が得られなかった場合、関数はこの値を返します。内部的には-1209（Informixの定義）と定義されています。

`ECPG_INFORMIX_DATE_CONVERT`

日付の書式付けの時にエラーが発生した場合、関数はこの値を返します。内部的には-1210（Informixの定義）と定義されています。

ECPG_INFORMIX_OUT_OF_MEMORY

操作時にメモリが不足した場合、関数はこの値を返します。内部的には-1211（Informixの定義）と定義されています。

ECPG_INFORMIX_ENOTDMY

解析処理が書式マスク（mmddyyのような）が存在することを前提としているが、すべてのフィールドが正しく列挙されていない場合、関数はこの値を返します。内部的には-1212（Informixの定義）と定義されています。

ECPG_INFORMIX_BAD_NUMERIC

解析処理がエラーのため数値のテキスト表現を解析できなかった場合や数値変数の少なくとも1つが無効のため数値変数を使用した計算を完了できなかった場合、関数はこの値を返します。内部的には-1213（Informixの定義）と定義されています。

ECPG_INFORMIX_BAD_EXPONENT

解析処理が指数の解析を行うことができなかった場合、関数はこの値を返します。内部的には-1216（Informixの定義）と定義されています。

ECPG_INFORMIX_BAD_DATE

解析処理が日付を解析できなかった場合、関数はこの値を返します。内部的には-1218（Informixの定義）と定義されています。

ECPG_INFORMIX_EXTRA_CHARS

解析処理が追加の文字列を解析できなかった場合、関数はこの値を返します。内部的には-1264（Informixの定義）と定義されています。

35.16. 内部

本節では内部的なECPGの動作を説明します。この情報はECPGの使用方法を理解する手助けとして有用なことがあります。

ecpgによって出力に書き込まれる最初の4行は固定されています。2行はコメントで、残り2行はライブラリとのインタフェースのために必要なインクルード行です。その後、プリプロセッサはファイル全体を読み取り、出力に書き出します。通常は、単にすべてそのまま出力に書き出します。

EXEC SQLを検出すると、間に入り、それを変更します。このコマンドはEXEC SQLで始まり、;で終わります。この間のすべてはSQL文として扱われ、変数の置換のために解析されます。

変数置換は、シンボルがコロン（:）から始まる場合に発生します。その名前の変数が、EXEC SQL DECLAREセクションで事前に宣言された変数の中から検索されます。

ライブラリ内で最も重要な関数はECPGdoです。これが、ほとんどのコマンドの実行を管理します。可変長の引数をとります。すべてのプラットフォームで問題にならないことを祈っていますが、これは50程度の引数まで簡単に追加できます。

引数を以下に示します。

行番号

元の行の行番号です。エラーメッセージ内でのみ使用されます。

文字列

発行すべきSQLコマンドです。入力変数、つまり、コンパイル時に未知だったがそのコマンド内に与えるべき変数によって変更されます。変数が文字列内に挿入される箇所は?となっています。

入力変数

すべての入力変数は10個の引数を作成します（後述）。

ECPGt_EOIT

入力変数がもうないことを表すenumです。

出力変数

すべての出力変数は10個の引数を作成します（後述）。これらの変数は関数によって埋められます。

ECPGt_EORT

変数がもうないことを表すenumです。

SQLコマンドの一部となるすべての変数に対して、この関数は以下の10個の引数を生成します。

1. 特別シンボルとしての型。
2. 値へのポインタ、もしくはポインタのポインタ。
3. 変数がcharかvarcharの場合はそのサイズ。
4. 配列の要素数（配列取り出し用）。
5. 配列の次の要素のオフセット（配列取り出し用）。
6. 特別シンボルとしての指示子変数の型。
7. 指示子変数へのポインタ。
8. 0
9. 指示子配列内の要素数（配列取り出し用）。
10. 指示子配列内の次要素へのオフセット（配列取り出し用）。

すべてのSQLコマンドがこの方法で扱われるわけではないことに注意してください。例えば、以下のカーソルを開くSQL文は出力にコピーされません。

```
EXEC SQL OPEN cursor;
```

その代わりにカーソルのDECLAREコマンドがOPENコマンドの場所で使用されます。実際にこのコマンドがカーソルを開くからです。

以下に、foo.pgcファイルに対するプリプロセッサの出力を完全に説明する例を示します（プリプロセッサのバージョンによって詳細が異なっているかもしれません）。

```
EXEC SQL BEGIN DECLARE SECTION;
int index;
int result;
EXEC SQL END DECLARE SECTION;
...
EXEC SQL SELECT res INTO :result FROM mytable WHERE index = :index;
```

これは以下に翻訳されます。

```
/* Processed by ecpg (2.6.0) */
/* These two include files are added by the preprocessor */
#include <ecpgtype.h>;
#include <ecpglib.h>;

/* exec sql begin declare section */

#line 1 "foo.pgc"

 int index;
 int result;
/* exec sql end declare section */
...
ECPGdo(__LINE__, NULL, "SELECT res FROM mytable WHERE index = ?     ",
        ECPGt_int,&(index),1L,1L,sizeof(int),
        ECPGt_NO_INDICATOR, NULL , 0L, 0L, 0L, ECPGt_EOIT,
        ECPGt_int,&(result),1L,1L,sizeof(int),
        ECPGt_NO_INDICATOR, NULL , 0L, 0L, 0L, ECPGt_EORT);
#line 147 "foo.pgc"
```

ここで可読性のためにインデントを付けています。プリプロセッサが行ったものではありません。

第36章 情報スキーマ

情報スキーマは、現在のデータベースで定義されたオブジェクトについての情報を持つビューの集合から構成されます。情報スキーマは標準SQLで定義されています。したがって、PostgreSQLに特化し、実装上の事項にならって作成されるシステムカタログとは異なり、移植性があり、安定性を保持できるものと期待できます。しかし、情報スキーマのビューには、PostgreSQL固有の機能についての情報が含まれていません。これに問い合わせを行うためには、システムカタログもしくはPostgreSQL固有のビューに問い合わせを行う必要があります。

> **注記**
>
> 制約情報についてデータベースに問い合わせるとき、一行を返すことが想定される標準に準拠したクエリが数行の結果を返す場合があります。これは、制約名がスキーマ内で一意になることをSQL標準が要求しているのに対して、PostgreSQLはこの制約を強制しないためです。PostgreSQLは自動生成される制約の名前がスキーマ内で重複することを防ぎますが、ユーザは重複する名前を指定できます。
>
> この問題は、check_constraint_routine_usageやcheck_constraints、domain_constraints、referential_constraintsといった情報スキーマビューを検索するときに表面化することがあります。いくつかの他のビューにも同様の問題がありますが、重複行を識別する助けになるテーブル名を含んでいます。例えばconstraint_column_usageやconstraint_table_usage、table_constraintsなどです。

36.1. スキーマ

情報スキーマ自身は、information_schemaという名前のスキーマです。このスキーマは自動的にすべてのデータベース内に存在します。このスキーマの所有者は、クラスタ内の最初のデータベースユーザであり、当然このユーザは、スキーマの削除を含むスキーマについてのすべての権限を持ちます（しかし、削除したとしても節約できる領域はわずかです）。

デフォルトでは、情報スキーマはスキーマの検索パスには含まれません。ですので、修飾した名前で情報スキーマ内のすべてのオブジェクトにアクセスする必要があります。情報スキーマ内の一部のオブジェクトの名前はユーザアプリケーションでも使用されるような一般的な名前であるため、情報スキーマをパスに追加する場合は注意しなければなりません。

36.2. データ型

情報スキーマのビューの列では、情報スキーマ内で定義された特殊なデータ型を使用します。これらは、通常の組み込み型の上位にあたる単純なドメインとして定義されます。情報スキーマ外部で操作する時にはこれらの型を使用してはなりません。しかし、情報スキーマを検索するようなアプリケーションではこれらの型への用意をしておかなければなりません。

これらの型を以下に記します。

cardinal_number

非負の整数です。

character_data

(最大文字長の指定がない)文字列です。

sql_identifier

文字列です。この型はSQL識別子用に使用され、character_dataはその他の任意のテキストデータに使用されます。

time_stamp

timestamp with time zone型の上位ドメインです。

yes_or_no

YESかNOのいずれかを持つ文字列ドメインです。情報スキーマ内で論理(真/偽)データを表すために使用されます。(情報スキーマはboolean型が標準SQLに追加される前に考案されました。このため情報スキーマの後方互換性を維持するために、この記法が必要です。)

情報スキーマ内の列はすべてこれら5つの型のいずれかを取ります。

36.3. information_schema_catalog_name

information_schema_catalog_nameは、常に現在のデータベース名(SQL用語では現在のカタログ)を持つ1行1列からなるテーブルです。

表36.1 information_schema_catalog_nameの列

列 型 説明
catalog_name sql_identifier この情報スキーマを持つデータベースの名前です。

36.4. administrable_role_authorizations

administrable_role_authorizationsビューは、現在のユーザがアドミンオプションを持つすべてのロールを識別します。

表36.2 administrable_role_authorizationsの列

列 型 説明
grantee sql_identifier このロールのメンバ資格を付与したロールの名前です。(現在のユーザかもしれませんし、入れ子状のロールメンバ資格の場合は異なるロールかもしれません。)
role_name sql_identifier

列 型 説明
ロール名です。
is_grantable yes_or_no 常にYESです。

36.5. applicable_roles

applicable_rolesビューは、その権限を現在のユーザが使用することができるすべてのロールを識別します。これは、現在のユーザから問題のロールに付与されたロールの連鎖が存在することを意味します。現在のユーザ自身もまた適用可能なロールです。適用可能なロール群は通常権限の検査に使用されます。

表36.3 applicable_rolesの列

列 型 説明
grantee sql_identifier このロールのメンバ資格を付与したロールの名前です。(現在のユーザかもしれませんし、入れ子状のロールメンバ資格の場合は異なるロールかもしれません。)
role_name sql_identifier ロール名です。
is_grantable yes_or_no 付与者がこのロールにアドミンオプションを持つ場合YES、さもなくばNOです。

36.6. attributes

attributesビューには、データベース内で定義された複合データ型の属性に関する情報が含まれます。(このビューが、PostgreSQLコンテキスト内でよく呼び出される属性である、テーブル列に関する情報を持たない点に注意してください。) 現在のユーザが(所有者であるかまたは複合データ型に対する権限を持っていて)アクセスする権限を持つ属性のみが表示されます。

表36.4 attributesの列

列 型 説明
udt_catalog sql_identifier データ型を含むデータベースの名前です(常に現在のデータベースです)。
udt_schema sql_identifier データ型を含むスキーマの名前です。
udt_name sql_identifier データ型の名前です。
attribute_name sql_identifier 属性の名前です。

列 型 説明
ordinal_position cardinal_number データ型の属性の序数位置です（1から始まります）。
attribute_default character_data 属性のデフォルト式です。
is_nullable yes_or_no 属性がNULLを持つことができる場合はYES、さもなくばNOです。
data_type character_data 属性のデータ型が組み込み型の場合、そのデータ型です。何らかの配列の場合、ARRAYです。（この場合、element_typesビューを参照してください。）さもなくばUSER-DEFINEDです。（この場合、型はattribute_udt_nameと関連する列により識別されます。）
character_maximum_length cardinal_number data_typeが文字列またはビット列を識別する場合、その宣言された最大長です。他のデータ型または最大長が宣言されていない場合はNULLです。
character_octet_length cardinal_number data_typeが文字列を識別する場合、オクテット（バイト）単位で表したデータの最大長です。他のデータ型ではNULLです。最大オクテット長は宣言された文字最大長(上述)とサーバ符号化方式に依存します。
character_set_catalog sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
character_set_schema sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
character_set_name sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
collation_catalog sql_identifier 属性の照合を含むデータベース(常に現在のデータベース)の名前で、デフォルトであるか属性のデータ型が照合可能でない場合はNULLです。
collation_schema sql_identifier 属性の照合を含むスキーマの名前で、デフォルトであるか属性のデータ型が照合可能でない場合はNULLです。
collation_name sql_identifier 属性の照合の名前で、デフォルトであるか属性のデータ型が照合可能でない場合はNULLです。
numeric_precision cardinal_number data_typeが数値型を識別する場合、この列は属性の型の（宣言された、あるいは暗黙的な）精度です。この精度は有効桁を意味します。numeric_precision_radix列の指定に従って、(10を基とした)10進数、または(2を基とした)2進数表記で表現されます。他のデータ型ではこの列はNULLです。
numeric_precision_radix cardinal_number data_typeが数値型を識別する場合、この列は、numeric_precisionおよびnumeric_scaleで表現される値の基が何かを識別します。この値は2または10です。他のデータ型ではこの列はNULLです。
numeric_scale cardinal_number

列 型 説明
data_typeが数値型を識別する場合、この列は、属性の型の(宣言された、あるいは暗黙的な)位取りが含まれます。位取りは小数点以下の有効桁数を意味します。numeric_precision_radix列の指定に従って、(10を基とした)10進数、または(2を基とした)2進数表記で表現されます。他のデータ型ではこの列はNULLです。
datetime_precision cardinal_number data_typeが日付、時刻、タイムスタンプ、または間隔型を示す場合、この列は(宣言されたか暗黙的な)この属性に対する分数秒精度を包含します。つまり、秒の値の小数点に続く保存された10進桁数です。他の全ての型に対してこの列はNULLです。
interval_type character_data もしdata_typeが時間間隔型を示す場合、この列はこの属性の時間間隔値がどのフィールドを含むかの仕様を含みます。例えば、YEAR TO MONTH、DAY TO SECONDなどです。もしフィールド制約が指定されていない(時間間隔が全てのフィールドを受け付ける)場合や、他の全てのデータ型の場合はこのフィールドはNULLです。
interval_precision cardinal_number PostgreSQLで利用できない機能に適用されるものです。(インターバル型の属性の秒未満の精度についてはdatetime_precisionを参照してください)
attribute_udt_catalog sql_identifier 属性のデータ型が定義されたデータベースの名前です。(常に現在のデータベースです。)
attribute_udt_schema sql_identifier 属性のデータ型が定義されたスキーマの名前です。
attribute_udt_name sql_identifier 属性のデータ型の名前です。
scope_catalog sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
scope_schema sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
scope_name sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
maximum_cardinality cardinal_number 常にNULLです。PostgreSQLでは配列の次数は無制限だからです。
dtd_identifier sql_identifier 列のデータ型記述子の、テーブルに属するデータ型記述子内で一意な識別子です。この識別子のインスタンスを結合する時に、主に有用です。(識別子の書式仕様は定義されておらず、今後のバージョンで同一性を維持する保証もありません。)
is_derived_reference_attribute yes_or_no PostgreSQLでは利用できない機能に適用されるものです。

後述の36.17も参照してください。ビューの構造が似ていますし、一部の列では更なる情報も記載されています。

36.7. character_sets

このcharacter_setsビューは、現在のデータベースで利用可能な文字セットを識別します。PostgreSQLはひとつのデータベース内で複数の文字セットをサポートしないので、このビューは常にデータベースエンコーディングの一行だけを表示します。

以下の用語のSQL標準での使われ方に注意してください。

文字集合

例えばUNICODEやUCS、LATIN1といった抽象的な文字集合です。SQLオブジェクトとしては出てきませんが、このビューで参照できます。

文字符号化形式

文字集合の符号化方式です。ほとんどの古い文字集合はひとつの符号化形式を使うため、それらについては分離した名称はありません(たとえば、LATIN1はLATIN1集合に適用可能な符号化形式です)。しかし例えばUnicodeにはUTF8、UTF16などの符号化形式があります(PostgreSQLでは一部だけサポートしています)。符号化形式はSQLオブジェクトとして表にでませんが、このビューで参照できます。

文字セット

文字集合、文字符号化方式とデフォルトの照合を識別する順序名前つきのSQLオブジェクトです。定義済みの文字セットは、一般的に符号化形式と同じ名前を持ちますが、ユーザは他の名前を定義できます。例えば、文字セットUTF8は一般的に文字集合UCS、符号化形式UTF8と何らかのデフォルト照合を識別します。

PostgreSQLにおける「encoding」は、文字セットまたは文字符号化形式のいずれかと考えられます。これらは同じ名前を持ち一つのデータベースでは一つだけ存在できます。

表36.5 character_setsの列

列 型 説明
character_set_catalog sql_identifier 文字セットはスキーマオブジェクトとして実装されていないので、この列はNULLです。
character_set_schema sql_identifier 文字セットはスキーマオブジェクトとして実装されていないので、この列はNULLです。
character_set_name sql_identifier 文字セットの名前で、現在はデータベースエンコーディングを表示するように実装されています。
character_repertoire sql_identifier 文字集合で、エンコーディングがUTF8の場合はUCSを、それ以外の場合は単にエンコーディング名を表示します。
form_of_use sql_identifier 文字符号化形式で, データベースエンコーディングと同じです。
default_collate_catalog sql_identifier デフォルト照合を含むデータベース(いずれかの照合が識別された場合は常に現在のデータベース)の名前です。
default_collate_schema sql_identifier

列 型 説明
デフォルト照合を含むスキーマの名前です。
default_collate_name sql_identifier デフォルト照合の名前です。デフォルト照合は、現在のデータベースのCOLLATEとCTYPE設定に一致する照合として識別されます。そのような照合が存在しない場合は、この列や対応するスキーマやカタログの列はNULLです。

36.8. check_constraint_routine_usage

check_constraint_routine_usageは検査制約で使用される処理（関数およびプロシージャ）を識別します。現在有効なロールが所有する処理のみが表示されます。

表36.6 check_constraint_routine_usageの列

列 型 説明
constraint_catalog sql_identifier 制約が含まれるデータベースの名前です（常に現在のデータベースです）。
constraint_schema sql_identifier 制約が含まれるスキーマの名前です。
constraint_name sql_identifier 制約の名前です。
specific_catalog sql_identifier 関数が含まれるデータベースの名前です（常に現在のデータベースです）。
specific_schema sql_identifier 関数が含まれるスキーマの名前です。
specific_name sql_identifier 関数の「仕様名称」です。詳細は36.41を参照してください。

36.9. check_constraints

check_constraintsビューには、現在有効なロールが所有している、テーブル上もしくはドメイン上のどちらかにある、全ての検査制約が含まれます（テーブルもしくはドメインの所有者がこの制約の所有者です）。

表36.7 check_constraintsの列

列 型 説明
constraint_catalog sql_identifier 制約が含まれるデータベースの名前です（常に現在のデータベースです）。
constraint_schema sql_identifier 制約が含まれるスキーマの名前です。

列 型 説明
constraint_name sql_identifier 制約の名前です。
check_clause character_data この検査制約の検査式です。

36.10. collations

collationsビューは現在のデータベースで利用可能な照合を含みます

表36.8 collationsの列

列 型 説明
collation_catalog sql_identifier 照合を含むデータベース(常に現在のデータベース)の名前です。
collation_schema sql_identifier 照合を含むスキーマの名前です。
collation_name sql_identifier デフォルト照合の名前です。
pad_attribute character_data 常にNO PADです。(もう一方のPAD SPACEはPostgreSQLではサポートされていません。)

36.11. collation_character_set_applicability

collation_character_set_applicabilityビューは、利用可能な照合がどの文字セットに適用可能かを識別します。PostgreSQLでは、データベースごとに一つの文字セットしか存在しない(36.7の説明を参照してください)ので、このビューは有益な情報を提供しません。

表36.9 collation_character_set_applicabilityの列

列 型 説明
collation_catalog sql_identifier 照合を含むデータベース(常に現在のデータベース)の名前です。
collation_schema sql_identifier 照合を含むスキーマの名前です。
collation_name sql_identifier デフォルト照合の名前です。
character_set_catalog sql_identifier 文字セットはスキーマオブジェクトとして実装されていないので、この列はNULLです。

列 型 説明
character_set_schema sql_identifier 文字セットはスキーマオブジェクトとして実装されていないので、この列はNULLです。
character_set_name sql_identifier 文字セットの名前です。

36.12. column_column_usage

column_column_usageビューは、同じテーブル内の別の基底列に基づくすべての生成列を識別します。現在有効なロールが所有するテーブルのみが含まれます。

表36.10 column_column_usageの列

列 型 説明
table_catalog sql_identifier テーブルを持つデータベースの名前です(常に現在のデータベースです)。
table_schema sql_identifier テーブルを持つスキーマの名前です。
table_name sql_identifier テーブルの名前です。
column_name sql_identifier 生成された列が基づく元の列の名前です。
dependent_column sql_identifier 生成された列の名前です。

36.13. column_domain_usage

column_domain_usageビューは、現在のデータベース内で定義され、現在有効なロールが所有するあるドメインを使用する(テーブルもしくはビューの)全ての列を識別します。

表36.11 column_domain_usageの列

列 型 説明
domain_catalog sql_identifier ドメインを持つデータベースの名前です(常に現在のデータベースです)。
domain_schema sql_identifier ドメインを持つスキーマの名前です。
domain_name sql_identifier ドメインの名前です。

列 型 説明
table_catalog sql_identifier テーブルを持つデータベースの名前です(常に現在のデータベースです)。
table_schema sql_identifier テーブルを持つスキーマの名前です。
table_name sql_identifier テーブルの名前です。
column_name sql_identifier 列の名前です。

36.14. column_options

column_optionsビューは、現在のデータベースにある外部テーブルの列に定義された全てのオプションを含みます。現在のユーザが(所有者であるかまたは権限を持っていて)アクセスする権限を持つ外部テーブル列のみが表示されます。

表36.12 column_optionsの列

列 型 説明
table_catalog sql_identifier 外部テーブルが含まれるデータベースの名前です(常に現在のデータベースです)。
table_schema sql_identifier 外部テーブルを含むスキーマの名前です。
table_name sql_identifier 外部テーブルの名前です。
column_name sql_identifier 列の名前です。
option_name sql_identifier オプションの名前です。
option_value character_data オプションの値です。

36.15. column_privileges

column_privilegesビューは、現在有効なロールに対し、または現在有効なロールによって、列に与えられた権限を全て識別します。列と許可を与えた者、許可を受けた者の組み合わせごとに1行があります。

権限がテーブル全体に付与されていた場合、このビューでは各列に権限が付与された場合と同じように表示されます。しかし、SELECT、INSERT、UPDATE、REFERENCESといった列単位で設定可能な種類の権限のみを対象範囲とします。

表36.13 column_privilegesの列

列 型 説明
grantor sql_identifier 権限を与えたロールの名前です。
grantee sql_identifier 権限を与えられたロールの名前です。
table_catalog sql_identifier その列を含むテーブルを持つデータベースの名前です(常に現在のデータベースです)。
table_schema sql_identifier その列を含むテーブルを持つスキーマの名前です。
table_name sql_identifier その列を含むテーブルの名前です。
column_name sql_identifier 列の名前です。
privilege_type character_data 権限の種類です。SELECT、INSERT、UPDATE、もしくはREFERENCESです。
is_grantable yes_or_no この権限を付与可能な場合はYES、さもなくばNOです。

36.16. column_udt_usage

column_udt_usageビューは、現在有効なロールが所有するデータ型を使用する全ての列を識別します。PostgreSQLでは、組み込みデータ型がユーザ定義型同様に振舞いますので、組み込みデータ型も同様にここに含まれます。詳細は36.17も参照してください。

表36.14 column_udt_usageの列

列 型 説明
udt_catalog sql_identifier 列データ型(もし適用されていたら背後にあるドメインの型)を定義したデータベースの名前です(常に現在のデータベースです)。
udt_schema sql_identifier 列データ型(もし適用されていたら背後にあるドメインの型)を定義したスキーマの名前です。
udt_name sql_identifier 列データ型(もし適用されていたら背後にあるドメインの型)の名前です。
table_catalog sql_identifier テーブルを持つデータベースの名前です(常に現在のデータベースです)。
table_schema sql_identifier テーブルを持つスキーマの名前です。

列 型 説明
table_name sql_identifier テーブルの名前です。
column_name sql_identifier 列の名前です。

36.17. columns

columnsビューには、データベース内の全てのテーブル列(もしくはビューの列)についての情報が含まれます。システム列(ctidなど)は含まれません。現在のユーザが(所有者である、権限を持っているなどの方法で)アクセスできる列のみが示されます。

表36.15 columnsの列

列 型 説明
table_catalog sql_identifier テーブルを持つデータベースの名前です(常に現在のデータベースです)。
table_schema sql_identifier テーブルを持つスキーマの名前です。
table_name sql_identifier テーブルの名前です。
column_name sql_identifier 列の名前です。
ordinal_position cardinal_number テーブル内の列の位置(1から始まる序数)です。
column_default character_data 列のデフォルト式です。
is_nullable yes_or_no 列がNULLを持つことができる場合はYES、NULLを持つことができなければNOです。非NULL制約は、列にNULLを持たせないようにする方法の1つですが、その他にも存在します。
data_type character_data 組み込み型の場合、列のデータ型、配列の場合、ARRAY(この場合はelement_typesビューを参照してください)、さもなくば、USER-DEFINEDです(この場合、型はudt_nameと関連する列で識別されます)。列がドメインに基づくものであれば、その列はドメインの背後にある型を示します (そして、ドメインはdomain_nameと関連する列によって識別されます)。
character_maximum_length cardinal_number data_typeが文字列またはビット列を識別する場合、その宣言された最大長です。他のデータ型または最大長が宣言されていない場合はNULLです。
character_octet_length cardinal_number data_typeが文字列を識別する場合、オクテット(バイト)単位で表したデータの最大長です。他のデータ型ではNULLです。最大オクテット長は宣言された文字最大長(上述)とサーバ符号化方式に依存します。

列 型 説明
numeric_precision cardinal_number data_typeが数値型を示す場合、ここには、その列の型の(宣言された、もしくは暗黙的な)精度が含まれます。この精度は有意な桁数を示します。numeric_precision_radix列の指定に従い、10進数(10を底)、もしくは2進数(2を底)で表現されます。この列は、他の全ての型ではNULLです。
numeric_precision_radix cardinal_number data_typeが数値型を識別する場合、この列は、numeric_precisionおよびnumeric_scaleで表現される値の基が何かを識別します。この値は2または10です。他のデータ型ではこの列はNULLです。
numeric_scale cardinal_number data_typeが高精度数値型を示す場合、ここには、その列の型の(宣言された、あるいは暗黙的な)位取りが含まれます。位取りとは、小数点より右側の有意な桁数です。numeric_precision_radix列の指定に従い、10進数(10を底)、もしくは2進数(2を底)で表現されます。この列は、他の全ての型ではNULLです。
datetime_precision cardinal_number data_typeが日付、時刻、タイムスタンプ、間隔型を示す場合、この列の型の秒以下の(宣言された、または暗黙的な)精度、つまり、秒値の小数点以降に保持する10進桁数、です。他のすべての型の場合ではこの列はNULLです。
interval_type character_data もしdata_typeが時間間隔型を示す場合、この列はこの属性の時間間隔値がどのフィールドを含むかの仕様を含みます。例えば、YEAR TO MONTH、DAY TO SECONDなどです。もしフィールド制約が指定されていない(時間間隔が全てのフィールドを受け付ける)場合や、他の全てのデータ型の場合はこのフィールドはNULLです。
interval_precision cardinal_number PostgreSQLで利用できない機能に適用されるものです。(時間間隔型の属性の秒未満の精度についてはdatetime_precisionを参照してください)
character_set_catalog sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
character_set_schema sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
character_set_name sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
collation_catalog sql_identifier 列の照合を含むデータベース(常に現在のデータベース)の名前で、デフォルトであるか属性のデータ型が照合可能でない場合はNULLです。
collation_schema sql_identifier 属性の照合を含むスキーマの名前で、デフォルトであるか属性のデータ型が照合可能でない場合はNULLです。
collation_name sql_identifier 列の照合の名前で、デフォルトであるか列のデータ型が照合可能でない場合はNULLです。
domain_catalog sql_identifier 列がドメイン型の場合、そのドメインを定義したデータベースの名前です (常に現在のデータベースです)。さもなくば、NULLです。
domain_schema sql_identifier 列がドメイン型の場合、そのドメインを定義したスキーマの名前です。さもなくば、NULLです。

列 型 説明
domain_name sql_identifier 列がドメイン型の場合、そのドメインの名前です。さもなくば、NULLです。
udt_catalog sql_identifier 列データ型（もし適用されていたら背後にあるドメインの型）を定義したデータベースの名前です（常に現在のデータベースです）。
udt_schema sql_identifier 列データ型（もし適用されていたら背後にあるドメインの型）を定義したスキーマの名前です。
udt_name sql_identifier 列データ型（もし適用されていたら背後にあるドメインの型）の名前です。
scope_catalog sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
scope_schema sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
scope_name sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
maximum_cardinality cardinal_number 常にNULLです。PostgreSQLでは配列の次数は無制限だからです。
dtd_identifier sql_identifier 列のデータ型記述子の、テーブルに属するデータ型記述子内で一意な識別子です。この識別子のインスタンスを結合する時に、主に有用です。（識別子の書式仕様は定義されておらず、今後のバージョンで同一性を維持する保証もありません。）
is_self_referencing yes_or_no PostgreSQLでは利用できない機能に適用されるものです。
is_identity yes_or_no 列が識別列であればYES、そうでなければNOです。
identity_generation character_data 列が識別列であれば、列の定義を反映してALWAYSまたはBY DEFAULTです。
identity_start character_data 列が識別列であれば内部シーケンスの開始値、そうでなければNULLです。
identity_increment character_data 列が識別列であれば内部シーケンスの増分、そうでなければNULLです。
identity_maximum character_data 列が識別列であれば内部シーケンスの最大値、そうでなければNULLです。
identity_minimum character_data 列が識別列であれば内部シーケンスの最小値、そうでなければNULLです。
identity_cycle yes_or_no 列が識別列であれば、内部シーケンスが周回する場合にYES、周回しない場合にNO、識別列でなければNULLです。
is_generated character_data 列が生成された列であればALWAYS、そうでなければNEVERです。

列 型 説明
generation_expression character_data 列が生成された列であれば生成式、そうでなければNULLです。
is_updatable yes_or_no 列が更新可能な場合YES、さもなくばNOです。(ベーステーブルの列は常に更新可能です。ビューの列では不要です。)

データ型は、複数の方法でSQLにより定義でき、さらにPostgreSQLにはデータ型定義に別の方法も追加されていますので、情報スキーマにおけるデータ型表現は多少難しくなっています。data_type列は、列の背後にある組み込み型を識別できるようになっていなければなりません。PostgreSQLでは、型がpg_catalogシステムカタログスキーマで定義されていることを意味します。よく知られた組み込み型を特別に扱うことができるアプリケーション(例えば、数値型を異なる書式にする、精度列内のデータを使用する)の場合、この列が有用な場合があります。udt_name、udt_schema、udt_catalog列は、たとえドメインに基づいた列であっても、常に列の背後にあるデータ型を識別します (PostgreSQLは、組み込み型をユーザ定義型と同様に扱いますので、ここには組み込み型も現れます。これは標準SQLからの拡張です)。アプリケーションが型に従って異なる処理を行う場合、これらの列を使用しなければなりません。このような場合、本当はドメインに基づいている列なのかどうかが関係ないからです。列がドメインに基づく場合、ドメインの識別子はdomain_name、domain_schema、domain_catalog列に保持されます。関連するデータ型と列の組み合わせを作りたい場合や、ドメインを別の型として扱いたい場合は、coalesce(domain_name, udt_name)などとすることができます。

36.18. constraint_column_usage

constraint_column_usageビューは、現在のデータベースで制約を使用する全ての列を識別します。現在有効なロールが所有するテーブル内の列のみが表示されます。検査制約では、このビューは検査式で使用される列を識別します。外部キー制約では、このビューは外部キーを参照する列を識別します。一意性制約もしくは主キー制約では、このビューは制約される列を識別します。

表36.16 constraint_column_usageの列

列 型 説明
table_catalog sql_identifier ある制約で使用される列を持つデータベースの名前です (データベースの名前です)。
table_schema sql_identifier ある制約で使用される列を持つテーブルを含むスキーマの名前です。
table_name sql_identifier ある制約で使用される列を持つテーブルの名前です。
column_name sql_identifier ある制約で使用される列の名前です。
constraint_catalog sql_identifier 制約を持つデータベースの名前です(常に現在のデータベースです)。
constraint_schema sql_identifier

列 型 説明
制約を持つスキーマの名前です。
constraint_name sql_identifier 制約の名前です。

36.19. constraint_table_usage

constraint_table_usageビューは、ある制約で使用され、かつ、現在有効なロールが所有する、現在のデータベース内の全てのテーブルを識別します（これは、全てのテーブル制約とそれを定義したテーブルを識別するtable_constraintsとは異なります）。外部キー制約では、このビューは外部キーが参照するテーブルを識別します。一意性制約もしくは主キー制約では、このビューは単に制約が属するテーブルを識別します。検査制約と非NULL制約はこのビューには含まれません。

表36.17 constraint_table_usageの列

列 型 説明
table_catalog sql_identifier ある制約で使用されるテーブルを持つデータベースの名前です（常に現在のデータベースです）。
table_schema sql_identifier ある制約で使用されるテーブルを持つスキーマの名前です。
table_name sql_identifier ある制約で使用されるテーブルの名前です。
constraint_catalog sql_identifier 制約を持つデータベースの名前です（常に現在のデータベースです）。
constraint_schema sql_identifier 制約を持つスキーマの名前です。
constraint_name sql_identifier 制約の名前です。

36.20. data_type_privileges

data_type_privilegesビューは、記述子が示すオブジェクトの所有者である、何かしらの権限を持っているといった方法で現在のユーザがアクセスできる全てのデータ型記述子を識別します。あるデータ型がテーブル列やドメイン、関数（パラメータとして、あるいは戻り値として）の定義内で使用されると、そのデータ型記述子は生成され、そして、データ型がそのインスタンスでどのように使用されるか（例えば、もし適切ならば、宣言された最大長）についての情報が格納されます。各データ型記述子は、1つのオブジェクト（テーブル、ドメイン、関数）に割り当てられたデータ型記述子の中で一意となる任意の識別子が割り振られます。このビューはおそらくアプリケーションではあまり使用されませんが、情報スキーマ内の他のビューを定義する際に使用されます。

表36.18 data_type_privilegesの列

列 型 説明
object_catalog sql_identifier 記述子が示すオブジェクトを持つデータベースの名前です(常に現在のデータベースです)。
object_schema sql_identifier 記述子が示すオブジェクトを持つスキーマの名前です。
object_name sql_identifier 記述子が示すオブジェクトの名前です。
object_type character_data 記述子が示すオブジェクトの種類です。TABLE(データ型記述子がそのテーブルの列に属します)、DOMAIN(データ型記述子がそのドメインに属します)、ROUTINE(データ型記述子がその関数のパラメータあるいは戻り値データ型に属します)のいずれかです。
dtd_identifier sql_identifier データ型記述子の識別子です。これは同一オブジェクトに対するデータ型記述子の中で一意なものです。

36.21. domain_constraints

domain_constraintsビューには、現在有効なロールが所有するドメインに属する全ての制約があります。現在のユーザが(所有者である、権限を持っているなどの方法で)アクセスできるドメインのみが示されます。

表36.19 domain_constraintsの列

列 型 説明
constraint_catalog sql_identifier 制約を持つデータベースの名前です(常に現在のデータベースです)。
constraint_schema sql_identifier 制約を持つスキーマの名前です。
constraint_name sql_identifier 制約の名前です。
domain_catalog sql_identifier ドメインを持つデータベースの名前です(常に現在のデータベースです)。
domain_schema sql_identifier ドメインを持つスキーマの名前です。
domain_name sql_identifier ドメインの名前です。
is_deferrable yes_or_no 制約が遅延可能ならばYES。さもなくばNO。
initially_deferred yes_or_no 制約が遅延可能で初期状態が遅延であればYES。さもなくばNO。

36.22. domain_udt_usage

domain_udt_usageビューは、現在有効なロールが所有するデータ型に基づいたすべてのドメインを識別します。PostgreSQLでは組み込みデータ型はユーザ定義型と同様に振舞いますので、ここにも同様に現れることに注意してください。

表36.20 domain_udt_usageの列

列 型 説明
udt_catalog sql_identifier ドメインデータ型を定義したデータベースの名前です(常に現在のデータベースです)。
udt_schema sql_identifier ドメインデータ型を定義したスキーマの名前です。
udt_name sql_identifier ドメインデータ型の名前です。
domain_catalog sql_identifier ドメインを持つデータベースの名前です(常に現在のデータベースです)。
domain_schema sql_identifier ドメインを持つスキーマの名前です。
domain_name sql_identifier ドメインの名前です。

36.23. domains

domainsビューには、現在のデータベースで定義された全てのドメインが含まれます。現在のユーザが(所有者である、権限を持っているなどの方法で)アクセスできるドメインのみが示されます。

表36.21 domainsの列

列 型 説明
domain_catalog sql_identifier ドメインを持つデータベースの名前です(常に現在のデータベースです)。
domain_schema sql_identifier ドメインを持つスキーマの名前です。
domain_name sql_identifier ドメインの名前です。
data_type character_data 組み込み型の場合は、ドメインのデータ型、何らかの配列の場合はARRAYです(後者の場合はelement_typesビューを参照してください)。さもなくば、USER-DEFINEDです(この場合、その型はudt_nameと関連する列で識別されます)。
character_maximum_length cardinal_number

列 型 説明
ドメインが、文字もしくはビット文字列型の場合、宣言された最大長です。他のデータ型、あるいは最大長の宣言がない場合はNULLです。
character_octet_length cardinal_number ドメインが文字型の場合、1つのデータの可能最大長をオクテット（バイト）で示します 他のデータ型の場合はNULLです。最大オクテット長は宣言された文字最大長(上述)とサーバ符号化方式に依存します。
character_set_catalog sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
character_set_schema sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
character_set_name sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
collation_catalog sql_identifier ドメインの照合を含むデータベース(常に現在のデータベース)の名前で、デフォルトであるかドメインのデータ型が照合可能でない場合はNULLです。
collation_schema sql_identifier ドメインの照合を含むスキーマの名前で、デフォルトであるかドメインのデータ型が照合可能でない場合はNULLです。
collation_name sql_identifier ドメインの照合の名前で、デフォルトであるかドメインのデータ型が照合可能でない場合はNULLです。
numeric_precision cardinal_number ドメインが数値型の場合、この列は、そのドメインの型の（宣言された、もしくは暗黙的な）精度を持ちます。この精度は有効桁数を示すものです。numeric_precision_radix 列が示す通り、10進数（10を底）でも2進数（2を底）でも表現できます。他の全てのデータ型では、この列はNULLです。
numeric_precision_radix cardinal_number ドメインが数値型の場合、この列は、numeric_precisionとnumeric_scaleで表現されるその列の値の底数がどちらかを示します。2もしくは10の値となります。他の全てのデータ型では、この列はNULLです。
numeric_scale cardinal_number ドメインが高精度数値型の場合、この列は、そのドメインの型の（宣言された、もしくは暗黙的な）位取りを持ちます。位取りは、小数点より右側の有効桁数を示すものです。numeric_precision_radix列の指定に従い、10進数（10を底）、もしくは2進数（2を底）で表現されます。他の全ての型ではこの列はNULLです。
datetime_precision cardinal_number data_typeが日付、時刻、タイムスタンプ、間隔型を示す場合、この列はこのドメインの型で（宣言された、または暗黙的な）秒の端数の精度、つまり、秒値の小数点以下で保持される10進の桁数です。他の全ての型の場合はNULLです。
interval_type character_data もしdata_typeが時間間隔型を示す場合、この列はこのドメインの時間間隔値がどのフィールドを含むかの仕様を含みます。例えば、YEAR TO MONTH、DAY TO SECONDなどです。もしフィールド制約が指定されていない(時間間隔が全てのフィールドを受け付ける)場合や、他の全てのデータ型の場合はこのフィールドはNULLです。
interval_precision cardinal_number PostgreSQLで利用できない機能に適用されるものです。(時間間隔型のドメインの秒未満の精度についてはdatetime_precisionを参照してください)
domain_default character_data

列 型 説明
ドメインのデフォルト式です。
udt_catalog sql_identifier ドメインデータ型を定義したデータベースの名前です（常に現在のデータベースです）。
udt_schema sql_identifier ドメインデータ型を定義したスキーマの名前です。
udt_name sql_identifier ドメインデータ型の名前です。
scope_catalog sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
scope_schema sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
scope_name sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
maximum_cardinality cardinal_number 常にNULLです。PostgreSQLでは配列の次数は無制限だからです。
dtd_identifier sql_identifier そのドメインに属するデータ型記述子間で一意な、データ型記述子の識別子です（ドメインには1つのデータ型記述子しかありませんのでこれは些細なものです）。これは主に、こうした識別子の他のインスタンスを結合する時に有用です（識別子の書式の仕様は定義されておらず、将来のバージョンでそのまま維持されるかどうかも保証されません）。

36.24. element_types

element_typesには、配列の要素のデータ型記述子が含まれます。テーブル列、複合データ型属性、ドメイン、関数パラメータ、関数の戻り値が配列型であると宣言された場合、情報スキーマの各ビューでは、data_type列にARRAYだけが含まれます。配列の要素の型についての情報を取り出すには、各ビューとこのビューを結合することで可能です。例えば、テーブルの列のデータ型と、もし適切ならば、配列の要素型を表示するには、以下のように行います。

```
SELECT c.column_name, c.data_type, e.data_type AS element_type
FROM information_schema.columns c LEFT JOIN information_schema.element_types e
     ON ((c.table_catalog, c.table_schema, c.table_name, 'TABLE', c.dtd_identifier)
       = (e.object_catalog, e.object_schema, e.object_name, e.object_type,
 e.collection_type_identifier))
WHERE c.table_schema = '...' AND c.table_name = '...'
ORDER BY c.ordinal_position;
```

このビューは、所有者である、適切な権限を持っているといった方法で、現在のユーザがアクセスできるオブジェクトのみが含まれます。

表36.22 element_typesの列

列 型 説明
object_catalog sql_identifier 記述される配列を使用するオブジェクトを持つデータベースの名前です（常に現在のデータベースです）。
object_schema sql_identifier 記述される配列を使用するオブジェクトを持つスキーマの名前です。
object_name sql_identifier 記述される配列を使用するオブジェクトの名前です。
object_type character_data 記述される配列を使用するオブジェクトの種類です。TABLE（その配列がテーブルの列によって使用される）、USER-DEFINED TYPE（その配列が複合データ型の属性によって使用される）、DOMAIN（その配列がドメインによって使用される）、ROUTINE（その配列が関数のパラメータ、もしくは戻り値の型によって使用される）のいずれかです。
collection_type_identifier sql_identifier 記述される配列のデータ型記述子の識別子です。他の情報スキーマビューのdtd_identifier列と結合するのに使用してください。
data_type character_data 組み込み型の場合は配列要素のデータ型です。さもなくば、USER-DEFINEDです（この場合、型はudt_nameと関連する列で識別されます）。
character_maximum_length cardinal_number 常にNULLです。この情報は、PostgreSQLにおける配列要素のデータ型には当てはまらないからです。
character_octet_length cardinal_number 常にNULLです。この情報は、PostgreSQLにおける配列要素のデータ型には当てはまらないからです。
character_set_catalog sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
character_set_schema sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
character_set_name sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
collation_catalog sql_identifier 要素データ型の照合を含むデータベース(常に現在のデータベース)の名前で、デフォルトであるか要素のデータ型が照合可能でない場合はNULLです。
collation_schema sql_identifier 要素データ型の照合を含むスキーマの名前で、デフォルトであるか要素のデータ型が照合可能でない場合はNULLです。
collation_name sql_identifier 要素データ型の照合の名前で、デフォルトであるか要素のデータ型が照合可能でない場合はNULLです。
numeric_precision cardinal_number 常にNULLです。この情報は、PostgreSQLにおける配列要素のデータ型には当てはまらないからです。
numeric_precision_radix cardinal_number 常にNULLです。この情報は、PostgreSQLにおける配列要素のデータ型には当てはまらないからです。

列 型 説明
numeric_scale cardinal_number 常にNULLです。この情報は、PostgreSQLにおける配列要素のデータ型には当てはまらないからです。
datetime_precision cardinal_number 常にNULLです。この情報は、PostgreSQLにおける配列要素のデータ型には当てはまらないからです。
interval_type character_data 常にNULLです。この情報は、PostgreSQLにおける配列要素のデータ型には当てはまらないからです。
interval_precision cardinal_number 常にNULLです。この情報は、PostgreSQLにおける配列要素のデータ型には当てはまらないからです。
domain_default character_data 未実装です。
udt_catalog sql_identifier 要素のデータ型を定義したデータベースの名前です（常に現在のデータベースです）。
udt_schema sql_identifier 要素のデータ型を定義したスキーマの名前です。
udt_name sql_identifier 要素のデータ型の名前です。
scope_catalog sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
scope_schema sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
scope_name sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
maximum_cardinality cardinal_number 常にNULLです。PostgreSQLでは配列の次数は無制限だからです。
dtd_identifier sql_identifier 要素のデータ型記述子の識別子です。現在は用途はありません。

36.25. enabled_roles

enabled_rolesビューは、現在「有効なロール」を識別します。有効なロールは、現在のユーザと自動継承によって有効なロールに付与されたすべてのロールとして再帰的に定義されます。言い換えると、これは、現在のユーザが直接または間接的に属するメンバ資格の継承により自動的に持つすべてのロールです。

権限検査では、「適用可能なロール」群が適用されます。これは、有効なロールよりも広範になる可能性があります。したがって一般的にはこのビューよりもapplicable_rolesビューを使用する方が良いでしょう。applicable_rolesビューの詳細については36.5を参照してください。

表36.23 enabled_rolesの列

列 型 説明
role_name sql_identifier ロール名です。

36.26. foreign_data_wrapper_options

foreign_data_wrapper_optionsビューには現在のデータベース内の外部データラッパー用に定義されたすべてのオプションが含まれます。現在のユーザが（所有者または何らかの権限を持つことで）アクセス可能な外部データラッパーのみが表示されます。

表36.24 foreign_data_wrapper_optionsの列

列 型 説明
foreign_data_wrapper_catalog sql_identifier 外部データラッパーを定義したデータベースの名前です（常に現在のデータベースです）。
foreign_data_wrapper_name sql_identifier 外部データラッパーの名前です。
option_name sql_identifier オプションの名前です。
option_value character_data オプションの値です。

36.27. foreign_data_wrappers

foreign_data_wrappersビューには現在のデータベース内で定義された、すべての外部データラッパーが含まれます。現在のユーザが（所有者である、または、何らかの権限を持つことにより）アクセス可能な外部データラッパーのみが表示されます。

表36.25 foreign_data_wrappersの列

列 型 説明
foreign_data_wrapper_catalog sql_identifier 外部データラッパーを含むデータベース名です（常に現在のデータベースです）。
foreign_data_wrapper_name sql_identifier 外部データラッパーの名前です。
authorization_identifier sql_identifier 外部サーバの所有者の名前です。

列 型 説明
library_name character_data この外部データラッパーを実装するライブラリの名前です。
foreign_data_wrapper_language character_data この外部データラッパーを実装するのに使用される言語です。

36.28. foreign_server_options

foreign_server_optionsビューには、現在のデータベース内の外部サーバ用に定義されたすべてのオプションが含まれます。現在のユーザが（所有者である、または何らかの権限を持つことにより）アクセス可能な外部サーバのみが表示されます。

表36.26 foreign_server_optionsの列

列 型 説明
foreign_server_catalog sql_identifier 外部サーバを定義したデータベースの名前です（常に現在のデータベースです）。
foreign_server_name sql_identifier 外部サーバの名前
option_name sql_identifier オプションの名前です。
option_value character_data オプションの値です。

36.29. foreign_servers

foreign_serversビューには、現在のデータベース内で定義されたすべての外部サーバが含まれます。現在のユーザが（所有者である、または何らかの権限を持つことにより）アクセス可能な外部サーバのみが表示されます。

表36.27 foreign_serversの列

列 型 説明
foreign_server_catalog sql_identifier 外部サーバを定義したデータベースの名前です（常に現在のデータベースです）。
foreign_server_name sql_identifier 外部サーバの名前
foreign_data_wrapper_catalog sql_identifier 外部サーバで使用される外部データラッパーを含むデータベースの名前です（常に現在のデータベースです）。

列 型 説明
foreign_data_wrapper_name sql_identifier 外部サーバにより使用される外部データラッパーの名前です。
foreign_server_type character_data 作成時点で指定されている場合、その外部サーバの種類情報です。
foreign_server_version character_data 作成時点で指定されている場合、その外部サーバのバージョン情報です。
authorization_identifier sql_identifier 外部サーバの所有者の名前です。

36.30. foreign_table_options

foreign_table_optionsビューは、現在のデータベースの外部テーブルに定義された全てのオプションを含みます。(所有者であるか何らかの権限を持っていて)現在のユーザがアクセスできる外部テーブルだけが表示されます。

表36.28 foreign_table_optionsの列

列 型 説明
foreign_table_catalog sql_identifier 外部テーブルが含まれるデータベースの名前です(常に現在のデータベースです)。
foreign_table_schema sql_identifier 外部テーブルを含むスキーマの名前です。
foreign_table_name sql_identifier 外部テーブルの名前です。
option_name sql_identifier オプションの名前です。
option_value character_data オプションの値です。

36.31. foreign_tables

foreign_tablesビューは、現在のデータベースで定義されている全ての外部テーブルを含みます。(所有者であるか何らかの権限を持っていて)現在のユーザがアクセスできる外部テーブルだけが表示されます。

表36.29 foreign_tablesの列

列 型 説明
foreign_table_catalog sql_identifier

列 型 説明
外部テーブルを含むデータベースの名前です（常に現在のデータベースです）。
foreign_table_schema sql_identifier 外部テーブルを含むスキーマの名前です。
foreign_table_name sql_identifier 外部テーブルの名前です。
foreign_server_catalog sql_identifier 外部サーバを定義したデータベースの名前です（常に現在のデータベースです）。
foreign_server_name sql_identifier 外部サーバの名前

36.32. key_column_usage

key_column_usageビューは、現在のデータベースにおいて、ある一意性制約、主キー制約、外部キー制約によって制限を受けている全ての列を識別します。検査制約はこのビューには含まれません。現在のユーザが所有者である、または何らかの権限を持ち、アクセスできるテーブル内のこうした列のみがここに示されます。

表36.30 key_column_usageの列

列 型 説明
constraint_catalog sql_identifier 制約を持つデータベースの名前です（常に現在のデータベースです）。
constraint_schema sql_identifier 制約を持つスキーマの名前です。
constraint_name sql_identifier 制約の名前です。
table_catalog sql_identifier この制約によって制限を受ける列を持つテーブルを持つデータベースの名前です。（常に現在のデータベースです。）
table_schema sql_identifier この制約によって制限を受ける列を持つテーブルを持つスキーマの名前です。
table_name sql_identifier この制約によって制限を受ける列を持つテーブルの名前です。
column_name sql_identifier この制約によって制限を受ける列の名前です。
ordinal_position cardinal_number 制約キー内の列の位置を（1から始まる）序数で表したものです。
position_in_unique_constraint cardinal_number 外部キー制約では、一意性制約内部の被参照列の位置の序数（1から始まります）です。その他の場合はNULLです。

36.33. parameters

parametersビューには、現在のデータベースにある全ての関数のパラメータ(引数)についての情報があります。現在のユーザが(所有している、あるいはある権限を持っているといった方法で)アクセスできる関数についてのみが示されます。

表36.31 parametersの列

列 型 説明
specific_catalog sql_identifier 関数が含まれるデータベースの名前です(常に現在のデータベースです)。
specific_schema sql_identifier 関数が含まれるスキーマの名前です。
specific_name sql_identifier 関数の「仕様名称」です。詳細は36.41を参照してください。
ordinal_position cardinal_number 関数の引数リストにおけるパラメータの位置の序数(1から始まる)です。
parameter_mode character_data 入力パラメータではIN、出力パラメータではOUT、入出力パラメータではINOUT です。
is_result yes_or_no PostgreSQLでは利用できない機能に適用されるものです。
as_locator yes_or_no PostgreSQLでは利用できない機能に適用されるものです。
parameter_name sql_identifier 名前付きパラメータです。無名のパラメータの場合はNULLです。
data_type character_data 組み込み型の場合、パラメータのデータ型です。何らかの配列の場合はARRAYです(この場合、element_typesビューを参照してください)。さもなくば、USER-DEFINEDです(この場合、型はudt_name と関連する列で識別されます)。
character_maximum_length cardinal_number 常にNULLです。PostgreSQLでは、この情報はパラメータデータ型に適用されないからです。
character_octet_length cardinal_number 常にNULLです。PostgreSQLでは、この情報はパラメータデータ型に適用されないからです。
character_set_catalog sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
character_set_schema sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
character_set_name sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
collation_catalog sql_identifier 常にNULLです。PostgreSQLでは、この情報はパラメータデータ型に適用されないからです。
collation_schema sql_identifier

列 型 説明
常にNULLです。PostgreSQLでは、この情報はパラメータデータ型に適用されないからです。
collation_name sql_identifier 常にNULLです。PostgreSQLでは、この情報はパラメータデータ型に適用されないからです。
numeric_precision cardinal_number 常にNULLです。PostgreSQLでは、この情報はパラメータデータ型に適用されないからです。
numeric_precision_radix cardinal_number 常にNULLです。PostgreSQLでは、この情報はパラメータデータ型に適用されないからです。
numeric_scale cardinal_number 常にNULLです。PostgreSQLでは、この情報はパラメータデータ型に適用されないからです。
datetime_precision cardinal_number 常にNULLです。PostgreSQLでは、この情報はパラメータデータ型に適用されないからです。
interval_type character_data 常にNULLです。PostgreSQLでは、この情報はパラメータデータ型に適用されないからです。
interval_precision cardinal_number 常にNULLです。PostgreSQLでは、この情報はパラメータデータ型に適用されないからです。
udt_catalog sql_identifier パラメータのデータ型を定義したデータベースの名前です（常に現在のデータベースです）。
udt_schema sql_identifier パラメータのデータ型を定義したスキーマの名前です。
udt_name sql_identifier パラメータのデータ型の名前です。
scope_catalog sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
scope_schema sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
scope_name sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
maximum_cardinality cardinal_number 常にNULLです。PostgreSQLでは配列の次数は無制限だからです。
dtd_identifier sql_identifier 関数に属するデータ型記述子内で一意なパラメータのデータ型記述子の識別子です。これは主に、こうした識別子の他のインスタンスと結合する時に有用です（識別子の書式の仕様は定義されておらず、また、今後のバージョンでも同一のままであるという保証もありません）。
parameter_default character_data パラメータのデフォルト式であり、存在しない場合または現在有効なロールがその関数の所有者でない場合にはNULLです。

36.34. referential_constraints

referential_constraintsビューには、現在のデータベース内にある全ての参照（外部キー）制約があります。現在のユーザが、参照テーブルに（所有者、またはSELECT以外の何らかの権限を持つという方法で）書き込みアクセスを持つと言う事でこれらの制約は示されます。

表36.32 referential_constraintsの列

列 型 説明
constraint_catalog sql_identifier 制約が含まれるデータベースの名前です（常に現在のデータベースです）。
constraint_schema sql_identifier 制約が含まれるスキーマの名前です。
constraint_name sql_identifier 制約の名前です。
unique_constraint_catalog sql_identifier 外部キー制約が参照する一意性制約、もしくは主キー制約を持つデータベースの名前です（常に現在のデータベースです）。
unique_constraint_schema sql_identifier 外部キー制約が参照する一意性制約、もしくは主キー制約を持つスキーマの名前です。
unique_constraint_name sql_identifier 外部キー制約が参照する一意性制約、もしくは主キー制約の名前です。
match_option character_data 外部キー制約の一致オプションです。FULL、PARTIAL、NONEのいずれかです。
update_rule character_data 外部キー制約の更新規則です。CASCADE、SET NULL、SET DEFAULT、RESTRICT、NO ACTIONのいずれかです。
delete_rule character_data 外部キー制約の削除規則です。CASCADE、SET NULL、SET DEFAULT、RESTRICT、NO ACTIONのいずれかです。

36.35. role_column_grants

role_column_grantsビューは、譲与者または被譲与者が現在有効なロールである場合、列に付与された全ての権限を識別します。詳細な情報はcolumn_privilegesの中にあります。このビューとcolumn_privilegesとの間の実質的な違いは、このビューでは現在のユーザがPUBLICに与えられた権限によりアクセスできるようになった列を省略していることだけです。

表36.33 role_column_grantsの列

列 型 説明
grantor sql_identifier 権限を与えたロールの名前です。

列 型 説明
grantee sql_identifier 権限を与えられたロールの名前です。
table_catalog sql_identifier その列を含むテーブルを持つデータベースの名前です(常に現在のデータベースです)。
table_schema sql_identifier その列を含むテーブルを持つスキーマの名前です。
table_name sql_identifier その列を含むテーブルの名前です。
column_name sql_identifier 列の名前です。
privilege_type character_data 権限の種類です。SELECT、INSERT、UPDATE、もしくはREFERENCESです。
is_grantable yes_or_no この権限を付与可能な場合はYES、さもなくばNOです。

36.36. role_routine_grants

role_routine_grantsビューは、現在有効なロールが譲与者、または被譲与者で関数上に与えられた全ての権限を識別します。詳細な情報はroutine_privilegesの中にあります。このビューとroutine_privilegesとの間の実質的な違いは、このビューでは現在のユーザがPUBLICに与えられた権限によりアクセスできるようになった関数を省略していることだけです。

表36.34 role_routine_grantsの列

列 型 説明
grantor sql_identifier 権限を与えたロールの名前です。
grantee sql_identifier 権限を与えられたロールの名前です。
specific_catalog sql_identifier 関数が含まれるデータベースの名前です(常に現在のデータベースです)。
specific_schema sql_identifier 関数が含まれるスキーマの名前です。
specific_name sql_identifier 関数の「仕様名称」です。詳細は36.41を参照してください。
routine_catalog sql_identifier 関数が含まれるデータベースの名前です(常に現在のデータベースです)。
routine_schema sql_identifier

列 型 説明
関数が含まれるスキーマの名前です。
routine_name sql_identifier 関数の名前です(オーバーロードされている場合は重複する可能性があります)。
privilege_type character_data 常にEXECUTEです(関数用の唯一の権限です)。
is_grantable yes_or_no この権限を付与可能な場合はYES、さもなくばNOです。

36.37. role_table_grants

role_table_grantsビューは、現在有効なロールが譲与者または被譲与者であるテーブルやビュー上に与えられた全ての権限を識別します。詳細な情報はtable_privilegesの中にあります。このビューとtable_privilegesとの間の実質的な違いは、このビューでは現在のユーザがPUBLICに与えられた権限によりアクセスできるようになったテーブルを省略していることだけです。

表36.35 role_table_grantsの列

列 型 説明
grantor sql_identifier 権限を与えたロールの名前です。
grantee sql_identifier 権限を与えられたロールの名前です。
table_catalog sql_identifier テーブルを持つデータベースの名前です(常に現在のデータベースです)。
table_schema sql_identifier テーブルを持つスキーマの名前です。
table_name sql_identifier テーブルの名前です。
privilege_type character_data 権限の種類は、SELECT、INSERT、UPDATE、DELETE、TRUNCATE、REFERENCES、またはTRIGGERのいずれかです。
is_grantable yes_or_no この権限を付与可能な場合はYES、さもなくばNOです。
with_hierarchy yes_or_no SQL標準では、WITH HIERARCHY OPTIONは、継承テーブル階層に対するある操作を許可する独立した(副次)権限です。PostgreSQLでは、これはSELECT権限に含まれているため、この列は権限がSELECTの場合はYES、それ以外の場合はNOです。

36.38. role_udt_grants

role_udt_grantsビューは、現在有効なロールが付与者また被付与者である、ユーザ定義型に付与されたUSAGE権限を識別することを意図したものです。それ以上の情報はudt_privilegesで見つかります。このビューとudt_privilegesビューとの間の実質的な違いは、このビューでは現在のユーザがPUBLICに与えられた権限によりアクセスできるようになったオブジェクトを省略していることだけです。PostgreSQLではデータ型は実際の権限を持たず、PUBLICに対する暗黙の権限付与しか持たないため、このビューは空です。

表36.36 role_udt_grantsの列

列 型 説明
grantor sql_identifier 権限を与えたロールの名前
grantee sql_identifier 権限が与えられたロールの名前
udt_catalog sql_identifier 型を持つデータベースの名前(常に現在のデータベースです)。
udt_schema sql_identifier 型を持つスキーマの名前
udt_name sql_identifier 型の名前
privilege_type character_data 常にTYPE USAGE
is_grantable yes_or_no この権限を付与可能な場合はYES、さもなくばNOです。

36.39. role_usage_grants

role_usage_grantsビューは、譲与者または被譲与者が現在有効なロールである多くの種類のオブジェクトに対し、USAGE権限を識別します。詳細な情報はusage_privilegesの中にあります。このビューとusage_privilegesビューとの間の実質的な違いは、このビューでは現在のユーザがPUBLICに与えられた権限によりアクセスできるようになったオブジェクトを省略していることだけです。

表36.37 role_usage_grantsの列

列 型 説明
grantor sql_identifier 権限を与えたロールの名前
grantee sql_identifier 権限が与えられたロールの名前
object_catalog sql_identifier

列 型 説明
オブジェクトを持つデータベースの名前(常に現在のデータベースです)。
object_schema sql_identifier 適用されるオブジェクトを持つスキーマの名前。そうでなければ空文字列
object_name sql_identifier オブジェクトの名前です。
object_type character_data COLLATIONまたはDOMAINまたはFOREIGN DATA WRAPPERまたはFOREIGN SERVERまたはSEQUENCE
privilege_type character_data 常にUSAGEです。
is_grantable yes_or_no この権限を付与可能な場合はYES、さもなくばNOです。

36.40. routine_privileges

routine_privilegesビューは、現在有効なロールに与えられた権限、あるいは現在有効なロールによって関数に与えられた権限を全て識別します。関数、権限の譲与者と被譲与者の組み合わせごとに1行あります。

表36.38 routine_privilegesの列

列 型 説明
grantor sql_identifier 権限を与えたロールの名前です。
grantee sql_identifier 権限を与えられたロールの名前です。
specific_catalog sql_identifier 関数が含まれるデータベースの名前です(常に現在のデータベースです)。
specific_schema sql_identifier 関数が含まれるスキーマの名前です。
specific_name sql_identifier 関数の「仕様名称」です。詳細は36.41を参照してください。
routine_catalog sql_identifier 関数が含まれるデータベースの名前です(常に現在のデータベースです)。
routine_schema sql_identifier 関数が含まれるスキーマの名前です。
routine_name sql_identifier 関数の名前です(オーバーロードされている場合は重複する可能性があります)。
privilege_type character_data 常にEXECUTEです(関数用の唯一の権限です)。

列 型 説明
is_grantable yes_or_no この権限を付与可能な場合はYES、さもなくばNOです。

36.41. routines

routinesビューには現在のデータベース内の全ての関数とプロシージャがあります。表示される関数とプロシージャは、現在のユーザが（所有者である、何らかの権限を持っているといった方法で）アクセスできるものだけです。

表36.39 routinesの列

列 型 説明
specific_catalog sql_identifier 関数が含まれるデータベースの名前です（常に現在のデータベースです）。
specific_schema sql_identifier 関数が含まれるスキーマの名前です。
specific_name sql_identifier 関数の「仕様名称」です。これは、その関数の実際の名前がオーバーロードされていたとしても、スキーマ内の関数を一意に識別する名前です。仕様名称の書式は定義されておらず、特定の関数名の他のインスタンスと比較するためにのみ使用されます。
routine_catalog sql_identifier 関数が含まれるデータベースの名前です（常に現在のデータベースです）。
routine_schema sql_identifier 関数が含まれるスキーマの名前です。
routine_name sql_identifier 関数の名前です（オーバーロードされている場合は重複する可能性があります）。
routine_type character_data 関数に対してはFUNCTION、プロシージャに対してはPROCEDUREです。
module_catalog sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
module_schema sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
module_name sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
udt_catalog sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
udt_schema sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
udt_name sql_identifier

列 型 説明
PostgreSQLでは利用できない機能に適用されるものです。
data_type character_data 関数の戻り値が組み込み型の場合、そのデータ型です。何らかの配列の場合はARRAYです（この場合は、element_typesビューを参照してください）。さもなくば、USER-DEFINEDです（この場合、その型はtype_udt_nameと関連する列によって識別されます）。プロシージャに対してはNULLです。
character_maximum_length cardinal_number 常にNULLです。PostgreSQLではこの情報は戻り値のデータ型に当てはまりません。
character_octet_length cardinal_number 常にNULLです。PostgreSQLではこの情報は戻り値のデータ型に当てはまりません。
character_set_catalog sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
character_set_schema sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
character_set_name sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
collation_catalog sql_identifier 常にNULLです。PostgreSQLではこの情報は戻り値のデータ型に当てはまりません。
collation_schema sql_identifier 常にNULLです。PostgreSQLではこの情報は戻り値のデータ型に当てはまりません。
collation_name sql_identifier 常にNULLです。PostgreSQLではこの情報は戻り値のデータ型に当てはまりません。
numeric_precision cardinal_number 常にNULLです。PostgreSQLではこの情報は戻り値のデータ型に当てはまりません。
numeric_precision_radix cardinal_number 常にNULLです。PostgreSQLではこの情報は戻り値のデータ型に当てはまりません。
numeric_scale cardinal_number 常にNULLです。PostgreSQLではこの情報は戻り値のデータ型に当てはまりません。
datetime_precision cardinal_number 常にNULLです。PostgreSQLではこの情報は戻り値のデータ型に当てはまりません。
interval_type character_data 常にNULLです。PostgreSQLではこの情報は戻り値のデータ型に当てはまりません。
interval_precision cardinal_number 常にNULLです。PostgreSQLではこの情報は戻り値のデータ型に当てはまりません。
type_udt_catalog sql_identifier 関数の戻り値のデータ型が定義されたデータベースの名前です（常に現在のデータベースです）。プロシージャに対してはNULLです。
type_udt_schema sql_identifier 関数の戻り値のデータ型が定義されたスキーマの名前です。プロシージャに対してはNULLです。

列 型 説明
type_udt_name sql_identifier 関数の戻り値のデータ型の名前です。プロシージャに対してはNULLです。
scope_catalog sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
scope_schema sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
scope_name sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
maximum_cardinality cardinal_number 常にNULLです。PostgreSQLでは配列の次数は無制限だからです。
dtd_identifier sql_identifier この関数に属するデータ型記述子中で一意な、関数の戻り値のデータ型のデータ型記述子の識別子です。これは主に、そうした識別子の他のインスタンスと結合する際に有用です。（識別子の書式仕様は定義されておらず、将来のバージョンで同じままであるという保証もありません。）
routine_body character_data 関数がSQL関数ならばSQL、さもなくばEXTERNALです。
routine_definition character_data 関数のソーステキストです（現在有効なロールがその関数の所有者でなければNULLです）。（標準SQLに従うと、routine_bodyがSQLの場合にのみ適用されます。しかし、PostgreSQLでは、関数が作成された時に指定されたソーステキストが常に含まれます。）
external_name character_data 関数がC関数の場合関数の外部名（リンクシンボル）、さもなくばNULLです。（これはroutine_definitionで示した値と同じになるように動作します。）
external_language character_data その関数を作成した言語です。
parameter_style character_data 常にGENERALです（標準SQLでは他のパラメータ様式も定義していますが、これらはPostgreSQLでは使用できません）。
is_deterministic yes_or_no 関数が不変である（標準SQLでは決定性があると呼びます）と宣言されている場合YES、さもなくばNOです。（情報スキーマ経由ではPostgreSQLで使用できる他の変動レベルを問い合わせることはできません。）
sql_data_access character_data 常に、関数がSQLデータを変更することができることを意味するMODIFIESです。この情報はPostgreSQLでは有用ではありません。
is_null_call yes_or_no その関数の引数のいずれかがNULLの場合に、自動的にNULLを返す場合はYES、さもなくばNOです。プロシージャに対してはNULLです。
sql_path character_data PostgreSQLでは利用できない機能に適用されるものです。
schema_level_routine yes_or_no

列 型 説明
常にYESです。(この反対はユーザ定義の種類による方法となります。これはPostgreSQLでは使用できない機能です。)
max_dynamic_result_sets cardinal_number PostgreSQLでは利用できない機能に適用されるものです。
is_user_defined_cast yes_or_no PostgreSQLでは利用できない機能に適用されるものです。
is_implicitly_invocable yes_or_no PostgreSQLでは利用できない機能に適用されるものです。
security_type character_data 現在のユーザ権限で関数が動作する場合INVOKER、定義したユーザの権限で関数が動作する場合DEFINERです。
to_sql_specific_catalog sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
to_sql_specific_schema sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
to_sql_specific_name sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
as_locator yes_or_no PostgreSQLでは利用できない機能に適用されるものです。
created time_stamp PostgreSQLでは利用できない機能に適用されるものです。
last_altered time_stamp PostgreSQLでは利用できない機能に適用されるものです。
new_savepoint_level yes_or_no PostgreSQLでは利用できない機能に適用されるものです。
is_udt_dependent yes_or_no 現在は常にNOです。もう一方のYESはPostgreSQLで利用できない機能に適用されるものです。
result_cast_from_data_type character_data PostgreSQLでは利用できない機能に適用されるものです。
result_cast_as_locator yes_or_no PostgreSQLでは利用できない機能に適用されるものです。
result_cast_char_max_length cardinal_number PostgreSQLでは利用できない機能に適用されるものです。
result_cast_char_octet_length cardinal_number PostgreSQLでは利用できない機能に適用されるものです。
result_cast_char_set_catalog sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
result_cast_char_set_schema sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
result_cast_char_set_name sql_identifier

列 型 説明
PostgreSQLでは利用できない機能に適用されるものです。
result_cast_collation_catalog sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
result_cast_collation_schema sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
result_cast_collation_name sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
result_cast_numeric_precision cardinal_number PostgreSQLでは利用できない機能に適用されるものです。
result_cast_numeric_precision_radix cardinal_number PostgreSQLでは利用できない機能に適用されるものです。
result_cast_numeric_scale cardinal_number PostgreSQLでは利用できない機能に適用されるものです。
result_cast_datetime_precision cardinal_number PostgreSQLでは利用できない機能に適用されるものです。
result_cast_interval_type character_data PostgreSQLでは利用できない機能に適用されるものです。
result_cast_interval_precision cardinal_number PostgreSQLでは利用できない機能に適用されるものです。
result_cast_type_udt_catalog sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
result_cast_type_udt_schema sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
result_cast_type_udt_name sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
result_cast_scope_catalog sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
result_cast_scope_schema sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
result_cast_scope_name sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
result_cast_maximum_cardinality cardinal_number PostgreSQLでは利用できない機能に適用されるものです。
result_cast_dtd_identifier sql_identifier PostgreSQLでは利用できない機能に適用されるものです。

36.42. schemata

schemataビューには、現在のデータベースの中で現在のユーザが（所有者である、または、何らかの権限を持つために）アクセスできるすべてのスキーマがあります。

表36.40 schemataの列

列 型 説明
catalog_name sql_identifier スキーマを持つデータベースの名前です（常に現在のデータベースです）。
schema_name sql_identifier スキーマの名前です。
schema_owner sql_identifier スキーマの所有者の名前です。
default_character_set_catalog sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
default_character_set_schema sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
default_character_set_name sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
sql_path character_data PostgreSQLでは利用できない機能に適用されるものです。

36.43. sequences

sequencesビューは、現在のデータベース内で定義されたすべてのシーケンスがあります。現在のユーザが（所有者である、または、何らかの権限を持つために）アクセスできるシーケンスのみが表示されます。

表36.41 sequencesの列

列 型 説明
sequence_catalog sql_identifier シーケンスを含むデータベースの名前です（常に現在のデータベースです）。
sequence_schema sql_identifier シーケンスを含むスキーマの名前です。
sequence_name sql_identifier シーケンスの名前です。
data_type character_data シーケンスのデータ型です。
numeric_precision cardinal_number

列 型 説明
この列は（宣言された、あるいは、暗黙的な）シーケンスデータ型の精度（上述）です。精度は有効桁数を意味します。numeric_precision_radixの指定に従い、これは10進数（10を基とする）または2進数（2を基とする）で表現されます。
numeric_precision_radix cardinal_number この列は、numeric_precisionおよびnumeric_scaleで表現される値の基が何かを識別します。値は2または10です。
numeric_scale cardinal_number この列はシーケンスデータ型の（宣言された、または、暗黙的な）位取り（上述）です。位取りは、小数点以下の有効桁数を意味します。numeric_precision_radixの指定に従い、これは10進数（10を基とする）または2進数（2を基とする）で表現されます。
start_value character_data シーケンスの開始値です。
minimum_value character_data シーケンスの最小値です。
maximum_value character_data シーケンスの最大値です。
increment character_data シーケンスの増加値です。
cycle_option yes_or_no シーケンスが周回する場合はYES、それ以外はNOです。

SQL標準に従い、開始、最小、最大および増加の値が文字列で返されることに注意してください。

36.44. sql_features

sql_featuresビューには、標準SQLで定義された公式な機能のどれがPostgreSQLでサポートされているかについての情報があります。これは付録D内に記された情報と同じものです。また、ここには背景について追加情報がいくつかあります。

表36.42 sql_featuresの列

列 型 説明
feature_id character_data 機能の識別文字列です。
feature_name character_data 機能の説明的な名前です。
sub_feature_id character_data 副機能の識別子です。副機能がない場合は空の文字列です。
sub_feature_name character_data 副機能の説明的な名前です。副機能がない場合は空の文字列です。
is_supported yes_or_no

列 型 説明
現在のバージョンのPostgreSQLで機能が完全にサポートされている場合はYESです。さもなくばNOです。
is_verified_by character_data 常にNULLです。PostgreSQL開発グループは機能が準拠しているかどうかについて公式な試験を行っていないからです。
comments character_data おそらく機能のサポート状況についてのコメントです。

36.45. sql_implementation_info

sql_implementation_infoビューには、標準SQLで実装依存で定義するものとされている各種状況に関する情報があります。この情報は主にODBCインタフェース環境での使用を意図しています。他のインタフェースのユーザはおそらくこの情報があまり役に立たないものと考えるでしょう。このため、個々の実装情報項目をここでは記載しません。ODBCインタフェースの説明でそれらを説明します。

表36.43 sql_implementation_infoの列

列 型 説明
implementation_info_id character_data 実装情報項目の識別文字列です。
implementation_info_name character_data 実装情報項目の説明的な名前です。
integer_value cardinal_number 実装情報項目の値です。その値がcharacter_valueにある場合はNULLです。
character_value character_data 実装情報項目の値です。その値がinteger_valueにある場合はNULLです。
comments character_data おそらく実装情報項目に関するコメントです。

36.46. sql_parts

sql_partsテーブルは、標準SQLのどの部分がPostgreSQLでサポートされているかに関する情報を持ちます。

表36.44 sql_partsの列

列 型 説明
feature_id character_data 部品番号を含む識別文字列です。

列 型 説明
feature_name character_data 部品の説明的な名称です。
is_supported yes_or_no PostgreSQLの現在のバージョンで部品が完全にサポートされている場合YES、さもなくばNOです。
is_verified_by character_data 常にNULLです。PostgreSQL開発グループは機能が準拠しているかどうかについて公式な試験を行っていないからです。
comments character_data おそらく部品のサポート状況に関するコメントです。

36.47. sql_sizing

sql_sizingテーブルには、PostgreSQL中の各種サイズ制限と上限値に関する情報があります。この情報は主にODBCインタフェース環境での使用を意図しています。他のインタフェースのユーザはおそらくこの情報があまり役に立たないものと考えるでしょう。このため、個々のサイズ調整項目はここでは記載しません。ODBCインタフェースの説明でそれらを説明します。

表36.45 sql_sizingの列

列 型 説明
sizing_id cardinal_number サイズ調整項目の識別子です。
sizing_name character_data サイズ調整項目の説明的な名前です。
supported_value cardinal_number サイズ調整項目の値です。サイズに制限がない場合や決定できない場合はゼロです。サイズ調整項目を適用する機能がサポートされない場合はNULLです。
comments character_data おそらくサイズ調整項目に関するコメントです。

36.48. table_constraints

table_constraintsビューには、現在のユーザが所有する、または何らかのSELECT以外の権限を持つテーブルに属する全ての制約があります。

表36.46 table_constraintsの列

列 型 説明
constraint_catalog sql_identifier

列 型 説明
制約を持つデータベースの名前です(常に現在のデータベースです)。
constraint_schema sql_identifier 制約を持つスキーマの名前です。
constraint_name sql_identifier 制約の名前です。
table_catalog sql_identifier テーブルを持つデータベースの名前です(常に現在のデータベースです)。
table_schema sql_identifier テーブルを持つスキーマの名前です。
table_name sql_identifier テーブルの名前です。
constraint_type character_data 制約の種類です。CHECK、FOREIGN KEY、PRIMARY KEY、UNIQUEのいずれかです。
is_deferrable yes_or_no 制約が遅延可能ならばYES。さもなくばNO。
initially_deferred yes_or_no 制約が遅延可能で初期状態が遅延であればYES。さもなくばNO。
enforced yes_or_no PostgreSQLで利用できない機能に適用されるものです(現在は常にYESです)。

36.49. table_privileges

table_privilegesビューは、現在有効なロールに対し、または現在有効なロールによって、テーブルもしくはビューに与えられた権限を全て識別します。テーブル、譲与者、被譲与者の組み合わせごとに1行があります。

表36.47 table_privilegesの列

列 型 説明
grantor sql_identifier 権限を与えたロールの名前です。
grantee sql_identifier 権限を与えられたロールの名前です。
table_catalog sql_identifier テーブルを持つデータベースの名前です(常に現在のデータベースです)。
table_schema sql_identifier テーブルを持つスキーマの名前です。
table_name sql_identifier

列 型 説明
テーブルの名前です。
privilege_type character_data 権限の種類は、SELECT、INSERT、UPDATE、DELETE、TRUNCATE、REFERENCES、またはTRIGGERのいずれかです。
is_grantable yes_or_no この権限を付与可能な場合はYES、さもなくばNOです。
with_hierarchy yes_or_no SQL標準では、WITH HIERARCHY OPTIONは、継承テーブル階層に対するある操作を許可する独立した(副次)権限です。PostgreSQLでは、これはSELECT権限に含まれているため、この列は権限がSELECTの場合はYES、それ以外の場合はNOです。

36.50. tables

tablesビューには、現在のデータベースで定義された全てのテーブルとビューがあります。現在のユーザが(所有している、何らかの権限を持っているといった方法で)アクセスできるテーブルとビューのみが表示されます。

表36.48 tablesの列

列 型 説明
table_catalog sql_identifier テーブルを持つデータベースの名前です(常に現在のデータベースです)。
table_schema sql_identifier テーブルを持つスキーマの名前です。
table_name sql_identifier テーブルの名前です。
table_type character_data テーブルの種類です。永続テーブル(普通のテーブルの種類)ではBASE TABLE、ビューではVIEW、外部テーブルではFOREIGN、一時テーブルではLOCAL TEMPORARYです。
self_referencing_column_name sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
reference_generation character_data PostgreSQLでは利用できない機能に適用されるものです。
user_defined_type_catalog sql_identifier テーブルが型付けされたテーブルの場合、背後のデータ型を持つデータベース(常に現在のデータベース)の名前です。さもなくばNULLです。
user_defined_type_schema sql_identifier テーブルが型付けされたテーブルである場合、背後のデータ型を含むスキーマの名前です。さもなくばNULLです。
user_defined_type_name sql_identifier テーブルが型付けされたテーブルである場合、背後のデータ型の名前です。さもなくばNULLです。

列 型 説明
is_insertable_into yes_or_no テーブルが挿入可能な場合YES、さもなくばNOです。(ベーステーブルは常に挿入可能ですが、ビューはそうとも限りません。)
is_typed yes_or_no テーブルが型付けされたテーブルの場合YES、そうでなければNOです。
commit_action character_data 未実装です。

36.51. transforms

ビューtransformsは現在のデータベースで定義されている変換についての情報を含んでいます。より正確に言えば、変換に含まれる各関数(「from SQL」あるいは「to SQL」関数)について1行ずつあります。

表36.49 transformsの列

列 型 説明
udt_catalog sql_identifier 変換の適用対象の型を含むデータベースの名前です(必ず現在のデータベースになります)。
udt_schema sql_identifier 変換の適用対象の型を含むスキーマの名前です。
udt_name sql_identifier 変換の適用対象の型の名前です。
specific_catalog sql_identifier 関数が含まれるデータベースの名前です(常に現在のデータベースです)。
specific_schema sql_identifier 関数が含まれるスキーマの名前です。
specific_name sql_identifier 関数の「仕様名称」です。詳細は36.41を参照してください。
group_name sql_identifier 標準SQLでは変換を「グループ」で定義して、実行時にグループを選択することを認めています。PostgreSQLはこれをサポートしていません。代わりに、変換は言語ごとに別々になっています。妥協として、このフィールドには変換の対象となる言語が入っています。
transform_type character_data FROM SQLまたはTO SQL

36.52. triggered_update_columns

列リスト(UPDATE OF column1, column2など)を指定する現在のデータベース内のトリガに関して、triggered_update_columnsビューはこれらの列を識別します。列リストを指定しないトリガはこのビューには

含まれません。これらの列の内、現在のユーザが所有するまたはSELECT以外の何らかの権限を持つもののみが示されます。

表36.50 triggered_update_columnsの列

列 型 説明
trigger_catalog sql_identifier トリガを持つデータベースの名前です(常に現在のデータベースです)。
trigger_schema sql_identifier トリガを持つスキーマの名前です。
trigger_name sql_identifier トリガの名前です。
event_object_catalog sql_identifier トリガが定義されたテーブルを持つデータベースの名前です。(常に現在のデータベースです。)
event_object_schema sql_identifier トリガが定義されたテーブルを持つスキーマの名前です。
event_object_table sql_identifier トリガが定義されたテーブルの名前です。
event_object_column sql_identifier トリガが定義された列の名前です。

36.53. triggers

triggersビューには、現在のデータベース内で、現在のユーザが所有するあるいは何らかのSELECT以外の権限を持つテーブルまたはビューに定義された、全てのトリガがあります。

表36.51 triggersの列

列 型 説明
trigger_catalog sql_identifier トリガを持つデータベースの名前です(常に現在のデータベースです)。
trigger_schema sql_identifier トリガを持つスキーマの名前です。
trigger_name sql_identifier トリガの名前です。
event_manipulation character_data トリガを発するイベントです (INSERT、UPDATEもしくはDELETEです)。
event_object_catalog sql_identifier トリガが定義されたテーブルを持つデータベースの名前です。(常に現在のデータベースです。)
event_object_schema sql_identifier トリガが定義されたテーブルを持つスキーマの名前です。

列 型 説明
event_object_table sql_identifier トリガが定義されたテーブルの名前です。
action_order cardinal_number 同じテーブルで同じevent_manipulation、action_timing、action_orientationのトリガを発行する順序です。PostgreSQLでは、トリガは名前順に発行されますので、この列はそれを反映しています。
action_condition character_data トリガのWHEN条件です。なければNULLです (現在有効なロールが所有していないテーブルの場合もNULLです)。
action_statement character_data トリガによって実行される文です (現在は常にEXECUTE FUNCTION function(...)です)。
action_orientation character_data トリガの発行が処理行ごとか文ごとかを識別します (ROWもしくはSTATEMENTです)。
action_timing character_data トリガを発する時期です (BEFORE、AFTERもしくはINSTEAD OFです)。
action_reference_old_table sql_identifier 「old」遷移テーブルの名前です。なければNULLです。
action_reference_new_table sql_identifier 「new」遷移テーブルの名前です。なければNULLです。
action_reference_old_row sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
action_reference_new_row sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
created time_stamp PostgreSQLでは利用できない機能に適用されるものです。

PostgreSQLにおけるトリガには、標準SQLと比べ、2つの非互換があり、これらは情報スキーマの表現に影響を与えます。1つ目は、PostgreSQLではトリガ名は、独立したスキーマオブジェクトではなく、それぞれのテーブル内で局所的であることです。そのため、別のテーブルに属している場合、1つのスキーマ内でトリガ名を重複させることができます。(trigger_catalogとtrigger_schemaは実際、そのトリガが定義されたテーブルに属する値となります。) 2つ目は、PostgreSQLではトリガは複数のイベントで発行できる点です (例えばON INSERT OR UPDATEです)。一方、標準SQLでは1つのみしか許されません。トリガが複数のイベントで発行するように定義された場合、それぞれのイベントで1行という形で、情報スキーマ内では複数の行として表現されます。これらの2つの問題の結果、triggersビューの主キーは実際、標準SQLで定義された(trigger_catalog, trigger_schema, trigger_name)ではなく、(trigger_catalog, trigger_schema, event_object_table, trigger_name, event_manipulation)となります。それでもなお、標準SQLに従う (スキーマ内でトリガ名を一意とし、トリガに対し1種類のイベントしか持たせないという) 手法でトリガを定義していれば、これは影響ありません。

注記

PostgreSQL 9.1 より前は、このビューの列の action_timing、action_reference_old_table、action_reference_new_table、action_reference_old_row、action_reference_new_row はそれ

ぞれ condition_timing、condition_reference_old_table、condition_reference_new_table、condition_reference_old_row、condition_reference_new_row という名前でした。これらの命名はSQL: 1999標準におけるものです。新しい名前はSQL:2003以降に準拠しています。

36.54. udt_privileges

udt_privilegesビューは、現在有効なロールが付与者または被付与者である、ユーザ定義型に付与されたUSAGE権限を識別します。型、付与者、被付与者の組み合わせごとに行があります。このビューは複合データ型のみを表示します(理由は36.56を参照してください)。ドメイン権限については36.55を参照してください。

表36.52 udt_privilegesの列

列 型 説明
grantor sql_identifier 権限を与えたロールの名前です。
grantee sql_identifier 権限を与えられたロールの名前です。
udt_catalog sql_identifier 型を持つデータベースの名前(常に現在のデータベースです)。
udt_schema sql_identifier 型を持つスキーマの名前
udt_name sql_identifier 型の名前
privilege_type character_data 常にTYPE USAGE
is_grantable yes_or_no この権限を付与可能な場合はYES、さもなくばNOです。

36.55. usage_privileges

usage_privilegesビューは、現在有効なロールに、もしくは現在有効なロールによって与えられた、各種オブジェクト上のUSAGE権限を識別します。これは今のところ、PostgreSQLでは照合、ドメイン、外部データラッパ、外部サーバ、およびシーケンスに適用します。オブジェクトと許可を与えた者、許可を受けた者の組み合わせごとに1行があります。

PostgreSQLでは、照合は実際の権限を所有しませんので、このビューは全ての照合に対して所有者からPUBLICに与えられた暗黙の付与できないUSAGE権限を示します。しかし、その他のオブジェクトの種類は実際の権限を示します。

PostgreSQLでは、シーケンスはUSAGEに加えてSELECTとUPDATE権限もサポートします。これらは非標準であるため、情報スキーマのビューでは参照できません。

表36.53 usage_privilegesの列

列 型 説明
grantor sql_identifier 権限を与えたロールの名前です。
grantee sql_identifier 権限を与えられたロールの名前です。
object_catalog sql_identifier オブジェクトを持つデータベースの名前(常に現在のデータベースです)。
object_schema sql_identifier 適用されるオブジェクトを持つスキーマの名前。そうでなければ空文字列
object_name sql_identifier オブジェクトの名前です。
object_type character_data COLLATIONまたはDOMAINまたはFOREIGN DATA WRAPPERまたはFOREIGN SERVERまたはSEQUENCE
privilege_type character_data 常にUSAGEです。
is_grantable yes_or_no この権限を付与可能な場合はYES、さもなくばNOです。

36.56. user_defined_types

user_defined_typesビューは、現在は現在のデータベースで定義された全ての複合データ型を含みます。表示される型は、現在のユーザが(所有者である、何らかの権限を持っているといった方法で)アクセスできるものだけです。

SQLは二種類のユーザ定義データ型を知っています。構造化型(PostgreSQLでは複合データ型として知られています)と特殊型(PostgreSQLでは実装されていません)。将来を見越して、user_defined_type_category列をこれらを区別するために使用します。PostgreSQLの拡張である基本型や列挙型といった他のユーザ定義型はここには表示されません。ドメインについては代わりに36.23を参照してください。

表36.54 user_defined_typesの列

列 型 説明
user_defined_type_catalog sql_identifier 型を持つデータベースの名前です(常に現在のデータベースです)。
user_defined_type_schema sql_identifier 型を持つスキーマの名前です。
user_defined_type_name sql_identifier 型の名前

列 型 説明
user_defined_type_category character_data 現在は常にSTRUCTUREDです。
is_instantiable yes_or_no PostgreSQLでは利用できない機能に適用されるものです。
is_final yes_or_no PostgreSQLでは利用できない機能に適用されるものです。
ordering_form character_data PostgreSQLでは利用できない機能に適用されるものです。
ordering_category character_data PostgreSQLでは利用できない機能に適用されるものです。
ordering_routine_catalog sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
ordering_routine_schema sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
ordering_routine_name sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
reference_type character_data PostgreSQLでは利用できない機能に適用されるものです。
data_type character_data PostgreSQLでは利用できない機能に適用されるものです。
character_maximum_length cardinal_number PostgreSQLでは利用できない機能に適用されるものです。
character_octet_length cardinal_number PostgreSQLでは利用できない機能に適用されるものです。
character_set_catalog sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
character_set_schema sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
character_set_name sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
collation_catalog sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
collation_schema sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
collation_name sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
numeric_precision cardinal_number PostgreSQLでは利用できない機能に適用されるものです。

列 型 説明
numeric_precision_radix cardinal_number PostgreSQLでは利用できない機能に適用されるものです。
numeric_scale cardinal_number PostgreSQLでは利用できない機能に適用されるものです。
datetime_precision cardinal_number PostgreSQLでは利用できない機能に適用されるものです。
interval_type character_data PostgreSQLでは利用できない機能に適用されるものです。
interval_precision cardinal_number PostgreSQLでは利用できない機能に適用されるものです。
source_dtd_identifier sql_identifier PostgreSQLでは利用できない機能に適用されるものです。
ref_dtd_identifier sql_identifier PostgreSQLでは利用できない機能に適用されるものです。

36.57. user_mapping_options

user_mapping_optionsビューは現在のデータベースでユーザマッピングに定義された全てのオプションを含みます。現在のユーザが対応する外部サーバへのアクセスを所有しているとき（所有者または何らかの権限を持っていることで）表示されます。

表36.55 user_mapping_optionsの列

列 型 説明
authorization_identifier sql_identifier マップされているユーザ名、またはマッピングがpublicの場合PUBLICです。
foreign_server_catalog sql_identifier このマッピングにより使用される外部サーバが定義されたデータベース名です（常に現在のデータベースです）。
foreign_server_name sql_identifier このマッピングに使用される外部サーバ名です。
option_name sql_identifier オプションの名前です。
option_value character_data オプションの値です。この列は、現在のユーザがマップされたユーザか、マッピングがPUBLIC用かつ現在のユーザがサーバ所有者、もしくは現在のユーザがスーパーユーザでない限りNULLを示します。ユーザマッピングオプションとして格納されているパスワード情報を保護するためです。

36.58. user_mappings

user_mappingsビューは現在のデータベースで定義されたユーザマッピングすべてを含みます。現在のユーザが対応する外部サーバへアクセスを持っているとき(所有者か、何らかの権限を持っていることとして)それらのユーザのマッピングのみを示します。

表36.56 user_mappingsの列

列 型 説明
authorization_identifier sql_identifier マップされているユーザ名、またはマッピングがpublicの場合PUBLICです。
foreign_server_catalog sql_identifier このマッピングにより使用される外部サーバが定義されたデータベース名です(常に現在のデータベースです)。
foreign_server_name sql_identifier このマッピングに使用される外部サーバ名です。

36.59. view_column_usage

view_column_usageビューは、ビューの問い合わせ式(ビューを定義するSELECT文)で使用される全ての列を識別します。現在有効なロールがその列を含むテーブルの所有者であるもののみが含まれます。

> **注記**
>
> システムテーブルの列は含まれません。これはいつか修正しなければなりません。

表36.57 view_column_usageの列

列 型 説明
view_catalog sql_identifier ビューを持つデータベースの名前です(常に現在のデータベースです)。
view_schema sql_identifier ビューを持つスキーマの名前です。
view_name sql_identifier ビューの名前です。
table_catalog sql_identifier ビューで使用される列を持つテーブルが含まれるデータベースの名前です (常に現在のデータベースです)。
table_schema sql_identifier ビューで使用される列を持つテーブルが含まれるスキーマの名前です。
table_name sql_identifier ビューで使用される列を持つテーブルの名前です。

列 型 説明
column_name sql_identifier ビューで使用される列の名前です。

36.60. view_routine_usage

view_routine_usageビューは、ビューの問い合わせ式(ビューを定義するSELECT文)で使用されるすべてのルーチン(関数およびプロシージャ)を識別します。現在有効なロールが所有するルーチンのみが含まれます。

表36.58 view_routine_usageの列

列 型 説明
table_catalog sql_identifier ビューを持つデータベースの名前です(常に現在のデータベースです)。
table_schema sql_identifier ビューを持つスキーマの名前です。
table_name sql_identifier ビューの名前です。
specific_catalog sql_identifier 関数が含まれるデータベースの名前です(常に現在のデータベースです)。
specific_schema sql_identifier 関数が含まれるスキーマの名前です。
specific_name sql_identifier 関数の「仕様名称」です。詳細は36.41を参照してください。

36.61. view_table_usage

view_table_usageビューは、ビューの問い合わせ式(ビューを定義するSELECT文)で使用されるすべてのテーブルを識別します。現在有効なロールがそのテーブルの所有者であるもののみが含まれます。

> **注記**
>
> システムテーブルは含まれません。これはいつか修正しなければなりません。

表36.59 view_table_usageの列

列 型 説明
view_catalog sql_identifier

列 型 説明
ビューを持つデータベースの名前です(常に現在のデータベースです)。
view_schema sql_identifier ビューを持つスキーマの名前です。
view_name sql_identifier ビューの名前です。
table_catalog sql_identifier ビューで使用されるテーブルを持つデータベースの名前です (常に現在のデータベースです)。
table_schema sql_identifier ビューで使用されるテーブルを持つスキーマの名前です。
table_name sql_identifier ビューで使用されるテーブルの名前です。

36.62. views

viewsビューには、現在のデータベースで定義されたすべてのビューがあります。現在のユーザが(所有者である、何らかの権限を持っているといった方法で)アクセスすることができるビューのみが表示されます。

表36.60 viewsの列

列 型 説明
table_catalog sql_identifier ビューを持つデータベースの名前です(常に現在のデータベースです)。
table_schema sql_identifier ビューを持つスキーマの名前です。
table_name sql_identifier ビューの名前です。
view_definition character_data ビューを定義する問い合わせ式です (現在有効なロールがビューの所有者でない場合はNULLです)。
check_option character_data ビューにCHECK OPTIONが定義されている場合はCASCADEDまたはLOCAL、さもなくばNONEです。
is_updatable yes_or_no ビューが更新可能(UPDATEおよびDELETEが可能)な場合YES、さもなくばNOです。
is_insertable_into yes_or_no ビューが挿入可能(INSERTが可能)な場合YES、さもなくばNOです。
is_trigger_updatable yes_or_no ビューにINSTEAD OF UPDATEトリガーが定義されている場合はYES、さもなくばNOです。
is_trigger_deletable yes_or_no ビューにINSTEAD OF DELETEトリガーが定義されている場合はYES、さもなくばNOです。

列 型 説明
is_trigger_insertable_into yes_or_no ビューにINSTEAD OF INSERTトリガーが定義されている場合はYES、さもなくばNOです。

パート V. サーバプログラミング

ここでは、ユーザ定義の関数、データ型、演算子、トリガなどを使用してサーバの機能を拡張する方法について説明します。これらはおそらく、PostgreSQLに関するユーザ向けの文書を理解した後にのみたどり着く先進的な話題です。また、最後の数章でPostgreSQLに附属するサーバサイドのプログラミング言語についても説明します。同時にサーバサイドのプログラミング言語に関する一般的な問題についても説明します。サーバサイドのプログラミング言語の章に進む前に、少なくとも、第37章（関数も説明しています）の最初の数節を読破することは重要です。

目次

第37章 SQLの拡張

本節では以下を追加することでPostgreSQLのSQL問い合わせ言語をどのように拡張できるかを説明します。

- 関数（37.3から）。
- 集約（37.12から）。
- データ型（37.13から）。
- 演算子（37.14から）。
- インデックス用演算子クラス（37.16から）。
- 関連オブジェクトのパッケージ（37.17から）。

37.1. 拡張の作用法

PostgreSQLの動作は、カタログに定義された方法で駆動されているため拡張が可能です。もし標準のリレーショナルデータベースシステムに慣れ親しんでいるのであれば、システムカタログとして一般に知られている中に、データベース、テーブル、列などの情報が格納されていることは知っていると思います。（システムによってはデータディクショナリと呼ぶものもあります。）このカタログはユーザの目には他のテーブルと同じように見えますが、DBMSは内部情報をそこに格納しているのです。PostgreSQLと標準的なリレーショナルデータベースシステムの重要な違いは、PostgreSQLはカタログにより多くの情報を格納するということです。テーブルと列に関する情報だけではなく、データ型、関数、アクセスメソッドなどの情報も格納されています。これらのテーブルはユーザが変更できます。そして、PostgreSQLは操作をこれらのテーブルに基づいて行うので、PostgreSQLはユーザによって拡張することができるのです。これに対して、一般のデータベースシステムでは、ソースコード内にハードコーディングされたプロシージャを変えるか、DBMSベンダによって特別に書かれたモジュールをロードしなければ拡張することができません。

さらにPostgreSQLサーバは動的ローディングによってユーザの作成したコードを取り入れることができます。つまり、ユーザが新しい型か関数を実装するオブジェクトコードファイル（例えば共有ライブラリ）を指定することができ、PostgreSQLは要求された時にロードします。SQLで作成されたコードをサーバに追加するのはさらに簡単です。このように演算を「その場で」変えることができるため、PostgreSQLは新しいアプリケーションや格納構造をラピッドプロトタイプする場合に適しています。

37.2. PostgreSQLの型システム

PostgreSQLのデータ型は、基本型、コンテナ型、ドメイン、疑似型に分類されます。

37.2.1. 基本型

基本型は`integer`のように、SQL言語レベル以下で実装されたものです（通常はCのような低レベル言語で作成されます）。一般的にこれらは抽象データ型とも呼ばれるものに対応します。PostgreSQLは、ユーザによって提供された関数を通してのみ、こうした型に対する操作が可能で、また、こうした型の動作をユーザが記述する限りにおいてのみ理解します。組み込みの基本型は、第8章に記載されています。

列挙(enum)型は基本型の一種とみなすことができます。主な違いは、列挙型は低レベルプログラミング無しに、SQLコマンドだけで作ることができることです。より詳細については、8.7を参照してください。

37.2.2. コンテナ型

PostgreSQLには三種類の「コンテナ」型があります。これは他の型の複数の値を含む型です。配列、複合型、範囲型があります。

配列は、全て同じ型の複数の値を保持することができます。配列型は各基本型、複合型、範囲型およびドメイン型に対して自動的に作られます。しかし、配列の配列はありません。この型システムにおいては多次元配列は一次元配列と同じです。より詳細については、8.15を参照してください。

ユーザがテーブルを作成すると、複合型、もしくは行型が作成されます。関連するテーブルを持たない「スタンドアローン」の複合型をCREATE TYPEを使用して定義することもできます。複合型は関連したフィールド名を持つ基本型の単なるリストです。複合型の値は行もしくはフィールド値のレコードです。より詳細については、8.16を参照してください。

範囲型は同じ型の二つの値を保持することができます。これらは範囲の下限と上限です。範囲型はユーザによって作られますが、少数の組み込みの型もあります。より詳細については、8.17を参照してください。

37.2.3. ドメイン

ドメインは、特定の元となる型に基づいたもので、多くの目的では、その元となる型と交換可能です。しかし、ドメインは元となる基本型で許可される範囲内で値の有効範囲を制限する制約を持つことができます。ドメインはSQLコマンドのCREATE DOMAINを使って作られます。より詳細については、8.18を参照してください。

37.2.4. 疑似型

特殊な目的用に数個の「疑似型」があります。疑似型はテーブルの列やコンテナ型の構成要素として現れることはありません。しかし、関数の引数や結果型を宣言する際に使用することができます。これは、型システム内で特殊な関数クラスを識別するための機構を提供します。表 8.27に既存の疑似型を列挙します。

37.2.5. 多様型

いくつかの特殊な用途を持つ疑似型は多様型で、多様関数を宣言するのに使われます。この強力な機能により、特定の呼び出しで実際に渡されたデータ型で決定される具体的なデータ型について、一つの関数定義で多数の異なるデータ型を処理できるようになります。多様型を表 37.1に示します。これらの使用例は37.5.10にあります。

表37.1 多様型

名前	族	説明
anyelement	Simple	関数があらゆるデータ型を受け付けることを示します
anyarray	Simple	関数があらゆる配列データ型を受け付けることを示します

名前	族	説明
anynonarray	Simple	関数があらゆる配列でないデータ型を受け付けることを示します
anyenum	Simple	関数があらゆる列挙型(8.7を参照)を受け付けることを示します
anyrange	Simple	関数があらゆる範囲データ型(8.17を参照)を受け付けることを示します
anycompatible	Common	さまざまな引数の共通データ型への自動昇格によって、関数があらゆるデータ型を受け付けることを示します
anycompatiblearray	Common	さまざまな引数の共通データ型への自動昇格によって、関数があらゆる配列データ型を受け付けることを示します
anycompatiblenonarray	Common	さまざまな引数の共通データ型への自動昇格によって、関数があらゆる配列でないデータ型を受け付けることを示します。
anycompatiblerange	Common	さまざまな引数の共通データ型への自動昇格によって、配列があらゆる範囲データ型を受け付けることを示します。

多様の引数と結果は互いに結びつけられていて、多様関数を呼ぶ問い合わせが解析されるときに特定のデータ型に決定されます。2つ以上の多様引数がある時には、入力値の実データ型は後述するように合わせなければなりません。関数の結果型が多様、あるいは、多様型の出力パラメータを持つ場合には、それらの結果の型は後述するように多様入力の実際の型から導出されます。

多様型の「simple」族では、一致と導出の規則は以下のように動作します。

anyelementとして宣言された位置(引数もしくは戻り値)にはそれぞれ、任意の実データ型を指定することができますが、1つの呼び出しでは、これらはすべて同一の実データ型でなければなりません。anyarrayとして宣言された位置にはそれぞれ、任意の配列データ型を持つことができますが、同様にこれらはすべて同じデータ型でなければなりません。また同様に、anyrangeとして宣言された位置はすべて同じ範囲型でなければなりません。さらに、anyarrayと宣言された位置とanyelementと宣言された位置の両方がある場合、anyarrayの位置の実際の配列型は、その要素の型がanyelement位置に現れる型と同じでなければなりません。同様にanyrangeと宣言された位置とanyelementもしくはanyarrayと宣言された位置の両方がある場合、anyrangeの位置の実際の範囲型は、その範囲の派生元型がanyelement位置に現れる型と同じであり、anyarray位置の要素の型と同じでなければなりません。anynonarrayは、実際の型が配列型であってはならないという制限が加わっている点を除き、anyelementとまったく同様に扱われます。anyenumは、実際の型が列挙型でなければならないという制約が加わっている点を除き、anyelementとまったく同様に扱われます。

このように、2つ以上の引数位置が多様型と宣言されると、全体の効果として、実引数型の特定の組み合わせのみが許されるようになります。例えば、equal(anyelement, anyelement)と宣言された関数は、2つの引数が同じデータ型である限り、任意の入力値を2つ取ることになります。

関数の戻り値を多様型として宣言する時、少なくとも1つの引数位置も多様でなければなりません。そして多様の引数として与えられる実データ型がその呼び出しの実結果型を決定します。例えば、配列添字機構がなかったとすると、subscript(anyarray, integer) returns anyelementとして添字機構を実装する関数

を定義できます。この宣言には、最初の実引数は配列型になり、パーサはこの最初の実引数の型より正しい結果型を推論することができます。他にも例えば、f(anyarray) returns anyenumと宣言された関数は列挙型の配列のみを受け付けます。

ほとんどの場合、パーサは同じ族の異なる多様型の引数から多様結果型の実データ型を推論できます。例えば、anyarrayをanyelementから、もしくはその逆から推定できます。例外はanyrange型の多様結果はanyrange型の引数を必要とします。anyarrayもしくはanyelementの引数からは推定できません。これは、同じ派生元型の複数の範囲型が存在する可能性があるためです。

anynonarray型とanyenum型が、別個の型変数を表していないことに注意してください。これはanyelementと同じ型で、追加の制約が付いているだけです。例えば、f(anyelement, anyenum)として関数を宣言することは、f(anyenum, anyenum)と宣言することと同一です。両方の実引数は同じ列挙型でなければなりません。

多様型の「common」族については、一致と導出の規則は概ね「simple」と同じに動作しますが、一つの大きな違いがあります。一つの共通型へ暗黙にキャスト可能である限り、引数の実際の型が同一である必要がありません。共通型はUNIONや関連する構文と同じ規則(10.5を参照)に従って選択されます。共通型の選択では、anycompatibleやanycompatiblenonarrayの入力の実際の型、anycompatiblearray入力の配列要素型、anycompatiblerange入力の範囲の派生元型が、考慮されます。anycompatiblenonarrayが存在する場合、共通型は配列でない型である必要があります。共通データ型が特定されたなら、anycompatibleおよびanycompatiblenonarrayの位置の引数は自動的にその型にキャストされ、anycompatiblearray位置の引数は自動的にその型に対する配列にキャストされます。

その派生元型だけ分かっている範囲型を選択する方法がないため、anycompatiblerangeの使用にはその型で宣言されている全ての引数が同じ実範囲型をとり、また、その型の派生元型は選択された共通型に即したものである必要があり、そのため、範囲値のキャストは必要ありません。anyrangeと同様に、関数の結果型としてのanycompatiblerangeの使用には、anycompatiblerangeの引数がある必要があります。

anycompatibleenum型は無いことに注意してください。通常、列挙型への暗黙キャストはありません。これは異なる列挙入力に対する共通型を決める方法が無いことを意味します。そのため、このような型はあまり有益ではないでしょう。

「simple」と「common」の多様族は型変数の二つの独立したセットに相当します。例えば以下を考えてください。

```
CREATE FUNCTION myfunc(a anyelement, b anyelement,
                       c anycompatible, d anycompatible)
RETURNS anycompatible AS ...
```

この関数の実際の呼び出しでは、最初の2つの入力は正確に同じ型を持たなければなりません。最後の2つの入力は共通型に昇格できなければなりませんが、この型が最初の2つの入力型と何らか関係がある必要はありません。結果は最後の2つの入力の共通型を持ちます。

可変長引数の関数(37.5.5で説明する可変個の引数を取る関数)を多様とすることができます。最後のパラメータをVARIADIC anyarrayまたはVARIADIC anycompatiblearrayと宣言することで実現されます。引数を一致させ、実際の結果型を決めるために、こうした関数はanynonarrayまたはanycompatiblenonarrayパラメータをあたかも適切な個数記述した場合と同様に動作します

37.3. ユーザ定義関数

PostgreSQLは4種類の関数を提供します。

- 問い合わせ言語関数（SQLで作成された関数）（37.5）
- 手続型言語関数（PL/pgSQLやPL/Tclなどで作成された関数）（37.8）
- 内部関数（37.9）
- C言語関数（37.10）

すべての関数は、基本型、複合型、またはこの組み合わせを引数（パラメータ）として受け付けることが可能です。また、すべての関数は基本型または複合型を返すことが可能です。関数は、基本型の集合または複合型の集合を返すように定義することもできます。

多くの関数は（多様型のような）特定の疑似型を引数としたり返したりすることができます。しかし、利用できる機能は様々です。詳細は各関数の種類の説明を参照してください。

SQL関数の定義の方法が最も簡単ですので、そちらから説明します。SQL関数にある概念のほとんどは、他の種類の関数にも適用できます。

本章の全体に関して、その例をより理解するために、CREATE FUNCTIONコマンドのマニュアルページを一読することが有用です。本章の例のいくつかはPostgreSQLソース配布物内の`src/tutorial`ディレクトリにある`funcs.sql`と`funcs.c`でも参照することができます。

37.4. ユーザ定義プロシージャ

プロシージャは関数と似たデータベースオブジェクトです。違いはプロシージャは値を返さず、そのため戻り型の宣言が無いことです。関数が問い合わせやDMLコマンドの一部として呼び出されるのに対して、プロシージャは明示的にCALLコマンドを使って呼び出されます。CALLコマンドが明示的なトランザクション内で無い場合、多くのサーバサイド言語のプロシージャはコミット、ロールバック、および新しいトランザクションの開始が実行出来ますが、関数には出来ません。

本章で後述するどのようにユーザ定義関数を定義するかの説明は、CREATE PROCEDUREコマンドを代わりに使う、戻り値が無い、変動性区分などいくつかの他の仕様が該当しないという点を除き、プロシージャにも同様にあてはまります。

関数とプロシージャは、ひとまとめに*ルーチン*とも言われます。関数とプロシージャを区別することなしに操作できるALTER ROUTINEやDROP ROUTINEなどのコマンドがあります。しかしながら、`CREATE ROUTINE`コマンドは無いことに注意してください。

37.5. 問い合わせ言語（SQL）関数

SQL関数は、任意のSQL文のリストを実行し、そのリストの最後の問い合わせの結果を返します。単純な（集合ではない）場合、最後の問い合わせの結果の最初の行が返されます。（複数行の結果のうちの「最初の行」は、`ORDER BY`を使用しない限り定義付けることができないことを覚えておいてください。）最後の問い合わせが何も行を返さない時はNULL値が返されます。

他にも、SQL関数は、SETOF sometype型を返すように指定すること、または同意のRETURNS TABLE(columns)と宣言することにより、集合(つまり複数の行)を返すように宣言することもできます。この場合、最後の問い合わせの結果のすべての行が返されます。詳細は後で説明します。

SQL関数の本体は、セミコロンで区切ったSQL文のリストでなければなりません。最後の文の後のセミコロンは省略可能です。関数がvoidを返すものと宣言されていない限り、最後の文はSELECT、またはRETURNING句を持つINSERT、UPDATE、またはDELETEでなければなりません。

SQL言語で作成された、任意のコマンド群はまとめて、関数として定義することができます。SELECT問い合わせ以外に、データ変更用の問い合わせ(つまり、INSERT、UPDATE、DELETE)やその他のSQLコマンドを含めることができます。(SQL関数ではCOMMIT、SAVEPOINTなどのトランザクション制御コマンドおよびVACUUMなどのユーティリティコマンドは使用することはできません。)しかし、最後のコマンドは、関数の戻り値型として定義したものを返すSELECT、またはRETURNING句があるものでなければなりません。その他にも、何か動作をさせるが、有用な値を返さないSQL関数を定義したいのであれば、voidを返すものと定義することで実現可能です。たとえば、以下の関数はempテーブルから負の給料となっている行を削除します。

```
CREATE FUNCTION clean_emp() RETURNS void AS '
    DELETE FROM emp
        WHERE salary < 0;
' LANGUAGE SQL;

SELECT clean_emp();

 clean_emp
-----------

(1 row)
```

注記

SQL関数の本体全体は、その一部が実行される前に解析されます。SQL関数はシステムカタログを変更するコマンド(例えばCREATE TABLE)を含むことができますので、そのようなコマンドの効果は関数の以降のコマンドの解析中は可視ではありません。それゆえ、例えば、CREATE TABLE foo (...); INSERT INTO foo VALUES(...);は単一のSQL関数にまとめられていると期待したようには動作しません。INSERTコマンドが解析されている時にはfooがまだ存在しないからです。このような場合にはSQL関数の代わりにPL/pgSQLを使うことを薦めます。

CREATE FUNCTIONコマンドの構文では、関数本体は文字列定数として作成される必要があります。この文字列定数の記述には、通常、ドル引用符付け(4.1.2.4)が最も便利です。文字列定数を単一引用符で括る通常の構文では、関数本体中で使用される単一引用符(')とバックスラッシュ(\)(エスケープ文字列構文を仮定)を二重にしなければなりません(4.1.2.1を参照)。

37.5.1. SQL関数用の引数

SQL関数の引数は関数本体内で名前または番号を用いて参照することができます。両方の方法の例を後で示します。

名前を使用するためには、関数引数を名前を持つものとして宣言し、その名前を関数本体内で記述するだけです。引数名が関数内の現在のSQLコマンドにおける任意の列名と同じ場合は、列名が優先されます。これを上書きするためには、function_name.argument_nameのように、引数名を関数自身の名前を付けて修飾してください。(もしこれも修飾された列名と競合する場合は、列名が優先されます。SQLコマンド内でテーブルに他の別名を付けることで、この曖昧さを防止することができます。)

古い番号による方法では、引数は関数本体内で$nという構文を用いて表すことができます。つまり、$1は第1引数を示し、$2は第2引数のようになります。これは特定の引数が名前付きで宣言されているかどうかに関係なく動作します。

引数が複合型の場合、argname.fieldnameや$1.fieldnameのようなドット表記を用いて引数の属性にアクセスすることができます。ここでも、引数名を持つ形式で曖昧さが発生する場合には関数名で引数名を修飾してください。

SQL関数の引数は、識別子としてではなく、データ値としてのみ使用することができます。したがって、例えば

```
INSERT INTO mytable VALUES ($1);
```

は正しいものですが、以下は動作しません。

```
INSERT INTO $1 VALUES (42);
```

> **注記**
>
> SQL関数の引数を参照するために名前を使用できる機能は、PostgreSQL 9.2で追加されました。これより古いサーバ内で使われる関数は$n記法を使用しなければなりません。

37.5.2. 基本型を使用するSQL関数

最も簡単なSQL関数は、引数を取らずに単にintegerのような基本型を返すものです。

```
CREATE FUNCTION one() RETURNS integer AS $$
    SELECT 1 AS result;
$$ LANGUAGE SQL;


-- 文字列リテラルの別の構文では
CREATE FUNCTION one() RETURNS integer AS '
    SELECT 1 AS result;
' LANGUAGE SQL;

SELECT one();

 one
-----
   1
```

関数本体内で関数の結果用に列の別名を（resultという名前で）定義したことに注目してください。しかし、この列の別名はこの関数の外部からは可視ではありません。したがって、その結果はresultではなく、oneというラベルで表示されています。

基本型を引数として取る、SQL関数を定義することはほとんどの場合簡単です。

```
CREATE FUNCTION add_em(x integer, y integer) RETURNS integer AS $$
    SELECT x + y;
$$ LANGUAGE SQL;

SELECT add_em(1, 2) AS answer;

 answer
--------
      3
```

この他に、引数に名前を付けることを省くことができます。この場合は番号を使用します。

```
CREATE FUNCTION add_em(integer, integer) RETURNS integer AS $$
    SELECT $1 + $2;
$$ LANGUAGE SQL;

SELECT add_em(1, 2) AS answer;

 answer
--------
      3
```

以下にもう少し役に立つ関数を示します。これは銀行口座からの引き落としに使用できます。

```
CREATE FUNCTION tf1 (accountno integer, debit numeric) RETURNS numeric AS $$
    UPDATE bank
        SET balance = balance - debit
        WHERE accountno = tf1.accountno;
    SELECT 1;
$$ LANGUAGE SQL;
```

以下のように、ユーザはこの関数を使用して、口座番号17から100ドルを引き出すことが可能です。

```
SELECT tf1(17, 100.0);
```

この例では、第一引数の名前にaccountnoを選びましたが、これはbankテーブルの列の名前と同じです。UPDATEコマンドの中では、accountnoはbank.accountno列を参照しますので、引数を参照するためにはtf1.accountnoを使用しなければなりません。もちろんこれは、引数に別の名前を使用することで防ぐことができます。

実際には、関数の結果を定数1よりもわかりやすい形にするために、以下のように定義するとよいでしょう。

```
CREATE FUNCTION tf1 (accountno integer, debit numeric) RETURNS numeric AS $$
    UPDATE bank
        SET balance = balance - debit
        WHERE accountno = tf1.accountno;
    SELECT balance FROM bank WHERE accountno = tf1.accountno;
$$ LANGUAGE SQL;
```

これは残高を調整し、更新後の残高を返します。同じことはRETURNINGを使用して1つのコマンドで行えます。

```
CREATE FUNCTION tf1 (accountno integer, debit numeric) RETURNS numeric AS $$
    UPDATE bank
        SET balance = balance - debit
        WHERE accountno = tf1.accountno
    RETURNING balance;
$$ LANGUAGE SQL;
```

SQL関数の最後のSELECT句やRETURNING句が関数で定義された結果型を正確に返さない場合、PostgreSQLは可能な場合に暗黙的キャストまたは代入キャストで必要な型に自動でキャストします。そうでない場合は明示的にキャストをする必要があります。例えば、前出のadd_em関数が代わりにfloat8型を返して欲しいとします。次のように記述すれば十分です。

```
CREATE FUNCTION add_em(integer, integer) RETURNS float8 AS $$
    SELECT $1 + $2;
$$ LANGUAGE SQL;
```

integerの和はfloat8に暗黙キャストできるからです。(キャストについての詳細は第10章またはCREATE CASTを参照して下さい)。

37.5.3. 複合型を使用するSQL関数

関数の引数に複合型を記述した場合、必要な引数を指定するだけではなく、必要とする引数の属性(フィールド)も指定する必要があります。例えば、empが従業員データを持つテーブルとすると、この名前はそのテーブル内の各行を表す複合型の名前でもあります。以下に示すdouble_salary関数は、該当する従業員の給料が倍増したらどうなるかを計算します。

```
CREATE TABLE emp (
    name        text,
    salary      numeric,
    age         integer,
    cubicle     point
);

INSERT INTO emp VALUES ('Bill', 4200, 45, '(2,1)');

CREATE FUNCTION double_salary(emp) RETURNS numeric AS $$
    SELECT $1.salary * 2 AS salary;
```

```
$$ LANGUAGE SQL;

SELECT name, double_salary(emp.*) AS dream
    FROM emp
    WHERE emp.cubicle ~= point '(2,1)';

 name | dream
------+-------
 Bill |  8400
```

$1.salaryという構文を使用して、引数の行値の1フィールドを選択していることに注目してください。また、table_name.*を使用したSELECTコマンドの呼び出しでは、複合型の値として、現在のテーブル行全体を表すテーブル名を使用していることにも注目してください。別の方法として、テーブル行は以下のようにテーブル名だけを使用して参照することができます。

```
SELECT name, double_salary(emp) AS dream
    FROM emp
    WHERE emp.cubicle ~= point '(2,1)';
```

しかし、この使用方法は混乱しやすいためお勧めしません。(テーブル行の複合型の値に対するこの二つの表記の詳細は8.16.5を参照してください)

その場で複合型の引数値を作成することが便利な場合があります。これはROW式で行うことができます。例えば、以下のようにして関数に渡すデータを調整することができます。

```
SELECT name, double_salary(ROW(name, salary*1.1, age, cubicle)) AS dream
    FROM emp;
```

複合型を返す関数を作成することもできます。以下に単一のemp行を返す関数の例を示します。

```
CREATE FUNCTION new_emp() RETURNS emp AS $$
    SELECT text 'None' AS name,
        1000.0 AS salary,
        25 AS age,
        point '(2,2)' AS cubicle;
$$ LANGUAGE SQL;
```

ここでは、各属性を定数で指定していますが、この定数を何らかの演算に置き換えることもできます。

関数を定義する上で、2つの重要な注意点を以下に示します。

- 問い合わせにおける選択リストの順番は、複合型に列が現れる順番と正確に一致する必要があります。(上で行ったように列に名前を付けても、システムは認識しません。)
- 各式の型が対応する複合型の列にキャスタができるようにする必要があります。さもなくば、以下のようなエラーとなります。

```
ERROR:  return type mismatch in function declared to return emp
```

```
DETAIL:  Final statement returns text instead of point at column 4.
```

基本型の場合と同様に、システムは明示的キャストを自動では挿入せず、暗黙または代入キャストのみをします。

同じ関数を以下のように定義することもできます。

```
CREATE FUNCTION new_emp() RETURNS emp AS $$
    SELECT ROW('None', 1000.0, 25, '(2,2)')::emp;
$$ LANGUAGE SQL;
```

ここで、正しい複合型の単一の列を単に返すSELECTを記述しました。今回の例ではこれはより優れたものとはいえませんが、例えば、必要な複合値を返す他の関数を呼び出して結果を計算しなければならない場合など、便利な解法になることがあります。他の例としては、単なる複合型ではなく複合型のドメインを返す関数を書こうとしてる場合に、単一列を返すように書くことが常に必要となります。なぜなら、行全体の結果を強制する方法がないからです。

この関数を、評価式で使って直接呼び出せますし、

```
SELECT new_emp();

         new_emp
-------------------------
 (None,1000.0,25,"(2,2)")
```

テーブル関数として呼び出しても直接呼び出せます。

```
SELECT * FROM new_emp();

 name | salary | age | cubicle
------+--------+-----+---------
 None | 1000.0 |  25 | (2,2)
```

2番目の方法については、37.5.7でより詳しく説明します。

複合型を返す関数を使用する時に、その結果から1つのフィールド（属性）のみを使用したいという場合があります。これは、以下のような構文で行うことができます。

```
SELECT (new_emp()).name;

 name
------
 None
```

パーサが混乱しないように、括弧を追加する必要があります。括弧なしで行おうとすると、以下のような結果になります。

```
SELECT new_emp().name;
```

```
ERROR:  syntax error at or near "."
LINE 1: SELECT new_emp().name;
                         ^
```

また、関数表記を使用して属性を抽出することもできます。

```
SELECT name(new_emp());

 name
------
 None
```

8.16.5で述べるように、フィールド表記と関数表記は等価です。

複合型を結果として返す関数を使用する他の方法は、その結果を、その行型を入力として受け付ける関数に渡す、以下のような方法です。

```
CREATE FUNCTION getname(emp) RETURNS text AS $$
    SELECT $1.name;
$$ LANGUAGE SQL;

SELECT getname(new_emp());
 getname
---------
 None
(1 row)
```

37.5.4. 出力パラメータを持つSQL関数

関数の結果の記述方法には、他にも出力パラメータを使用して定義する方法があります。以下に例を示します。

```
CREATE FUNCTION add_em (IN x int, IN y int, OUT sum int)
AS 'SELECT x + y'
LANGUAGE SQL;

SELECT add_em(3,7);
 add_em
--------
     10
(1 row)
```

37.5.2で示したadd_em版と基本的な違いはありません。複数列を返す関数を定義する簡単な方法を提供することが出力パラメータの本来の価値です。以下に例を示します。

```
CREATE FUNCTION sum_n_product (x int, y int, OUT sum int, OUT product int)
AS 'SELECT x + y, x * y'
```

```
LANGUAGE SQL;

 SELECT * FROM sum_n_product(11,42);
 sum | product
-----+---------
  53 |     462
(1 row)
```

これは基本的に、関数結果用の無名の複合型の作成を行います。上の例では、

```
CREATE TYPE sum_prod AS (sum int, product int);

CREATE FUNCTION sum_n_product (int, int) RETURNS sum_prod
AS 'SELECT $1 + $2, $1 * $2'
LANGUAGE SQL;
```

と同じ最終結果になります。しかし、独立した複合型定義に悩まされることがなくなり、便利であるともいえます。出力パラメータに割り振られた名前が単なる飾りではなく、無名複合型の列名を決定するものであることに注意してください。(出力パラメータの名前を省略した場合、システム自身が名前を選びます。)

SQLからこうした関数を呼び出す時、出力パラメータが呼び出し側の引数リストに含まれないことに注意してください。PostgreSQLでは入力パラメータのみが関数の呼び出しシグネチャを定義するとみなしているためです。これはまた、関数を削除することなどを目的に関数を参照する場合、入力パラメータのみが考慮されることを意味しています。上の関数は、次のいずれかの方法で削除することができます。

```
DROP FUNCTION sum_n_product (x int, y int, OUT sum int, OUT product int);
DROP FUNCTION sum_n_product (int, int);
```

パラメータには、IN(デフォルト)、OUT、INOUT、またはVARIADICという印を付与できます。INOUTパラメータは、入力パラメータ(呼び出し引数リストの一部)と出力パラメータ(結果のレコード型の一部)の両方を提供します。VARIADICパラメータは入力パラメータですが、次に説明するように特別に扱われます。

37.5.5. 可変長引数を取るSQL関数

すべての「オプションの」引数が同じデータ型の場合、SQL関数は可変長の引数を受け付けるように宣言できます。オプションの引数は配列として関数に渡されます。この関数は最後のパラメータをVARIADICと印を付けて宣言されます。このパラメータは配列型であるとして宣言されなければなりません。例をあげます。

```
CREATE FUNCTION mleast(VARIADIC arr numeric[]) RETURNS numeric AS $$
    SELECT min($1[i]) FROM generate_subscripts($1, 1) g(i);
$$ LANGUAGE SQL;

SELECT mleast(10, -1, 5, 4.4);
 mleast
--------
     -1
(1 row)
```

実際、VARIADICの位置以降の実引数はすべて、あたかも以下のように記述したかのように、1次元の配列としてまとめられます。

```
SELECT mleast(ARRAY[10, -1, 5, 4.4]);    -- 動作しません
```

しかし、実際にこのように記述することはできません。少なくとも、この関数定義に一致しません。VARIADIC印の付いたパラメータは、自身の型ではなく、その要素型が1つ以上存在することに一致します。

時として、variadic関数に既に構築された配列を渡せることは有用です。1つのvariadic関数が、自身の配列パラメータを他のものに渡したいとき特に便利です。また、これが、信用できないユーザがオブジェクトを作成できるスキーマにあるvariadic関数を呼び出す唯一の安全な方法です。10.3を参照してください。これは、呼び出しにVARIADICを指定することで行えます。

```
SELECT mleast(VARIADIC ARRAY[10, -1, 5, 4.4]);
```

これは関数のvariadicパラメータがその要素型に拡張するのを防ぎます。その結果、配列引数値が標準的にマッチされるようになります。VARIADICは関数呼び出しの最後の実引数としてのみ付加できます。

呼び出しでVARIADICを指定することは、variadic関数に空の配列を渡す唯一の方法でもあります。例えば、

```
SELECT mleast(VARIADIC ARRAY[]::numeric[]);
```

variadicパラメータが少なくとも1つの実引数と一致しなければなりませんので、単にSELECT mleast()と書くだけでは上手くいきません。(もしそのような呼び出しを許可したいのなら、mleastという名前のパラメータのない第2の関数を定義することもできます。)

variadicパラメータから生成される配列要素パラメータは、それ自身にはまったく名前を持たないものとして扱われます。これは、名前付き引数(4.3)を使用して可変長の関数を呼び出すことができないことを意味します。ただし、VARIADICを指定する場合は例外です。たとえば、

```
SELECT mleast(VARIADIC arr => ARRAY[10, -1, 5, 4.4]);
```

は動作しますが、

```
SELECT mleast(arr => 10);
SELECT mleast(arr => ARRAY[10, -1, 5, 4.4]);
```

は動作しません。

37.5.6. 引数にデフォルト値を持つSQL関数

一部またはすべての入力引数にデフォルト値を持つ関数を宣言することができます。デフォルト値は、関数が実際の引数の数に足りない数の引数で呼び出された場合に挿入されます。引数は実引数リストの終端から省略することができますので、デフォルト値を持つパラメータの後にあるパラメータはすべて、同様にデフォルト値を持たなければなりません。(名前付きの引数記法を使用してこの制限を緩和させることもできますが、まだ位置引数記法が実用的に動作できることが強制されています。) 使うかどうかに関わりなく、この能力は、あるユーザが他のユーザを信用しないデータベースで関数を呼び出す時に、セキュリティの事前の対策を必要とします。10.3を参照してください。

以下に例を示します。

```
CREATE FUNCTION foo(a int, b int DEFAULT 2, c int DEFAULT 3)
RETURNS int
LANGUAGE SQL
AS $$
    SELECT $1 + $2 + $3;
$$;

SELECT foo(10, 20, 30);
 foo
-----
  60
(1 row)

SELECT foo(10, 20);
 foo
-----
  33
(1 row)

SELECT foo(10);
 foo
-----
  15
(1 row)


SELECT foo();  -- 最初の引数にデフォルトがないため失敗
ERROR:  function foo() does not exist
```

=記号をDEFAULTキーワードの代わりに使用することもできます。

37.5.7. テーブルソースとしてのSQL関数

すべてのSQL関数は問い合わせのFROM句で使用できますが、複合型を返す関数に特に便利です。関数が基本型を返すよう定義されている場合、テーブル関数は1列からなるテーブルを作成します。関数が複合型を返すよう定義されている場合、テーブル関数は複合型の列のそれぞれに対して1つの列を作成します。

以下に例を示します。

```
CREATE TABLE foo (fooid int, foosubid int, fooname text);
INSERT INTO foo VALUES (1, 1, 'Joe');
INSERT INTO foo VALUES (1, 2, 'Ed');
INSERT INTO foo VALUES (2, 1, 'Mary');

CREATE FUNCTION getfoo(int) RETURNS foo AS $$
```

```
    SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;

SELECT *, upper(fooname) FROM getfoo(1) AS t1;

 fooid | foosubid | fooname | upper
-------+----------+---------+-------
     1 |        1 | Joe     | JOE
(1 row)
```

例からわかる通り、関数の結果の列を通常のテーブルの列と同じように扱うことができます。

この関数の結果得られたのは1行のみであることに注意してください。これはSETOFを指定しなかったためです。これについては次節で説明します。

37.5.8. 集合を返すSQL関数

SQL関数がSETOF *sometype*を返すよう宣言されている場合、関数の最後の問い合わせは最後まで実行され、各出力行は結果集合の要素として返されます。

この機能は通常、関数をFROM句内で呼び出す時に使用されます。この場合、関数によって返される各行は、問い合わせによって見えるテーブルの行になります。例えば、テーブルfooの内容が上記と同じであれば以下のようになります。

```
CREATE FUNCTION getfoo(int) RETURNS SETOF foo AS $$
    SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;

SELECT * FROM getfoo(1) AS t1;
```

この出力は以下の通りです。

```
fooid | foosubid | fooname
-------+----------+---------
     1 |        1 | Joe
     1 |        2 | Ed
(2 rows)
```

また、以下のように出力パラメータで定義された列を持つ複数の行を返すことも可能です。

```
CREATE TABLE tab (y int, z int);
INSERT INTO tab VALUES (1, 2), (3, 4), (5, 6), (7, 8);

CREATE FUNCTION sum_n_product_with_tab (x int, OUT sum int, OUT product int)
RETURNS SETOF record
AS $$
    SELECT $1 + tab.y, $1 * tab.y FROM tab;
$$ LANGUAGE SQL;
```

```
SELECT * FROM sum_n_product_with_tab(10);
 sum | product
-----+---------
  11 |      10
  13 |      30
  15 |      50
  17 |      70
(4 rows)
```

ここで重要な点は、関数が1行だけではなく複数行を返すことを示すためにRETURNS SETOF recordを記述しなければならない点です。出力パラメータが1つしか存在しない場合は、recordではなく、そのパラメータの型を記述してください。

集合を返す関数を、それぞれの呼び出し時に連続するテーブル行または副問い合わせに由来するパラメータを付けて、複数回呼び出すことで問い合わせ結果を構築することはしばしば有用です。お勧めする方法は、7.2.1.5で説明するLATERALキーワードを使用することです。以下は集合を返す関数を使用して、ツリー構造の要素を模擬する例です。

```
SELECT * FROM nodes;
   name    | parent
-----------+--------
 Top       |
 Child1    | Top
 Child2    | Top
 Child3    | Top
 SubChild1 | Child1
 SubChild2 | Child1
(6 rows)

CREATE FUNCTION listchildren(text) RETURNS SETOF text AS $$
    SELECT name FROM nodes WHERE parent = $1
$$ LANGUAGE SQL STABLE;

SELECT * FROM listchildren('Top');
 listchildren
--------------
 Child1
 Child2
 Child3
(3 rows)

SELECT name, child FROM nodes, LATERAL listchildren(name) AS child;
  name  |   child
--------+-----------
 Top    | Child1
 Top    | Child2
```

```
 Top    | Child3
 Child1 | SubChild1
 Child1 | SubChild2
(5 rows)
```

この例は単純な結合でできない何かを行うものではありません。しかしより複雑な計算では、何らかの作業を関数内に押し込むオプションはかなり便利です。

集合を返す関数は問い合わせの選択リスト内でも呼び出すことができます。問い合わせ自身によって生成する各行に対し、集合を返す関数が呼び出され、関数の結果集合の各要素に対して出力行が生成されます。上の例は以下のような問い合わせでも実現することができます。

```
SELECT listchildren('Top');
 listchildren
--------------
 Child1
 Child2
 Child3
(3 rows)

SELECT name, listchildren(name) FROM nodes;
  name  | listchildren
--------+--------------
 Top    | Child1
 Top    | Child2
 Top    | Child3
 Child1 | SubChild1
 Child1 | SubChild2
(5 rows)
```

最後のSELECTにおいて、Child2とChild3などが出力行に表示されていないことに注意してください。これは、listchildrenがこの入力に対して空の集合を返すため出力行が生成されないからです。LATERAL構文を使用した時の関数の結果との内部結合から得る場合と同じ動作です。

選択リストにある集合を返す関数に対するPostgreSQLの振舞いは、集合を返す関数がLATERAL FROM句に書かれている場合とほとんど同じです。例えば

```
SELECT x, generate_series(1,5) AS g FROM tab;
```

は

```
SELECT x, g FROM tab, LATERAL generate_series(1,5) AS g;
```

とほぼ同じです。この特定の例では、gは実際にはtabにLATERALには依存しませんので、プランナがネステッドループ結合の外にgを置くことを選ぶかもしれないという点を除いて、全く同じです。そのため、出力行の順番が異なる結果になるかもしれません。選択リスト内の集合を返す関数は、FROM句からの次の行が考慮される前に関数の実行が完了するよう、FROM句の残りとのネステッドループ結合の中にあるかのように必ず評価されます。

問い合わせの選択リスト内に集合を返す関数が2つ以上ある場合には、振舞いは一つのLATERAL ROWS FROM(...) FROM句に関数を置いた場合に得られるものと似ています。元となる問い合わせからの各行に対して、各関数からの最初の結果を使った出力行、2番目の結果を使った出力行、と続きます。集合を返す関数の中に他のものより出力の数が少ないものがある場合には、欠けたデータの代わりにNULL値が使われますので、1つの元となる行から作られる行の合計の数は、一番多くの出力を出力する集合を返す関数に対するのと同じだけになります。そのため、集合を返す関数はすべてが尽きるまで「歩調を合わせて」実行され、それから次の元となる行へと実行が続きます。

集合を返す関数は、FROM句内では許されていませんが、選択リスト内では入れ子にできます。その場合、入れ子の各階層は、別々のLATERAL ROWS FROM(...)であるかのように別々に扱われます。例えば、

```
SELECT srf1(srf2(x), srf3(y)), srf4(srf5(z)) FROM tab;
```

では、集合を返す関数srf2、srf3、srf5はtabの各行に対して歩調を合わせて実行され、次に階層の低い関数が生成した各行に対してsrf1とsrf4が歩調を合わせて適用されます。

集合を返す関数はCASEやCOALESCEのような条件を評価する構成の中では使えません。例えば、

```
SELECT x, CASE WHEN x > 0 THEN generate_series(1, 5) ELSE 0 END FROM tab;
```

を考えてください。これは、x > 0である入力行の5回の繰り返しとそうでないものの1回の繰り返しを生成するように思えるかもしれません。しかし、実際には、generate_series(1, 5)はCASEが評価される前に暗黙のLATERAL FROMの中で実行されますので、各入力行に対して5回の繰り返しを生成します。混乱を減らすため、そのような場合にはその代わりに解析時エラーになります。

注記

もし関数の最後のコマンドがRETURNINGを持つINSERT、UPDATE、またはDELETEである場合、関数がSETOF付きで宣言されていない、または呼び出す問い合わせがすべての結果行を取り出さなくても、そのコマンドは完了まで実行されます。RETURNING句で生成される余計な行はすべて警告無しに削除されますが、コマンド対象のテーブルの変更はそれでも起こります（そして、関数から戻る前にすべて完了します）。

注記

PostgreSQL 10より前では、集合を返す関数を2つ以上同じ選択リストに置くと常に等しい数の行を生成しない限りあまり賢くは振舞いませんでした。そうでなければ、得られるのは、集合を返す関数が生成する行の数の最小公倍数に等しい数の出力行でした。また、入れ子の集合を返す関数は上に書いたようには動作しませんでした。代わりに、集合を返す関数は多くても1つの集合を返す引数を持ち、集合を返す関数の各入れ子は独立に実行されました。また、条件実行(CASE等の内側にある集合を返す関数)は以前は認められており、事態をより複雑にしていました。PostgreSQLの古いバージョンで動作することが必要な問い合わせを書く場合には、バージョンが異なっても一貫した結果を返しますので、LATERAL構文を使うことを勧めます。集合を返す関数の条件実行に頼った問い合わせがあるのなら、条件確認を独自の集合を返す関数の中に移動することで修正できます。例えば

```
SELECT x, CASE WHEN y > 0 THEN generate_series(1, z) ELSE 5 END FROM tab;
```

は

```
CREATE FUNCTION case_generate_series(cond bool, start int, fin int, els int)
  RETURNS SETOF int AS $$
BEGIN
  IF cond THEN
    RETURN QUERY SELECT generate_series(start, fin);
  ELSE
    RETURN QUERY SELECT els;
  END IF;
END$$ LANGUAGE plpgsql;

SELECT x, case_generate_series(y > 0, 1, z, 5) FROM tab;
```

になります。この定式化はPostgreSQLのバージョンすべてで同じように動作します。

37.5.9. TABLEを返すSQL関数

集合を返すものとして関数を宣言するには、他にも方法があります。RETURNS TABLE(columns)構文を使用することです。これは1つ以上のOUTパラメータを使い、さらに、関数をSETOF record(または、適切ならば単一の出力パラメータの型のSETOF)を返すものと印を付けることと等価です。この記法は標準SQLの最近の版で規定されたものですので、SETOFを使用するより移植性がより高いかもしれません。

例えば前述の合計と積の例はこのように書けます。

```
CREATE FUNCTION sum_n_product_with_tab (x int)
RETURNS TABLE(sum int, product int) AS $$
    SELECT $1 + tab.y, $1 * tab.y FROM tab;
$$ LANGUAGE SQL;
```

RETURNS TABLE記法と一緒に、明示的OUTまたはINOUTパラメータを使用することはできません。すべての出力列をTABLEリストに含めなければなりません。

37.5.10. 多様SQL関数

SQL関数は、37.2.5の多様型を受け付け、返すように宣言することができます。以下のmake_array多様関数は、任意の2つのデータ型要素から配列を作成します。

```
CREATE FUNCTION make_array(anyelement, anyelement) RETURNS anyarray AS $$
    SELECT ARRAY[$1, $2];
$$ LANGUAGE SQL;

SELECT make_array(1, 2) AS intarray, make_array('a'::text, 'b') AS textarray;
 intarray | textarray
----------+-----------
 {1,2}    | {a,b}
(1 row)
```

'a'::textという型キャストを使用して、引数がtext型であることを指定していることに注目してください。これは引数が単なる文字列リテラルである場合に必要です。さもないと、unknown型として扱われてしまうため、無効なunknownの配列を返そうとしてしまいます。型キャストがないと、以下のようなエラーが発生します。

```
ERROR:  could not determine polymorphic type because input has type unknown
```

上記のようにmake_arrayを宣言した場合、まったく同じデータ型の2つの引数を指定する必要があります。システムは型の違いを解決しようとしません。したがって、例えばこれはうまくいきません。

```
SELECT make_array(1, 2.5) AS numericarray;
ERROR:  function make_array(integer, numeric) does not exist
```

別の方法として、「共通」族の多様型を使用する方法があります。これにより、システムは適切な共通の型を特定できます。

```
CREATE FUNCTION make_array2(anycompatible, anycompatible)
RETURNS anycompatiblearray AS $$
    SELECT ARRAY[$1, $2];
$$ LANGUAGE SQL;

SELECT make_array2(1, 2.5) AS numericarray;
 numericarray
--------------
 {1,2.5}
(1 row)
```

すべての入力が未知の型である場合、共通の型を解決するルールはデフォルトでtext型を選択するので、これも動作します。

```
SELECT make_array2('a', 'b') AS textarray;
 textarray
-----------
 {a,b}
(1 row)
```

固定の戻り値型を持ちながら多様引数を持つことは許されますが、逆は許されません。以下に例を示します。

```
CREATE FUNCTION is_greater(anyelement, anyelement) RETURNS boolean AS $$
    SELECT $1 > $2;
$$ LANGUAGE SQL;

SELECT is_greater(1, 2);
 is_greater
------------
 f
```

```
(1 row)

CREATE FUNCTION invalid_func() RETURNS anyelement AS $$
    SELECT 1;
$$ LANGUAGE SQL;
ERROR:  cannot determine result data type
DETAIL:  A result of type anyelement requires at least one input of type anyelement, anyarray,
 anynonarray, anyenum, or anyrange.
```

出力引数を持つ関数でも多様性を使用することができます。以下に例を示します。

```
CREATE FUNCTION dup (f1 anyelement, OUT f2 anyelement, OUT f3 anyarray)
AS 'select $1, array[$1,$1]' LANGUAGE SQL;

SELECT * FROM dup(22);
 f2 |   f3
----+---------
 22 | {22,22}
(1 row)
```

多様性はvariadic関数とともに使用できます。例をあげます。

```
CREATE FUNCTION anyleast (VARIADIC anyarray) RETURNS anyelement AS $$
    SELECT min($1[i]) FROM generate_subscripts($1, 1) g(i);
$$ LANGUAGE SQL;

SELECT anyleast(10, -1, 5, 4);
 anyleast
----------
       -1
(1 row)

SELECT anyleast('abc'::text, 'def');
 anyleast
----------
 abc
(1 row)

CREATE FUNCTION concat_values(text, VARIADIC anyarray) RETURNS text AS $$
    SELECT array_to_string($2, $1);
$$ LANGUAGE SQL;

SELECT concat_values('|', 1, 4, 2);
 concat_values
---------------
 1|4|2
```

```
(1 row)
```

37.5.11. 照合順序を持つSQL関数

SQL関数が照合順序の変更が可能なデータ型のパラメータを1つ以上持つ場合、23.2で説明されているように、それぞれの関数呼び出しに対して、実引数に割り当てられた照合順序に応じて、照合順序が識別されます。照合順序の識別に成功した(つまり、暗黙的な照合順序がすべての引数で競合しない)場合、すべての照合順序の変更が可能なパラメータは暗黙的に照合順序を持つものとして扱われます。これは関数内の照合順序に依存する操作の振舞いに影響します。例えば、上記のanyleastを使って考えます。

```
SELECT anyleast('abc'::text, 'ABC');
```

この結果はデータベースのデフォルト照合順序に依存します。Cロケールではabcという結果になりますが、他の多くのロケールではabcになります。使用される照合順序をCOLLATE句を付与することで強制することができます。例を以下に示します。

```
SELECT anyleast('abc'::text, 'ABC' COLLATE "C");
```

この他、呼び出し元の照合順序とは関係なく特定の照合順序で動作する関数にしたければ、関数定義において必要な所にCOLLATE句を付けてください。以下のanyleastでは、文字列を比較する際に常にen_USを使用します。

```
CREATE FUNCTION anyleast (VARIADIC anyarray) RETURNS anyelement AS $$
    SELECT min($1[i] COLLATE "en_US") FROM generate_subscripts($1, 1) g(i);
$$ LANGUAGE SQL;
```

しかし、もし照合順序の変更ができないデータ型が与えられた場合にエラーになってしまうことに注意してください。

実引数全体で共通の照合順序を識別できない場合、SQL関数はパラメータがそのデータ型のデフォルト照合順序(通常はデータベースのデフォルトの照合順序ですが、ドメイン型のパラメータでは異なる可能性があります)を持つものとみなします。

照合順序の変更ができるパラメータの動作は、テキストのデータ型にのみ適用できる、限定された多様性と考えることができます。

37.6. 関数のオーバーロード

使用する引数が異なるのであれば、同じSQL名の関数を1つ以上定義することができます。つまり、関数名はオーバーロードが可能です。使うかどうかに関わりなく、この能力は、あるユーザが他のユーザを信用しないデータベースで関数を呼び出す時に、セキュリティの事前の対策を必要とします。10.3を参照してください。問い合わせが実行された時、サーバは与えられた引数のデータ型と数によって呼び出すべき関数を決定します。またオーバーロードを使用して、有限個の可変長引数を持つ関数を模擬することができます。

オーバーロード関数を作成する時、曖昧さが発生しないように注意しなければなりません。例えば、以下のような関数を考えてみます。

```
CREATE FUNCTION test(int, real) RETURNS ...
CREATE FUNCTION test(smallint, double precision) RETURNS ...
```

test(1, 1.5)のような平凡な入力でも、どちらの関数を呼び出すのかはすぐには明確ではありません。現在実装されている解決規則は第10章にて説明していますが、この動作に巧妙に依存するようにシステムを設計することは推奨しません。

一般的に、1つの複合型の引数を取る関数は、その型の属性（フィールド）と同じ名前を持ってはいけません。attribute(table)がtable.attributeと等価とみなされることを思い出してください。複合型に対する関数と複合型の属性との間に曖昧さがあるような場合、属性の方が常に使用されます。この振舞いは関数名をスキーマで修飾する（つまり、schema.func(table)）ことにより変更できますが、競合する名前を使用しないことで問題を防ぐ方が良いでしょう。

可変長引数を取る関数と可変長引数を取らない関数の間に、他にも競合する可能性があります。例えば、foo(numeric)とfoo(VARIADIC numeric[])の両方を作成することが可能です。この場合、単一の数値引数を取った呼び出し、例えばfoo(10.1)をどちらに一致するものとすべきか不明瞭です。検索パスのより前にある関数が使われる、もし2つの関数が同一スキーマにあれば可変長引数を取らない関数が優先されるというのが、この場合の規則です。

C言語関数をオーバーロードする場合、さらに制限があります。オーバーロードされた関数群内の各関数のCの名前は、内部か動的ロードされたかに関係なく他のすべての関数のCの名前と異なる必要があります。この規則に反した場合は、この動作は移植性がありません。実行時リンカエラーになるかもしれませんし、関数群のどれか（たいていは内部関数）が呼び出されるかもしれません。CREATE FUNCTION SQLコマンドの別形式のAS句は、SQL関数名とCソースコード内の関数名とを分離します。以下に例を示します。

```
CREATE FUNCTION test(int) RETURNS int
    AS 'filename', 'test_1arg'
    LANGUAGE C;
CREATE FUNCTION test(int, int) RETURNS int
    AS 'filename', 'test_2arg'
    LANGUAGE C;
```

ここでのC関数の名前は多くの取り得る規約の1つを反映しています。

37.7. 関数の変動性分類

すべての関数は*変動性*区分を持ちます。取り得る区分は、VOLATILE、STABLE、もしくはIMMUTABLEです。CREATE FUNCTIONコマンドで分類の指定がなければデフォルトでVOLATILEになります。変動性に関する分類は、その関数の動作に関するオプティマイザへの約束事です。

- VOLATILE関数は、データベースの変更を含む、すべてを行うことができます。同一引数で続けて呼び出したとしても異なる結果を返すことができます。オプティマイザはこうした関数の振舞いに対する前提を持ちません。VOLATILE関数を使用した問い合わせは、その行の値を必要とするすべての行においてその関数を再評価します。

- STABLE関数はデータベースを変更することができません。また、単一の文内ですべての行に対して同一の引数を渡した場合に同一の結果を返すことが保証されています。この区分により、オプティマイザは複数

の関数の呼び出しを1つの呼び出しに最適化することができます。特に、インデックススキャン条件内でこうした関数を含んだ式を使用することは安全です。(インデックススキャンは行ごとに一度ではなく、一度だけ比較値の評価を行いますので、インデックススキャン条件内でVOLATILE関数を使用することは意味がありません。)

- IMMUTABLE関数はデータベースを変更することができません。また、同一引数に対する呼び出しは常に同一の結果を返すことが保証されています。問い合わせが定数の引数でこうした関数を呼び出した場合、オプティマイザはこの関数を事前に評価することができます。例えば、SELECT ... WHERE x = 2 + 2といった問い合わせは、SELECT ... WHERE x = 4のように単純化することができます。これは、整数加算演算子の基になる関数がIMMUTABLEとして宣言されているためです。

最適化の結果を最善にするためには、関数に対して有効かつ最も厳密な変動性区分を付けなければなりません。

副作用を持つ関数はすべてVOLATILEと付けなければなりません。こうした関数は最適化することができないためです。関数が副作用を持たなかったとしても、単一問い合わせ内で値が変動する場合はVOLATILEと付けなければなりません。例えば、random()、currval()、timeofday()などです。

その他の重要な例は、current_timestamp系列の関数は、それらの値がトランザクション内で変わらないことから、STABLEと見なされます。

計画作成を行い、すぐに実行されるような単一の対話式問い合わせを考えた場合、相対的にSTABLE区分とIMMUTABLE区分との違いはあまりありません。このような場合、関数が計画作成中に一度実行されるか、問い合わせ実行中に一度実行されるかがあまり問題になりません。しかし、計画が保存され、後で再利用される場合は大きな違いが現れます。本来ならば関数が計画作成段階で早めに定数を保持することができない場合にIMMUTABLEを付けると、その後にこの計画を使用する時に古くて意味のない値が再利用されてしまうことになります。これは、プリペアド文や計画をキャッシュする関数言語(PL/pgSQLなど)を使用する場合は危険です。

SQLもしくは標準手続き言語で作成された関数では、変動性分類で決定される2番目に重要な性質があります。すなわち、その関数を呼び出すSQLコマンドによりなされてきたすべてのデータ変更の可視性です。VOLATILE関数はそのような変更を捕らえますが、STABLEまたはIMMUTABLE関数はそうしません。この動作はMVCC(第13章を参照)のスナップショット処理の動作を使用して実装されています。STABLEとIMMUTABLE関数は、呼び出す問い合わせの開始時点で成立したスナップショットを使用しますが、VOLATILE関数はそれぞれの問い合わせの実行開始時点の作りたてのスナップショットを取得します。

> **注記**
>
> しかし、C言語で作成された関数は、どのようにでもスナップショットを管理することができますが、通常C関数でもこのように動作させることは良い考えです。

このスナップショット処理の動作のため、同時実行の問い合わせによって別途変更されている可能性があるテーブルから選択していたとしても、SELECTコマンドのみを含む関数は、安全にSTABLEとすることができます。PostgreSQLは、呼び出し元の問い合わせに対して確立されたスナップショットを使用してSTABLE関数のすべてのコマンドを実行します。したがってその問い合わせの間、データベースに対して固定された視点で値を参照することになります。

IMMUTABLE関数内のSELECTコマンドも同様のスナップショット処理の動作を使用します。ただし、一般的に、IMMUTABLE関数内でデータベースのテーブルを検索(SELECT)することは勧められません。テーブルの内

容が変わってしまった場合にその不変性が壊れてしまうためです。しかし、PostgreSQLでは強制的に検索（SELECT）できないようにはしていません。

よくあるエラーは、設定パラメータに依存する結果となる関数にIMMUTABLEと付けることです。例えば、タイムスタンプを操作する関数は、おそらくTimeZoneの設定に依存した結果になります。こうした関数は、安全のため代わりにSTABLEと付けてください。

> **注記**
>
> PostgreSQLはデータの変更を防ぐためにSTABLE関数とIMMUTABLE関数がSELECT以外のSQLコマンドを含まないことを要求します。（こうした関数はまだデータベースを変更するVOLATILE関数を呼び出すことができますので、これは防弾条件として完全ではありません。これを行うと、STABLEもしくはIMMUTABLE関数は、そのスナップショットからそれらが隠されていることから、呼び出した関数によるデータベースの変更に気がつきません。）

37.8. 手続き型言語関数

PostgreSQLではSQLやC言語以外の言語でユーザ定義の関数を作成することができます。これらの他の言語は一般に*手続き言語*（PL）と呼ばれます。手続き言語はPostgreSQLサーバに組み込まれておらず、ロード可能モジュールとして提供されています。詳細は第41章と以下の章を参照してください。

37.9. 内部関数

内部関数とは、Cで作成された、PostgreSQLサーバに静的にリンクされた関数です。関数定義の「本体」では関数のC言語における名前を指定します。この名前をSQLでの使用のために宣言される名前と同じにする必要はありません。（後方互換性のため、C言語関数名がSQL名と同じであるという意味として、空の本体も受け付けられます。）

通常、サーバに存在するすべての内部関数は、データベースクラスタの初期化（18.2参照）の際に宣言されますが、ユーザはCREATE FUNCTIONを使用して、内部関数の別名をさらに作成することができます。内部関数はinternalという言語名を付けたCREATE FUNCTIONによって宣言されます。例えば、sqrt関数の別名を作成するには以下のようにします。

```
CREATE FUNCTION square_root(double precision) RETURNS double precision
    AS 'dsqrt'
    LANGUAGE internal
    STRICT;
```

（ほとんどの内部関数は「strict」として宣言されることを想定しています。）

> **注記**
>
> 「定義済みの」関数のすべてが上の意味での「内部」ではありません。SQLで作成された定義済み関数もあります。

37.10. C言語関数

ユーザ定義の関数はC（もしくはC++のようなCと互換性のある言語）で作成することができます。そのような関数は動的ロード可能オブジェクト（共有ライブラリとも呼ばれます）としてコンパイルされ、必要に応じてサーバにロードされます。動的ロード機能が、「C言語」関数を「内部」関数と区別するものです。コーディング方法は基本的に両方とも同じです。（したがって、標準内部関数ライブラリはユーザ定義のC関数のコーディング例の豊富な情報源となります。）

現在、1つの呼び出し規約だけがC関数で使用されています(「version 1」)。その呼び出し規約をサポートしていることは、以下に示すように、その関数用に呼び出しマクロPG_FUNCTION_INFO_V1()を書くことで示されます。

37.10.1. 動的ロード

特定のロード可能オブジェクト内のユーザ定義の関数がセッションで最初に呼び出されると、動的ローダは、その関数を呼び出すことができるように、オブジェクトファイルをメモリ内に読み込みます。そのため、ユーザ定義のC関数用のCREATE FUNCTIONはその関数について、ロード可能オブジェクトファイルの名前とオブジェクトファイル中の呼び出される特定の関数のC名称（リンクシンボル）という2つの情報を指定しなければなりません。C名称が明示的に指定されなかった場合、SQLにおける関数名と同じものと仮定されます。

CREATE FUNCTIONコマンドで与えられた名前に基づいて、共有オブジェクトファイルの場所を見つける際に以下のアルゴリズムが使用されます。

1. 名前が絶対パスの場合、指定されたファイルが読み込まれます。
2. 名前が$libdirという文字列から始まる場合、その部分はPostgreSQLパッケージのライブラリディレクトリで置き換えられます。このディレクトリはビルド時に決定されます。
3. 名前にディレクトリ部分がない場合、そのファイルはdynamic_library_path設定変数で指定されたパス内から検索されます。
4. 上記以外の場合（ファイルがパス内に存在しない場合や相対ディレクトリ部分を持つ場合）、動的ローダは指定された名前をそのまま使用し、ほとんどの場合は失敗します。（これは現在の作業ディレクトリに依存するため信頼できません。）

ここまでの流れがうまくいかなかった場合、プラットフォーム独自の共有ライブラリファイル拡張子（多くの場合.so）が指定された名前に追加され、再度この流れを試みます。同様に失敗した場合は、読み込みは失敗します。

共有ライブラリを$libdirから相対的に、もしくは動的ライブラリパスの通った所に配置することを推奨します。異なる場所に新しいインストレーションを配置する場合にバージョンアップを簡単にします。$libdirが示す実際のディレクトリはpg_config --pkglibdirコマンドを使用することでわかります。

PostgreSQLサーバの実効ユーザIDはロード予定のファイルのパスまで到達できなければなりません。よくある失敗として、postgresユーザに対して読み込み、実行、または両方の権限がそのファイルとその上位ディレクトリに与えられていないことがあります。

どの場合でも、CREATE FUNCTIONコマンドに与えたファイル名はそのままシステムカタログに保存されます。ですので、もしそのファイルを再度読み込む必要がある場合、同じ処理が適用されます。

> **注記**
>
> PostgreSQLはC関数を自動的にコンパイルしません。CREATE FUNCTIONコマンドで参照する前に、そのオブジェクトファイルはコンパイルされていなければなりません。さらなる情報については37.10.5を参照してください。

確実に、動的にロードされるモジュールが互換性がないサーバにロードされないように、PostgreSQLは、そのファイルに適切な内容を持つ「マジックブロック」が含まれているかどうか検査します。これによりサーバは、メジャーバージョンが異なるPostgreSQL用にコンパイルされたモジュールなど、明確に互換性がないことを検知することができます。マジックブロックを含めるためには、以下をモジュールのソースファイルに一度（一度だけ）、fmgr.hヘッダファイルをincludeさせた後で、記述してください。

```
PG_MODULE_MAGIC;
```

最初に使用された後も、動的にロードされたオブジェクトファイルはメモリ内に保持されます。同一セッションにおいてそのファイル内の関数をその後に呼び出した場合、シンボルテーブルの検索に要する小さなオーバーヘッドしかかかりません。例えば再コンパイルした後など、そのオブジェクトファイルを強制的に再度読み込ませる必要がある場合は、新しいセッションを開始してください。

省略することもできますが、動的にロードされるファイルに初期化処理関数と最終処理関数を含めることができます。_PG_initという関数がファイルに存在すると、この関数はファイルがロードされた直後に呼び出されます。この関数は引数を取らずvoid型を返さなければなりません。_PG_finiという関数がファイルに存在すると、この関数はファイルがアンロードされる直前に呼び出されます。この関数も同様に引数を取らずvoid型を返さなければなりません。_PG_finiがファイルのアンロード時にのみ呼び出されるものであり、処理の終了時に呼び出されるものではないことに注意してください。（現在、アンロードは無効となっていますので、決して発生しません。将来変更される可能性があります。）

37.10.2. C言語関数における基本型

C言語関数の作成方法を理解するためには、PostgreSQLが基本データ型を内部でどのように表現し、どのようにそれらを関数とやり取りしているかを理解する必要があります。内部的にPostgreSQLは基本型を「メモリの小さな塊」とみなします。ある型を定義するユーザ定義関数は、言い換えると、PostgreSQLがそれを操作できる方法を定義します。つまり、PostgreSQLはデータの格納、ディスクからの取り出しのみを行い、データの入力や処理、出力にはユーザ定義関数を使用します。

基本型は下記の3つのいずれかの内部書式を使用しています。

- 固定長の値渡し
- 固定長の参照渡し
- 可変長の参照渡し

値渡しは、1、2、4バイト長の型のみで使用することができます（使用するマシンのsizeof(Datum)が8の場合は8バイトも使用できます）。データ型を定義する際、その型がすべてのアーキテクチャにおいて同一の大きさ（バイト数）となるように定義するように注意してください。例えば、long型はマシンによっては4バイトで

あったり、8バイトであったりして危険ですが、int型はほとんどのUnixマシンでは4バイトです。Unixマシンにおけるint4の理論的な実装は以下のようになります。

```
/* 4 バイト整数、値渡し */
typedef int int4;
```

（実際のPostgreSQLのCコードではこの型をint32と呼びます。intXXがXXビットであることはCにおける規約だからです。したがってint8というCの型のサイズは1バイトであることに注意してください。int8というSQLの型はCではint64と呼ばれます。表 37.2も参照してください。）

一方、任意の大きさの固定長の型は参照として引き渡すことができます。例として以下にPostgreSQLの型の実装サンプルを示します。

```
/* 16 バイト構造体、参照渡し */
typedef struct
{
    double  x, y;
} Point;
```

それらの型のポインタのみがPostgreSQL関数の入出力時に使用できます。それらの型の値を返すためには、palloc()を使用して正しい大きさのメモリ領域を割り当て、そのメモリ領域に値を入力し、それのポインタを返します。（また、入力引数の1つと同じ型かつ同じ値を返したいのであれば、pallocを行う手間を省くことができます。この場合は入力値へのポインタを単に返してください。）

最後に、すべての可変長型は参照として引き渡す必要があります。また、すべての可変長型は正確に4バイトの不透明なlengthフィールドから始まる必要があります。このフィールドはSET_VARSIZEで設定されます。決して直接このフィールドを設定してはいけません。その型に格納されるすべてのデータはlengthフィールドのすぐ後のメモリ領域に置かれる必要があります。lengthフィールドにはその構造体の総長が格納されます。つまり、lengthフィールドそのものもその大きさに含まれます。

この他の重要な点は、データ型の値の中で初期化されていないビットを残さないことです。例えば、構造体内に存在する可能性がある整列用のパディングバイトを注意してすべてゼロクリアしてください。こうしないと、独自データ型の論理的に等価な定数がプランナにより一致しないものと判断され、（不正確ではありませんが）非効率的な計画をもたらすかもしれません。

> **警告**
>
> 参照渡しの入力値の内容を決して変更しないでください。指定したポインタがディスクバッファを直接指し示している可能性がよくありますので、変更すると、ディスク上のデータを破壊してしまうかもしれません。この規則の唯一の例外について37.12で説明します。

例えば、text型を定義するには、下記のように行えます。

```
typedef struct {
    int32 length;
```

```
    char data[FLEXIBLE_ARRAY_MEMBER];
} text;
```

[FLEXIBLE_ARRAY_MEMBER]表記は、データ部分の実際の長さはこの宣言では指定されないことを意味します。

可変長型を操作する時、正確な大きさのメモリを割り当て、lengthフィールドを正確に設定することに注意する必要があります。例えば、40バイトをtext構造体に保持させたい場合、下記のようなコードを使用します。

```
#include "postgres.h"
...

char buffer[40]; /* 私たちの元のデータ */
...
text *destination = (text *) palloc(VARHDRSZ + 40);
SET_VARSIZE(destination, VARHDRSZ + 40);
memcpy(destination->data, buffer, 40);
...
```

VARHDRSZはsizeof(int32)と同一ですが、可変長型のオーバーヘッド分の大きさを参照する時には、VARHDRSZマクロを使用する方が好ましい形式とみなされています。また長さフィールドを単なる代入ではなくSET_VARSIZEマクロを使用して設定しなければなりません。

表 37.2は、PostgreSQLの組み込み型を使用するC言語関数を作成する時の、Cの型とSQL型との対応を規定したものです。「定義場所」列では、型定義を取り出すためにインクルードしなければならないヘッダファイルを示しています。(実際の定義は一覧中のファイルからインクルードされた、別のファイルであるかもしれません。ユーザは定義されたインタフェースを厳守することを推奨されています。) postgres.hには必ず必要になる多くのものが宣言されていますので、ソースファイルの中で必ず初めにこのファイルをインクルードしなければならないことに注意してください。

表37.2 組み込みSQL型に相当するCの型

SQL型	C 言語型	定義場所
boolean	bool	postgres.h(コンパイラで組み込み済みの可能性があります)
box	BOX*	utils/geo_decls.h
bytea	bytea*	postgres.h
"char"	char	(コンパイラで組み込み済み)
character	BpChar*	postgres.h
cid	CommandId	postgres.h
date	DateADT	utils/date.h
smallint (int2)	int16	postgres.h
int2vector	int2vector*	postgres.h
integer (int4)	int32	postgres.h
real (float4)	float4*	postgres.h

SQL型	C 言語型	定義場所
double precision(float8)	float8*	postgres.h
interval	Interval*	datatype/timestamp.h
lseg	LSEG*	utils/geo_decls.h
name	Name	postgres.h
oid	Oid	postgres.h
oidvector	oidvector*	postgres.h
path	PATH*	utils/geo_decls.h
point	POINT*	utils/geo_decls.h
regproc	regproc	postgres.h
text	text*	postgres.h
tid	ItemPointer	storage/itemptr.h
time	TimeADT	utils/date.h
time with time zone	TimeTzADT	utils/date.h
timestamp	Timestamp	datatype/timestamp.h
varchar	VarChar*	postgres.h
xid	TransactionId	postgres.h

ここまでで基本型に関してあり得る構造体のすべてを記述しましたので、実際の関数の例をいくつか示すことができます。

37.10.3. Version 1 呼び出し規約

Version-1呼び出し規約では、引数と結果の引き渡しの複雑さをなくすためにマクロを使用しています。Version-1関数のC言語宣言は必ず下記のように行います。

```
Datum funcname(PG_FUNCTION_ARGS)
```

さらに、マクロ呼び出し

```
PG_FUNCTION_INFO_V1(funcname);
```

が同じソースファイルに書かれている必要があります。(一般には、関数の直前に書かれます。)PostgreSQLではすべての内部関数はVersion-1であると認識するので、このマクロの呼び出しはinternal言語関数では必要ありません。しかし、動的にロードされる関数では必要です。

Version-1関数では、それぞれの実引数は、引数のデータ型に合ったPG_GETARG_xxx()マクロを使用して取り出されます。(厳格でない関数では、PG_ARGISNULL()を使って引数がNULLかどうか事前に確認することが必要です。下記参照。) 結果は戻り値の型に合ったPG_RETURN_xxx()マクロを使用して返されます。PG_GETARG_xxx()は、その引数として、取り出す関数引数の番号(ゼロから始まります)を取ります。PG_RETURN_xxx()は、その引数として、実際に返す値を取ります。

Version-1呼出し規約を使った例をいくつか以下に示します。

```
#include "postgres.h"
#include <string.h>
#include "fmgr.h"
#include "utils/geo_decls.h"

PG_MODULE_MAGIC;


/* 値渡し */

PG_FUNCTION_INFO_V1(add_one);

Datum
add_one(PG_FUNCTION_ARGS)
{
    int32   arg = PG_GETARG_INT32(0);

    PG_RETURN_INT32(arg + 1);
}


/* 固定長の参照渡し */

PG_FUNCTION_INFO_V1(add_one_float8);

Datum
add_one_float8(PG_FUNCTION_ARGS)
{

    /* FLOAT8用のマクロは参照渡しという性質を隠します */
    float8   arg = PG_GETARG_FLOAT8(0);

    PG_RETURN_FLOAT8(arg + 1.0);
}

PG_FUNCTION_INFO_V1(makepoint);

Datum
makepoint(PG_FUNCTION_ARGS)
{

    /* ここのPoint型の参照渡しという性質は隠されていません */
    Point     *pointx = PG_GETARG_POINT_P(0);
    Point     *pointy = PG_GETARG_POINT_P(1);
```

```
    Point     *new_point = (Point *) palloc(sizeof(Point));

    new_point->x = pointx->x;
    new_point->y = pointy->y;

    PG_RETURN_POINT_P(new_point);
}


/* 可変長の参照渡し */

PG_FUNCTION_INFO_V1(copytext);

Datum
copytext(PG_FUNCTION_ARGS)
{
    text     *t = PG_GETARG_TEXT_PP(0);

    /*

     * VARSIZEは、
      そのヘッダのVARHDRSZまたはVARHDRSZ_SHORTを引いた
     * 構造体の総長をバイト数で表したものです。
     * 完全な長さのヘッダと合わせたコピーを作成します。
     */
    text     *new_t = (text *) palloc(VARSIZE_ANY_EXHDR(t) + VARHDRSZ);
    SET_VARSIZE(new_t, VARSIZE_ANY_EXHDR(t) + VARHDRSZ);

    /*

     * VARDATAは新しい構造体のデータ領域へのポインタです。
     * コピー元はshortデータかもしれませんので、
      VARDATA_ANYでデータを取り出します。
     */

    memcpy((void *) VARDATA(new_t), /* コピー先 */
           (void *) VARDATA_ANY(t), /* コピー元 */
           VARSIZE_ANY_EXHDR(t));   /* バイト数 */
    PG_RETURN_TEXT_P(new_t);
}

PG_FUNCTION_INFO_V1(concat_text);

Datum
concat_text(PG_FUNCTION_ARGS)
{
```

```
    text  *arg1 = PG_GETARG_TEXT_PP(0);
    text  *arg2 = PG_GETARG_TEXT_PP(1);
    int32 arg1_size = VARSIZE_ANY_EXHDR(arg1);
    int32 arg2_size = VARSIZE_ANY_EXHDR(arg2);
    int32 new_text_size = arg1_size + arg2_size + VARHDRSZ;
    text *new_text = (text *) palloc(new_text_size);

    SET_VARSIZE(new_text, new_text_size);
    memcpy(VARDATA(new_text), VARDATA_ANY(arg1), arg1_size);
    memcpy(VARDATA(new_text) + arg1_size, VARDATA_ANY(arg2), arg2_size);
    PG_RETURN_TEXT_P(new_text);
}
```

上のコードがファイルfuncs.cに用意されていて、共有オブジェクトにコンパイルされているとしたら、以下のようにPostgreSQLにコマンドで関数を定義できます。

```
CREATE FUNCTION add_one(integer) RETURNS integer
     AS 'DIRECTORY/funcs', 'add_one'
     LANGUAGE C STRICT;


-- SQL関数名"add_one"のオーバーロードに注意
CREATE FUNCTION add_one(double precision) RETURNS double precision
     AS 'DIRECTORY/funcs', 'add_one_float8'
     LANGUAGE C STRICT;

CREATE FUNCTION makepoint(point, point) RETURNS point
     AS 'DIRECTORY/funcs', 'makepoint'
     LANGUAGE C STRICT;

CREATE FUNCTION copytext(text) RETURNS text
     AS 'DIRECTORY/funcs', 'copytext'
     LANGUAGE C STRICT;

CREATE FUNCTION concat_text(text, text) RETURNS text
     AS 'DIRECTORY/funcs', 'concat_text'
     LANGUAGE C STRICT;
```

ここでは、DIRECTORYは共有ライブラリファイルのディレクトリ(例えばPostgreSQLのチュートリアルのディレクトリ、そこにはこの節で使われている例のコードがあります)を表しています。(DIRECTORYを検索パスに追加した後にAS句で'funcs'だけを使うのがより良いやり方でしょう。どの場合でも、共有ライブラリを表すシステムに特有の拡張子、普通は.soを省略できます。)

関数を「strict」と指定したことに注意してください。これは入力値のいずれかがNULLだった場合、システムが自動的に結果をNULLと決めてしまうことを意味します。こうすることで、関数のコード内でNULLの入力を確認しなければならないことを避けています。これがなければ、PG_ARGISNULL()を使ってNULL値を明示的に確認しなければなりません。

PG_ARGISNULL(n)マクロにより関数は各入力がNULLであるかどうかの検査を行うことができます。(もちろんこれは、「厳密」と宣言されていない関数でのみ必要です。) PG_GETARG_xxx()マクロと同様、入力引数の番号はゼロから始まります。引数がNULLでないことを確認するまでは、PG_GETARG_xxx()の実行は控えなければなりません。結果としてNULLを返す場合は、PG_RETURN_NULL()を実行します。これは、厳密な関数と厳密でない関数の両方で使用可能です。

一見、Version-1のコーディング規約は、普通のCの呼出し規約と比較すると、無意味なあいまいなものの様に見えるかもしれません。しかし、NULLになりうる引数や戻り値、「TOASTされた」(圧縮または行外)値を扱うことができます。

Version 1のインタフェースでは、その他のオプションとしてPG_GETARG_xxx()マクロの変形を2つ提供しています。1つ目のPG_GETARG_xxx_COPY()によって、安全に書き込むことができる指定引数のコピーが確実に返されます。(通常のマクロは、物理的にテーブルに格納されている値へのポインタを返すことがあるので、書き込んではなりません。PG_GETARG_xxx_COPY()マクロの結果は書き込み可能であることが保証されています。) 2つ目の変形は、引数を3つ取るPG_GETARG_xxx_SLICE()マクロからなります。1つ目は関数の引数の番号(上記の通り)です。2つ目と3つ目は、オフセットと返されるセグメントの長さです。オフセットはゼロから始まり、負の長さは残りの値を返すことを要求します。これらのマクロを使用すると、ストレージ種類が「external」(外部)である大きな値の一部へアクセスする際に非常に効果的です。(列のストレージ種類はALTER TABLE tablename ALTER COLUMN colname SET STORAGE storagetypeを使用して指定できます。storagetypeは、plain、external、extended、またはmainのいずれかです。)

最後に、Version-1関数呼び出し規約では、結果集合(37.10.8)を返すこと、およびトリガ関数(第38章)と手続型言語の呼び出しハンドラ(第55章)を実装することができます。詳細についてはソース配布物内のsrc/backend/utils/fmgr/READMEを参照してください。

37.10.4. コードの作成

より先進的な話題に入る前に、PostgreSQL C言語関数のコーディングについての規則をいくつか説明します。C言語以外の言語で記述した関数をPostgreSQLに組み込みむことは可能であるかもしれませんが、例えばC++、FORTRANやPascalといった言語はC言語と同じ呼び出し規約に従いませんので、多くの場合、(可能であったとしても)困難です。それはつまり、他の言語では同じ方法で関数に引数を渡したり、関数から結果を返すことを行わないということです。このため、C言語関数は実際にC言語で書かれているものと仮定します。

C関数の作成と構築の基本規則を以下に示します。

- pg_config --includedir-serverを使用して、使用中のシステム(もしくはユーザが実行するシステム)にてPostgreSQLサーバのヘッダファイルがインストールされた場所を見つけます。
- PostgreSQLに動的にロードできるように独自コードをコンパイル/リンクする時には常に、特別なフラグが必要となります。特定のオペレーティングシステムにおけるコンパイル/リンク方法については37.10.5を参照してください。
- 忘れずに37.10.1で説明した「マジックブロック」を共有ライブラリで定義してください。
- メモリを割り当てる際、Cライブラリのmallocとfreeではなく、PostgreSQLのpallocとpfreeを使用してください。pallocで割り当てられたメモリは各トランザクションの終わりに自動的に解放され、メモリリークを防ぎます。

- memsetを使用して、構造体を必ずゼロクリアしてください（または最初の段階でpalloc0を用いて割り当ててください）。構造体の各フィールドを割り当てたとしても、ゴミの値を持つ整列用のパディング（構造体内の穴）があるかもしれません。こうしないと、ハッシュインデックスやハッシュ結合をサポートすることが困難です。ハッシュを計算するには、データ構造体内の有意なビットのみを取り出す必要があるためです。プランナはまた時折ビット単位の等価性を用いて定数の比較を行います。このため論理的にな値がビット単位で等価でない場合に望まない計画になってしまう可能性があります。

- ほとんどのPostgreSQLの内部型はpostgres.hに宣言されています。一方、関数管理インタフェース（PG_FUNCTION_ARGSなど）はfmgr.hで宣言されています。したがって、少なくともこの2つのファイルをインクルードする必要があります。移植性に関する理由により、postgres.hをその他のシステムヘッダファイル、ユーザヘッダファイルよりも先にインクルードしておくことが最善です。postgres.hをインクルードすることはelog.h、palloc.hもインクルードすることになります。

- オブジェクトファイルで定義されているシンボル名は、互いに、またはPostgreSQLサーバの実行ファイルで定義されているものと異なっている必要があります。これに関するエラーが表示される場合は、関数名または変数名を変更する必要があります。

37.10.5. 動的にロードされる関数のコンパイルとリンク

Cで書かれたPostgreSQLの拡張関数を使うためには、サーバが動的にロードできるように特別な方法でコンパイルとリンクを行う必要があります。正確には共有ライブラリを作る必要があります。

本節の説明以上の詳しい情報はオペレーティングシステムのドキュメント、特にCコンパイラccとリンクエディタldのマニュアルページを参照してください。さらに、PostgreSQLのソースコードのcontribディレクトリにいくつか実例があります。しかし、もしこれらの例に頼るとPostgreSQLソースコードが利用できることに依存したモジュールが作られてしまいます。

共有ライブラリの作成は一般的に実行プログラムのリンクに類似しています。まずソースファイルがオブジェクトファイルにコンパイルされ、そのオブジェクトファイル同士がリンクされます。これらのオブジェクトファイルは位置独立なコード（PIC）として作られる必要があります。 それは概念的には、実行プログラムから呼び出される時にメモリの適当な場所に置くことができるということです （実行プログラム用として作られたオブジェクトファイルはそのようにはコンパイルされません）。共有ライブラリをリンクするコマンドは実行プログラムのリンクと区別するための特別なフラグがあります（少なくとも理論上ではそのようになっています。システムによってはもっと醜い実際が見受けられます)。

次の例ではソースコードはfoo.cファイルにあると仮定し、foo.soという共有ライブラリを作るとします。中間のオブジェクトファイルは特別な記述がない限りfoo.oと呼ばれます。共有ライブラリは1つ以上のオブジェクトファイルを持つことができますが、ここでは1つしか使いません。

FreeBSD

PICを作るためのコンパイラフラグは-fPICです。共有ライブラリを作るコンパイラフラグは-sharedです。

```
gcc -fPIC -c foo.c
gcc -shared -o foo.so foo.o
```

これはFreeBSDのバージョン3.0に適用されます。

HP-UX

PICを作るためのシステムコンパイラのコンパイラフラグは+zです。GCCを使う場合は-fPICです。共有ライブラリのためのリンカフラグは-bです。したがって、以下のようになります。

```
cc +z -c foo.c
```

または

```
gcc -fPIC -c foo.c
```

そして

```
ld -b -o foo.sl foo.o
```

HP-UXは他のほとんどのシステムと異なり共有ライブラリに.slという拡張子を使います。

Linux

PICを作るためのコンパイラフラグは-fPICです。共有ライブラリを作るコンパイラフラグは-sharedです。完全な例は下記のようになります。

```
cc -fPIC -c foo.c
cc -shared -o foo.so foo.o
```

macOS

例を以下に示します。開発者用ツールがインストールされていることが前提です。

```
cc -c foo.c
cc -bundle -flat_namespace -undefined suppress -o foo.so foo.o
```

NetBSD

PICを作るためのコンパイラフラグは-fPICです。ELFシステムでは-sharedコンパイラフラグを使用して共有ライブラリをリンクします。より古い非ELFシステムではld -Bshareableが使われます。

```
gcc -fPIC -c foo.c
gcc -shared -o foo.so foo.o
```

OpenBSD

PICを作成するためのコンパイラフラグは-fPICです。共有ライブラリをリンクするにはld -Bshareableを使用します。

```
gcc -fPIC -c foo.c
ld -Bshareable -o foo.so foo.o
```

Solaris

PICを作るためのコンパイラフラグはSunコンパイラでは-KPICで、GCCでは-fPICです。共有ライブラリをリンクするためには、どちらのコンパイラでもコンパイラオプションは-Gで、GCCの場合、代わりに-sharedオプションを使うこともできます。

```
cc -KPIC -c foo.c
cc -G -o foo.so foo.o
```

もしくは

```
gcc -fPIC -c foo.c
gcc -G -o foo.so foo.o
```

ヒント

これがあまりに難しいようであれば、GNU Libtool[1]の使用を検討すべきです。これはプラットフォームの違いを、統一されたインタフェースで判らないようにします。

これで完成した共有ライブラリファイルはPostgreSQLにロードすることができます。CREATE FUNCTIONコマンドにファイル名を指定する時には、中間オブジェクトファイルではなく共有ライブラリファイルの名前を与えてください。システムの標準共有ライブラリ用の拡張子（通常.soあるいは.sl）はCREATE FUNCTIONで省略することができ、そして移植性を最も高くするため通常は省略されます。

サーバがライブラリファイルをどこに見つけるかに関しては37.10.1を見直してください。

37.10.6. 複合型引数

複合型ではCの構造体のような固定のレイアウトがありません。複合型のインスタンスはNULLフィールドを持つことができます。さらに、複合型で継承階層の一部であるものは、同じ継承階層の他のメンバとは異なるフィールドを持つこともできます。そのため、PostgreSQLはC言語から複合型のフィールドにアクセスするための関数インタフェースを提供します。

以下のような問い合わせに答える関数を書こうとしていると仮定します。

```
SELECT name, c_overpaid(emp, 1500) AS overpaid
    FROM emp
    WHERE name = 'Bill' OR name = 'Sam';
```

Version 1呼び出し規約を使用すると、c_overpaidは以下のように定義できます。

```
#include "postgres.h"

#include "executor/executor.h"  /* GetAttributeByName()用 */

PG_MODULE_MAGIC;
```

[1] https://www.gnu.org/software/libtool/

```
PG_FUNCTION_INFO_V1(c_overpaid);

Datum
c_overpaid(PG_FUNCTION_ARGS)
{
    HeapTupleHeader  t = PG_GETARG_HEAPTUPLEHEADER(0);
    int32            limit = PG_GETARG_INT32(1);
    bool isnull;
    Datum salary;

    salary = GetAttributeByName(t, "salary", &isnull);
    if (isnull)
        PG_RETURN_BOOL(false);

    /* この他、salaryがNULLの場合用にPG_RETURN_NULL()を行った方が良いでしょう */

    PG_RETURN_BOOL(DatumGetInt32(salary) > limit);
}
```

GetAttributeByNameは、指定された行から属性を返す、PostgreSQLシステム関数です。これには3つの引数があります。それらは、関数に渡されたHeapTupleHeader型の引数、求められた属性の名前、属性がNULLであるかどうかを通知する返りパラメータです。GetAttributeByNameは適切なDatumGetXXX()マクロを使用して適切なデータ型に変換可能なDatum型の値を返します。このNULLフラグが設定されている場合、戻り値の意味がないことに注意し、この結果で何かを行おうとする前に常に、NULLフラグを検査してください。

対象列を名前ではなく列番号で選択するGetAttributeByNumもあります。

下記のコマンドでc_overpaid関数をSQLで宣言します。

```
CREATE FUNCTION c_overpaid(emp, integer) RETURNS boolean
    AS 'DIRECTORY/funcs', 'c_overpaid'
    LANGUAGE C STRICT;
```

入力引数がNULLかどうかを検査する必要がないようにSTRICTを使用していることに注意してください。

37.10.7. 行(複合型)を返す

C言語関数から行もしくは複合型の値を返すために、複合型の複雑な作成のほとんどを隠蔽するマクロや関数を提供する、特別なAPIを使用することができます。このAPIを使用するためには、ソースファイルで以下をインクルードする必要があります。

```
#include "funcapi.h"
```

複合型のデータ値(以降「タプル」と記す)を作成する2つの方法があります。Datum値の配列から作成する方法、もしくはタプルのある列の型の入力変換関数に渡すことができるC文字列の配列から作成することで

す。どちらの方法でも、まずタプル構造体用のTupleDesc記述子を入手、あるいは作成しなければなりません。Datumを使用する場合は、TupleDescをBlessTupleDescに渡し、各行に対してheap_form_tupleを呼び出します。C文字列を使用する場合は、TupleDesc をTupleDescGetAttInMetadataに渡し、各行に対してBuildTupleFromCStringsを呼び出します。タプルの集合を返す関数の場合、この設定段階を最初の関数呼び出しで一度にまとめて行うことができます。

必要なTupleDescの設定用の補助用関数がいくつかあります。ほとんどの複合型を返す関数での推奨方法は、以下の関数を呼び出し、呼び出し元の関数自身に渡されるfcinfo構造体と同じものを渡すことです。

```
TypeFuncClass get_call_result_type(FunctionCallInfo fcinfo,
                                   Oid *resultTypeId,
                                   TupleDesc *resultTupleDesc)
```

（これにはもちろん、version 1呼び出し規約を使用していることが必要です。）resultTypeIdをNULLとすることも、ローカル変数のアドレスを指定して関数の戻り値型のOIDを受け取ることができます。resultTupleDescはローカルなTupleDesc変数のアドレスでなければなりません。結果がTYPEFUNC_COMPOSITEかどうかを確認してください。TYPEFUNC_COMPOSITEであった場合、resultTupleDescには必要なTupleDescが格納されています。（TYPEFUNC_COMPOSITEではなかった場合、「レコード型を受け付けない文脈でレコードを返す関数が呼び出されました」というエラーを報告することができます。）

ヒント

get_call_result_typeは、多様性関数の結果の実際の型を解決することができます。ですので、複合型を返す関数だけではなく、スカラの多様結果を返す関数でも有意です。resultTypeId出力は主にスカラの多様結果を返す関数で有意です。

注記

get_call_result_typeは、get_expr_result_typeと似たような関数で、関数呼び出しで想定される出力型を式のツリー構造として解決します。関数自身以外から結果型を決定したい場合に、これを使用することができます。また、get_func_result_typeという関数もあります。これは関数のOIDが利用できる場合にのみ使用することができます。しかし、これらの関数は、record型を返すものと宣言された関数では使用できません。また、get_func_result_typeは多様型を解決することができません。したがって、優先してget_call_result_typeを使用すべきです。

古く、廃止予定のTupleDescを入手するための関数を以下に示します。

```
TupleDesc RelationNameGetTupleDesc(const char *relname)
```

これを指名したリレーションの行型用のTupleDescを取り出すために使用してください。また、

```
TupleDesc TypeGetTupleDesc(Oid typeoid, List *colaliases)
```

これを型のOIDに基づいてTupleDescを取り出すために使用してください。これは、基本型もしくは複合型のTupleDescを取り出すために使用可能です。これはrecordを返す関数ではうまく動作しません。また、多様型を解決することもできません。

TupleDescを獲得した後に、Datumを使用する場合は以下を呼び出してください。

```
TupleDesc BlessTupleDesc(TupleDesc tupdesc)
```

C文字列を使用する場合は以下を呼び出してください。

```
AttInMetadata *TupleDescGetAttInMetadata(TupleDesc tupdesc)
```

集合を返す関数を作成する場合は、これらの関数の結果をFuncCallContext構造体に格納してください。それぞれtuple_descとattinmetaを使用します。

Datumを使用する場合は、ユーザデータをDatum形式に格納したHeapTupleを構築するために以下を使用します。

```
HeapTuple heap_form_tuple(TupleDesc tupdesc, Datum *values, bool *isnull)
```

C文字列を使用する場合は、ユーザデータをC文字列形式に格納したHeapTupleを構築するために以下を使用します。

```
HeapTuple BuildTupleFromCStrings(AttInMetadata *attinmeta, char **values)
```

valuesは行の各属性を1要素としたC文字列の配列です。各C文字列は、属性のデータ型用の入力関数が受け付け可能な形式でなければなりません。属性の値をNULL値として返すためには、values配列の対応するポインタにNULLを設定してください。この関数は返す行それぞれに対して繰り返し呼び出す必要があります。

関数から返すタプルを構築し終わったら、それをDatumに変換しなければなりません。以下を使用して、HeapTupleを有効なDatumに変換してください。

```
HeapTupleGetDatum(HeapTuple tuple)
```

単一行のみを返すのであれば、このDatumを直接返すことができます。さもなくば、集合を返す関数における現在の戻り値として使用することができます。

次節に例を示します。

37.10.8. 集合を返す

C言語関数から集合（複数行）を返すには2つ選択肢があります。一つは、*ValuePerCall*モードと呼ばれる方法で、集合を返す関数が繰り返し呼び出され(毎回同じ引数を渡します)、返す行がなくなるまで呼び出しごとに1つの新しい行を返し、返す行がなくなったらNULLを返します。したがって、集合を返す関数（SRF）は、呼び出し間に十分な状態を保存し何をしていたかを記憶して、呼び出しの度に次の項目を返す必要があります。もう一つは、*Materialize*モードと呼ばれる方法で、集合を返す関数は結果全体を含むタプルストアオブジェクトを埋めて返します。結果全体に対して1つの呼び出しだけが発生し、呼び出し間の状態は必要ありません。

ValuePerCallモードを使用する場合、問い合わせが完全に実行される保証はないことに注意してください。つまり、LIMITなどのオプションがあるため、全ての行をフェッチする前に、エクゼキュータが集合を返す関数の呼び出しを中止することがあります。これは、実行されない可能性があるため、最後の呼び出しでクリー

ンアップ活動を実行するのは安全ではないことを意味します。ファイル記述子などの外部リソースにアクセスする必要がある関数には、Materializeモードを使用することをお勧めします。

本節の残りの部分では、ValuePerCallモードを使用する集合を返す関数で一般に使用される補助マクロのセット（ただし、使用は必須ではありませんが）について説明します。Materializeモードの詳細については、src/backend/utils/fmgr/READMEを参照してください。また、PostgreSQLソース配布物内のcontribモジュールには、ValuePerCallとMaterializeモードの両方を使用する、集合を返す関数のより多くの例があります。

ここで説明するValuePerCallサポートマクロを使用するには、funcapi.hをインクルードします。これらのマクロは、複数の呼び出しにわたって保存する必要がある状態を含むFuncCallContext構造体が備わっています。集合を返す関数内では、fcinfo->flinfo->fn_extraは、呼び出し間でFuncCallContextへのポインタを保持するために使用されます。マクロは、最初の使用時に自動的にそのフィールドを埋め、その後の使用時に同じポインタを見つけることを期待します。

```
typedef struct FuncCallContext
{
    /*

     * 既に行われた呼び出しの回数。
     *

     * SRF_FIRSTCALL_INIT()によってcall_cntrが0に初期化され、

     * SRF_RETURN_NEXT()が呼び出される度に増分されます。
     */
    uint64 call_cntr;

    /*

     * 省略可能 : 呼び出しの最大数
     *
     * max_callsは、
      便宜上用意されているだけで、
      設定は省略可能です。
     * 設定されていなければ、
      関数が終了したことを知るための別の方法を
     * 用意する必要があります。
     */
    uint64 max_calls;

    /*

     * 省略可能 : 様々なユーザによるコンテキスト情報へのポインタ
     *
     * user_fctxは、
      関数の呼び出し間の任意のコンテキスト情報を
```

```
     * 取得するためのユーザ独自の構造へのポインタとして使用されます。
     */
    void *user_fctx;

    /*

     * 省略可能 : 属性型入力メタ情報を含んだ構造体へのポインタ
     *
     * attinmeta はタプル（つまり複合データ型）を返す際に使用され、

     * 基本データ型を返す場合には必要ありません。
     * BuildTupleFromCStrings()を使用して返されるタプルを作成する場合にのみ必要です。
     */
    AttInMetadata *attinmeta;

    /*

     *  複数の呼び出しで必要とされる構造体に使われるメモリコンテキスト
     *
     * multi_call_memory_ctxは、
      SRF_FIRSTCALL_INIT()によってに設定され、

     * SRF_RETURN_DONE()がクリーンアップの際に使用します。
     * これはSRFの複数呼び出しで再利用される全てのメモリ用に最も適切なメモリコンテキストです。
     */
    MemoryContext multi_call_memory_ctx;

    /*

     * 省略可能: タプル説明を含む構造体へのポインタ。
     * tuple_descはタプル（つまり複合データ型）を返す場合に使用され、BuildTupleFromCStrings()
     * ではなくheap_form_tuple()を使用してタプルを作成する場合にのみ必要です。
     * 通常ここに格納されるTupleDescは最初にBlessTupleDesc()を最初に実行したものでなければなり
     * ません。
     */
    TupleDesc tuple_desc;

} FuncCallContext;
```

この基盤を使用して、SRFが使用するマクロは以下の通りです。

```
SRF_IS_FIRSTCALL()
```

これを使用して、関数が初めて呼び出されたのか、2回目以降に呼び出されたのかを判別します。最初の呼び出し(のみ)で、

```
SRF_FIRSTCALL_INIT()
```

を呼び出し、FuncCallContextを初期化します。最初の呼び出しを含むすべての呼び出しで、

```
SRF_PERCALL_SETUP()
```

を呼び出し、FuncCallContextを使用するように設定します。

現在の呼び出しで返すべきデータが関数にある場合は、次を使用します。

```
SRF_RETURN_NEXT(funcctx, result)
```

を使用して、そのデータを呼び出し側に返します。(先に説明した通りresultはDatum型、つまり1つの値またはタプルである必要があります。) 最後に、関数がデータを返し終わったら、

```
SRF_RETURN_DONE(funcctx)
```

を使用してSRFを片付け、終了します。

SRFの呼び出し時に現行になっているメモリコンテキストは一時的なコンテキストで、各呼び出しの間に消去されます。つまりpallocを使用して割り当てたもののすべてをpfreeする必要はありません。これらはいずれ消去されるものだからです。しかし、データ構造体を複数の呼び出しに渡って使用するように割り当てる場合は、どこか別の場所に置いておく必要があります。multi_call_memory_ctxによって参照されるメモリコンテキストは、SRFの実行が終わるまで使用可能にしなければならないデータの保管場所として適しています。つまり、ほとんどの場合、最初の呼び出しのセットアップ中にmulti_call_memory_ctxへ切り替える必要があるということです。funcctx->user_fctxを使用して、このような複数の呼び出しに渡るデータ構造体へのポインタを保持します。(multi_call_memory_ctxに配置したデータは、問い合わせが終了すると自動的に削除されるので、そのデータを手動で解放する必要はありません。)

> **警告**
>
> 関数の実引数は呼出しの間変わらないままですが、一時的なコンテキストで引数の値をTOAST解除した場合には(これは通常、PG_GETARG_xxxマクロにより透過的に行なわれます)、TOAST解除されたコピーが各サイクルで解放されます。従って、user_fctx内のその値への参照を保持する場合には、TOAST解除した後にmulti_call_memory_ctxにそれらをコピーするか、その値をTOAST解除するのはそのコンテキストの中だけであること確実にしなければなりません。

完全な疑似コードの例を示します。

```
Datum
my_set_returning_function(PG_FUNCTION_ARGS)
{
    FuncCallContext  *funcctx;
    Datum             result;
    further declarations as needed
```

```
    if (SRF_IS_FIRSTCALL())
    {
        MemoryContext oldcontext;

        funcctx = SRF_FIRSTCALL_INIT();
        oldcontext = MemoryContextSwitchTo(funcctx->multi_call_memory_ctx);

        /* 一度限りのセットアップコードがここに入ります: */
        user code
        if returning composite
            build TupleDesc, and perhaps AttInMetadata
        endif returning composite
        user code
        MemoryContextSwitchTo(oldcontext);
    }


    /* 毎回実行するセットアップコードがここに入ります: */
    user code
    funcctx = SRF_PERCALL_SETUP();
    user code


    /* これは、
      終了したかどうかをテストする方法の1つです: */
    if (funcctx->call_cntr < funcctx->max_calls)
    {

        /* ここで、
      別の項目を返します: */
        user code
        obtain result Datum
        SRF_RETURN_NEXT(funcctx, result);
    }
    else
    {

        /* これで項目を返し終わりました。 その事実を報告します。 */
        /* （ここにクリーンアップコードを置く誘惑に抵抗してください。） */
        SRF_RETURN_DONE(funcctx);
    }
}
```

複合型を返す単純なSRFの完全な例は以下の通りです。

```
PG_FUNCTION_INFO_V1(retcomposite);
```

```
Datum
retcomposite(PG_FUNCTION_ARGS)
{
    FuncCallContext     *funcctx;
    int                  call_cntr;
    int                  max_calls;
    TupleDesc            tupdesc;
    AttInMetadata       *attinmeta;


     /* 関数の最初の呼び出し時にのみ実行 */
    if (SRF_IS_FIRSTCALL())
    {
        MemoryContext   oldcontext;


        /* 呼び出し間で永続化する関数コンテキストを作成 */
        funcctx = SRF_FIRSTCALL_INIT();


        /* 複数関数呼び出しに適切なメモリコンテキストへの切り替え */
        oldcontext = MemoryContextSwitchTo(funcctx->multi_call_memory_ctx);


        /* 返されるタプルの合計数 */
        funcctx->max_calls = PG_GETARG_UINT32(0);


        /*  結果型用のタプル記述子を作成 */
        if (get_call_result_type(fcinfo, NULL, &tupdesc) != TYPEFUNC_COMPOSITE)
            ereport(ERROR,
                    (errcode(ERRCODE_FEATURE_NOT_SUPPORTED),
                     errmsg("function returning record called in context "
                            "that cannot accept type record")));

        /*

         * 後で未加工のC文字列からタプルを作成するために必要となる
         * 属性メタデータの生成
         */
        attinmeta = TupleDescGetAttInMetadata(tupdesc);
        funcctx->attinmeta = attinmeta;

        MemoryContextSwitchTo(oldcontext);
    }
```

```
/* 全ての関数呼び出しで実行 */
funcctx = SRF_PERCALL_SETUP();

call_cntr = funcctx->call_cntr;
max_calls = funcctx->max_calls;
attinmeta = funcctx->attinmeta;


if (call_cntr < max_calls)    /* 他にも送るものがある場合  */
{
    char       **values;
    HeapTuple    tuple;
    Datum        result;

    /*

     * 返すタプルを構築するためのvalues配列を用意します。
     * これは、
  後で適切な入力関数で処理される
     * C文字列の配列でなければなりません。
     */
    values = (char **) palloc(3 * sizeof(char *));
    values[0] = (char *) palloc(16 * sizeof(char));
    values[1] = (char *) palloc(16 * sizeof(char));
    values[2] = (char *) palloc(16 * sizeof(char));

    snprintf(values[0], 16, "%d", 1 * PG_GETARG_INT32(1));
    snprintf(values[1], 16, "%d", 2 * PG_GETARG_INT32(1));
    snprintf(values[2], 16, "%d", 3 * PG_GETARG_INT32(1));


    /* タプルの作成 */
    tuple = BuildTupleFromCStrings(attinmeta, values);


    /* タプルをdatumに変換 */
    result = HeapTupleGetDatum(tuple);


    /* クリーンアップ（これは必須ではありません） */
    pfree(values[0]);
    pfree(values[1]);
    pfree(values[2]);
    pfree(values);
```

```
        SRF_RETURN_NEXT(funcctx, result);
    }

    else    /* 何も残っていない場合 */
    {
        SRF_RETURN_DONE(funcctx);
    }
}
```

以下にこの関数をSQLで宣言する一例を示します。

```
CREATE TYPE __retcomposite AS (f1 integer, f2 integer, f3 integer);

CREATE OR REPLACE FUNCTION retcomposite(integer, integer)
    RETURNS SETOF __retcomposite
    AS 'filename', 'retcomposite'
    LANGUAGE C IMMUTABLE STRICT;
```

他にも以下のようにOUTパラメータを使用する方法もあります。

```
CREATE OR REPLACE FUNCTION retcomposite(IN integer, IN integer,
    OUT f1 integer, OUT f2 integer, OUT f3 integer)
    RETURNS SETOF record
    AS 'filename', 'retcomposite'
    LANGUAGE C IMMUTABLE STRICT;
```

この方法では、関数の出力型は形式上無名のrecord型になることに注意してください。

37.10.9. 引数と戻り値の多様性

C言語関数は、37.2.5で説明されている多様型を受け付ける、または返すように宣言することができます。多様関数の詳細な説明は37.2.5を参照してください。関数の引数もしくは戻り値が多様型として定義される時、関数の作成者は前もって呼び出しにおけるデータ型や返すべきデータ型が何であるかを知ることはできません。Version-1 C関数で引数の実データ型と、返すべきと想定された型を発見できるための2つのルーチンがfmgr.hに用意されています。このルーチンはget_fn_expr_rettype(FmgrInfo *flinfo)とget_fn_expr_argtype(FmgrInfo *flinfo, int argnum)という名前です。これらは結果もしくは引数型のOIDを返します。ただし、もし情報が利用できなければInvalidOidを返します。flinfo構造体は通常fcinfo->flinfoとしてアクセスされます。argnumパラメータは0から始まります。また、get_fn_expr_rettypeの代わりにget_call_result_typeを使用することもできます。また、variadic変数が配列に吸収されたかどうかを判定するために使用できるget_fn_expr_variadicがあります。そのような吸収はvariadic関数が普通の配列型をとる場合に必ず起こりますので、これは特にVARIADIC "any"の場合に有用です。

例えば、任意の型の単一要素を受け付け、その型の1次元配列を返す関数を考えてみます。

```
PG_FUNCTION_INFO_V1(make_array);
```

```
Datum
make_array(PG_FUNCTION_ARGS)
{
    ArrayType  *result;
    Oid         element_type = get_fn_expr_argtype(fcinfo->flinfo, 0);
    Datum       element;
    bool        isnull;
    int16       typlen;
    bool        typbyval;
    char        typalign;
    int         ndims;
    int         dims[MAXDIM];
    int         lbs[MAXDIM];

    if (!OidIsValid(element_type))
        elog(ERROR, "could not determine data type of input");


    /* 与えられた要素がNULLかどうか注意しつつ、
      要素を取り出します。*/
    isnull = PG_ARGISNULL(0);
    if (isnull)
        element = (Datum) 0;
    else
        element = PG_GETARG_DATUM(0);


    /* 次元数は1 */
    ndims = 1;

    /* 要素を1つ */
    dims[0] = 1;

    /* 下限は1 */
    lbs[0] = 1;


    /* この要素型に関する必要情報を取り出す。 */
    get_typlenbyvalalign(element_type, &typlen, &typbyval, &typalign);


    /* ここで配列を作成 */
    result = construct_md_array(&element, &isnull, ndims, dims, lbs,
                                element_type, typlen, typbyval, typalign);

    PG_RETURN_ARRAYTYPE_P(result);
```

```
}
```

以下のコマンドはSQLでmake_array関数を宣言します。

```
CREATE FUNCTION make_array(anyelement) RETURNS anyarray
    AS 'DIRECTORY/funcs', 'make_array'
    LANGUAGE C IMMUTABLE;
```

C言語関数でのみ使用できる多様性の変異体があります。"any"型のパラメータを取るように宣言できます。(この型名は、SQL予約語でもあるため二重引用符で括らなくてはならないことに注意してください。) これは、他の"any"引数が同じ型になることを強要することも、関数の結果型の決定を支援することもない点を除いて、anyelementのように動作します。C言語関数は最終パラメータがVARIADIC "any"であるように宣言可能です。これは任意の型の1つ以上の実引数と一致します(同じ型である必要はありません)。これらの引数は、通常のvariadic関数で起こったように、配列の中にまとめられません。それらは単に別々に関数に渡されるだけです。PG_NARGS()マクロと上に記載したメソッドは、この機能を使用するときに実際の引数とその型を決定するため使用されなければなりません。また、こうした関数のユーザは、その関数呼び出しにおいて、関数が配列要素を分離した引数として扱うだろうという予想のもとでVARIADICキーワードを良く使用するかもしれません。関数自身は必要ならば、get_fn_expr_variadicを実行した後で、実引数がVARIADIC付きであることを検出した場合に、その動作を実装しなければなりません。

37.10.10. 共有メモリとLWLocks

アドインはLWLocks(軽量ロック)とサーバ起動時に共有メモリの割り当てを保持することができます。shared_preload_librariesで指定して、こうしたアドインの共有ライブラリを事前にロードしなければなりません。共有メモリは、その_PG_init関数で以下を呼び出すことで保持されます。

```
void RequestAddinShmemSpace(int size)
```

LWLocksはその_PG_init関数で以下を呼び出すことで保持されます。

```
void RequestNamedLWLockTranche(const char *tranche_name, int num_lwlocks)
```

num_lwlocks個のLWLockの配列がtranche_nameという名前で確実に利用できるようにします。この配列へのポインタを得るにはGetNamedLWLockTrancheを使ってください。

競合状態の可能性を防止するために、割り当てられた共有メモリへの接続やその初期化時に、以下のように各バックエンドでAddinShmemInitLock軽量ロックを使用しなければなりません。

```
static mystruct *ptr = NULL;

if (!ptr)
{
        bool    found;
```

```
        LWLockAcquire(AddinShmemInitLock, LW_EXCLUSIVE);
        ptr = ShmemInitStruct("my struct name", size, &found);
        if (!found)
        {
                initialize contents of shmem area;
                acquire any requested LWLocks using:
                ptr->locks = GetNamedLWLockTranche("my tranche name");
        }
        LWLockRelease(AddinShmemInitLock);
}
```

37.10.11. 拡張へのC++の利用

以下のガイドラインに従うことで、PostgreSQLの拡張を構築するためC++モードのコンパイラを利用できます。

- バックエンドからアクセスされる関数はすべてバックエンドに対してCインタフェースを提供しなければなりません。このC関数はC++関数を呼びだすことができます。例えば、バックエンドからアクセスされる関数にはextern Cリンクが必要です。これはバックエンドとC++コードの間でポインタとして渡される関数にも必要です。

- 適切な解放メソッドを使ってメモリを解放してください。例えば、ほとんどのバックエンドメモリはpalloc()で確保されますので、pfree()を使って解放してください。この場合にC++のdelete()を使うと失敗するでしょう。

- 例外がCコードへ伝播しないようにしてください(extern C関数すべての最上位ですべての例外を捕捉するブロックを使ってください)。メモリ不足のようなイベントにより例外が発生する可能性がありますので、C++コードが何も例外を発生させない場合であっても、これは必要です。例外はすべて捕捉しなければなりません。そして適切なエラーをCインタフェースに渡してください。可能であれば、例外を完全に除去できるように-fno-exceptionsを付けてC++をコンパイルしてください。その場合、例えばnew()で返されるNULLの検査など、C++コード内で失敗の検査を行わなければなりません。

- C++コードからバックエンド関数を呼び出す場合には、C++呼び出しスタック内にC言語互換構造体(POD)のみが含まれていることを確認してください。バックエンドのエラーは、非PODオブジェクトを持つC++呼び出しスタックを適切に戻すことができない、長距離longjmp()を生成しますので、これは必要です。

まとめると、バックエンドとやりとりするための壁の役割を担うextern C関数の背後にC++コードを配置して、例外、メモリ、呼び出しスタックそれぞれの漏れを避けるのが最善です。

37.11. 関数最適化に関する情報

デフォルトでは、関数は、データベースシステムがその振舞いについてごく一部しか知らない単なる「ブラックボックス」です。しかし、これは、関数を使う問い合わせがその実力よりもずっと効率悪く実行されるかもし

れないことを意味します。プランナが関数呼び出しを最適化するのを助ける補足の情報を提供することが可能です。

基本的な事実のいくつかはCREATE FUNCTIONコマンドで宣言的な注釈として提供されます。この中でも最も重要なものは、関数の変動性分類(IMMUTABLE、STABLEまたはVOLATILE)です。関数を定義する時にはこれを正しく指定するよう常に注意すべきです。並列問い合わせでその関数を使いたいのなら、並列処理での安全性の性質(PARALLEL UNSAFE、PARALLEL RESTRICTEDまたはPARALLEL SAFE)も指定しなければなりません。関数の推定実行コストや集合を返す関数が返すと推定される行数を指定することも有用な場合があります。しかし、この2つの事実を指定する宣言的な方法は定数を指定することしか許しておらず、それは多くの場合不適切です。

SQLで呼び出せる関数(対応する*対象関数*と呼ばれます)に*プランナサポート関数*を結び付け、それによって複雑すぎて宣言的に表現できない対象関数に関する知識を提供することも可能です。(対象関数はそうではありませんが)プランナサポート関数はCで書かなければなりませんので、これは比較的少数の人が使う先進的な機能です。

プランナサポート関数には以下のSQLシグネチャがなければなりません。

```
supportfn(internal) returns internal
```

対象関数を作成する時にSUPPORT句を指定することで対象関数に結び付けられます。

プランナサポート関数のAPIの詳細は、PostgreSQLソースコードのファイルsrc/include/nodes/supportnodes.hで見つけられます。ここではプランナサポート関数ができることの概略を説明するにとどめます。サポート関数へ可能なリクエストの集合は拡張可能ですので、将来のバージョンではより多くのことが可能になっているでしょう。

一部の関数呼び出しでは、関数固有の属性に基づいて計画作成中に単純化できます。例えば、int4mul(n, 1)はnだけに単純化できます。この種の変形は、SupportRequestSimplifyリクエスト型プランナサポート関数に実装することにより実行されます。問い合わせ解析木で見つかった対象関数それぞれに対して、サポート関数が呼び出されます。特定の呼出しが別の形に単純化できることが分かれば、その式を表現する解析木を作成して返します。これは、その関数に基づく演算子に対しても自動的に行なわれます—上の例ではn * 1もnへと単純化されます。(しかし、これは単なる例であることに注意してください。この特定の最適化は、標準のPostgreSQLでは実際には行なわれません。) サポート関数が単純化する状況では、PostgreSQLが対象関数を呼び出すことはないとは保証しません。単純化された式と対象関数の実際の実行が厳密に等しいことを確実にしてください。

booleanを返す対象関数に対しては、その関数を使ったWHERE句により選択される行の割合を推定するのが有用な場合がよくあります。これはSupportRequestSelectivityリクエスト型を実装したサポート関数で行なえます。

対象関数の実行時間が、その入力に大きく依存する場合には、それに対応する定数でないコスト推定を提供するのが有用でしょう。これはSupportRequestCostリクエスト型を実装したサポート関数で行なえます。

集合を返す対象関数に対しては、その関数が返す行の数の定数でない推定を提供するのが有用な場合がよくあります。これはSupportRequestRowsリクエスト型を実装したサポート関数で行なえます。

booleanを返す対象関数に対しては、WHERE句に現れる関数呼び出しをインデックス可能な演算子句に変換できる場合があります。変換された句は、正確にその関数の条件と等しいか幾分弱い(すなわち、関数の条件が受け付けない値も受け付けるかもしれません)でしょう。後者の場合、インデックスの条件は*損失がある*と

言われます。それでもインデックスの走査には使えますが、それが本当にWHERE条件を満たすのかどうか、インデックスにより返された各行に対して関数呼び出しを実行しないといけません。そのような条件を作るには、サポート関数はSupportRequestIndexConditionリクエスト型を実装しなければなりません。

37.12. ユーザ定義の集約

PostgreSQLにおける集約関数は、状態値と状態遷移関数で定義されています。つまり集約は、入力行を順次処理して更新される状態値を使用することで動作します。新しい集約関数を定義するためには、状態値のデータ型、初期状態値、そして状態遷移関数のデータ型を選択します。状態遷移関数は、前の状態値と現在行の集約のための入力値(複数可)を取り、新たな状態値を返します。実行中に保持する状態値と求めている集約の結果のデータが違う場合は、最終関数を指定することもできます。最終関数は、最後の状態値を取り、そして集約の結果として望まれているものを返します。原則として、遷移関数と最終関数は、通常の関数であり集約以外の状況でも使用することができます。(実際には、集約の一部として呼び出されて動作する専用の遷移関数を作成することは、多くの場合パフォーマンス上の理由から役立ちます。)

したがって、集約のユーザに見える引数と結果のデータ型に加え、引数と結果の型のどちらとも違う可能性がある内部状態値のデータ型があります。

最終関数を使わない集約を定義した場合は、列の値を行ごとに計算する関数を実行することで集約ができます。sumはそのような集約の一例です。sumは0から始まり、常に現在の行の値をその時点までの総和に追加します。例えば、もしsum集約を複素数(complex)のデータ型で動作するようにしたければ、そのデータ型の加算関数だけが必要になります。集約の定義は以下のようになります。

```
CREATE AGGREGATE sum (complex)
(
    sfunc = complex_add,
    stype = complex,
    initcond = '(0,0)'
);
```

これは以下のように使用します。

```
SELECT sum(a) FROM test_complex;

   sum
-----------
 (34,53.9)
```

(関数のオーバーロード機能に依存していることに注意してください。sumという名前の集約関数は複数存在しますが、PostgreSQLは列のcomplex型に適用できるsum関数を見つけ出すことができます。)

上記のsumの定義は、もし非NULLの入力値がなければ0(初期状態)を返します。本来はこの場合NULLを返したいのではないかと思いますし、標準SQLではsumがそのように動作することを期待しています。そうするためには、単にinitcond句を省略すれば、初期状態がNULLになります。通常このことは、sfuncがNULL状態の入力をチェックする必要があることを意味します。しかしsumや、その他max、minのような単純な集約にとっては、状態変数に最初の非NULL入力値を挿入し、2番目の非NULL入力値で状態遷移関数の適用を開始

すれば十分です。PostgreSQLは、もし初期状態がNULLで状態遷移関数が「strict(厳密)」と宣言されている場合、自動的にそのように動作します(つまりNULL入力では呼び出されないようになります)。

もう1つの「strict」な状態遷移関数のデフォルト動作としては、NULL入力値が現れると前の状態値が変わらずに維持されるということがあります。したがって、NULL値は無視されます。もしNULL入力に対し他の動作が必要な場合は、状態遷移関数をstrict宣言しないようにします。その代わりにNULL入力の検査をおこなうようにコーディングし、必要なことをすればよいのです。

avg(平均値計算)はもっと複雑な集約の一例です。それには2つの変動する状態が必要になります。入力の合計と入力数のカウントです。最終的な結果はこれらの値を割算することによって得られます。平均値計算は配列を状態遷移値として使う典型的な実装です。例えば、avg(float8)の組み込みの実装は以下のようになっています。

```
CREATE AGGREGATE avg (float8)
(
    sfunc = float8_accum,
    stype = float8[],
    finalfunc = float8_avg,
    initcond = '{0,0,0}'
);
```

注記

float8_accumは、入力の総和と個数だけではなく二乗和も蓄積しますので、2要素ではなく、3要素の配列を必要とします。それは、avg以外の他の集約でも使用できるようにするためです。

SQLの集約関数はオプションによりDISTINCTとORDER BYを許可します。それは集約の遷移関数に渡される行や順序を制御します。これらのオプションは裏側で実装されるので、集約のサポート関数が気にする必要はありません。

さらなる詳細については、CREATE AGGREGATEコマンドを参照してください。

37.12.1. 移動集約モード

集約関数は、移動集約モードをオプションでサポートします。それは、ウィンドウ内のフレーム開始点を移動することで、集約関数の実行を大幅に高速にすることができます。(集約関数としてのウィンドウ関数の使用に関する情報は3.5と 4.2.8 を参照してください。) 基本的な考え方は、通常の「順方向」の遷移関数に加えて、集約は逆方向遷移関数を提供します。これによりウィンドウフレームが終了した時点で、集約の実行中の状態値から行を除外することが可能になります。例えば、sum集約では、順方向遷移関数として加算を使用しており、逆方向遷移関数として減算を使用します。逆方向遷移関数を持たないとウィンドウ関数は、フレームの開始点に移動するたびに一から集約を再計算しなければなりません。その実行時間は、入力行の数のフレーム長の平均回数倍に比例します。逆遷移関数を使用すると実行時間は、入力行の数にのみ比例します。

逆遷移関数には、現在の状態値と現在の状態が含まれる最も古い行の集約入力値(複数可)を渡されます。与えられた入力行が集約されていなかった場合は、それに続く行のみ状態値を再構築する必要があります。これは時々、順方向遷移関数は通常の集約モードよりも必要な状態を持つことが必要になります。そのため、移動集約モードは、通常のモードから完全に分離した実装を使用します。必要に応じて、独自の状態データ

型、独自の順方向遷移関数、及びそれ独自の最終関数を持ちます。これらは必要がない場合、通常モードのデータ型および関数と同じでも構いません。

例として、移動集約モードをサポートするために、以下のようにsum集約を拡張できます。

```
CREATE AGGREGATE sum (complex)
(
    sfunc = complex_add,
    stype = complex,
    initcond = '(0,0)',
    msfunc = complex_add,
    minvfunc = complex_sub,
    mstype = complex,
    minitcond = '(0,0)'
);
```

mで始まる名前のパラメータは、移動集約の実装を定義します。逆遷移関数minvfunc以外はmのない通常の集約パラメータに対応しています。

移動集約モードのための順方向遷移関数は、新しい状態値としてnullを返すことが許されていません。逆遷移関数がnullを返した場合、関数はこの特定の入力に対して状態計算を逆にできないことを示すものと考えます。そのような集約計算は、現在のフレーム開始位置からやり直すことになります。この規則は、実行中の状態値から逆転することが現実的でないような、まれなケースで使用することが出来ます。逆遷移関数はこれらのケースで「諦め」ますが、大部分のケースで動作することが出来ます。例として、浮動小数点数を扱う集約は、NaN(非数)の入力が実行されている状態値から除去されなければならない時に諦めることを選択するかもしれません。

移動集約サポート関数を記述する際には、逆遷移関数が正しい状態値を正確に再構築できていることを確認することが重要です。それ以外の場合は、移動集約モードが使用されているかどうかに応じてユーザに見える結果に違いがあるかもしれません。逆遷移関数を追加する最初の簡単な例は、要件を満たせていないfloat4やfloat8入力のsumです。稚拙なsum(float8)の宣言です。

```
CREATE AGGREGATE unsafe_sum (float8)
(
    stype = float8,
    sfunc = float8pl,
    mstype = float8,
    msfunc = float8pl,
    minvfunc = float8mi
);
```

この集約は、逆遷移関数を持たない場合よりも激しく異なる結果になります。例を考えます。

```
SELECT
  unsafe_sum(x) OVER (ORDER BY n ROWS BETWEEN CURRENT ROW AND 1 FOLLOWING)
FROM (VALUES (1, 1.0e20::float8),
             (2, 1.0::float8)) AS v (n,x);
```

このクエリは 2行目の結果が期待した1ではなく0を返します。原因は、浮動小数点値で制限された精度です：1e20に1を加えても結果は再び1e20になります。その結果から1e20を引くと1ではなく0になります。これは、PostgreSQL限定ではなくて、一般的な浮動小数点演算の制限であることに注意してください。

37.12.2. 多様引数と可変長引数集約

集約関数は多様状態遷移関数や多様最終関数を使用することができます。これにより、同じ関数を使用して複数の集約を実装することができます。37.2.5に多様関数の説明があります。もう少し細かく言うと、集約関数自体が、単一の集約定義で複数の入力データ型を扱うことができるように、多様入力型(複数可)と多様状態型を指定することができるということです。以下に多様型の集約の例を示します。

```
CREATE AGGREGATE array_accum (anyelement)
(
    sfunc = array_append,
    stype = anyarray,
    initcond = '{}'
);
```

ここでは、任意の呼び出しが出来る集約として実際の状態型を（実際の入力型がその要素となる）配列型にしています。この集約の動作は、その配列型に全ての入力を連結することです。（組み込みの集約関数array_aggは、この定義での動作よりもより良い性能で、類似の機能を提供しています。）

以下に2つの異なる実データ型を引数として使用した出力例を示します。

```
SELECT attrelid::regclass, array_accum(attname)
    FROM pg_attribute
    WHERE attnum > 0 AND attrelid = 'pg_tablespace'::regclass
    GROUP BY attrelid;

   attrelid    |              array_accum
---------------+---------------------------------------
 pg_tablespace | {spcname,spcowner,spcacl,spcoptions}
(1 row)

SELECT attrelid::regclass, array_accum(atttypid::regtype)
    FROM pg_attribute
    WHERE attnum > 0 AND attrelid = 'pg_tablespace'::regclass
    GROUP BY attrelid;

   attrelid    |        array_accum
---------------+---------------------------
 pg_tablespace | {name,oid,aclitem[],text[]}
(1 row)
```

通常、上記の例のように多様型の結果を返す集約関数は多様状態型を持ちます。それは、最終関数を適正に宣言するために以下が必要になります。結果の型は多様型であり、引数の型は多様型でない必要がありま

す。そうでないとCREATE FUNCTIONは、呼び出しから結果の型を推定することができないので拒否されます。しかし、状態型として多様型を使用するのは時に不便です。最も一般的なケースでは集約サポート関数は、C言語で状態型をinternal（内部データ）と宣言して書かれる必要があります。なぜなら、SQLには同等のものがないためです。このケースに対処するために、集約の入力引数と一致する追加の「ダミー」引数を取るように最終関数を宣言することが可能です。最終関数が呼び出されたときに特定の値を使用できないため、このようなダミー引数は常にnull値として渡されます。それらは、多様最終関数の結果の型を集約の入力型（複数可）に合わせる場合のみ使用します。例えば以下の定義は、組み込み集約のarray_aggと等価です。

```
CREATE FUNCTION array_agg_transfn(internal, anynonarray)
  RETURNS internal ...;
CREATE FUNCTION array_agg_finalfn(internal, anynonarray)
  RETURNS anyarray ...;

CREATE AGGREGATE array_agg (anynonarray)
(
    sfunc = array_agg_transfn,
    stype = internal,
    finalfunc = array_agg_finalfn,
    finalfunc_extra
);
```

ここで、finalfunc_extraオプションは最終関数が状態値に加えて、集約の入力引数（複数可）に対応する追加のダミー引数（複数可）を受け取れることを指定します。array_agg_finalfnの追加引数anynonarrayにより有効であると宣言をすることができます。

集約関数は、通常の関数の場合とほとんど同じ方法で、最後の引数をVARIADIC配列として宣言することで、可変長の引数を受け入れるようにすることができます。37.5.5を参照してください。集約の遷移関数(複数可)は、それら最後の引数と同じ配列型を持っている必要があります。遷移関数(複数可)は、典型的には、VARIADIC付きになりますが、これは必須ではありません。

注記

可変長集約は、ORDER BYオプション（4.2.7を参照してください）との組み合わせでは、パーサが実引数かどうかを見分けることができないので、簡単に誤用されるようになります。ORDER BYの右側にあるすべてのものは、集約への引数ではなく、ソートキーであることに留意してください。例えば、

```
SELECT myaggregate(a ORDER BY a, b, c) FROM ...
```

パーサには集約関数の引数1つと3つのソートキーと見えます。しかし、これは以下のようにユーザーが意図している可能性があります。

```
SELECT myaggregate(a, b, c ORDER BY a) FROM ...
```

もしmyaggregateが可変長引数の場合、これらの呼び出しが両方とも妥当かもしれません。

同じ理由で、通常の引数の数とは違う同じ名前の集約関数を作成する前に二度考えるのが賢明です。

37.12.3. 順序集合の集約

これまでに記述された集約は「通常の」集約です。PostgreSQLは、順序集合集約もサポートします。それは、通常の集約とは2つの大きな違いがあります。第一に、入力行ごとに評価される通常の集約引数に加えて、順序集合集約は、集約の呼び出しの時に一度だけ評価される「直接」引数を持つことが出来ます。第二に、集約引数の構文は通常、明示的にソート順を指定します。順序集合集約は通常、呼び出すソート順が必要な局面、例えば順位や百分位数(パーセンタイル)のような特定の行の順序に依存して計算する実装のために使用されます。例えば、以下は組み込み関数percentile_discの定義と同じです。

```
CREATE FUNCTION ordered_set_transition(internal, anyelement)
  RETURNS internal ...;
CREATE FUNCTION percentile_disc_final(internal, float8, anyelement)
  RETURNS anyelement ...;

CREATE AGGREGATE percentile_disc (float8 ORDER BY anyelement)
(
    sfunc = ordered_set_transition,
    stype = internal,
    finalfunc = percentile_disc_final,
    finalfunc_extra
);
```

この集約は、float8型の直接引数(百分位数)と、任意のソート可能なデータ型を集約の入力として取ります。それは、以下のように家計所得の中央値を得ることができます。

```
SELECT percentile_disc(0.5) WITHIN GROUP (ORDER BY income) FROM households;
 percentile_disc
-----------------
           50489
```

ここで0.5は直接の引数です。百分位数が行毎に変化する値であったら意味がありません。

通常の集約の場合とは違って、順序集合集約のための入力行のソートは、裏側でおこなわれていません。それは集約のサポート関数の責任です。典型的な実装方法は、集約の状態値に「tuplesort」オブジェクトへの参照を保持し、そのオブジェクトに入ってくる行を供給した後、ソートを完了し、最終関数内でデータを読み出すことです。この設計は、最終関数がソートされるデータに追加の「架空」行を注入するなどの特別な操作を実行するのを可能にします。通常の集約は多くの場合、PL/pgSQLまたは別のPL言語で書かれたサポート関数で実装することができますが、順序集合集約は状態値が任意のSQLデータ型のように定義可能ではないため一般的にC言語で書かれます。(上の例では、状態値が内部型 — として宣言されていることに気づくでしょう。これは典型的なものです。) また、最終関数がソートを実行しますので、遷移関数を後で再び実行し、引き続き入力行を追加することはできません。これは最終関数がREAD_ONLYではないことを意味します。追加の最終関数呼び出しで既にソートされた状態を使えるのなら、CREATE AGGREGATEでREAD_WRITEまたはSHAREABLEと宣言しなければなりません。

順序集合集約のための状態遷移関数は、現在の状態値を加えた行ごとに集約入力値を受信し、更新された状態値を返します。これは通常の集約と同じ定義ですが、(もしあっても)直接の引数が提供さ

れていないことに注意してください。最終関数は、最後の状態値、もしあれば直接の引数の値、および（finalfunc_extraが指定された場合）集約入力（複数）に対応するnull値を受信します。通常の集約と同様に、finalfunc_extraは集約が多様(型)である場合にのみ便利です。そのとき集約の入力型(複数可)が、最終関数の結果の型と合わせるために追加のダミー引数が必要になります。

現在、順序集合集約は、ウィンドウ関数として使用することができないので移動集約モードをサポートする必要はありません。

37.12.4. 部分集約

省略可能ですが、集約関数は部分集約をサポート出来ます。部分集約の考え方は、入力データの異なるサブセットに状態遷移関数を独立して実行し、その後、それらのサブセットから得られた状態値を結合します。こうすることで、単一の操作ですべての入力をスキャンした結果であったのと同じ状態値を生成します。このモードは、別のワーカプロセスをテーブルの異なる部分をスキャンさせることによって並列集約のために使用することが出来ます。それぞれのワーカが、部分状態値を生成し、最後にこれらの状態値を結合して最終状態値を生成します。（将来このモードは、ローカルとリモートのテーブルの集計を結合させるなどの目的のために使用されるかもしれません。それはまだ実装されていません。）

部分集約をサポートするためには、集約定義が結合関数を提供しなければなりません。それは、2つの集約の状態型（入力行の2つのサブセットに対する集約した結果を表わす）の値を取り、状態型の新しい値を生成します。状態は、それらの行の集合の組み合わせを集約した後であろうものを表します。2つのセットからの入力行の相対的な順序であったであろうものが指定されません。これは入力行の順序に敏感な集約のための結合関数を定義することは通常不可能だということを意味します。

簡単な例を示します。MAXとMIN集約は、その遷移関数として使用される「2つの大なり」比較、又は「2つの小なり」比較関数と同じ結合関数を指定することで部分集約をサポートすることが出来ます。SUM集約は結合関数として加算関数が必要になります。（ここでも、入力データ型よりも状態値が広い場合を除き遷移関数と同じです。）

結合関数は、2番目の引数として、基本となる入力型ではなく状態型の値を取りますが、遷移関数のように扱われています。具体的には、null値とstrict関数に対処するためのルールは似ています。また、initcondが非nullである集約定義を指定する場合、各部分集約の実行のための初期状態として使用されるだけでなく、各部分の結果をその状態に結合するために呼び出される結合関数の初期状態としても使用されることに留意してください。

集約の状態型がinternalで宣言されている場合、その結果が集約状態値の正しいメモリコンテキストに割り当てられていることは結合関数の責任です。これは特に、以下のことを意味します。最初の入力がNULLだと、 単純に2番目の入力を返すのは無効です。なぜなら、その値が間違ったコンテキストになり、そして十分な寿命を持っていないことになります。

集約の状態型をinternalで宣言することは、シリアライズ関数とデシリアライズ関数を提供するために通常適切です。これらの関数は、状態値を1つのプロセスから別のプロセスにコピーすることを可能にします。これらの関数がなければ、並列集約を行うことができず、ローカル/リモート集約などの将来のアプリケーションも、おそらく動作しません。

シリアライズ関数は、internalの単一の引数を取り、フラットなblobのバイト状態値パッケージを表わすbytea型を返します。逆にデシリアライズ関数はその変換を逆にします。bytea型とinternal型の2つの引

数を取り、internal型を返します。(第2引数は使用せず常にOですが、型の安全性の理由のために必要とされます。) デシリアライズ関数の結果は単純に、現在のメモリコンテキストに割り当てる必要があります。結合関数の結果とは異なり、長寿命でありません。

集約を並列に実行するために、集約自体にPARALLEL SAFEマークが、されなければならないというのは注目する価値があります。それのサポート関数のパラレルセーフマークは参照されません。

37.12.5. 集約サポート関数

C言語で作成された関数は、AggCheckCallContextを呼び出して、集約サポート関数として呼び出されているかを検出することができます。例えば、

```
if (AggCheckCallContext(fcinfo, NULL))
```

この検査を行う理由の1つは、これが真の場合、先頭の入力は一時的な状態値であるはずなので、新規に割り当ててコピーを持つことなくそのまま変更しても安全だからです。例としてint8inc()を参照してください。(集約遷移関数は常に遷移値をその場で変更できますが、集約最終関数ではそのようなことをするのは一般には勧められません。もし、そうするなら集約を定義する時にその振舞いを宣言しなければなりません。より詳しくはCREATE AGGREGATEを見てください。)

AggCheckCallContextの第2引数は、集約の状態値が保管されているメモリコンテキストを取得するために使用できます。これは状態値として「展開された」オブジェクト(37.13.1を参照)を使用する遷移関数に便利です。最初の呼び出しで、遷移関数はメモリコンテキストが集約状態のコンテキストの子である展開されたオブジェクトを返し、その後の呼び出しで同じ展開されたオブジェクトを保持し続ける必要があります。array_append()の例を参照してください。(array_append()は組み込み集約の遷移関数ではありませんが、カスタム集約の遷移関数で使用すると効率的に動作するように書かれています。)

別のサポートルーチンとしてC言語で書かれたAggGetAggref集約関数が利用可能です。それは、集約の呼び出しを定義するAggrefパースノードを返します。これは主に順序集合集約で有用です。これはソートの順序をどう実現するかAggrefノードの内部構造まで検査することができます。その例は、PostgreSQLソースコード中のorderedsetaggs.cから見つけることができます。

37.13. ユーザ定義の型

37.2に述べられているように、PostgreSQLは、新しい型をサポートするように拡張することができます。本節では、SQL言語以下のレベルで定義されるデータ型である基本型を新しく定義する方法について説明します。新しい基本型の作成には、低レベル言語、通常Cで作成された型を操作する関数の実装が必要です。

本節で使用する例は、ソース配布物内のsrc/tutorialディレクトリにcomplex.sqlとcomplex.cという名前で置いてあります。この例の実行方法についてはディレクトリ内のREADMEを参照してください。

ユーザ定義データ型では必ず入力関数と出力関数が必要です。これらの関数は、その型が(ユーザによる入力とユーザへの出力のための)文字列としてどのように表現されるかと、その型がメモリ中でどう構成されるかを決定します。入力関数は引数としてヌル終端文字列を取り、その型の(メモリ中の)内部表現を返します。出力関数は引数としてその型の内部表現を取り、ヌル終端文字列を返します。単に格納するだけではな

く、その型に操作を加えたいのであれば、その型に持たせたいすべての操作を実装した関数をさらに提供しなければなりません。

例えば、複素数を表現するcomplex型を定義することを考えます。おそらく、次のようなC構造体で複素数をメモリ中で表現することがごく自然な方法です。

```
typedef struct Complex {
    double      x;
    double      y;
} Complex;
```

単一のDatum値で扱うには大き過ぎるので、これは参照渡し型にしなければなりません。

この型の外部文字列表現として(x,y)形式の文字列を使用することを選択します。

入出力関数、特に出力関数を作成することは困難ではありません。しかし、この型の外部表現文字列を定義する時、その表現のための完全で堅牢なパーサを入力関数として作成しなければなりません。以下に例を示します。

```
PG_FUNCTION_INFO_V1(complex_in);

Datum
complex_in(PG_FUNCTION_ARGS)
{
    char       *str = PG_GETARG_CSTRING(0);
    double      x,
                y;
    Complex    *result;

    if (sscanf(str, " ( %lf , %lf )", &x, &y) != 2)
        ereport(ERROR,
                (errcode(ERRCODE_INVALID_TEXT_REPRESENTATION),
                 errmsg("invalid input syntax for type %s: \"%s\"",
                        "complex", str)));

    result = (Complex *) palloc(sizeof(Complex));
    result->x = x;
    result->y = y;
    PG_RETURN_POINTER(result);
}
```

出力関数は以下のように簡単にできます。

```
PG_FUNCTION_INFO_V1(complex_out);

Datum
complex_out(PG_FUNCTION_ARGS)
{
```

```
    Complex    *complex = (Complex *) PG_GETARG_POINTER(0);
    char       *result;

    result = psprintf("(%g,%g)", complex->x, complex->y);
    PG_RETURN_CSTRING(result);
}
```

入出力関数は各々の逆関数になるように注意しなければなりません。そうしないと、データをファイルにダンプし、それを読み戻そうとする際に、深刻な問題が発生するでしょう。これは特に浮動小数点数が関係する際によく発生する問題です。

省略することができますが、ユーザ定義型はバイナリ入出力関数を提供することができます。バイナリ入出力は通常テキスト入出力より高速ですが、テキスト入出力より移植性がありません。テキスト入出力と同様に、外部バイナリ表現を正確に定義することは作成者の責任です。ほとんどの組み込みデータ型は、マシンに依存しないバイナリ表現を提供しようとしています。complex型ではfloat8型のバイナリ入出力コンバータを元にします。

```
PG_FUNCTION_INFO_V1(complex_recv);

Datum
complex_recv(PG_FUNCTION_ARGS)
{
    StringInfo  buf = (StringInfo) PG_GETARG_POINTER(0);
    Complex    *result;

    result = (Complex *) palloc(sizeof(Complex));
    result->x = pq_getmsgfloat8(buf);
    result->y = pq_getmsgfloat8(buf);
    PG_RETURN_POINTER(result);
}

PG_FUNCTION_INFO_V1(complex_send);

Datum
complex_send(PG_FUNCTION_ARGS)
{
    Complex    *complex = (Complex *) PG_GETARG_POINTER(0);
    StringInfoData buf;

    pq_begintypsend(&buf);
    pq_sendfloat8(&buf, complex->x);
    pq_sendfloat8(&buf, complex->y);
    PG_RETURN_BYTEA_P(pq_endtypsend(&buf));
}
```

入出力関数を作成し共有ライブラリ内にコンパイルすれば、SQLでcomplex型を定義することができます。まずシェル型として宣言します。

```
CREATE TYPE complex;
```

これは、入出力関数を定義する時にこの型を参照することができるプレースホルダとして動作します。この後以下のように、入出力関数を定義することができます。

```
CREATE FUNCTION complex_in(cstring)
    RETURNS complex
    AS 'filename'
    LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_out(complex)
    RETURNS cstring
    AS 'filename'
    LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_recv(internal)
   RETURNS complex
   AS 'filename'
   LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_send(complex)
   RETURNS bytea
   AS 'filename'
   LANGUAGE C IMMUTABLE STRICT;
```

最後にデータ型の完全な定義を提供することができます。

```
CREATE TYPE complex (
   internallength = 16,
   input = complex_in,
   output = complex_out,
   receive = complex_recv,
   send = complex_send,
   alignment = double
);
```

新しい基本型を定義すると、PostgreSQLは自動的にその型の配列のサポートを提供します。配列型は通常、基本型の名前の前にアンダースコア文字_が付いた名前になります。

データ型が存在するようになると、そのデータ型に対する有用な操作を提供する関数を宣言することができます。そしてその関数を使用する演算子も定義できます。また、必要に応じて、そのデータ型用のインデックスをサポートするための演算子クラスも作成することができます。こうした追加層については後の節で説明します。

データ型の内部表現が可変長であるなら、内部表現は可変長データの標準配置に従わなければなりません。先頭の4バイトはchar[4]フィールドで、直接アクセスされることは決してありません（慣習的にvl_len_と呼ばれます）。SET_VARSIZE()マクロを使用してデータの総量をこのフィールドに格納し、また、VARSIZE()を

使用して取り出さなければなりません。(長さフィールドはプラットフォームに依存してエンコードされるかもしれませんので、このマクロが存在します。)

詳細についてはCREATE TYPEコマンドの説明を参照してください。

37.13.1. TOASTの考慮

データ型の値により(内部形式で)容量が変動する場合、そのデータ型をTOAST可能とすることが通常は望ましいです(68.2を参照してください)。ヘッダのオーバーヘッドを減らすことでTOASTは小さなデータに対しても容量を抑えることができますので、データが常に圧縮したり外部に格納したりするには小さ過ぎる場合でも、これを行なうことを推奨します。

TOAST格納をサポートするために、そのデータ型を扱うC関数は常に、PG_DETOAST_DATUMを使用して、渡されたTOAST化値を注意深く展開しなければなりません。(通常、こうした詳細は型独自のGETARG_DATATYPE_Pマクロを定義して隠蔽します。) その後、CREATE TYPEコマンドを実行する際に、内部長をvariableと指定し、また、plain以外の適当な格納オプションを選択してください。

データの整列が(単なる特定の関数向けやデータ型が常にバイト単位の整列を規定しているため)重要でない場合、PG_DETOAST_DATUMのオーバヘッドの一部を省くことができます。代わりにPG_DETOAST_DATUM_PACKEDを使用してください(通常はGETARG_DATATYPE_PPマクロを定義することで隠蔽されます)。そして、VARSIZE_ANY_EXHDRおよびVARDATA_ANYマクロを使用して、圧縮されている可能性があるデータにアクセスしてください。繰り返しますが、これらのマクロから返されるデータは、たとえデータ型定義で整列を規定していたとしても、整列されません。整列が重要であれば、通常のPG_DETOAST_DATUMインタフェースを介して実行してください。

> **注記**
>
> 古めのコードではしばしばvl_len_をchar[4]ではなくint32として宣言しています。この構造体定義が少なくともint32で整列されたフィールドを持っている限り、これは問題ありません。しかし、整列されていない可能性があるデータを扱う場合に、こうした構造体定義を使用することは危険です。データが実際に整列されていると仮定することをコンパイラの規則としているかもしれず、この場合、整列に厳密なアーキテクチャではコアダンプしてしまいます。

TOASTのサポートにより有効になるもう一つの機能は以下のような可能性です。ディスクに格納されたフォーマットよりも扱うのにより便利な展開されたインメモリデータ表現を持てるかもしれません。通常のもしくは「単純な」varlena格納フォーマットは結局のところ単なるバイトのblobです。例えば、メモリの別の場所にコピーされるかもしれませんのでポインタを含むことができません。複雑なデータ型に対しては、単純なフォーマットは扱うのにかなり高価になるかもしれません。そこで、PostgreSQLは計算するのにより適した表現に単純なフォーマットを「展開する」方法を提供し、そのフォーマットをインメモリでそのデータ型の関数から関数へと渡します。

展開された格納を使うためには、データ型はsrc/include/utils/expandeddatum.hにある規則に従う展開されたフォーマットを定義し、単純なvarlenaの値を展開されたフォーマットに「展開する」関数や展開されたフォーマットを通常のvarlena表現に「戻す」関数を提供しなければなりません。そのデータ型のC関数はすべてどちらの表現でも確実に受け付けられるようにしてください。おそらく、受け取ったらすぐに一方からもう一方に変換することによって実現することになるでしょう。これはそのデータ型の既存の関数をすべて一度に修正することを要求するものでありません。なぜなら、PG_DETOAST_DATUMマクロが展開された入力を通常の単純なフォーマットに変換するために定義されているからです。そのため、単純なvarlenaフォーマットを扱う既

存の関数は、わずかに非効率ではありますが、展開された入力も続けて扱えるでしょう。より良いパフォーマンスが重要になるまで、変更は必要ありません。

展開された表現の扱い方を知っているC関数は典型的には以下の2つに分類されます。展開されたフォーマットのみを扱えるものと、展開されたものも単純なvarlena入力も扱えるものです。前者は書くのが簡単ですが、全般にあまり効率的ではないかもしれません。なぜなら、一つの関数による使用のために単純な入力を展開された形に変換することは、展開されたフォーマットで操作することで節約されることよりコストが掛かるかもしれないからです。展開されたフォーマットのみ扱うことが必要であるなら、単純な入力の展開された形への変換は引数を取得するマクロの中に隠すことができます。それゆえ、関数は伝統的なvarlena入力を扱うものよりもより複雑に見えることはありません。両方の型の入力を扱うためには、外部やショートヘッダや圧縮されたvarlenaの入力はトースト解除をするけれども展開された入力に対してはトースト解除をしないような、引数を取得する関数を書いてください。そのような関数は、単純なvarlenaフォーマットと展開されたフォーマットの共用体へのポインタを返すよう定義できます。呼び出し側はどちらのフォーマットを受け取ったのか確定するのに`VARATT_IS_EXPANDED_HEADER()`マクロを使えます。

TOAST基盤により、通常のvarlenaの値を展開された値から区別できるようになるだけでなく、展開された値への「読み書き可能」なポインタと「読み取りのみ」のポインタを区別できるようになります。展開された値を検査することが必要なだけのものや安全で意味論的に不可視の方法で変更するC関数は、受け取ったポインタがどちらの種類であるか気にする必要はありません。入力値の修正されたバージョンを生成するC関数は、読み書き可能なポインタを受け取ったのであれば展開された入力値をその場で修正できますが、読み取りのみのポインタを受け取ったのであれば入力を変更してはなりません。その場合には、まず値をコピーして、修正するための新しい値を生成しなければなりません。展開された値を新しく作成したC関数は、必ずそこへの読み書き可能なポインタを返すことを推奨します。また、読み書き可能な展開された値をその場で修正するC関数は、途中で失敗した場合に気をつけて値を健全な状態のままにしておくことを推奨します。

展開された値を扱う例は、標準配列基盤、特に`src/backend/utils/adt/array_expanded.c`を見てください。

37.14. ユーザ定義の演算子

演算子は裏側で実際の作業を行う関数を呼び出す「構文上の飾り」です。ですから、演算子を作成する前にまずこの基礎となる関数を作成する必要があります。しかし、演算子は単なる構文上の飾りではありません。問い合わせプランナによる演算子を使用する問い合わせの最適化を補助する追加情報をやり取りするからです。次節では、この追加情報について重点的に説明します。

PostgreSQLでは左単項演算子、右単項演算子、二項演算子をサポートしています。演算子はオーバーロード可能です。つまり、同じ演算子名をオペランドの数と型が異なる演算子に対して使用することができるということです。問い合わせが実行されると、システムは与えられたオペランドの数と型より呼び出すべき演算子を決定します。

以下に2つの複素数の加算を行う演算子を作成する例を示します。既にcomplex型の定義（37.13を参照）を作成していることを前提としています。まず、実作業を行う関数が必要です。その後、演算子を定義できます。

```
CREATE FUNCTION complex_add(complex, complex)
    RETURNS complex
    AS 'filename', 'complex_add'
```

```
    LANGUAGE C IMMUTABLE STRICT;

CREATE OPERATOR + (
    leftarg = complex,
    rightarg = complex,
    function = complex_add,
    commutator = +
);
```

これで以下のような問い合わせを実行できるようになります。

```
SELECT (a + b) AS c FROM test_complex;

        c
-----------------
 (5.2,6.05)
 (133.42,144.95)
```

ここでは二項演算子をどのように作成するのかを示しました。単項演算子を作成するには、単にleftarg(左単項の場合)もしくはrightarg(右方単項の場合)を省略するだけです。function句と引数用の句のみがCREATE OPERATORでの必須項目です。例で示したcommutator句は省略可能で、問い合わせオプティマイザへのヒントとなります。commutatorやその他のオプティマイザへのヒントについての詳細は次節で説明します。

37.15. 演算子最適化に関する情報

PostgreSQLの演算子定義では、システムに演算子がどう振舞うかに関する有用なことを通知する、いくつかのオプション句を持つことができます。これらの句により演算子を使用する問い合わせの実行速度がかなり向上しますので、これらの句は適切な時には常に提供しなければなりません。しかし、提供する時にはそれらが正しいことを確認しなければいけません! 間違って最適化用の句を使用すると、問い合わせの低速化、わけのわからないおかしな出力、その他有害な事が起こり得ます。最適化用の句についてわからなければ、使用しなくても構いません。使用された時よりも問い合わせの実行が遅くなるかもしれないというだけです。

PostgreSQLの今後のバージョンで、最適化用の句はさらに追加される可能性があります。ここで説明するものはすべて、バージョン13.1で有効なものです。

演算子の基となる関数にプランナサポート関数を結び付けて、システムに演算子の振舞いを通知する別の方法を提供することも可能です。より詳細な情報については37.11を参照してください。

37.15.1. COMMUTATOR

COMMUTATOR句が与えられた場合、それは定義する演算子の交代演算子となる演算子の名前です。取り得る全ての入力値x、yに対して、(x A y)が(y B x)と等しい時、演算子Aは演算子Bの交代演算子であると言います。また、BはAの交代演算子となることにも注意してください。例えば、通常、特定のデータ型用の演算子<と>は互いの交代演算子になります。また、通常、演算子+は自身が交代演算子となります。しかし、通常、演算子-は交代演算子を持ちません。

交代可能な演算子の左オペランドの型は、その交代演算子の右オペランドの型と同一で、その逆もまた同様です。したがって、PostgreSQLで交代演算子を検索する時に必要なものは交代演算子の名前のみになりますので、COMMUTATOR句でそれのみを与えておけば十分です。

インデックスや結合句で使用される演算子では交代演算子の情報を提供することが必須です。これにより、問い合わせオプティマイザがその句を他の種類の実行計画で必要とされる形式に「ひっくり返す」ことができるためです。例えば、tab1.x = tab2.yのようなWHERE句を持った問い合わせを考えてみます。ここでtab1.xとtab2.yはユーザ定義型で、tab2.yにはインデックスが付いていると仮定します。オプティマイザは、この句をtab2.y = tab1.xという形にひっくり返す方法を知らない限り、インデックススキャンを生成できません。インデックススキャン機構は演算子の左側にインデックス付けされた列があることを想定しているためです。PostgreSQLは簡単にこの変形が有効であると前提しません。=演算子の作成者がこれが有効であることを、交換演算子情報を持つ演算子であると印付けて指定しなければなりません。

自己交代演算子を定義する場合は、単にそれを指定するだけです。交代演算子の対を定義する場合は少し複雑になります。最初に他の未定義のものを参照するものをどう定義するのかということが問題となります。この問題には下記の2つの解決方法があります。

- 1つ目の方法は、最初の演算子を定義する際にCOMMUTATOR句を省略し、2番目の演算子の定義では、COMMUTATOR句に最初の演算子を与えるという方法です。PostgreSQLは交代演算子が対になっていることがわかっていますので、2番目の定義を見た時に、自動的に最初の定義に戻ってその未定義になっているCOMMUTATOR句を設定します。

- もう1つの方法は、両方の定義にCOMMUTATOR句を含めるというもっと素直な方法です。PostgreSQLは最初の定義を処理する際に、COMMUTATORが存在しない演算子を参照していることがわかると、システムはその演算子用の仮のエントリをシステムカタログに作成します。この仮エントリには、PostgreSQLがこの時点で推定できる、演算子名、左オペランドの型、右オペランドの型、および結果の型についてのみの有効なデータが入ります。最初の演算子のカタログエントリはこの仮エントリに結び付きます。この後、2番目の演算子が定義されたら、システムはその仮エントリに2番目の定義から得られる追加情報を更新します。更新される前に仮の演算子を使用すると、エラーメッセージが出力されます。

37.15.2. NEGATOR

NEGATOR句が与えられた場合、それは定義する演算子の否定子となる演算子の名前です。入力値xとyの取り得るすべての値に対して両方の演算子が論理値を返し、(x A y)がNOT (x B y)と等しい場合、演算子Aは演算子Bの否定子であると言います。また、BはAの否定子でもあることに注意してください。例えば、ほとんどのデータ型では<と>=は否定子の対となります。演算子が自身の否定子になることは決してありません。

交代演算子と異なり、単項演算子の対は互いに否定子として有効に指定されます。つまりすべてのxに対して(A x)がNOT (B x)と等しいことを意味します。右単項演算子でも同様です。

ある演算子の否定子は、その演算子定義の左オペランド、右オペランドと同じ型を持たなければなりません。ですので、COMMUTATOR句と同様に演算子の名前のみNEGATOR句で与えるだけで済みます。

NOT (x = y)という式をx <> yという形に単純化させることが可能なので、否定子があると問い合わせオプティマイザにとって非常に役に立ちます。他の再配置の結果としてNOT操作が挿入されることがありますので、この現象は思ったより頻繁に起こります。

否定子の対は、上記の交代演算子のペアで説明した方法と同じ方法で定義することができます。

37.15.3. RESTRICT

RESTRICT句が与えられた場合、それは、その演算子用の制限選択評価関数を指定します。(演算子名ではなく関数名であることに注意してください。) RESTRICT句はboolean型を返す二項演算子に対してのみ有効です。制限選択評価の目的は、現在の演算子と特定の定数値についてのWHERE句

```
column OP constant
```

の条件を満たすテーブル内の行の割合を推測することです。この形式を持ったWHERE句によって、どのくらいの行が除外されるのかを通知することで、オプティマイザの手助けをします。(定数値が左項にあったら何が起こるかという疑問が生じるかもしれませんが、それはCOMMUTATORが存在する理由の1つでもあります。)

新しい制限選択評価関数の記述方法は本章の内容を超えていますが、幸いなことに、数多いユーザ定義の演算子に対し通常いくつかのシステム標準の評価関数を使用すれば事足ります。システム標準の制限評価関数には下記のものがあります。

=用のeqsel
<>用のneqsel
<用のscalarltsel
<=用のscalarlesel
>用のscalargtsel
>=用のscalargesel

非常に高いもしくは低い選択性を所有する演算子が、まったく等しいか等しくないかにかかわらず、eqselまたはneqselを使用しないことも往々にして可能です。例えば、近似等号用の幾何演算子はテーブルのエントリの小部分にのみに合致すると仮定してeqselを使用します。

範囲比較のために数値スカラに変換することに多少の有意性があるデータ型を比較するために、scalarltsel、scalarlesel、scalargtsel、scalargeselを使用することも可能です。できればsrc/backend/utils/adt/selfuncs.cのconvert_to_scalar()のルーチンで理解できるところにデータ型を追加してください (今後、このルーチンはpg_typeシステムカタログの列で識別された、データ型ごとの関数で置き換えられなければなりませんが、まだ行われていません)。これを行わなくても動きますが、オプティマイザは本来の推測機能を十分発揮することができません。

もう1つの便利な組み込みの選択評価関数はmatchingselで、入力データ型に対して標準的な最頻値やヒストグラム統計を収集する場合、ほぼすべての二項演算子に対して動作します。デフォルトの評価はeqselで設定されているデフォルトの評価の2倍に設定されており、等価性ほど厳密でないような比較演算子には最適です。(裏で実行されるgeneric_restriction_selectivity関数に異なるデフォルトの評価を与えて呼び出すこともできます。)

さらにsrc/backend/utils/adt/geo_selfuncs.cには、幾何演算子に対する選択評価関数areasel、positionsel、contselがあります。本章の執筆時点では、これらは単なるスタブですが、ともかく使いたい(あるいは改良したい)こともあるでしょう。

37.15.4. JOIN

JOIN句が与えられた場合、それはその演算子用の結合選択評価関数の名前を指定します。(これが演算子名ではなく関数名であることに注意してください。) JOIN句はboolean型を返す二項演算子に対してのみ有効です。結合選択評価の目的は、現在の演算子について、WHERE句

```
table1.column1 OP table2.column2
```

を満たすテーブルの組み合わせの行の割合を推測することです。RESTRICT句の使用と同様、これはいくつかの取り得る結合手順のうち、どれが最も仕事量が少ないように考えられるのかをオプティマイザに計算させることで、大きなオプティマイザへの援助となります。

以前と同様、本章でも結合選択評価関数の作成方法は説明しません。しかし適用できるものがあれば、単に標準の評価関数を使用することをお勧めします。

=用のeqjoinsel
<>用のneqjoinsel
<用のscalarltjoinsel
<=用のscalarlejoinsel
>用のscalargtjoinsel
>=用のscalargejoinsel
汎用の一致演算子用のmatchingjoinsel
2次元面積を基にした比較用のareajoinsel
2次元位置を基にした比較用のpositionjoinsel
2次元包含関係を基にした比較用のcontjoinsel

37.15.5. HASHES

HASHES句が存在する場合、それはシステムに対して、この演算子に基づいた結合にハッシュ結合方法を使っても問題がないことを伝えます。HASHES句はboolean型を返す二項演算子にのみ有効です。実際には、この演算子はあるデータ型またはデータ型の組み合わせの等価性を表現しなければなりません。

ハッシュ結合の基礎となっている仮定は、結合演算子は左項と右項の値が同じハッシュコードを持つ時にのみ真を返すことができるということです。2つの値が異なるハッシュのバケットに置かれた場合、結合演算子の結果が必ず偽であるという仮定を、結合は暗黙的に行い、それらを比べることをしません。したがって、何らかの等価性を表さない演算子にHASHES句を指定することはまったく意味がありません。ほとんどの場合、両辺に同一のデータ型をとる演算子に対してハッシュ機能をサポートすることが現実的です。しかし時として、2つ以上のデータ型に対して互換的なハッシュ関数、つまり、値自体が異なる表現形態を持っていたとしても「等しい」値に対して同一のハッシュコードを生成する関数を設計することもできます。例えば、サイズが異なる整数に対するハッシュでは、この性質を調整することで大変単純になります。

HASHES印を付けるためには、結合演算子はハッシュインデックスの演算子族内になければなりません。演算子を作成する時には参照する演算子族がまだ存在しませんので、演算子の作成時にこれは強制されていません。しかし、演算子族が存在しない場合に、この演算子をハッシュ結合で使用しようとすると、実行時に失敗します。システムは、演算子の入力データ型用のデータ型特有のハッシュ関数を検索するために、演算子族を必要とします。もちろん、演算子族を作成する前に適切なハッシュ関数を作成しなければなりません。

ハッシュ関数を準備する時には注意が必要です。マシンに依存することから、ハッシュ結合が適切な処理を行わずに失敗することがあるからです。例えば、データ型が不要な部分を埋めるビットを持つ可能性がある構造体である場合、(推奨する戦略である、他の演算子と関数を作成して、不要なビットが常にゼロになることを保証しない限り、)その構造体全体を単にhash_anyに渡すことはできません。この他の例として、IEEE浮動小数点標準を満たすマシンでは、マイナス0とプラス0は異なる値(異なるビット列)になりますが、この比

較は等価と定義されます。浮動小数点数値がマイナス0を持つ可能性があるのであれば、それがプラス0と同じハッシュコードを確実に生成するような処置が必要です。

ハッシュ結合可能な演算子は、同一演算子族内に存在する交代演算子を持たなければなりません。(2つの入力データ型が同じ場合はその演算子自体が交代演算子となります。異なる場合は関連する等価性演算子となります。) これを満たさないと、演算子の使用時にプランナエラーが発生します。また、複数のデータ型をサポートするハッシュ演算子族に対して、データ型の組み合わせすべてに対する等価性演算子を持たせることを推奨します(必要ではありません)。これにより、より優れた最適化が可能になります。

> **注記**
>
> ハッシュ結合可能演算子の基となる関数はimmutableもしくはstableでなければなりません。volatileの場合、システムはその演算子を決してハッシュ結合に使用しません。

> **注記**
>
> ハッシュ結合可能演算子の基となる関数が厳密(strict)な場合、その関数は完全、つまり2つの非NULL入力に対して、真または偽を返し、決してNULLを返さないものである必要があります。この規則に従わないと、IN操作におけるハッシュ最適化は間違った結果を生成する可能性があります。(特に、標準に従うとNULLが正しい答えになるところでINは偽を返すかもしれません。もしくは、NULLという結果に対する準備をしていないといったエラーを生成するかもしれません。)

37.15.6. MERGES

MERGES句が存在する場合、それはシステムに対して、この演算子に基づいた結合にマージ結合方法を使っても問題がないことを伝えます。MERGES句はboolean型を返す二項演算子にのみ有効です。実際には、演算子がデータ型またはデータ型の組み合わせの等価性を表すものであることが必要です。

マージ結合は、左側のテーブル、右側のテーブルを順序よくソートし、並列にスキャンするという考えに基づいています。したがって、両データ型には完全な順序付け機能が必要であり、結合演算子はソート順で「同じ場所」にある値の対のみを成功したものとするものである必要があります。実際問題として、これは、結合演算子は等価性のような振舞いをしなければならないことを意味しています。しかし、マージ結合は論理的な互換性を持つ別の2つのデータ型を取ることができます。例えば、smallint対integerの等価性演算子はマージ結合が可能です。両方のデータ型を論理的な互換性を保つ順番にソートする演算子のみが必要です。

MERGES印を付けるためには、結合演算子は、btreeインデックス演算子族の等価性メンバとして存在しなければなりません。演算子を作成する時には参照する演算子族がまだ存在しませんので、演算子の作成時にこれは強制されていません。しかし、対応する演算子族が存在しない限り、実際にマージ結合に使用されることはありません。このように、MERGESフラグは、プランナが対応する演算子族を検索すべきかどうかを決定する際のヒントとして動作します。

マージ結合可能な演算子は、同一演算子族内に存在する交代演算子を持たなければなりません。(2つの入力データ型が同じ場合はその演算子自体が交代演算子となります。異なる場合は関連する等価性演算子となります。) これを満たさないと、演算子の使用時にプランナエラーが発生します。また、複数のデータ型を

サポートするbtree演算子族に対して、データ型の組み合わせすべてに対する等価性演算子を持たせることを推奨します（必要ではありません）。これにより、より優れた最適化が可能になります。

> **注記**
>
> マージ結合可能演算子の背後にある関数はimmutableもしくはstableでなければなりません。volatileの場合、システムはその演算子を決してマージ結合に使用しようとはしません。

37.16. インデックス拡張機能へのインタフェース

これまでのところでは、新しい型や新しい関数、および新しい演算子をどの様に定義するかについて説明してきました。しかしながら、新しい型の列に対するインデックスをまだ作成することができません。このためには、新しいデータ型に対する*演算子クラス*を定義する必要があります。本節では、複素数を値の絶対値の昇順にソートし格納するB-treeインデックスメソッドを使った新しい演算子クラスについての実行例を用いて、演算子クラスの概念を説明します。

演算子クラスを*演算子族*にまとめ、意味的に互換性を持つクラス間の関係を表すことができます。1つのデータ型のみが含まれる場合、演算子クラスで十分です。そこでまずこうした状況に注目し、その後で演算子族に戻ります。

37.16.1. インデックスメソッドと演算子クラス

pg_amテーブルには各インデックスメソッド（内部ではアクセスメソッドとして知られています）に対して1つの行が含まれています。テーブルへの通常のアクセスのサポートはPostgreSQLに組み込まれていますが、すべてのインデックスメソッドは、pg_amで記述されています。必要なコードを書いた後、pg_amにエントリを作成することによって、新しいインデックスアクセスメソッドを追加することができます。しかし、この方法についての説明は本章での範囲を超えています（第61章を参照してください）。

インデックスメソッドのルーチンには、直接的にインデックスメソッドが演算するデータ型の情報は何も与えられていません。代わりに、*演算子クラス*が、特定のデータ型の操作においてインデックスメソッドを使用する必要がある演算の集合を識別します。演算子クラスという名前の由来は、それらが指定するものの1つにインデックスで使用できる（つまり、インデックススキャン条件に変換できる）WHERE句演算子の集合があるからです。また、演算子クラスは、インデックスメソッドの内部演算で必要な、しかしインデックスで使用できるWHERE句演算子には直接的には対応しない、*サポート関数*をいくつか指定することができます。

同じ入力データ型およびインデックスメソッドに対して複数の演算子クラスを定義することが可能です。これにより、1つのデータ型に対して、複数のインデックス付けセマンティクスの集合を定義することができます。例えば、B-treeインデックスでは、処理するデータ型ごとにソート順を定義する必要があります。複素数データ型では、複素数の絶対値によりデータをソートするB-tree演算子クラスと、実部の数値によりソートするB-tree演算子クラスを持つといった方法は、有用かもしれません。通常は演算子クラスの1つが一般的に最も有用であると判断され、そのデータ型およびインデックスメソッドに対するデフォルトの演算子クラスとして設定されます。

複数の異なるインデックスメソッドに、同一の演算子クラス名を使用することができます（例えば、B-treeとハッシュインデックスメソッドは、両方ともint4_opsという名前の演算子クラスを持つことができます）。ただし、そのような各クラスは独立した実体であり、別々に定義される必要があります。

37.16.2. インデックスメソッドのストラテジ

演算子クラスに関連付けられている演算子は、「ストラテジ番号」により識別されます。「ストラテジ番号」は、演算子クラスのコンテキスト内における各演算子のセマンティクスを識別するためのものです。例えば、B-treeの場合、キーが小さい方から大きい方へ厳密に並んでいなければなりません。したがって、B-treeに関しては、「より小さい」および「以上」のような演算子は興味深いと言えます。PostgreSQLではユーザが演算子を定義できるため、PostgreSQLは演算子の名前（例えば<や>=）を見つけても、その演算子がどのような比較を行うかを判断することはできません。その代わり、インデックスメソッドは「ストラテジ」の集合を定義します。「ストラテジ」は汎用演算子と考えることができます。各演算子クラスは、特定のデータ型およびインデックスセマンティクスの解釈において、実際のどの演算子が各ストラテジに対応しているかを指定します。

表 37.3に示すように、B-treeインデックスメソッドではストラテジを5つ定義します。

表37.3 B-treeストラテジ

演算	ストラテジ番号
小なり	1
以下	2
等しい	3
以上	4
大なり	5

ハッシュインデックスは等価性のみをサポートします。したがって、表 37.4に示すように、ストラテジを1つのみ定義します。

表37.4 ハッシュストラテジ

演算	ストラテジ番号
等しい	1

GiSTインデックスはより柔軟です。固定のストラテジの集合をまったく持ちません。代わりに、特定のGiST演算子クラスの「consistent」サポートルーチンが、ストラテジ番号が何を意味するかを解釈します。例として、2次元幾何オブジェクトをインデックス付けし、「R-tree」ストラテジを提供する組み込みのGiSTインデックス演算子クラスのいくつかを表 37.5に示します。この内4個は2次元に対する（重複、合同、包含、被包含）試験です。残りの内4個はX方向のみに対する、残り4個はY方向のみに対する同一の試験を提供します。

表37.5 GiSTによる2次元の「R-tree」ストラテジ

演算	ストラテジ番号
完全に左側	1
右側にはみ出さない	2
重なる	3
左側にはみ出さない	4
完全に右側	5
同じ	6

演算	ストラテジ番号
含む	7
含まれる	8
上側にはみ出さない	9
完全に下側	10
完全に上側	11
下側にはみ出さない	12

SP-GiSTインデックスは柔軟性という点でGiSTと似ており、固定のストラテジ群を持ちません。その代わりに、各演算子クラスのサポートルーチンが演算子クラスの定義に従ってストラテジ番号を解釈します。例として、点に対する組み込みの演算子クラスで使用されるストラテジ番号を表 37.6に示します。

表37.6 SP-GiSTの点に関するストラテジ

演算	ストラテジ番号
厳密に左側	1
厳密に右側	5
同一	6
包含される	8
厳密に下	10
厳密に上	11

GINインデックスは、いずれも固定のストラテジ群を持たないという点で、GiSTおよびSP-GiSTインデックスと似ています。その代わりに、各演算子クラスのサポートルーチンが演算子クラスの定義に従ってストラテジ番号を解釈します。例として、配列に対する組み込みの演算子クラスで使用されるストラテジ番号を表 37.7に示します。

表37.7 GIN 配列のストラテジ

演算	ストラテジ番号
重複	1
包含	2
包含される	3
等しい	4

BRINインデックスは、いずれも固定のストラテジ群を持たないという点で、GiST、SP-GiSTおよびGINインデックスと似ています。その代わりに、各演算子クラスのサポートルーチンが演算子クラスの定義に従ってストラテジ番号を解釈します。例として、組み込みのMinmax演算子クラスで使用されるストラテジ番号を表 37.8に示します。

表37.8 BRIN Minmaxストラテジ

演算	ストラテジ番号
小なり	1

演算	ストラテジ番号
以下	2
等しい	3
以上	4
大なり	5

上記の演算子はすべて論理値を返すことに注意してください。実際、インデックスで使用されるためにWHEREの最上位レベルで現れなければなりませんので、インデックスメソッド検索演算子として定義された、すべての演算子の戻り値の型はbooleanでなければなりません。(一部のインデックスアクセスメソッドは、通常論理型の値を返さない順序付け演算子もサポートします。この機能については37.16.7で説明します。)

37.16.3. インデックスメソッドのサポートルーチン

ストラテジは通常、システムがインデックスを使う方法を判断するために十分な情報ではありません。実際には、インデックスメソッドが動作するためには、さらにサポートルーチンを必要とします。例えばB-treeインデックスメソッドは、2つのキーを比較し、より大きいのか、等しいのか、より小さいのかを決定できなければなりません。同様に、ハッシュインデックスは、キー値のハッシュコードを計算できなければなりません。これらの操作はSQLコマンドの条件内で使用される演算子とは対応しません。これらはインデックスメソッドで内部的に使用される管理用ルーチンです。

ストラテジと同じように、演算子クラスにより、与えられたデータ型およびセマンティクス解釈に対して、どの特定の関数がこれらの各役割を果たすべきであるかが識別されます。インデックスメソッドは必要な関数の集合を定義し、演算子クラスは、これらをインデックスメソッドで指定された「サポート関数番号」に代入することによって、使用すべき正しい関数を識別します。

さらに、演算子クラスの中には、ユーザがその振る舞いを制御するパラメータを指定できるものもあります。各組み込みインデックスアクセスメソッドには省略可能なoptionsサポート関数があり、演算子クラスに固有のパラメータの集合を定義しています。

表 37.9に示すように、B-treeは比較サポート関数を必須とし、演算子クラスの作者が望めば4つの追加サポート関数を与えることができます。これらのサポート関数の要件は63.3でさらに詳しく解説されています。

表37.9 B-treeサポート関数

関数	サポート番号
2つのキーを比較し、最初のキーが2番目のキーより小さいか、等しいか、大きいかを示す、0未満、0、もしくは0より大きい整数を返します。	1
C言語から呼び出し可能なソートサポート関数のアドレスを返します(省略可能)。	2
テスト値をベース値にオフセットを加減算したものと比較して、比較結果に従って真または偽を返します(省略可能)。	3
演算子クラスを使うインデックスがB-tree重複排除最適化を安全に適用できるかどうかを決定します(省略可能)。	4
この演算子クラスに固有のオプションの集合を定義します(省略可能)。	5

表 37.10に示すようにハッシュインデックスでは一つのサポート関数が必須で、演算子クラス作者が望むなら、さらに2つのサポート関数を与えることができます。

表37.10 ハッシュサポート関数

関数	サポート番号
キーの32ビットハッシュ値を計算	1
64bitソルトが与えられたキーに対する64ビットハッシュ値を計算します。ソルトが0なら結果の下位32ビットは関数1で計算された値と一致しなければなりません。(省略可能)	2
この演算子クラスに固有のオプションの集合を定義します(省略可能)。	3

表 37.11に示すように、GiSTインデックスには10のサポート関数があり、また、そのうち3つは省略可能です。(詳細については第64章を参照してください。)

表37.11 GiSTサポート関数

関数	説明	サポート番号
consistent	キーが問い合わせ条件を満たすかどうかを決定します。	1
union	キー集合の和集合を計算します。	2
compress	キーまたはインデックス付けされる値の圧縮表現を計算します。	3
decompress	圧縮されたキーを伸張した表現を計算します。	4
penalty	指定された副ツリーキーを持つ副ツリーに新しいキーを挿入する時のペナルティを計算します。	5
picksplit	ページのどのエントリを新しいページに移動させるかを決定し、結果ページ用の統合キーを計算します。	6
equal	2つのキーを比較し、等しければ真を返します。	7
distance	キーと問い合わせ値との間の距離を決定します(省略可能)。	8
fetch	インデックスオンリースキャンのために圧縮されたキーの元の表現を計算します(省略可能)。	9
options	この演算子クラスに固有のオプションの集合を定義します(省略可能)。	10

表 37.12に示すように、SP-GiSTインデックスでは6つのサポート関数があり、また、そのうち1つは省略可能です。(詳細については第65章を参照してください。)

表37.12 SP-GiSTサポート関数

関数	説明	サポート番号
config	演算子クラスに関する基本情報を提供します。	1
choose	新しい値を内部タプルに挿入する方法を決定します。	2
picksplit	値集合を分割する方法を決定します。	3
inner_consistent	ある問い合わせでサブパーティションの検索が必要かどうか決定します。	4

関数	説明	サポート番号
leaf_consistent	キーが問い合わせ修飾子を満たすかどうか決定します。	5
options	この演算子クラスに固有のオプションの集合を定義します(省略可能)。	6

表 37.13に示すように、GINインデックスには、7つのサポート関数があり、また、そのうち4つは省略可能です。(詳細については第66章を参照してください。)

表37.13 GINサポート関数

関数	説明	サポート番号
compare	2つのキーを比較し、0未満、0、0より大きな整数を返します。それぞれ最初のキーの方が大きい、等しい、小さいを示します。	1
extractValue	インデックス付けされる値からキーを抽出します。	2
extractQuery	問い合わせ条件からキーを抽出します。	3
consistent	問い合わせ条件に一致する値かどうかを決定します(2値の亜種)。(サポート関数6があれば、省略可能)	4
comparePartial	問い合わせからの部分キーとインデックスからのキーを比較し、それぞれ、GINがこのインデックス項目を無視しなければならないか、一致する項目として扱わなければならないか、インデックススキャンを中止しなければならないかを示す、ゼロより小さい、ゼロ、ゼロより大きい整数値のいずれかを返します(省略可能)。	5
triConsistent	問い合わせ条件に一致する値かどうかを決定します(3値の亜種)。(サポート関数4があれば、省略可能)	6
options	この演算子クラスに固有のオプションの集合を定義します(省略可能)。	7

表 37.14に示すようにBRINインデックスには、5つの基本サポート関数があり、また、そのうち1つは省略可能です。基本関数の版には追加のサポート関数の提供を要求するものもあります。(詳細については67.3を参照してください。)

表37.14 BRINサポート関数

関数	説明	サポート番号
opcInfo	インデックスが貼られた列の要約データを記述する内部情報を返します	1
add_value	既存のサマリーインデックスタプルに新しい値を足します	2
consistent	値が問い合わせ条件に一致するかどうかを決めます	3
union	2つのサマリータプルの結合を計算します	4
options	この演算子クラスに固有のオプションの集合を定義します(省略可能)。	5

検索演算子と異なり、サポート関数は特定のインデックスメソッドが想定するデータ型、例えばB-tree用の比較関数の場合、符号付き整数を返します。同様に各サポート関数に渡す引数の数と型はインデックスメ

ソッドに依存します。B-treeとハッシュでは、比較関数とハッシュ処理サポート関数はその演算子クラスに含まれる演算子と同じ入力データ型を取りますが、GIN、SP-GiST、GiST、およびBRINサポート関数のほとんどはそうではありません。

37.16.4. 例

ここまでで概念について説明してきました。ここで、新しい演算子クラスを作成する有用な例を紹介します。(この例を作業できるように、ソース配布物内のsrc/tutorial/complex.cとsrc/tutorial/complex.sqlにコピーがあります。) この演算子クラスは、複素数をその絶対値による順番でソートする演算子をカプセル化します。ですので、その名前にcomplex_abs_opsを選びました。最初に演算子の集合が必要になります。演算子を定義する処理は37.14で説明しました。B-tree上の演算子クラスでは、以下の演算子が必要です。

- 絶対値による、小なり(ストラテジ1)
- 絶対値による、以下(ストラテジ2)
- 絶対値による、等しい(ストラテジ3)
- 絶対値による、以上(ストラテジ4)
- 絶対値による、大なり(ストラテジ5)

比較演算子の関連する集合を定義する時にエラーの発生を最小にする方法は、まず、B-tree比較サポート関数を作成し、その後に、他の関数をサポート関数に対する1行のラッパとして作成することです。これにより、境界となる条件で一貫性のない結果を得る確率が減少します。この手法に従って、まず以下を作成します。

```
#define Mag(c)  ((c)->x*(c)->x + (c)->y*(c)->y)

static int
complex_abs_cmp_internal(Complex *a, Complex *b)
{
    double      amag = Mag(a),
                bmag = Mag(b);

    if (amag < bmag)
        return -1;
    if (amag > bmag)
        return 1;
    return 0;
}
```

これで、小なり関数は以下のようになります。

```
PG_FUNCTION_INFO_V1(complex_abs_lt);

Datum
complex_abs_lt(PG_FUNCTION_ARGS)
{
    Complex    *a = (Complex *) PG_GETARG_POINTER(0);
```

```
    Complex    *b = (Complex *) PG_GETARG_POINTER(1);

    PG_RETURN_BOOL(complex_abs_cmp_internal(a, b) < 0);
}
```

他の4関数での違いは、内部関数の結果とゼロとをどのように比べるかだけです。

次に、関数と、この関数に基づく演算子をSQLで宣言します。

```
CREATE FUNCTION complex_abs_lt(complex, complex) RETURNS bool
    AS 'filename', 'complex_abs_lt'
    LANGUAGE C IMMUTABLE STRICT;

CREATE OPERATOR < (
   leftarg = complex, rightarg = complex, procedure = complex_abs_lt,
   commutator = > , negator = >= ,
   restrict = scalarltsel, join = scalarltjoinsel
);
```

正しく交代演算子と否定演算子を指定する他、適切な制限選択性関数と結合関数を指定することが重要です。さもないと、オプティマイザはインデックスを効率的に使用することができません。

他にも注意すべきことがここで発生します。

- 例えば、complex型を両オペランドに取る=という名前の演算子を1つしか作成できません。この場合、complex用の他の=演算子を持てません。しかし、実際にデータ型を作成しているとしたら、おそらく、複素数の(絶対値の等価性ではない)通常の等価性演算を行う=を欲するでしょう。この場合、complex_abs_eq用の演算子名に別の名前を使用しなければなりません。

- PostgreSQLでは異なる引数のデータ型であれば同じSQL名の演算子を使うことができますが、Cでは1つの名前で1つのグローバル関数が使えるだけです。ですから、C関数はabs_eqのような単純な名前にするべきではありません。通常は、他のデータ型の関数と衝突しないように、C関数名にデータ型名を入れておくことを勧めます。

- abs_eq関数のSQL名は、PostgreSQLが引数のデータ型によって同じ名前を持つ他のSQL関数から区別してくれることを期待して作ることができます。ここでは例を簡単にするために、関数にCレベルとSQLレベルで同じ名前を与えています。

次のステップは、B-treeに必要なサポートルーチンの登録です。これを実装するCコードは、演算子関数と同じファイルに入っています。以下は、関数をどのように宣言するかを示します。

```
CREATE FUNCTION complex_abs_cmp(complex, complex)
    RETURNS integer
    AS 'filename'
    LANGUAGE C IMMUTABLE STRICT;
```

これまでで、必要な演算子およびサポートルーチンを持つようになりました。最後に演算子クラスを作成することができます。

```
CREATE OPERATOR CLASS complex_abs_ops
    DEFAULT FOR TYPE complex USING btree AS
        OPERATOR        1       < ,
        OPERATOR        2       <= ,
        OPERATOR        3       = ,
        OPERATOR        4       >= ,
        OPERATOR        5       > ,
        FUNCTION        1       complex_abs_cmp(complex, complex);
```

これで終わりです！これでcomplex列にB-treeインデックスを作って使用することが可能になったはずです。

以下のように、演算子エントリをより冗長に記述することができます。

```
OPERATOR        1       < (complex, complex) ,
```

しかし、演算子が、演算子クラスの定義と同一のデータ型を取る場合、このような記述をする必要はありません。

上記の例は、ユーザがこの新しい演算子クラスをcomplexデータ型のデフォルトのB-tree演算子クラスにしようとしていると仮定しています。このようにしない場合、DEFAULTという単語を取り除いてください。

37.16.5. 演算子クラスと演算子族

これまでは暗黙的に、演算子クラスは1つのデータ型のみを扱うものと仮定してきました。確かに特定のインデックス列にはたった1つのデータ型しかあり得ませんが、異なるデータ型の値とインデックス列の比較を行うインデックス操作はよく役に立ちます。また、演算子クラスと関連したデータ型を跨る演算子を使用できる場合、他のデータ型は独自の関連した演算子クラスを持つことがよくあります。SQL問い合わせを最適化する際にプランナを補助することができますので、関連したクラスを明示的に関連付けることは（どのように動作するかに関する知識をプランナは多く持ちますので、特にB-tree演算子クラスで）有用です。

こうした要望に応えるためにPostgreSQLは*演算子族*という概念を使用します。演算子族は1つ以上の演算子クラスから構成されます。また、演算子族全体に属するが、演算子族内の個々のクラスには属さないインデックス可能演算子や対応するサポート関数を含めることもできます。こうした演算子や関数を、特定のクラスに束縛されていないことから、演算子族内で「自由」であると呼びます。通常、各演算子クラスは1つのデータ型演算子を持ちますが、データ型を跨る演算子は演算子族内で自由になります。

演算子族内の演算子と関数はすべて、意味的な互換性を持たなければなりません。この互換性についての必要条件はインデックスメソッドによって設定されます。このため、演算子族の特定の部分集合を演算子クラスとして選び出す方法に疑問を持つかもしれません。実際多くの目的では、クラスの分類は不適切で、演算子族が唯一の興味深いグループ化です。演算子クラスを定義する理由は、どれだけ多くの演算子族が何らかのインデックスをサポートするために必要かを指定することです。ある演算子クラスを使用するインデックスが存在する場合、演算子クラスはそのインデックスを削除しない限り削除することができません。しかし、演算子族の他の部分、すなわち、他の演算子クラスや自由な演算子を削除することができます。したがって、演算子クラスは、特定のデータ型に対するインデックスを操作する上で理論上必要となる最少の演算子と関数の集合を含むように指定すべきです。そして、関連するが基本的なものではない演算子を演算子族の自由なメンバとして追加することができます。

例えばPostgreSQLにはinteger_opsという組み込みのB-tree演算子族があります。ここにはbigint (int8)、integer (int4)、smallint (int2)型の列上へのインデックスにそれぞれ対応したint8_ops、int4_ops、int2_opsという演算子クラスが含まれています。また、上記の型の内任意の2つの型を比較できるように、この演算子族にはデータ型を跨る比較演算子も含まれます。このため、上記の型のいずれかに対するインデックスを他の型の値との比較の際に使用することができます。この演算子族は以下の定義により多重化されています。

```
CREATE OPERATOR FAMILY integer_ops USING btree;

CREATE OPERATOR CLASS int8_ops
DEFAULT FOR TYPE int8 USING btree FAMILY integer_ops AS

  -- 標準int8比較
  OPERATOR 1 < ,
  OPERATOR 2 <= ,
  OPERATOR 3 = ,
  OPERATOR 4 >= ,
  OPERATOR 5 > ,
  FUNCTION 1 btint8cmp(int8, int8) ,
  FUNCTION 2 btint8sortsupport(internal) ,
  FUNCTION 3 in_range(int8, int8, int8, boolean, boolean) ,
  FUNCTION 4 btequalimage(oid) ;

CREATE OPERATOR CLASS int4_ops
DEFAULT FOR TYPE int4 USING btree FAMILY integer_ops AS

  -- 標準int4比較
  OPERATOR 1 < ,
  OPERATOR 2 <= ,
  OPERATOR 3 = ,
  OPERATOR 4 >= ,
  OPERATOR 5 > ,
  FUNCTION 1 btint4cmp(int4, int4) ,
  FUNCTION 2 btint4sortsupport(internal) ,
  FUNCTION 3 in_range(int4, int4, int4, boolean, boolean) ,
  FUNCTION 4 btequalimage(oid) ;

CREATE OPERATOR CLASS int2_ops
DEFAULT FOR TYPE int2 USING btree FAMILY integer_ops AS

  -- 標準int2比較
  OPERATOR 1 < ,
  OPERATOR 2 <= ,
  OPERATOR 3 = ,
  OPERATOR 4 >= ,
  OPERATOR 5 > ,
```

```
  FUNCTION 1 btint2cmp(int2, int2) ,
  FUNCTION 2 btint2sortsupport(internal) ,
  FUNCTION 3 in_range(int2, int2, int2, boolean, boolean) ,
  FUNCTION 4 btequalimage(oid) ;

ALTER OPERATOR FAMILY integer_ops USING btree ADD

  -- 型を跨ぐ比較 int8対int2
  OPERATOR 1 < (int8, int2) ,
  OPERATOR 2 <= (int8, int2) ,
  OPERATOR 3 = (int8, int2) ,
  OPERATOR 4 >= (int8, int2) ,
  OPERATOR 5 > (int8, int2) ,
  FUNCTION 1 btint82cmp(int8, int2) ,


  -- 型を跨ぐ比較 int8対int4
  OPERATOR 1 < (int8, int4) ,
  OPERATOR 2 <= (int8, int4) ,
  OPERATOR 3 = (int8, int4) ,
  OPERATOR 4 >= (int8, int4) ,
  OPERATOR 5 > (int8, int4) ,
  FUNCTION 1 btint84cmp(int8, int4) ,


  -- 型を跨ぐ比較 int4対int2
  OPERATOR 1 < (int4, int2) ,
  OPERATOR 2 <= (int4, int2) ,
  OPERATOR 3 = (int4, int2) ,
  OPERATOR 4 >= (int4, int2) ,
  OPERATOR 5 > (int4, int2) ,
  FUNCTION 1 btint42cmp(int4, int2) ,


  -- 型を跨ぐ比較 int4対int8
  OPERATOR 1 < (int4, int8) ,
  OPERATOR 2 <= (int4, int8) ,
  OPERATOR 3 = (int4, int8) ,
  OPERATOR 4 >= (int4, int8) ,
  OPERATOR 5 > (int4, int8) ,
  FUNCTION 1 btint48cmp(int4, int8) ,


  -- 型を跨ぐ比較 int2対int8
  OPERATOR 1 < (int2, int8) ,
  OPERATOR 2 <= (int2, int8) ,
```

```
  OPERATOR 3 = (int2, int8) ,
  OPERATOR 4 >= (int2, int8) ,
  OPERATOR 5 > (int2, int8) ,
  FUNCTION 1 btint28cmp(int2, int8) ,


  -- 型を跨ぐ比較 int2対int4
  OPERATOR 1 < (int2, int4) ,
  OPERATOR 2 <= (int2, int4) ,
  OPERATOR 3 = (int2, int4) ,
  OPERATOR 4 >= (int2, int4) ,
  OPERATOR 5 > (int2, int4) ,
  FUNCTION 1 btint24cmp(int2, int4) ,

  -- cross-type in_range functions
  FUNCTION 3 in_range(int4, int4, int8, boolean, boolean) ,
  FUNCTION 3 in_range(int4, int4, int2, boolean, boolean) ,
  FUNCTION 3 in_range(int2, int2, int8, boolean, boolean) ,
  FUNCTION 3 in_range(int2, int2, int4, boolean, boolean) ;
```

この定義は演算子ストラテジ関数番号とサポート関数番号を「上書き」していることに注意してください。各番号は演算子族内で複数回現れます。特定番号のインスタンスがそれぞれ異なる入力データ型を持つ限り、これは許されます。入力型の両方が演算子クラスの入力型と同じインスタンスは、演算子クラスの主演算子および主サポート関数であり、ほとんどの場合、演算子族の自由メンバではなく演算子クラスの一部として宣言しなければなりません。

詳細が63.2で示されている通り、B-tree演算子族では演算子族内のすべての演算子は互換性をもってソートしなければなりません。演算子族内の各演算子では、演算子と同じデータ型の2つのデータ型を取るサポート関数が存在しなければなりません。演算子族を完結させること、つまり、データ型の組み合わせそれぞれに対する演算子をすべて含めることを推奨します。各演算子クラスは、自身のデータ型に対してデータ型を跨らない演算子とサポート関数だけを含めなければなりません。

複数データ型のハッシュ演算子族を構築するには、演算子族でサポートされるデータ型それぞれに対する互換性を持つハッシュサポート関数を作成しなければなりません。ここで、互換性とは、関数がその演算子族の等価性演算子で等価であるとみなされる任意の2つの値では同一のハッシュコードが生成されることを保証することを意味します。通常、型が異なる物理表現を持つ場合、これを実現することは困難ですが、実現可能な場合もあります。さらに、暗黙的またはバイナリ変換により、ある演算子族で表現されるデータ型から同じ演算子族で表現されるデータ型に値をキャストしても、計算されたハッシュ値を変更してはいけません。データ型1つに対してサポート関数が1つしか存在しないことに注意してください。等価性演算子ごとに1つではありません。演算子族を完結させること、つまり、データ型の組み合わせそれぞれに対する等価性演算子をすべて含めることを推奨します。各演算子クラスは、自身のデータ型に対してデータ型を跨らない演算子とサポート関数だけを含めなければなりません。

GiST、SP-GiST、GINインデックスではデータ型を跨る操作についての明示的な記法はありません。サポートされる演算子群は単に指定演算子クラスの主サポート関数が扱うことができるものです。

BRINでは、要求は演算子クラスを提供するフレームワークに依存します。minmaxに基づく演算子クラスに対しては、求められる振る舞いはB-tree演算子クラスに対するものと同じです。族内のすべての演算子はソート互換でなければならず、キャストは関連するソート順序を変更してはいけません。

> **注記**
>
> PostgreSQL8.3より前のバージョンでは演算子族という概念はありませんでした。そのため、インデックスで使用する予定のデータ型を跨る演算子はすべて、インデックスの演算子クラスに結びつけなければなりませんでした。この手法もまだ使用できますが、インデックスの依存性を広げる点、および、両データ型が同一演算子族内で演算子を持つ場合、プランナがデータ型を跨った比較をより効率的に扱うことができる点より、廃止予定です。

37.16.6. システムの演算子クラスに対する依存性

PostgreSQLは演算子クラスを、単にインデックスで使用できるかどうかだけではなく、多くの方式で演算子の性質を推定するために使用します。したがって、データ型の列をインデックス付けするつもりがなくても、 演算子クラスを作成した方が良い可能性があります。

具体的には、ORDER BYやDISTINCTなど、値の比較とソートを必要とするSQL機能があります。ユーザ定義のデータ型に対してこの機能を実装するために、PostgreSQLはそのデータ型用のデフォルトのB-tree演算子クラスを検索します。この演算子クラスの「等価判定」メンバが、GROUP BYやDISTINCT用の値の等価性についてのシステムの意向を定義し、この演算子クラスによって強制されるソート順序が、デフォルトのORDER BY順序を定義します。

データ型用のデフォルトのB-tree演算子クラスが存在しないと、システムはデフォルトのハッシュ演算子クラスを検索します。しかし、この種類の演算子クラスは等価性のみを提供しますので、ソートではなくグループ化のみサポートできます。

データ型用のデフォルトの演算子クラスが存在しない場合に、こうしたSQL機能をデータ型に使用しようとすると、「順序付け演算子を識別できなかった」といったエラーとなります。

> **注記**
>
> PostgreSQLバージョン7.4より前まででは、ソートやグループ化演算は暗黙的に=、<、>という名前の演算子を使用していました。この新しい、デフォルトの演算子クラスに依存する振舞いによって、特定の名前を持つ演算子の振舞いについて何らかの仮定を立てることを防止しています。

演算子クラスの小なり演算子をUSINGオプションに指定することで、デフォルトでないB-tree演算子クラスによるソートが可能です。以下に例を示します。

```
SELECT * FROM mytable ORDER BY somecol USING ~<~;
```

代わりにUSINGで演算子クラスの大なり演算子を指定すると降順ソートが行われます。

ユーザ定義型の配列の比較も型のデフォルトB-tree演算子クラスで定義された意味に依存します。デフォルトのB-tree演算子クラスが無く、しかしデフォルトのハッシュ演算子クラスがある場合、配列の順比較ではなく同等比較がサポートされます。

データ型特有の知識をさらに必要とする他のSQL仕様としては、ウィンドウ関数（4.2.8を参照してください）のRANGE `offset` PRECEDING/FOLLOWINGフレームオプションがあります。下記のような問い合わせに対して、

```
SELECT sum(x) OVER (ORDER BY x RANGE BETWEEN 5 PRECEDING AND 10 FOLLOWING)
  FROM mytable;
```

これはどのようにxで整列するかを知るのに十分ではありません。データベースは現在のウィンドウフレームの境界を識別するためにどのように現在行のxの値に「5を減算」や「10を加算」を行うかを理解する必要もあります。ORDER BY整列を定義するB-tree演算子クラスで提供される比較演算子を使って結果として生じる他の行のx値への範囲を比較することは可能です。しかし、加算、減算演算子は演算子クラスの一部ではありません。では、どの演算子が使われるべきでしょうか。異なるソート順序（異なるB-tree演算子クラス）では異なる振る舞いを要するかもしれないため、選択を決め打ちすることは望ましくありません。そのため、B-tree演算子クラスはそのソート順に意味がある加算と減算の振る舞いをカプセル化する*in_range*サポート関数を指定することができます。RANGE句のオフセットとして使う意味のある複数のデータ型がある場合にむけて、複数のin_rangeサポート関数を提供することもできます。ウィンドウのORDER BY句と関連しているB-tree演算子クラスが、一致するin_rangeサポート関数を持たない場合、PRECEDING/FOLLOWINGオプションはサポートされません。

他の重要な点として、ハッシュ演算子族内に現れる等価性演算子がハッシュ結合、ハッシュ集約、関連する最適化の候補となることがあります。使用するハッシュ関数を識別するため、ここでのハッシュ演算子族は基本的なものです。

37.16.7. 順序付け演算子

一部のインデックスアクセスメソッド（現時点ではGiSTとSP-GiSTのみ）は*順序付け演算子*という概念をサポートします。これまで説明してきたものは*検索演算子*でした。検索演算子は、WHERE `indexed_column operator constant`を満たすすべての行を見つけるために、インデックスを検索可能にするためのものです。一致した行がどの順序で返されるかについては保証がないことに注意してください。反対に、順序付け演算子は返すことができる行集合を限定しませんが、その順序を決定します。順序付け演算子は、ORDER BY `indexed_column operator constant`で表される順序で行を返すために、インデックスをスキャン可能にするためのものです。このように順序付け演算子を定義する理由は、その演算子が距離を測るものであれば最近傍検索をサポートすることです。例えば以下のような問い合わせを考えます。

```
SELECT * FROM places ORDER BY location <-> point '(101,456)' LIMIT 10;
```

これは指定した対象地点に最も近い10地点を見つけ出します。<->は順序付け演算子ですので、location列上のGiSTインデックスは、これを効率的に行うことができます。

検索演算子が論理値結果を返さなければなりませんが、順序付け演算子は普通、距離を表す浮動小数点や数値型など、何らかの他の型を返します。この型は通常、インデックス対象のデータ型と同じにはなりません。異なるデータ型の動作についての固定化された前提を防ぐために、順序付け演算子の定義では、結果データ型のソート順序を指定するB-tree演算子族の名前を必要とします。前節で述べたように、B-tree演算

子族はPostgreSQLの順序付け記法を定義します。ですのでこれは自然な表現です。pointに対する<->演算子はfloat8を返しますので、演算子クラスを作成するコマンド内で以下のように指定します。

```
OPERATOR 15    <-> (point, point) FOR ORDER BY float_ops
```

ここでfloat_opsは、float8に対する操作を含んだ組み込みの演算子族です。この宣言は、インデックスが<->演算子の値が増加する方向で行を返すことができることを表しています。

37.16.8. 演算子クラスの特殊な機能

演算子クラスには、まだ説明していない2つの特殊な機能があります。説明していない主な理由は、最もよく使用するインデックスメソッドでは、これらがあまり有用ではないためです。

通常、演算子を演算子クラス（または演算子族）のメンバとして宣言すると、インデックスメソッドでその演算子を使用して、WHERE条件を満たす行の集合を正確に抽出することができます。以下に例を示します。

```
SELECT * FROM table WHERE integer_column < 4;
```

この式は、整数列にB-treeインデックスを使用することにより、正確に満たすことができます。しかし、一致する行へ厳密ではなくとも導く手段としてインデックスが有用である場合があります。例えば、GiSTインデックスで、幾何オブジェクトの外接矩形のみを格納したとします。その結果、多角形のような長方形でないオブジェクトとの重なりをテストするWHERE条件は正確に満たすことができません。もっとも、このインデックスを使用して、対象オブジェクトの外接矩形に重なる外接矩形を持つオブジェクトを検索し、さらに、検索されたオブジェクトのみに対して正確に重なるかどうかをテストすることはできます。この筋書きを適用する場合、インデックスは演算子に対して「非可逆」と言われます。非可逆インデックス検索は、ある行が問い合わせ条件を実際に満足するかしないかの時に*recheck*フラグを返すインデックスメソッドを持つことで実装されます。コアシステムは、そこで有効なマッチとして行が返されるか否かを確認するために、抽出された行に対して元の問い合わせ条件を検査します。この手法はインデックスがすべての必要な行を返すことが保証された上で、元の演算子呼び出しを実行することによって除外することができる、いくつか余分な行を返す可能性がある場合に動作します。非可逆検索を提供するインデックス方式（現時点ではGiST、SP-GiSTおよびGIN）は個々の演算子クラスのサポート関数がrecheckフラグを設定することを許可します。このためこれは原則的に演算子クラスの機能です。

再度、多角形のような複雑なオブジェクトの外接矩形のみをインデックスに格納している状況を考えてみてください。この場合、インデックスエントリに多角形全体を格納するのは、それほど有用なことではありません。単に、より単純なbox型のオブジェクトを格納した方が良いかもしれません。このような状況は、CREATE OPERATOR CLASSのSTORAGEオプションによって表現することができます。例えば、以下のように記述します。

```
CREATE OPERATOR CLASS polygon_ops
    DEFAULT FOR TYPE polygon USING gist AS
        ...
        STORAGE box;
```

現時点では、GiST、GINおよびBRINインデックスメソッドが、列のデータ型と異なるSTORAGE型をサポートしています。STORAGEが使用された場合、GiSTのcompressおよびdecompressサポートルーチンは、データ型

を変換する必要があります。GINでは、STORAGE型は「キー」の値の型を識別します。通常これはインデックス付けされる列の型とは異なります。例えば、整数配列の列用の演算子クラスは単なる整数をキーとして持つかもしれません。GINのextractValueおよびextractQueryサポートルーチンが、インデックス付けされた値からキーを取り出す責任を負います。BRINはGINと同様です。STORAGE型は格納された要約値の型を識別し、演算子クラスのサポートプロシージャは要約値を正しく解釈する責任を負います。

37.17. 関連するオブジェクトを拡張としてパッケージ化

PostgreSQLへの有用な拡張は通常、複数のSQLオブジェクトを含んでいます。例えば、新しいデータ型は新しい関数、新しい演算子、おそらく新しいインデックス演算子クラスを必要とします。これらのオブジェクトをすべて単一のパッケージとしてまとめることは、データベース管理を単純化するために役に立ちます。PostgreSQLではこうしたパッケージを*拡張*とよびます。拡張を定義するためには、少なくとも、拡張のオブジェクトを作成するためのSQLコマンドを含む*スクリプトファイル*、拡張自身の数個の基本属性を指定する*制御ファイル*が必要です。また拡張がCコードを含む場合、通常Cコードで構築された共有ライブラリが存在します。これらのファイルがあれば、単純なCREATE EXTENSIONコマンドがそのオブジェクトをデータベース内に読み込みます。

拡張を使用する主な利点は、SQLスクリプトを実行するだけでデータベースに「粗な」なオブジェクトの群をロードできることではなく、PostgreSQLが拡張のオブジェクトをまとまったものと理解できることです。単一のDROP EXTENSIONコマンドでオブジェクトすべてを削除することができます（個々の「アンインストール」スクリプトを保守する必要はありません）。もっと有用なことは、pg_dumpが拡張の個々のメンバオブジェクトを削除してはならないことを把握していることです。代わりにダンプ内にはCREATE EXTENSIONコマンドだけが含まれます。これは、古いバージョンよりも多くのまたは異なるオブジェクトを含む可能性がある、拡張の新しいバージョンへの移行を大きく単純化します。しかし、こうしたダンプを新しいデータベースにロードする際には、拡張の制御ファイル、スクリプトファイル、その他のファイルが利用できるようにしておく必要があります。

PostgreSQLはユーザに、拡張全体を削除させる以外に、拡張内に含まれる個々のオブジェクトを削除させません。また、拡張のメンバオブジェクトの定義を変更する（例えば関数ではCREATE OR REPLACE FUNCTIONを介して変更する）ことはできますが、変更した定義はpg_dumpによりダンプされないことに留意してください。こうした変更は通常、同時に拡張のスクリプトファイルにも同じ変更を行った場合のみ認識することができます。（しかし設定データを持つテーブルに対しては特殊な準備があります。37.17.3を参照してください。）本番環境では、拡張メンバオブジェクトへの変更を処理するために拡張更新スクリプトを作成するのが一般により良い方法です。

拡張スクリプトは、GRANT文とREVOKE文を使って拡張の一部のオブジェクトに権限を設定するかもしれません。それぞれのオブジェクト（どれかが設定される場合）の最終的な権限のセットは、pg_init_privsシステムカタログに格納されます。pg_dumpが使用されると、CREATE EXTENSIONコマンドがダンプ内に含まれ、オブジェクトの権限をダンプが取られた時点のものに設定するために必要となるGRANT文とREVOKE文が後に続きます。

PostgreSQLは、現在拡張スクリプトにてCREATE POLICY文やSECURITY LABEL文の発行をサポートしていません。これらは拡張が作成された後に設定されるべきです。拡張オブジェクトのすべての行セキュリティポリシーとセキュリティラベルはpg_dumpによって作成されたダンプに含まれます。

また拡張機構は、拡張に含まれるSQLオブジェクトの定義を調整するパッケージ調整スクリプトを準備しています。例えば、拡張のバージョン1.1でバージョン1.0と比べて1つの関数を追加し、他の関数本体を変更する場合、拡張の作成者はこれらの2つの変更のみを行う更新スクリプトを提供することができます。そしてALTER EXTENSION UPDATEコマンドを使用して、これらの変更を適用し、指定されたデータベース内に実際にインストールされた拡張のバージョンが何かを記録します。

拡張のメンバとなり得るSQLオブジェクトの種類をALTER EXTENSIONで説明します。拡張は1つのデータベースの中でのみ認識されますので、データベース、ロール、テーブル空間などデータベースクラスタ全体のオブジェクトは拡張のメンバにすることができないことに注意してください。(拡張のスクリプトでこうしたオブジェクトを生成することは禁止されていませんが、作成したとしても、拡張の一部として記録されません。)また、テーブルは拡張のメンバになることができますが、インデックスなどそれに付随するオブジェクトは拡張の直接的なメンバとはみなされません。もう一つの重要な点は、スキーマは拡張に属すことがありますがその逆はないということです。拡張は非修飾名でいかなるスキーマ「の中に」も存在しません。しかし、拡張のメンバオブジェクトはオブジェクトの型が適切であればスキーマに属します。拡張が自身のメンバオブジェクトが属するスキーマを所有することは適切かも知れませんし、そうでないかも知れません。

ある拡張のスクリプトが(一時テーブルのような)一時オブジェクトを作成する場合、現在のセッションで、以降そのオブジェクトは拡張のメンバーとして扱われます。しかしすべての一時オブジェクト同様、セッションの終わりに削除されます。これは、拡張全体を削除することなしに、拡張のメンバーオブジェクトは削除できない、という規則の例外です。

37.17.1. 拡張のファイル

CREATE EXTENSIONコマンドは各拡張に関して、拡張と同じ名前に.controlという拡張子を持つファイル名である必要がある、制御ファイルに依存します。また、このファイルはインストレーションのSHAREDIR/extensionディレクトリ内に存在しなければなりません。また少なくとも1つの、extension--old_version--target_version.sqlという命名規約(例えばfoo拡張のバージョン1.0ではfoo--1.0.sql)に従ったSQLスクリプトファイルが存在しなければなりません。デフォルトでは、このスクリプトファイルもSHAREDIR/extensionディレクトリに格納されますが、制御ファイルでスクリプトファイルを別のディレクトリに指定することができます。

拡張の制御ファイルのファイル書式はpostgresql.confファイルと同じです。すなわち、parameter_name = valueという代入を1行当たり1つ記述します。空行および#から始まるコメントが許されます。単一の単語または数字ではない値にはすべて引用符で確実にくくってください。

制御ファイルは以下のパラメータを設定することができます。

directory (string)

拡張のSQLスクリプトファイルを含むディレクトリです。絶対パスで指定されていない限り、この名前はインストレーションのSHAREDIRディレクトリからの相対パスになります。デフォルトの動作はdirectory = 'extension'と指定した場合と同じです。

default_version (string)

拡張のデフォルトのバージョン(CREATE EXTENSIONでバージョン指定がない場合にインストールされるバージョン)です。これは省略することができますが、その場合VERSIONオプションがないCREATE EXTENSIONは失敗します。ですので通常省略しようとは思わないでしょう。

comment (string)

拡張に関するコメント(任意の文字列)です。最初に拡張が作成されるときにコメントは適用されますが、拡張が更新される間はされません(ユーザが追加したコメントを上書いてしまうため)。この他の方法として、スクリプトファイル内でCOMMENTコマンドを使用してコメントを設定することができます。

encoding (string)

スクリプトファイルで使用される文字セット符号化方式です。スクリプトファイルに何らかの非ASCII文字が含まれる場合に指定しなければなりません。指定がなければ、ファイルはデータベース符号化方式であると仮定されます。

module_pathname (string)

このパラメータの値でスクリプトファイル内のMODULE_PATHNAMEの出現箇所が置換されます。設定されていない場合は置換は行われません。通常これは、スクリプトファイル内で共有ライブラリの名前を直接書き込む必要がなくなるように$libdir/shared_library_nameに設定され、C言語関数ではCREATE FUNCTIONコマンド中でMODULE_PATHNAMEを使用します。

requires (string)

拡張が依存する拡張の名前のリストです。例えばrequires = 'foo, bar'です。対象の拡張がインストールできるようになる前に、これらの拡張がインストールされていなければなりません。

superuser (boolean)

このパラメータがtrue(デフォルト)の場合、スーパーユーザのみが拡張を作成または新しいバージョンに更新することができます(ただし、後述するtrustedも参照してください)。falseに設定されている場合は、インストール中のコマンド実行またはスクリプト更新のために必要な権限のみが必要とされます。いずれかスクリプトコマンドがスーパーユーザ権限を必要とするなら、通常はtrueに設定されるべきです。(このようなコマンドはいずれにせよ失敗するでしょうけれども、前もってエラーを出す方がよりユーザフレンドリです。)

trusted (boolean)

このパラメータは、true(デフォルトではありません)に設定されている場合、一部の非スーパーユーザがsuperuserにtrueを設定している拡張をインストールできるようにします。具体的には、現在のデータベースにCREATE権限を持っているユーザにインストールが許可されるようになります。CREATE EXTENSIONを実行するユーザがスーパーユーザでないけれども本パラメータの効力でインストールできるとき、インストールやスクリプトの更新は実行したユーザではなく、サービス起動のスーパーユーザとして実行されます。superuserがfalseの場合、本パラメータは無意味です。一般に、ファイルシステムのアクセスなど、別の方法ではスーパーユーザのみができることにアクセスができる拡張に対して、これをtrueにすべきではありません。また、拡張をtrustedとして作成するには、拡張のインストールとスクリプト更新を安全に記述するために、かなりの追加の労力が影響が必要です。37.17.6を参照してください。

relocatable (boolean)

拡張を最初に作成した後に拡張により含まれるオブジェクトを別のスキーマに移動することができる場合、拡張は再配置可能です。デフォルトはfalse、つまり、拡張は再配置可能ではありません。詳しくは37.17.2を参照してください。

schema (string)

このパラメータは再配置可能ではない拡張に対してのみ設定することができます。拡張が指名したスキーマのみにロードされ、他にはロードされないことを強制します。schemaパラメータは、拡張を最初に作成するときにのみ参照され、拡張が更新される間はされません。詳しくは37.17.2を参照してください。

主制御ファイルextension.controlに加え、拡張はextension--version.controlという形の名前の副制御ファイルを持つことができます。これらを提供する場合は、スクリプトファイルディレクトリに格納しなければなりません。副制御ファイルは主制御ファイルと同じ書式に従います。拡張の対応するバージョンをインストールまたは更新する時、副制御ファイル内で設定されるパラメータはいずれも、主制御ファイルを上書きします。しかしdirectoryおよびdefault_versionパラメータは副制御ファイルで設定することはできません。

拡張のSQLスクリプトファイルにはトランザクション制御コマンド（BEGIN、COMMITなど）およびトランザクションブロックの内側で実行することができないコマンド（VACUUMなど）を除く任意のSQLコマンドを含めることができます。スクリプトファイルが暗黙的にトランザクションブロック内で実行されるためです。

拡張のSQLスクリプトファイルには、\echoから始まる行を含めることができます。この行は拡張の機構では無視されます（コメントとして扱われます）。これは、このスクリプトがCREATE EXTENSION（37.17.7のスクリプト例を参照）ではなくpsqlに渡された場合にエラーを発生するために一般的に使用するために用意されたものです。これがないと、ユーザは間違って拡張としてではなく、「まとまっていない」オブジェクトとして拡張の内容をロードしてしまい、復旧が多少困難な状態になる可能性があります。

拡張のスクリプトに文字列@extowner@が含まれている場合、この文字列は、CREATE EXTENSIONやALTER EXTENSIONを実行した（適切にクォートされた）ユーザ名で置き換えられます。典型的には、この機能はtrustedと印付けされた拡張が選択されたオブジェクトにサービス起動のスーパーユーザではなく実行したユーザを所有者として割り当てる際に使われます。（とはいえ、このようにするには注意深くすべきです。例えば、C言語関数の所有者を非スーパーユーザに割り当てると、そのユーザに権限昇格の経路を作ることになるでしょう）

スクリプトファイルは指定した符号化方式で認められる任意の文字を含めることができますが、PostgreSQLが制御ファイルの符号化方式が何かを把握する方法がありませんので、制御ファイルにはASCII文字のみを含めなければなりません。実際には、拡張のコメントに非ASCII文字を含めたい場合にのみ、これが問題になります。このような場合には、制御ファイルのcommentを使用せず、代わりにコメントを設定するためにスクリプトファイル内でCOMMENT ON EXTENSIONを使用することを勧めます。

37.17.2. 拡張の再配置性

ユーザは拡張に含まれるオブジェクトを拡張の作成者が考えていたスキーマとは別のスキーマにロードしたいとよく考えます。再配置性に関して3つのレベルがサポートされます。

- 完全な再配置可能な拡張は、いつでも、データベースにロードされた後であっても、他のスキーマに移動させることができます。これは、自動的にすべてのメンバオブジェクトを新しいスキーマに名前を変更する、ALTER EXTENSION SET SCHEMAを用いて行います。通常これは、拡張がオブジェクトが含まれるスキーマが何かに関して内部的な仮定を持たない場合のみ可能です。また、拡張のオブジェクト（手続き言語など何らかのスキーマに属さないオブジェクトは無視して）はすべて最初に1つのスキーマ内に存在しなければなりません。制御ファイル内でrelocatable = trueと設定することで、完全な再配置可能と印付けます。

- 拡張はインストール処理の間再配置可能ですが、その後再配置することはできません。通常これは、拡張のスクリプトファイルが、SQL関数用のsearch_path属性の設定など、対象のスキーマを明示的に参照する必要がある場合です。こうした拡張では、制御ファイルでrelocatable = falseと設定し、スクリプトファイル内で対象のスキーマを参照するために@extschema@を設定してください。この文字列の出現箇所はすべて、スクリプトが実行される前に、実際の対象のスキーマ名に置換されます。ユーザはCREATE EXTENSIONのSCHEMAオプションを使用して対象のスキーマを設定することができます。

- 拡張が再配置をまったくサポートしない場合、制御ファイルでrelocatable = falseを設定し、かつ、schemaを意図している対象スキーマの名前に設定してください。これは、制御ファイル内で指定されたスキーマと同じ名前が指定されていない限り、CREATE EXTENSIONのSCHEMAオプションの指定を阻止します。この選択は通常、拡張が@extschema@を使用して置き換えることができないスキーマ名について内部的な仮定を持つ場合に必要です。@extschema@置換機構はこの場合でも使用することができますが、スキーマ名が制御ファイルによって決定されますので、用途は限定されます。

すべての場合において、スクリプトファイルは対象のスキーマを指し示すようにあらかじめ設定したsearch_pathを用いて実行されます。つまりCREATE EXTENSIONは以下と同じことを行います。

```
SET LOCAL search_path TO @extschema@, pg_temp;
```

これによりスクリプトファイルで作成されるオブジェクトを対象のスキーマ内に格納することができます。スクリプトファイルは要望に応じてsearch_pathを変更することができますが、一般的には望まれません。CREATE EXTENSIONの実行後、search_pathは以前の設定に戻されます。

対象のスキーマは制御ファイル内のschemaパラメータがあればこのパラメータにより決定されます。このパラメータがなければ、CREATE EXTENSIONのSCHEMAがあればこの値で決まり、これ以外の場合は現在のデフォルトのオブジェクト生成用スキーマ（呼び出し元のsearch_pathの最初のもの）になります。制御ファイルのschemaパラメータが使用される時、対象のスキーマが存在しない場合は作成されますが、これ以外の2つの場合ではすでに存在しなければなりません。

何らかの事前に必要な拡張が制御ファイル内のrequiresに列挙されていた場合、それらのターゲットスキーマは新しい機能拡張のターゲットスキーマに続いてsearch_pathの初期設定に追加されます。これにより新しい拡張のスクリプトファイルからそれらのオブジェクトが可視になります。

安全のため、全てのケースにおいてpg_tempは自動的にsearch_pathの最後に追記されます。

再配置不可能な拡張は複数スキーマにまたがるオブジェクトを含めることができますが、通常、外部使用を意図したオブジェクトはすべて単一スキーマに格納することが望まれます。この単一スキーマが拡張の対象のスキーマとみなされます。こうした調整は依存する拡張を作成する間、デフォルトのsearch_path設定を都合に合わせて扱います。

37.17.3. 拡張設定テーブル

一部の拡張は、拡張をインストールした後でユーザにより追加または変更される可能性があるデータを持つ設定テーブルを含みます。通常、テーブルが拡張の一部である場合、テーブル定義もその内容もpg_dumpによりダンプされません。しかしこの振舞いは設定テーブルの場合望まれません。ユーザによってなされたデータ変更はダンプ内に含まれなければなりません。さもないとダンプしリストアした後で拡張の動作が変わってしまいます。

この問題を解消するために、拡張のスクリプトファイルでは設定リレーションとして作成されるテーブル、またはシーケンスに印を付け、pg_dumpにテーブルの、またはシーケンスの内容をダンプに含める(定義は含まれません)ようにさせることができます。このためには、以下の例のようにテーブル、またはシーケンスを作成した後にpg_extension_config_dump(regclass, text)関数を呼び出してください。

```
CREATE TABLE my_config (key text, value text);
CREATE SEQUENCE my_config_seq;

SELECT pg_catalog.pg_extension_config_dump('my_config', '');
SELECT pg_catalog.pg_extension_config_dump('my_config_seq', '');
```

任意数のテーブル、またはシーケンスをこの方法で印付けることができます。serial列またはbigserial列に関連したシーケンスが、同様に印付けることができます。

pg_extension_config_dumpの第2引数が空文字列である場合、テーブルのすべての内容がpg_dumpによりダンプされます。これは、拡張のスクリプトによって作成された初期段階においてテーブルが空である場合のみ正しいものです。テーブルの中で初期データとユーザが提供したデータが混在する場合、pg_extension_config_dumpの第2引数においてダンプすべきデータを選択するWHERE条件を提供します。以下に例を示します。

```
CREATE TABLE my_config (key text, value text, standard_entry boolean);

SELECT pg_catalog.pg_extension_config_dump('my_config', 'WHERE NOT standard_entry');
```

このようにした後、拡張のスクリプトで作成される行のみでstandard_entryが確実に真になるようにします。

シーケンスにおいて、pg_extension_config_dumpの第2引数は何も影響を及ぼしません。

初期状態で提供される行がユーザによって変更されるようなもっと複雑な状況では、設定テーブルに対するトリガを作成して、変更された行が正しく印付けられることを確実にするように取り扱うことができます。

pg_extension_config_dumpを再度呼び出すことにより、設定テーブルに関連付いたフィルタ条件を変更することができます。(通常これは拡張の更新スクリプト内で役に立つでしょう。) 設定ファイルからテーブルを取り除くように印付ける方法は、ALTER EXTENSION ... DROP TABLEを用いてテーブルを拡張から分離するしかありません。

このテーブルとの外部キーの関係は、テーブルがpg_dumpによってダンプされる順序に影響します。特に、pg_dumpは参照しているテーブルの前に参照されているテーブルをダンプしようとします。外部キーの関係はCREATE EXTENSION時(データがテーブルにロードされる前)に設定されますので、循環依存はサポートされません。循環依存が存在すれば、データはダンプされますが、そのダンプを直接はリストアできず、ユーザの介入が必要になります。

serial列またはbigserial列に関連したシーケンスは、それらの状態をダンプするために直接印付けする必要があります。親リレーションを印付けすることは、この目的に十分ではありません。

37.17.4. 拡張の更新

拡張機構の1つの利点は、拡張のオブジェクトを定義するSQLコマンドの更新を簡便に管理する方法を提供していることです。これは、拡張のインストール用スクリプトのリリース版それぞれにバージョン

名称またはバージョン番号を関連付けることで行われます。さらに、ユーザにあるバージョンから次のバージョンへ動的にデータベースを更新させることができるようにしたい場合、あるバージョンから次のバージョンまでの間に行われる必要な変更を行う更新スクリプトを提供しなければなりません。更新スクリプトはextension--old_version--target_version.sqlというパターンに従った名前（例えば、foo--1.0--1.1.sqlはfoo拡張のバージョン1.0からバージョン1.1に変更するコマンドを含みます。）を持たなければなりません。

適切な更新スクリプトが利用可能である場合、ALTER EXTENSION UPDATEコマンドはインストール済みの拡張を指定した新しいバージョンへ更新します。更新スクリプトは、CREATE EXTENSIONがインストール用スクリプト向けに提供する環境と同じ環境で実行されます。具体的にはsearch_pathは同じ方法で設定され、スクリプトにより作成される新しいオブジェクトはすべて自動的に拡張に追加されます。また、スクリプトが拡張のメンバーオブジェクトを削除する場合には、それらのメンバーオブジェクトは拡張から自動的に分離されます。

拡張が副制御ファイルを持つ場合、更新スクリプトで使用される制御パラメータは、スクリプトの対象の（新しい）バージョンに関連付けされたものになります。

ALTER EXTENSIONは、要求される更新を実現するために更新スクリプトを連続して実行することができます。例えばfoo--1.0--1.1.sqlとfoo--1.1--2.0.sqlのみが利用可能であるとすると、現在1.0がインストールされている時にバージョン2.0への更新が要求された場合、ALTER EXTENSIONはこれらを順番に適用します。

PostgreSQLはバージョン名称の特性についてまったく仮定を行いません。例えば1.0の次が1.1であるかどうかを把握しません。これは利用可能なバージョン名をかみ合わせ、もっとも少ない数の更新スクリプトを適用するために必要な経路を続けるだけです。（バージョン名には、--を含まず先頭または最後に-が付かなければ、任意の文字を取ることができます。）

「ダウングレード」スクリプトを提供することが便利な場合があります。例えばfoo--1.1--1.0.sqlは、バージョン1.1に関連した変更を元に戻すことができます。この場合、ダウングレードスクリプトがより短いパスを生成するために、予期せず適用されてしまう可能性に注意してください。複数のバージョンをまたがって更新する「近道」更新スクリプトと近道の開始バージョンへのダウングレードスクリプトが存在する場合に危険性があります。ダウングレードしてから近道となる更新スクリプトを実行する方が、バージョンを1つずつ進めるよりも少ない処理で済んでしまうかもしれません。ダウングレードスクリプトが取り返しがつかないオブジェクトを何か削除してしまう場合、望まない結果になってしまいます。

想定外の更新経路かどうかを検査するためには、以下のコマンドを使用してください。

```
SELECT * FROM pg_extension_update_paths('extension_name');
```

これは指定した拡張の個々の既知のバージョン名の組み合わせをそれぞれ、元のバージョンから対象のバージョンへ進む時に取られる更新経路順、またはもし利用できる更新経路がなければNULLを付けて、表示します。経路は--を区切り文字として使用したテキスト形式で表示されます。配列形式の方が良ければregexp_split_to_array(path,'--')を使用することができます。

37.17.5. 更新スクリプトを利用した拡張のインストール

以前から存在している拡張は、おそらく複数のバージョンに渡って存在しているので、拡張の作者は更新スクリプトを開発する必要性が出てきます。たとえば、拡張fooがバージョン1.0、1.1、1.2をリリースしていたとすると、更新スクリプトfoo--1.0--1.1.sqlとfoo--1.1--1.2.sqlが存在しなければなりません。PostgreSQL 10より前では、新しい拡張のバージョンを直接作成するスクリプトファイ

ルfoo--1.1.sqlとfoo--1.2.sqlも新規に作る必要がありました。これらがないと、新しいバージョンの拡張を直接インストールすることはできず、1.0をインストールしてから更新するしかありませんでした。それにはうんざりしますし、また冗長です。しかし、今ではCREATE EXTENSIONが自動的に更新連鎖を追跡してくるので、それは不要になりました。たとえば、foo--1.0.sql、foo--1.0--1.1.sql、foo--1.1--1.2.sqlだけしかない場合、バージョン1.2のインストールのリクエストは、これらのスクリプトを順に実行することによって達成されます。この手順は、最初に1.0をインストールして、1.2にアップデートする場合でも同じです。(ALTER EXTENSION UPDATEは、複数の手順がある場合には、最短の手順を選びます。) この方法で拡張のスクリプトを調整することにより、小さな更新を複数作成するための保守の手間を減らすことができます。

この方法で保守している拡張に二次的な(バージョン固有の)制御ファイルがある場合は、スタンドアローンのインストールスクリプトがない場合でも、各バージョンで制御ファイルが必要になることに注意してください。そのバージョンへと更新する暗黙的な方法を、制御ファイルが決定するからです。たとえば、foo--1.0.controlがrequires = 'bar'を指定しているのに、fooの他の制御ファイルが指定していないとすると、1.0から他のバージョンに更新した際にbarへの依存性が削除されてしまうでしょう。

37.17.6. 拡張のためのセキュリティに関する考慮事項

広く配布される拡張では、インストールされるデータベースについて想定していないはずです。このため、拡張はサーチパスに基づく攻撃を受けないよう、安全なスタイルで拡張によって提供される関数記述するのが適切です。

superuserプロパティを真にしている拡張はインストールや更新スクリプトの中で行われるアクションのセキュリティ面での危険も考慮しなければなりません。悪意あるユーザが不用意に書かれた拡張スクリプトを悪用してトロイの木馬を作成し、スーパーユーザ権限を獲得できるようにすることは、そう難しくありません。

拡張がtrustedと印付けされている場合、そのインストールスキーマはインストールするユーザにより選択できます。そのユーザはスーパーユーザ権限を獲得することを狙って意図的に安全でないスキーマを使用するかもしれません。したがって、trustedの拡張はセキュリティ観点から極めて危険で、そのスクリプトのコマンドは危険性がないことを確実にするため注意深く検証されなければなりません。

関数を安全に書くためのアドバイスは以下のリンクから提供されます。37.17.6.1 また、インストールスクリプトを安全に書くためのアドバイスは以下のリンクから提供されます。37.17.6.2

37.17.6.1. 拡張関数のためのセキュリティに関する考慮事項

拡張により提供されるSQL言語とPL言語関数は実行されるときにサーチパスに基づく攻撃を受ける危険性があります。これらの関数は作成時ではなく、実行時に解析されるためです。

CREATE FUNCTIONのリファレンスページにはSECURITY DEFINER関数を安全に書くためのアドバイスが記載されています。拡張が提供するあらゆる関数は、強い権限を持つユーザから実行されることがあるので、これらのテクニックを適用することは、良い習慣です。

search_pathに安全なスキーマだけを設定できない場合は、修飾されていない名前は悪意あるユーザが定義したオブジェクトとして名前解決されうることを想定してください。暗黙的にsearch_pathに依存する構文に注意してください。例えば、INや CASE 式 WHENは常にサーチパスを使って演算子を選びます。これらの場所には、OPERATOR(スキーマ.=) ANYやCASE WHEN 式を使用してください。

汎用の拡張は通常、安全なスキーマにインストールされることを想定するべきではありません。これはスキーマ修飾された自身のオブジェクトであっても完全にリスクがないわけではないことを意味しています。例え

ば、拡張が myschema.myfunc(bigint)という関数を定義しているとき、myschema.myfunc(42)というような呼び出しは、悪意ある関数myschema.myfunc(integer)に捕捉される可能性があります。必要に応じて明示的なキャストを使用して関数と演算子のデータ型が引数の型と厳密に一致するように注意してください。

37.17.6.2. 拡張スクリプトのためのセキュリティに関する考慮事項

拡張のインストールや更新スクリプトはスクリプト実行時にサーチパスに基づく攻撃を防ぐように記述されなければなりません。スクリプトが参照するオブジェクトがスクリプトの著者が意図したものではないオブジェクトとして解決されうる場合、即座もしくは、誤って定義された拡張オブジェクトが使われたときに攻撃を受ける可能性があります。

CREATE FUNCTIONやCREATE OPERATOR CLASSなどのDDLコマンドは一般的には安全ですが、汎用的な式を構成に持つコマンドには注意が必要です。例えば、CREATE FUNCTIONのDEFAULT式に行うのと同様にCREATE VIEWには審査が必要です。

拡張スクリプトには汎用SQLを実行する必要があることがあります。例えば、DDLではできないカタログの調整などです。そのようなコマンドは安全なsearch_path使って実行するように気をつけてください。CREATE/ALTER EXTENSIONで提供されるパスが安全であると信用しないでください。最も良い方法は一時的にsearch_pathを'pg_catalog, pg_temp'にセットし、必要な箇所に明示的に拡張のインストールスキーマの参照を記述する方法です(この方法はビューを作成する場合にも参考になります)。例は配布されるPostgreSQLソースコードのcontribに見つけることができます。

拡張をまたがる参照を完璧に安全にすることは極めて困難です。理由の一つに、他の拡張がどのスキーマにあるのか定かではないことがあります。この危険性は両方の拡張を同じスキーマにインストールすることで軽減できます。悪意あるオブジェクトをインストール時のsearch_path内で参照された拡張の前に置くことができないからです。しかしながら、現時点ではこれを要求するメカニズムはありません。今のところ、最良の手段は他の拡張に依存する拡張であるなら、依存する拡張が常にpg_catalogにインストールされるのでない限り、拡張をtrustedと印付けしないことです。

拡張のメンバーであることが分かっている関数の定義を変更しなければならない更新スクリプトを除いて、CREATE OR REPLACE FUNCTIONは使用しないでください(他のOR REPLACEでも同様です)。OR REPLACE を不必要に使うことは他の誰かの別の関数を誤って上書きしてしまうリスクがあるだけでなく、上書きされた関数は元の所有者のままなので元の所有者が変更できる状態となってしまい、セキュリティの脆弱性になります。

37.17.7. 拡張の例

ここでは、SQLのみの拡張の完全な例を示します。「k」と「v」という名称の2つの要素からなる複合型であり、 そのスロットには任意の型の値を格納することができるものです。格納の際テキスト以外の値は自動的にテキストに変換されます。

pair--1.0.sqlスクリプトファイルは以下のようになります。

```
-- スクリプトが、CREATE EXTENSION経由ではなく、psqlのソースとして使われた場合には文句を言う
\echo Use "CREATE EXTENSION pair" to load this file. \quit

CREATE TYPE pair AS ( k text, v text );
```

```
CREATE FUNCTION pair(text, text)
RETURNS pair LANGUAGE SQL AS 'SELECT ROW($1, $2)::@extschema@.pair;';

CREATE OPERATOR ~> (LEFTARG = text, RIGHTARG = text, FUNCTION = pair);

-- "SET search_path" is easy to get right, but qualified names perform better.
CREATE FUNCTION lower(pair)
RETURNS pair LANGUAGE SQL
AS 'SELECT ROW(lower($1.k), lower($1.v))::@extschema@.pair;'
SET search_path = pg_temp;

CREATE FUNCTION pair_concat(pair, pair)
RETURNS pair LANGUAGE SQL
AS 'SELECT ROW($1.k OPERATOR(pg_catalog.||) $2.k,
              $1.v OPERATOR(pg_catalog.||) $2.v)::@extschema@.pair;';
```

pair.control制御ファイルは以下のようになります。

```
# pair extension
comment = 'A key/value pair data type'
default_version = '1.0'
# cannot be relocatable because of use of @extschema@
relocatable = false
```

これらの2つのファイルを正しいディレクトリにインストールするためにメークファイルを作成する必要はほとんどありませんが、以下を含むMakefileを使用することができます。

```
EXTENSION = pair
DATA = pair--1.0.sql

PG_CONFIG = pg_config
PGXS := $(shell $(PG_CONFIG) --pgxs)
include $(PGXS)
```

このメークファイルは37.18で説明するPGXSに依存します。make installコマンドは制御ファイルとスクリプトファイルをpg_configで報告される正しいディレクトリにインストールします。

ファイルがインストールされた後、CREATE EXTENSIONコマンドを使用してオブジェクトを任意の特定のデータベースにロードしてください。

37.18. 拡張構築基盤

PostgreSQL拡張モジュールの配布を考えているのであれば、移植可能な構築システムを準備することはかなり難しいものになるかもしれません。このためPostgreSQLインストレーションは単純な拡張モジュールをすでにインストールされているサーバに対して簡単に構築することができるように、PGXSと呼ばれる拡張向けの構築基盤を提供します。PGXSは主にCコードを含む拡張を意図していますが、SQLのみからなる拡張

でも使用することができます。PGXSがPostgreSQLと相互に作用する任意のソフトウェアを構築するために使用できるような万能な構築システムを意図したものではないことに注意してください。これは単に、単純なサーバ拡張用の一般的な構築規則を自動化するものです。より複雑なパッケージでは、独自の構築システムを作成する必要があるかもしれません。

独自の拡張でPGXS基盤を使用するためには、簡単なメークファイルを作成する必要があります。このメークファイルの中で、いくつか変数を設定し、大域的なPGXSメークファイルをインクルードする必要があります。以下にisbn_issnという名称の拡張モジュールを構築する例を示します。このモジュールはいくつかのCコードを含む共有ライブラリ、拡張の制御ファイル、SQLスクリプト、インクルードファイル（他のモジュールが拡張の関数にSQLを経由せずにアクセスする必要があるかもしれない場合にのみ必要です）、ドキュメントテキストファイルから構成されます。

```
MODULES = isbn_issn
EXTENSION = isbn_issn
DATA = isbn_issn--1.0.sql
DOCS = README.isbn_issn
HEADERS_isbn_issn = isbn_issn.h

PG_CONFIG = pg_config
PGXS := $(shell $(PG_CONFIG) --pgxs)
include $(PGXS)
```

最後の3行は常に同じです。ファイルのこの前に変数の設定と独自のmakeルールを記載してください。

以下の3個の変数の1つを構築対象に指定してください。

MODULES

同じ家系のソースファイルから構築される共有ライブラリのリストです。（このリストにはライブラリ接頭辞を含めないでください。）

MODULE_big

複数のソースファイルから構築される共有ライブラリです。（OBJSにオブジェクトファイルを列挙します。）

PROGRAM

構築する実行プログラムです。（OBJSにオブジェクトファイルを列挙します。）

以下の変数も設定することができます。

EXTENSION

拡張の名前です。各名前に対して、prefix/share/extensionにインストールされるextension.controlを提供しなければなりません。

MODULEDIR

DATAおよびDOCSファイルのインストール先となるはずのprefix/shareサブディレクトリです。（設定がない場合、デフォルトはEXTENSIONが設定されている場合はextensionに、設定されていない場合はcontribになります。）

DATA

prefix/share/$MODULEDIRにインストールされる様々なファイルです。

DATA_built

prefix/share/$MODULEDIRにインストールされる、最初に構築しなければならない様々なファイルです。

DATA_TSEARCH

prefix/share/tsearch_data以下にインストールされる様々なファイルです。

DOCS

prefix/doc/$MODULEDIR以下にインストールされる様々なファイルです。

HEADERS
HEADERS_built

(必要に応じてビルドして)prefix/include/server/$MODULEDIR/$MODULE_big以下にインストールをするファイル。

DATA_builtと違って、HEADERS_builtのファイルはcleanターゲットによって削除されません。削除したい場合には、それらをEXTRA_CLEANにも加えるか、削除を行う独自のルールを追加してください。

HEADERS_$MODULE
HEADERS_built_$MODULE

(指定されていたならビルド後に)prefix/include/server/$MODULEDIR/$MODULEの下にインストールするファイル。ここでの$MODULEはMODULESかMODULE_bigで使われているモジュール名でなければなりません。

DATA_builtと違って、HEADERS_built_$MODULEのファイルはcleanターゲットによって削除されません。削除したい場合には、これらをEXTRA_CLEANにも加えるか、削除を行う独自のルールを追加してください。

同じモジュールあるいは任意の組み合わせに対して両方の変数を使うことは正当ですが、MODULESリストにプレフィックスbuilt_の有無しか異ならない二つのモジュール名を書く場合を除きます。これは両義解釈をひき起こすでしょう。このような(おそらくありそうにない)場合、HEADERS_built_$MODULE変数だけを使うべきです。

SCRIPTS

prefix/binにインストールされるスクリプトファイルです(バイナリファイルではありません)。

SCRIPTS_built

prefix/binにインストールされる、最初に構築しなければならないスクリプトファイルです(バイナリファイルではありません)。

REGRESS

リグレッションテストケース(接尾辞がない)のリストです。後述します。

REGRESS_OPTS

pg_regressに渡す追加オプションです。

ISOLATION

分離性試験のリストです。詳細は後述します。

ISOLATION_OPTS

pg_isolation_regressに渡す追加オプションです。

TAP_TESTS

TAPテストを実行する必要があるかどうかを定義するオプションです。後述します。

NO_INSTALLCHECK

installcheckターゲットを定義しません。テストの際に特殊な設定が必要、あるいはpg_regressを使用しない場合などに有用です。

EXTRA_CLEAN

make cleanで削除される追加ファイルです。

PG_CPPFLAGS

CPPFLAGSの先頭に加えられます。

PG_CFLAGS

CFLAGSに加えられます。

PG_CXXFLAGS

CXXFLAGSに加えられます。

PG_LDFLAGS

LDFLAGSの先頭に加えられます。

PG_LIBS

PROGRAMのリンク行に追加されます。

SHLIB_LINK

MODULE_bigリンク行に追加されます。

PG_CONFIG

構築対象のPostgreSQLインストレーション用のpg_configプログラムへのパスです。(通常はPATH内の最初に見つかるpg_configが単純に使用されます)

このメークファイルをMakefileとして拡張を保管するディレクトリ内に保管してください。その後コンパイルするためにmakeを、モジュールをインストールするためにmake installを行うことができます。デフォルトでは、PATHの中で最初に見つかるpg_configプログラムが対応するPostgreSQLインストレーション用に拡張はコンパイルされ、インストールされます。メークファイルまたはmakeのコマンドラインのいずれかでPG_CONFIGを別のpg_configプログラムを指し示すように設定することで、別のインストレーションを使用することができます。

構築ディレクトリを別にしておきたいのであれば、拡張のソースツリーの外のディレクトリでmakeを実行することもできます。この方法は*VPATH*構築とも呼ばれます。以下にやり方を示します。

```
mkdir build_dir
cd build_dir
make -f /path/to/extension/source/tree/Makefile
make -f /path/to/extension/source/tree/Makefile install
```

あるいは、コアコードと同様な方法でVPATH構築用のディレクトリを設定できます。そのようにする1つの方法は、コアスクリプトconfig/prep_buildtreeを使うことです。一度そうすれば、make変数VPATHを以下のように設定することで、構築できます。

```
make VPATH=/path/to/extension/source/tree
make VPATH=/path/to/extension/source/tree install
```

この方法はより様々なディレクトリのレイアウトで機能します。

REGRESS変数に列挙されたスクリプトは、make installを実行した後でmake installcheckによって呼び出すことができる、作成したモジュールのリグレッションテストで使用されます。これが動作するためには、PostgreSQLサーバが実行していなければなりません。REGRESS変数に列挙されたスクリプトは、拡張のディレクトリ内のsql/という名前のサブディレクトリ内に存在しなければなりません。これらのファイルは.sqlという拡張子を持たなければなりません。この拡張子はメークファイル内のREGRESSリストには含まれません。また試験ごとにexpected/という名前のサブディレクトリ内に想定出力を内容として含む、同じステムに.out拡張子を付けた名前のファイルがなければなりません。make installcheckはpsqlを用いて各試験スクリプトを実行し、結果出力が想定ファイルに一致するかどうか比較します。何らかの差異はdiff -c書式でregression.diffsに書き出されます。想定ファイルがない試験を実行しようとすると「問題」として報告されます。このためすべての想定ファイルがあることを確認してください。

ISOLATION変数に列挙されたスクリプトは、make installを実行した後でmake installcheckによって呼び出すことができるモジュールでの同時実行中のセッションの振舞いの負荷テストで使用されます。これが動作するためには、PostgreSQLサーバが実行していなければなりません。ISOLATION変数に列挙されたスクリプトは、拡張のディレクトリ内のspecs/という名前のサブディレクトリ内に存在しなければなりません。これらのファイルは.specという拡張子を持たなければなりません。この拡張子はmakefile内のISOLATIONリストには含まれません。また試験ごとにexpected/という名前のサブディレクトリ内に想定出力を内容として含む、同じステムに.out拡張子を付けた名前のファイルがなければなりません。make installcheckは各試験スクリプトを実行し、結果出力が想定ファイルに一致するかどうか比較します。何らかの差異はdiff -c書式でoutput_iso/regression.diffsに書き出されます。想定ファイルがない試験を実行しようとすると「問題」として報告されます。このためすべての想定ファイルがあることを確認してください。

TAP_TESTSはTAPテストの指定を有効にします。各試験の実行によるデータはtmp_check/という名前のサブディレクトリに含まれます。詳細は32.4を参照してください。

ヒント

想定ファイルを作成する最も簡単な方法は、空のファイルを作成し、試験を実行する(当然差異が報告されます)ことです。(REGRESSの試験による)results/ディレクトリまたは(ISOLATIONの試験による)output_iso/results/ディレクトリ内で見つかる実際の結果ファイルを確認し、テストの想定結果と合致するのであれば、expected/にコピーしてください。

第38章 トリガ

本章ではトリガ関数の作成に関する一般的な情報を示します。トリガ関数は、PL/pgSQL (第42章)、PL/Tcl (第43章)、PL/Perl (第44章)、PL/Python (第45章)など、利用可能な手続き言語のほとんどで作成することができます。本章を読んだ後、好みの手続き言語に関する章を参照して、トリガ作成に関する言語特有の詳細を確認すべきです。

また、C言語でトリガ関数を作成することができます。しかし、ほとんどの方は、手続き言語のいずれかで作成する方が簡単であることに気づくでしょう。現時点では、普通のSQL関数言語ではトリガ関数を作成することはできません。

38.1. トリガ動作の概要

トリガとは、データベースが、ある特定の操作が行われた時に常に自動的に実行しなければならない特定の機能に関する規定です。トリガはテーブル（パーティション化されているかどうかにかかわらず）、ビュー、外部テーブルに付与することができます。

テーブルおよび外部テーブル上では、トリガをINSERT、UPDATEまたはDELETE操作の前後に、行を変更する度、あるいはSQL文ごとに実行するように定義することができます。さらに、UPDATEトリガについては、特定のカラムがUPDATE文のSET句の対象になった時のみ発動するよう設定することができます。また、トリガはTRUNCATE文についても発動できます。トリガイベントが起こると、トリガ関数がそのイベントを扱う適切な時点で呼び出されます。

ビュー上では、トリガをINSERT、UPDATEまたはDELETE操作の代わりに実行するものとして定義できます。そうしたINSTEAD OFトリガは、ビュー内の変更を行うために必要となる行それぞれに対して一度発行されます。ビューの元になっている基底テーブルへの必要な変更の実施、そして必要に応じて、ビュー上で見えるであろう変更された行を返却するのは、トリガ関数の責任です。ビューへのトリガは、SQL文ごとに、INSERT、UPDATEまたはDELETE操作の前後で実行させるよう定義することもできます。しかし、そうしたトリガは、ビューにINSTEAD OFトリガがあるときにだけ発行されます。INSTEAD OFトリガを定義しない場合は、ビューを操作しようとする文は、元になる基底テーブルに影響を与える文に書き換えなければなりません。その結果、発行されるトリガは、基底テーブルに付けられたトリガとなります。

トリガ関数は、トリガ自体が作成される前までに定義しておく必要があります。トリガ関数は、引数を取らない、trigger型を返す関数として宣言される必要があります（トリガ関数は、通常の関数で使用される引数という形ではなく、TriggerData構造体で入力を受け取ります）。

適切なトリガ関数が作成されると、CREATE TRIGGERを使用してトリガを構築することができます。同一のトリガ関数を複数のトリガに使用することができます。

PostgreSQLは、*行単位の*トリガと*文単位の*トリガの両方を提供します。行単位のトリガでは、トリガを発行した文によって影響を受ける行ごとにトリガ関数が呼び出されます。反対に、文単位のトリガでは、適切な文が実行された時に、その文で何行が影響を受けたかどうかは関係なく、一度だけ呼び出されます。特に、行に影響を与えない文であっても、適切な文単位のトリガがあれば実行されます。この2種類のトリガはそれぞれ*行レベル*トリガと*文レベル*トリガと呼ばれることがあります。TRUNCATEに対するトリガは、行単位ではなく、文レベルにのみに定義することができます。

また、トリガはそれらが操作の*前*、*後*または*代わり*のどれで実行されるかに応じて分けられます。これらはそれぞれBEFOREトリガ、AFTERトリガ、そしてINSTEAD OFトリガと呼ばれます。文レベルのBEFOREトリガは、もちろ

ん文が何かを始める前に発行され、文レベルのAFTERトリガは文の本当に最後に発行されます。これらのタイプのトリガはテーブル、ビュー、あるいは外部テーブルに定義できます。行レベルのBEFOREトリガは、特定の行が操作される直前に発行され、行レベルのAFTERトリガは文の終わり(ただし、全ての文レベルのAFTERトリガの前)に発行されます。これらのタイプのトリガは、外部テーブルに定義できますが、ビューには定義できません。INSTEAD OFトリガはビューにのみ定義され、行レベルのみが許されます。つまり、ビュー上のそれぞれの行で処理が必要と判断された場合には、即座に発動します。

継承あるいはパーティション階層において、親テーブルをターゲットとする文は、影響を受けた子テーブルの文レベルトリガを発動しません。すなわち、親テーブルの文レベルトリガのみが発動します。しかし、影響を受けた子テーブルの行レベルトリガは発動します。

INSERTがON CONFLICT DO UPDATE句を含む場合、EXCLUDED列が参照されていると、行単位BEFORE INSERTトリガおよび行単位BEFORE UPDATEトリガの両方の効果が適用され、それが更新後の行の最後の状態から明らかな場合がありえます。ただし、両方の行レベルのBEFOREトリガを実行するためにEXCLUDEDの参照が必要なわけではありません。驚くような結果の可能性について、BEFORE INSERTとBEFORE UPDATEの両方の文単位トリガーがあり、それらがいずれも挿入あるいは更新対象の行に影響を与える場合に考慮すべきです(これは更新が冪等ではないが、ほぼ同等であるときには、それでも問題になります)。文単位のUPDATEトリガはON CONFLICT DO UPDATEが指定されたとき、そのUPDATEによって行が影響を受けたかどうかに関わらず(そしてその代替であるUPDATE部分が実行されたかどうかに関わらず)実行されることに注意してください。ON CONFLICT DO UPDATE句のあるINSERTでは、まず文単位のBEFORE INSERTトリガ、次に文単位のBEFORE UPDATEトリガ、次いで文単位のAFTER UPDATEトリガ、最後に文単位のAFTER INSERTトリガを実行します。

あるパーティション化されたテーブルに適用されたUPDATEの結果、行が他のパーティションに移動することになるなら、元のパーティションでDELETEし、続いて新しいパーティションにINSERTする操作として実行されます。この場合、すべての行レベルBEFORE UPDATEトリガとBEFORE DELETEトリガが元のパーティションで発動します。そして、すべての行レベルBEFORE INSERTトリガが移動先のパーティションで発動します。これらのトリガが移動対象の行に対して影響を及ぼす際に、驚くべき結果となる可能性を考慮しておくべきでしょう。AFTER ROWトリガに関しては、AFTER DELETEとAFTER INSERTトリガが適用されます。しかし、AFTER UPDATEトリガは適用されません。なぜなら、UPDATEはDELETEとINSERTに変換されるからです。文レベルのトリガに関しては、たとえ行の移動が起こったとしてもDELETEトリガもINSERTトリガも発動されません。UPDATE文中に現れた対象テーブルに定義されたUPDATEトリガだけが発動されます。

文単位のトリガによって呼び出されるトリガ関数は常にNULLを返さなければなりません。行単位のトリガによって呼び出されるトリガ関数は呼び出し元のエクゼキュータにテーブル行(HeapTuple型の値)を返すように選択することができます。操作前に発行された行レベルのトリガでは以下の選択肢があります。

- NULLを返して、現在の行への操作を飛ばすことができます。これは、エクゼキュータにトリガの元になった行レベルの操作(特定のテーブル行の挿入、更新、削除)を行わないよう指示します。

- 行レベルのINSERTおよびUPDATEトリガの場合のみ、返される行が挿入される、もしくは実際に更新される行になります。これにより、トリガ関数で、挿入される行もしくは更新される行を変更することができます。

これらの動作をさせたくない行レベルのBEFOREトリガについては、渡された行(つまり、INSERTおよびUPDATEトリガではNEW行、DELETEの場合はOLD行)と同じ行結果を返すように気を付ける必要があります。

行レベルのINSTEAD OFトリガは、ビューの元となった元テーブルのデータをまったく変更しないことを表すNULL、または、渡されたビューの行(INSERTとUPDATE操作の場合NEW行、DELETE操作の場合OLD行)を返さなければなりません。非NULLの戻り値は、そのトリガがビューにおいて必要なデータ変更を実行したことを通知するために使用されます。これにより影響を受けた行数を数えるカウンタは増加されます。

INSERTとUPDATE操作のみ、トリガは戻す前にNEW行を変更することができます。これはINSERT RETURNINGまたはUPDATE RETURNINGで返されるデータを変更しますので、ビューが提供されたデータと正確に同じ結果を返さない場合に有益です。

操作の後に発生する行レベルトリガでは戻り値は無視されますので、これらはNULLを返すことができます。

生成列に対してはいくつか考慮が必要です。格納された生成列は、BEFOREトリガの後、AFTERトリガの前に計算されます。そのため、生成される値はAFTERトリガで調べることができます。BEFOREトリガでは、皆さんが期待している通りOLD行は以前の生成された値を含んでいますが、NEW行は新しく生成される値をまだ含んでおらず、アクセスすべきではありません。C言語インタフェースでは、この時点では列の内容は未定義です。高レベルプログラム言語は、BEFOREトリガ内ではNEW行の生成列へのアクセスを避けるべきです。BEFOREトリガでの生成列の値の変更は無視され、上書きされます。

同一リレーション、同一イベントに対して1つ以上のトリガが定義された場合、トリガはその名前のアルファベット順に発生します。BEFOREトリガとINSTEAD OFトリガの場合では、各トリガで返される、変更された可能性がある行が次のトリガの入力となります。もし、あるBEFOREトリガやINSTEAD OFトリガがNULLを返したら、(いまのところ)操作はその行で中断し、残りのトリガは発生しません。

トリガ定義は、トリガを発動するかどうかをWHEN句の論理条件で指定することも可能です。行レベルトリガにおいて、WHEN条件は行の列の古い値と(あるいは)新しい値を検索することができます。(あまり有用ではありませんが、文レベルトリガでもWHEN条件で同じことができます。)BEFOREトリガでは、実質的にトリガ関数の開始時と同じ条件で検査できるように、WHEN条件の評価が関数の実施直前になされます。しかしAFTERトリガでは、WHEN条件の評価は行の更新直後に行われ、文の終わり(コミット時)にトリガを発動するためのイベントを待ち行列に入れるかどうかを決めます。そのため、あるAFTERトリガのWHEN条件が真を返さなかった場合は、イベントを待ち行列に入れる必要も文の終わりに行を再取得する必要もありません。これは、大量の行の変更が発生するけれども、トリガがその内の少数の行に対してのみ発動させる必要がある、といった文の処理速度を大幅に上げる効果があります。INSTEAD OFトリガはWHEN条件をサポートしていません。

通常、行レベルのBEFOREトリガは、挿入あるいは更新される予定のデータの検査や変更のために使用されます。例えば、BEFOREトリガは、timestamp型の列に現在時刻を挿入するために、あるいは行の2つの要素の整合性を検査するために使用される可能性があります。行レベルのAFTERトリガは、ほとんど常識的に他のテーブルに更新を伝播させるために、あるいは他のテーブルとの整合性を検査するために使用されます。こうした仕事の切り分け理由は、AFTERトリガは行の最終値を見ることができ、BEFOREトリガは見ることができないという点です。トリガをBEFOREにするかAFTERにするかを決める時に特別な理由がないのであれば、操作の情報を行が終わるまで保持する必要がない分、BEFOREを使う方が効率的です。

トリガ関数がSQLコマンドを処理する場合、これらの問い合わせがトリガを再度発行することがあります。これはカスケードされたトリガと呼ばれます。カスケードの段数に直接的な制限はありません。カスケードの場合、同じトリガを再帰的に呼び出すことが可能です。例えば、INSERTトリガで同じテーブルに追加の行を挿入する問い合わせが実行された場合、その結果としてINSERTトリガが再度発行されます。こうした状況で無限再帰を防ぐのは、トリガプログラマの責任です。

トリガを定義する時、そのトリガ用の引数を指定することができます。トリガ定義に引数を含めた目的は、似たような要求の異なるトリガに同じ関数を呼び出すことができるようにすることです。例えば、2つの列名を引数とし、片方に現在のユーザをもう片方に現在のタイムスタンプを取る、汎化トリガ関数があるとします。適切に作成すれば、この関数が特定のトリガの発行元となるテーブルに依存することはなくなります。同じ関数を使用して、例えば、トランザクションテーブルに作成記録を自動的に登録させるために、適切な列を持つ任意のテーブルのINSERTイベントに使用することができます。また、UPDATEとして定義すれば、最終更新イベントを追跡するために使用することも可能です。

トリガをサポートするプログラミング言語はそれぞれ独自の方法で、トリガ関数で利用できるトリガの入力データを作成します。この入力データにはトリガイベント種類（例えばINSERTやUPDATEなど、CREATE TRIGGERで指定された全ての引数）が含まれます。行レベルトリガの入力データには、INSERTおよびUPDATEトリガの場合はNEW行が、UPDATEおよびDELETEトリガの場合はOLD行が含まれます。

デフォルトでは、文レベルトリガには文によって変更された個々の行を検査するための手段がありません。しかし、トリガがアクセスできる影響を受けた行の集合を作成するために、AFTER STATEMENTトリガは、遷移テーブル*(transition tables)*の作成を依頼することができます。AFTER ROWトリガも遷移テーブルを依頼できるので、発動中の個々の行における変更だけでなく、テーブル全体におけるすべての変更を見ることができます。遷移テーブルを検査する方法も使用中のプログラミング言語に依存しますが、典型的な方法は、トリガ関数の中で発行するSQLコマンドでアクセスできる、読み込み専用の一時テーブルのように振る舞う遷移テーブルを作成することです。

38.2. データ変更の可視性

トリガ関数内でSQLコマンドを実行し、このコマンドがトリガの元となったテーブルにアクセスする場合、データの可視性規則に注意する必要があります。この規則が、SQLコマンドがトリガの発行原因となったデータ変更を見ることができるかどうかを決定するからです。簡単に以下に示します。

- 文レベルトリガは次に示す簡単な可視性規則に従います。文によってなされた変更は、文レベルのBEFOREトリガでは不可視です。一方、文レベルのAFTERトリガでは全ての変更が可視です。
- 当然ながら行レベルのBEFOREトリガ内のSQLコマンドでは、トリガの発生原因となったデータ変更（挿入、更新、削除）はまだ発生していませんので、可視ではありません。
- しかし、行レベルのBEFOREトリガで実行されるSQLコマンドは、その外側のコマンドで以前に処理された行へのデータ変更の影響を見ることになるでしょう。これらの変更イベントの順序は一般的に予測できませんので、注意が必要です。複数行に影響するSQLコマンドはどのような順番でもその行を更新することができます。
- 同様に、行レベルのINSTEAD OFトリガは、同じ外側のコマンドで以前に処理されたINSTEAD OFトリガよる変更結果を見ることになるでしょう。
- 行レベルのAFTERトリガが発生すると、その外側のコマンドによってなされた全ての変更は既に完了していますので、呼び出されたトリガ関数から可視になります。

もし、あなたのトリガが標準的な手続き型言語のいずれかで記述されている時、上記の可視性は関数がVOLATILEで定義されている場合のみ適用されます。STABLE、もしくはIMMUTABLEで定義されている関数は、どのようなケースにおいても、呼び出しコマンドによる変更は見ないでしょう。

データ可視性規則に関する詳細は46.5にあります。38.4の例にこの規則を示します。

38.3. Cによるトリガ関数の作成

本節ではトリガ関数とのインタフェースについて低レベルな詳細を説明します。この情報はC言語でトリガ関数を作成する時にのみ必要です。高レベルな言語で作成すれば、こうした詳細は代わりに扱ってもらえます。たいていの場合、Cでトリガを作成する前に手続き言語を使用することを検討すべきです。各手続き言語の文書で、その言語を使用したトリガの作成方法を説明します。

トリガ関数は「version 1」関数マネージャインタフェースを使わなくてはいけません。

関数がトリガマネージャから呼び出される時は、通常の引数が渡されるのではなく、TriggerData構造体を指す「context」ポインタが渡されます。C関数は、トリガマネージャから呼び出されたのかどうかを以下のマクロを実行することで検査することができます。

```
CALLED_AS_TRIGGER(fcinfo)
```

これは以下に展開されます。

```
((fcinfo)->context != NULL && IsA((fcinfo)->context, TriggerData))
```

もしこれが真を返す場合、fcinfo->contextをTriggerData *型にキャストし、指されたTriggerData構造体を使用することは安全です。その関数は、TriggerData構造体やそれが指すどのようなデータも変更してはいけません。

struct TriggerDataはcommands/trigger.hの中で定義されています。

```
typedef struct TriggerData
{
    NodeTag          type;
    TriggerEvent     tg_event;
    Relation         tg_relation;
    HeapTuple        tg_trigtuple;
    HeapTuple        tg_newtuple;
    Trigger         *tg_trigger;
    TupleTableSlot  *tg_trigslot;
    TupleTableSlot  *tg_newslot;
    Tuplestorestate *tg_oldtable;
    Tuplestorestate *tg_newtable;
    const Bitmapset *tg_updatedcols;
} TriggerData;
```

メンバは下記のように定義されています。

type

常にT_TriggerDataです。

tg_event

その関数が呼び出されたイベントを記述します。tg_eventを調べるためには下記のマクロを使うことができます。

TRIGGER_FIRED_BEFORE(tg_event)

トリガが操作の前に(before)発行された場合真を返します。

TRIGGER_FIRED_AFTER(tg_event)

トリガが操作の後に(after)発行された場合真を返します。

TRIGGER_FIRED_INSTEAD(tg_event)

トリガがINSTEAD OFで発行された場合真を返します。

TRIGGER_FIRED_FOR_ROW(tg_event)

トリガが行レベルのイベントで発行された場合真を返します。

TRIGGER_FIRED_FOR_STATEMENT(tg_event)

トリガが文レベルのイベントで発行された場合真を返します。

TRIGGER_FIRED_BY_INSERT(tg_event)

トリガがINSERTコマンドで発行された場合真を返します。

TRIGGER_FIRED_BY_UPDATE(tg_event)

トリガがUPDATEコマンドで発行された場合真を返します。

TRIGGER_FIRED_BY_DELETE(tg_event)

トリガがDELETEコマンドで発行された場合真を返します。

TRIGGER_FIRED_BY_TRUNCATE(tg_event)

トリガがTRUNCATEコマンドで発行された場合真を返します。

tg_relation

トリガの発行元のリレーションを記述する構造体へのポインタです。この構造体についての詳細は、utils/rel.hを参照してください。最も興味深いのは、tg_relation->rd_att(リレーションタプルの記述子)とtg_relation->rd_rel->relnameです(リレーション名、これはchar*ではなくNameDataです。名前のコピーが必要な場合は、char*を得るためにSPI_getrelname(tg_relation)を使用してください)。

tg_trigtuple

トリガが発行された行へのポインタです。これは挿入される、削除される、あるいは更新される行です。もしINSERT/DELETEでこのトリガが発行された時、この行を別のもので置き換えたくない(INSERTの場合)場合や、その操作を飛ばしたくない場合は、これをこの関数から返してください。外部テーブルのトリガに対しては、システム列の値はここでは指定されません。

tg_newtuple

トリガがUPDATEで発行された場合は、行の新しいバージョンへのポインタです。INSERTもしくはDELETEの場合は、NULLです。UPDATEイベントの時、この行を別のもので置き換えたくない場合や操作を飛ばしたくない場合は、これをこの関数から返してください。外部テーブルのトリガに対しては、システム列の値はここでは指定されません。

tg_trigger

以下のようにutils/reltrigger.hで定義された、Trigger構造体へのポインタです。

```
typedef struct Trigger
{
```

```
    Oid         tgoid;
    char       *tgname;
    Oid         tgfoid;
    int16       tgtype;
    char        tgenabled;
    bool        tgisinternal;
    Oid         tgconstrrelid;
    Oid         tgconstrindid;
    Oid         tgconstraint;
    bool        tgdeferrable;
    bool        tginitdeferred;
    int16       tgnargs;
    int16       tgnattr;
    int16      *tgattr;
    char      **tgargs;
    char       *tgqual;
    char       *tgoldtable;
    char       *tgnewtable;
} Trigger;
```

ここで、tgnameがトリガの名前、tgnargsがtgargs内の引数の数、tgargsはCREATE TRIGGER文で指定された引数へのポインタの配列です。他のメンバは内部でのみ使用されます。

tg_trigslot

tg_trigtupleを含むスロット、またはタプルが存在しない場合はNULLポインタです。

tg_newslot

tg_newtupleを含むスロット、またはタプルが存在しない場合はNULLポインタです。

tg_oldtable

tg_relationで指定するフォーマットの0以上の行を含むTuplestorestate型の構造体へのポインタです。OLD TABLE遷移リレーションが存在しない場合はNULLポインタです。

tg_newtable

tg_relationで指定するフォーマットの0以上の行を含むTuplestorestate型の構造体へのポインタです。NEW TABLE遷移リレーションが存在しない場合はNULLポインタです。

tg_updatedcols

UPDATEトリガに対しては、トリガコマンドにより更新された列を示すビットマップ集合です。汎用のトリガ関数はこれを使って、変更されていない列を扱わないことで動作を最適化できます。

例として、属性番号attnum(1始まり)の列がこのビットマップ集合のメンバであるかどうか判定するために、bms_is_member(attnum - FirstLowInvalidHeapAttributeNumber, trigdata->tg_updatedcols))を呼び出します。

UPDATEトリガ以外のトリガに対しては、これはNULLになります。

SPIを使って遷移テーブルを参照するクエリを発行する方法については、SPI_register_trigger_dataを参照してください。

トリガ関数はHeapTupleポインタもしくはNULLポインタ(SQLのNULLではありません。したがって、isNullは真にはなりません)のどちらかを返さなければなりません。操作対象の行を変更したくない場合は、注意して、tg_trigtupleかtg_newtupleの適切な方を返してください。

38.4. 完全なトリガの例

C言語で作成したトリガ関数に関するとても簡単な例をここに示します(手続き言語で作成したトリガの例は、その手続き言語の文書に記載されています。)

trigf関数は、ttestテーブル内にある行数を報告し、問い合わせがxにNULL値を挿入しようとしていた場合は、その操作を飛ばします(つまり、このトリガは、トランザクションを中断させないNOT NULL制約のような動作をします。)

まず、以下のようにテーブルを定義します。

```
CREATE TABLE ttest (
    x integer
);
```

以下がトリガ関数のソースコードです。

```
#include "postgres.h"
#include "fmgr.h"









#include "executor/spi.h"       /* これはSPIを使用する場合に必要なもの */
#include "commands/trigger.h"   /* これはトリガで必要なもの */
#include "utils/rel.h"          /* これはリレーションで必要なもの */

PG_MODULE_MAGIC;

PG_FUNCTION_INFO_V1(trigf);

Datum
trigf(PG_FUNCTION_ARGS)
{
    TriggerData *trigdata = (TriggerData *) fcinfo->context;
    TupleDesc   tupdesc;
    HeapTuple   rettuple;
    char        *when;
    bool        checknull = false;
```

```
    bool        isnull;
    int         ret, i;









    /* トリガとして呼び出されたかどうかを確認 */
    if (!CALLED_AS_TRIGGER(fcinfo))
        elog(ERROR, "trigf: not called by trigger manager");









    /* エクゼキュータに返すタプル */
    if (TRIGGER_FIRED_BY_UPDATE(trigdata->tg_event))
        rettuple = trigdata->tg_newtuple;
    else
        rettuple = trigdata->tg_trigtuple;









    /* NULL値をチェック */
    if (!TRIGGER_FIRED_BY_DELETE(trigdata->tg_event)
        && TRIGGER_FIRED_BEFORE(trigdata->tg_event))
        checknull = true;

    if (TRIGGER_FIRED_BEFORE(trigdata->tg_event))
        when = "before";
    else
        when = "after ";

    tupdesc = trigdata->tg_relation->rd_att;









    /* SPIマネージャに接続 */
    if ((ret = SPI_connect()) < 0)
        elog(ERROR, "trigf (fired %s): SPI_connect returned %d", when, ret);
```

```
    /* テーブル中の行数を取得 */
    ret = SPI_exec("SELECT count(*) FROM ttest", 0);

    if (ret < 0)
        elog(ERROR, "trigf (fired %s): SPI_exec returned %d", when, ret);



    /* count(*)はint8を返す。変換に注意してください*/
    i = DatumGetInt64(SPI_getbinval(SPI_tuptable->vals[0],
                                    SPI_tuptable->tupdesc,
                                    1,
                                    &isnull));

    elog (INFO, "trigf (fired %s): there are %d rows in ttest", when, i);

    SPI_finish();

    if (checknull)
    {
        SPI_getbinval(rettuple, tupdesc, 1, &isnull);
        if (isnull)
            rettuple = NULL;
    }

    return PointerGetDatum(rettuple);
}
```

ソースコードをコンパイル（37.10.5を参照してください）した後に、以下の様に関数とトリガを宣言します。

```
CREATE FUNCTION trigf() RETURNS trigger
    AS 'filename'
    LANGUAGE C;

CREATE TRIGGER tbefore BEFORE INSERT OR UPDATE OR DELETE ON ttest
    FOR EACH ROW EXECUTE FUNCTION trigf();

CREATE TRIGGER tafter AFTER INSERT OR UPDATE OR DELETE ON ttest
    FOR EACH ROW EXECUTE FUNCTION trigf();
```

これで、トリガの操作を確認することができます。

```
=> INSERT INTO ttest VALUES (NULL);
INFO:  trigf (fired before): there are 0 rows in ttest
INSERT 0 0


-- 挿入操作は飛ばされ、
      また、
      AFTERトリガも発行されません。

=> SELECT * FROM ttest;
 x
---
(0 rows)

=> INSERT INTO ttest VALUES (1);
INFO:  trigf (fired before): there are 0 rows in ttest
INFO:  trigf (fired after ): there are 1 rows in ttest
                                       ^^^^^^^^

                             可視性の説明を思い出してください。
INSERT 167793 1
vac=> SELECT * FROM ttest;
 x
---
 1
(1 row)

=> INSERT INTO ttest SELECT x * 2 FROM ttest;
INFO:  trigf (fired before): there are 1 rows in ttest
INFO:  trigf (fired after ): there are 2 rows in ttest
                                       ^^^^^^

                             可視性の説明を思い出してください。
INSERT 167794 1
=> SELECT * FROM ttest;
 x
---
 1
 2
(2 rows)

=> UPDATE ttest SET x = NULL WHERE x = 2;
INFO:  trigf (fired before): there are 2 rows in ttest
UPDATE 0
```

```
=> UPDATE ttest SET x = 4 WHERE x = 2;
INFO:  trigf (fired before): there are 2 rows in ttest
INFO:  trigf (fired after ): there are 2 rows in ttest
UPDATE 1
vac=> SELECT * FROM ttest;
 x
---
 1
 4
(2 rows)

=> DELETE FROM ttest;
INFO:  trigf (fired before): there are 2 rows in ttest
INFO:  trigf (fired before): there are 1 rows in ttest
INFO:  trigf (fired after ): there are 0 rows in ttest
INFO:  trigf (fired after ): there are 0 rows in ttest
                                       ^^^^^^

                            可視性の説明を思い出してください。
DELETE 2
=> SELECT * FROM ttest;
 x
---
(0 rows)
```

src/test/regress/regress.cとspiにはもっと複雑な例があります。

第39章 イベントトリガ

第38章で議論されたトリガの機能を補完するために、PostgreSQLはイベントトリガを提供します。一つのテーブルに付与され、DMLイベントのみ対象にしたこれまでのトリガと違い、イベントトリガは特定のデータベースに大域的であり、DDLイベントを対象に実行できます。

これまでのトリガと違い、イベントトリガは普通のSQLではなく、イベントトリガがサポートする手続き言語やC言語で記述することができます。

39.1. イベントトリガ動作の概要

イベントトリガは、関連づけられたイベントが、定義されたデータベースで起こるたびに実行されます。現在の所サポートされているイベントは、ddl_command_start、ddl_command_end、table_rewrite、sql_dropです。今後のリリースで新たなイベントが追加されるかもしれません。

ddl_command_startイベントは、CREATE、ALTER、DROP、SECURITY LABEL、COMMENT、GRANT、REVOKEコマンドの実行の直前に発生します。影響するオブジェクトが存在するかどうかのチェックはイベントトリガが実行されるまで行われません。しかしながら例外として、共有オブジェクト — データベース、ロール、テーブルスペース — を対象としているDDLコマンド、もしくは、イベントトリガ自体をターゲットにしたコマンドに対してのイベントは起こりません。ddl_command_startはまた、SELECT INTOコマンドの実行直前にも発生します。このコマンドはCREATE TABLE ASコマンドと同等だからです。

ddl_command_endイベントは、ddl_command_startイベントが対象とする同様のコマンドの実行後に発生します。発生したDDL操作のより詳細を取得するには、ddl_command_endイベントトリガコード(9.29を参照してください)で集合を返す関数pg_event_trigger_ddl_commands()を使ってください。トリガはアクションが起きた後(ただし、トランザクションのコミットの前)に起動し、システムカタログは既に変更されたものとして読まれることに注意してください。

sql_dropイベントは、データベースオブジェクトを削除する命令に対するddl_command_endイベントトリガの直前に発生します。削除されたオブジェクトを一覧として出力するために、sql_dropイベントトリガコード(9.29を参照してください)からpg_event_trigger_dropped_objects()という集合を返す関数を使用します。トリガはオブジェクトがシステムカタログから削除された後に実行されるため、以後、そのオブジェクトを見ることができないことに注意してください。

コマンドALTER TABLEやALTER TYPEのアクションによりテーブルが書き換えられる直前に、table_rewriteイベントは発生します。CLUSTERやVACUUMのような他の制御文でもテーブルを書き換えられますが、それらではtable_rewriteイベントは引き起こされません。

イベントトリガは(他の関数のように)中断したトランザクションでは実行されません。従って、DDLコマンドがエラーで失敗した場合、関連するddl_command_endトリガは実行されません。逆に、もしddl_command_startトリガがエラーで失敗した場合、他のイベントトリガは起動されず、コマンド自体も実行されません。同様に、もしddl_command_endトリガがエラーで失敗した場合、それを含むトランザクションが失敗した場合のようにDDL文はロールバックされます。

イベントトリガでサポートされているコマンドの一覧は、39.2を参照してください。

イベントトリガは、コマンドCREATE EVENT TRIGGERを使用して作成されます。イベントトリガを作成するために、まず特別な型event_triggerを返す関数を作る必要があります。この関数は値を返す必要はありませ

ん。というのも、その戻り値型は単にシグナルとして、その関数がイベントトリガを呼び出していることを示しているだけだからです。

特定のイベントに対して複数のイベントトリガが定義された場合、トリガ名のアルファベット順で起動されます。

トリガ定義はWHEN条件で特定されます。そのため、例えばddl_command_startトリガはユーザが望む特定のコマンドのみを契機に実行させることができます。このようなトリガの一般的な使い方として、ユーザが実行するかもしれないDDL文の範囲を狭めることができます。

39.2. イベントトリガ起動マトリクス

表 39.1にて、イベントトリガがサポートするすべてのコマンドが参照できます。

表39.1 コマンドタグによりサポートされるイベントトリガ

コマンドタグ	ddl_command_start	ddl_command_end	sql_drop	table_rewrite	注意
ALTER AGGREGATE	X	X	-	-	
ALTER COLLATION	X	X	-	-	
ALTER CONVERSION	X	X	-	-	
ALTER DOMAIN	X	X	-	-	
ALTER DEFAULT PRIVILEGES	X	X	-	-	
ALTER EXTENSION	X	X	-	-	
ALTER FOREIGN DATA WRAPPER	X	X	-	-	
ALTER FOREIGN TABLE	X	X	X	-	
ALTER FUNCTION	X	X	-	-	
ALTER LANGUAGE	X	X	-	-	
ALTER LARGE OBJECT	X	X	-	-	
ALTER MATERIALIZED VIEW	X	X	-	-	
ALTER OPERATOR	X	X	-	-	
ALTER OPERATOR CLASS	X	X	-	-	
ALTER OPERATOR FAMILY	X	X	-	-	
ALTER POLICY	X	X	-	-	
ALTER PROCEDURE	X	X	-	-	
ALTER PUBLICATION	X	X	-	-	
ALTER SCHEMA	X	X	-	-	
ALTER SEQUENCE	X	X	-	-	
ALTER SERVER	X	X	-	-	
ALTER STATISTICS	X	X	-	-	

コマンドタグ	ddl_command_start	ddl_command_end	sql_drop	table_rewrite	注意
ALTER SUBSCRIPTION	X	X	-	-	
ALTER TABLE	X	X	X	X	
ALTER TEXT SEARCH CONFIGURATION	X	X	-	-	
ALTER TEXT SEARCH DICTIONARY	X	X	-	-	
ALTER TEXT SEARCH PARSER	X	X	-	-	
ALTER TEXT SEARCH TEMPLATE	X	X	-	-	
ALTER TRIGGER	X	X	-	-	
ALTER TYPE	X	X	-	X	
ALTER USER MAPPING	X	X	-	-	
ALTER VIEW	X	X	-	-	
COMMENT	X	X	-	-	ローカルオブジェクトに対してのみ
CREATE ACCESS METHOD	X	X	-	-	
CREATE AGGREGATE	X	X	-	-	
CREATE CAST	X	X	-	-	
CREATE COLLATION	X	X	-	-	
CREATE CONVERSION	X	X	-	-	
CREATE DOMAIN	X	X	-	-	
CREATE EXTENSION	X	X	-	-	
CREATE FOREIGN DATA WRAPPER	X	X	-	-	
CREATE FOREIGN TABLE	X	X	-	-	
CREATE FUNCTION	X	X	-	-	
CREATE INDEX	X	X	-	-	
CREATE LANGUAGE	X	X	-	-	
CREATE MATERIALIZED VIEW	X	X	-	-	
CREATE OPERATOR	X	X	-	-	
CREATE OPERATOR CLASS	X	X	-	-	
CREATE OPERATOR FAMILY	X	X	-	-	
CREATE POLICY	X	X	-	-	
CREATE PROCEDURE	X	X	-	-	
CREATE PUBLICATION	X	X	-	-	
CREATE RULE	X	X	-	-	
CREATE SCHEMA	X	X	-	-	
CREATE SEQUENCE	X	X	-	-	

コマンドタグ	ddl_command_start	ddl_command_end	sql_drop	table_rewrite	注意
CREATE SERVER	X	X	-	-	
CREATE STATISTICS	X	X	-	-	
CREATE SUBSCRIPTION	X	X	-	-	
CREATE TABLE	X	X	-	-	
CREATE TABLE AS	X	X	-	-	
CREATE TEXT SEARCH CONFIGURATION	X	X	-	-	
CREATE TEXT SEARCH DICTIONARY	X	X	-	-	
CREATE TEXT SEARCH PARSER	X	X	-	-	
CREATE TEXT SEARCH TEMPLATE	X	X	-	-	
CREATE TRIGGER	X	X	-	-	
CREATE TYPE	X	X	-	-	
CREATE USER MAPPING	X	X	-	-	
CREATE VIEW	X	X	-	-	
DROP ACCESS METHOD	X	X	X	-	
DROP AGGREGATE	X	X	X	-	
DROP CAST	X	X	X	-	
DROP COLLATION	X	X	X	-	
DROP CONVERSION	X	X	X	-	
DROP DOMAIN	X	X	X	-	
DROP EXTENSION	X	X	X	-	
DROP FOREIGN DATA WRAPPER	X	X	X	-	
DROP FOREIGN TABLE	X	X	X	-	
DROP FUNCTION	X	X	X	-	
DROP INDEX	X	X	X	-	
DROP LANGUAGE	X	X	X	-	
DROP MATERIALIZED VIEW	X	X	X	-	
DROP OPERATOR	X	X	X	-	
DROP OPERATOR CLASS	X	X	X	-	
DROP OPERATOR FAMILY	X	X	X	-	
DROP OWNED	X	X	X	-	
DROP POLICY	X	X	X	-	
DROP PROCEDURE	X	X	X	-	
DROP PUBLICATION	X	X	X	-	

コマンドタグ	ddl_command_start	ddl_command_end	sql_drop	table_rewrite	注意
DROP RULE	X	X	X	-	
DROP SCHEMA	X	X	X	-	
DROP SEQUENCE	X	X	X	-	
DROP SERVER	X	X	X	-	
DROP STATISTICS	X	X	X	-	
DROP SUBSCRIPTION	X	X	X	-	
DROP TABLE	X	X	X	-	
DROP TEXT SEARCH CONFIGURATION	X	X	X	-	
DROP TEXT SEARCH DICTIONARY	X	X	X	-	
DROP TEXT SEARCH PARSER	X	X	X	-	
DROP TEXT SEARCH TEMPLATE	X	X	X	-	
DROP TRIGGER	X	X	X	-	
DROP TYPE	X	X	X	-	
DROP USER MAPPING	X	X	X	-	
DROP VIEW	X	X	X	-	
GRANT	X	X	-	-	ローカルオブジェクトに対してのみ
IMPORT FOREIGN SCHEMA	X	X	-	-	
REFRESH MATERIALIZED VIEW	X	X	-	-	
REVOKE	X	X	-	-	ローカルオブジェクトに対してのみ
SECURITY LABEL	X	X	-	-	ローカルオブジェクトに対してのみ
SELECT INTO	X	X	-	-	

39.3. C言語によるイベントトリガ関数の書き方

本節ではトリガ関数とのインタフェースについて低レベルな詳細を説明します。この情報はC言語でトリガ関数を作成する時にのみ必要です。高レベルな言語で作成すれば、こうした詳細は代わりに扱ってもらえます。たいていの場合、Cでトリガを作成する前に手続き言語を使用することを検討すべきです。各手続き言語の文書で、その言語を使用したイベントトリガの作成方法を説明します。

トリガ関数は「version 1」関数マネージャインタフェースを使わなくてはいけません。

関数がイベントトリガマネージャから呼び出される時は、通常の引数が渡されるのではなく、EventTriggerData構造体を指す「context」ポインタが渡されます。C関数は、イベントトリガマネージャから呼び出されたのかどうかを以下のマクロを実行することで検査することができます。

```
CALLED_AS_EVENT_TRIGGER(fcinfo)
```

これは以下に展開されます。

```
((fcinfo)->context != NULL && IsA((fcinfo)->context, EventTriggerData))
```

もしこれが真を返す場合、fcinfo->contextをEventTriggerData *型にキャストし、指されたEventTriggerData構造体を使用することは安全です。その関数は、TriggerData構造体やそれが指すどのようなデータも変更してはいけません。

struct EventTriggerDataはcommands/event_trigger.hの中で定義されています。

```
typedef struct EventTriggerData
{
    NodeTag     type;

    const char *event;      /* イベント名 */
    Node       *parsetree;  /* 解析ツリー */
    CommandTag  tag;        /* コマンドタグ */
} EventTriggerData;
```

メンバは下記のように定義されています。

type

常にT_EventTriggerDataです。

event

その関数が呼び出されたイベント、"ddl_command_start"、"ddl_command_end"、"sql_drop"、"table_rewrite"のうちの1つを記述します。これらのイベントの内容は、39.1を参照してください。

parsetree

コマンドの解析ツリーへのポインタです。詳細はPostgreSQLのソースコードを確認してください。解析ツリーの構造は予告なく変更されることがあります。

tag

イベントトリガの実行対象となるイベントに関連するコマンドタグです。たとえば、"CREATE FUNCTION"です。

イベントトリガ関数はNULLポインタ（SQLのNULLではありません。したがって、isNullは真にはなりません）を返さなければなりません。

39.4. 完全なイベントトリガの例

C言語で作成したイベントトリガ関数に関するとても簡単な例をここに示します（手続き言語で作成したトリガの例は、その手続き言語の文書に記載されています）。

noddl関数は、呼ばれるたびに例外を発生させます。このイベントトリガは、この関数とddl_command_startイベントを関連づけます。そのため、(39.1で言及した例外はありますが例外を含む)すべてのDDLコマンドは、実行できません。

以下がトリガ関数のソースコードです。

```
#include "postgres.h"
#include "commands/event_trigger.h"


PG_MODULE_MAGIC;

PG_FUNCTION_INFO_V1(noddl);

Datum
noddl(PG_FUNCTION_ARGS)
{
    EventTriggerData *trigdata;


    if (!CALLED_AS_EVENT_TRIGGER(fcinfo))  /* 内部エラー */
        elog(ERROR, "not fired by event trigger manager");

    trigdata = (EventTriggerData *) fcinfo->context;

    ereport(ERROR,
        (errcode(ERRCODE_INSUFFICIENT_PRIVILEGE),
                 errmsg("command \"%s\" denied", trigdata->tag)));

    PG_RETURN_NULL();
}
```

ソースコードをコンパイル(37.10.5を参照してください)した後に、以下の様に関数とトリガを宣言します。

```
CREATE FUNCTION noddl() RETURNS event_trigger
    AS 'noddl' LANGUAGE C;

CREATE EVENT TRIGGER noddl ON ddl_command_start
    EXECUTE FUNCTION noddl();
```

これで、トリガの操作を確認することができます。

```
=# \dy
                     List of event triggers
 Name  |       Event       | Owner | Enabled | Function | Tags
-------+-------------------+-------+---------+----------+------
 noddl | ddl_command_start | dim   | enabled | noddl    |
```

```
(1 row)

=# CREATE TABLE foo(id serial);
ERROR:  command "CREATE TABLE" denied
```

この状況では、DDLコマンドを必要なときに実行できるようにするには、このイベントトリガを削除するか、無効化しなければなりません。以下のように、トランザクションの期間中だけトリガを無効化するのが、便利かもしれません。

```
BEGIN;
ALTER EVENT TRIGGER noddl DISABLE;
CREATE TABLE foo (id serial);
ALTER EVENT TRIGGER noddl ENABLE;
COMMIT;
```

(イベントトリガ自体が関係するDDLコマンドは、イベントトリガの影響を受けないことを思い出してください。)

39.5. テーブル書き換えイベントトリガの例

table_rewriteイベントのおかげで、メンテナンスウィンドウでの書き換えを許可するだけでテーブル書き換えポリシーを実装できます。

これが、そのようなポリシーを実装した例です。

```
CREATE OR REPLACE FUNCTION no_rewrite()
 RETURNS event_trigger
 LANGUAGE plpgsql AS
$$
---

--- ローカルテーブル書き換えポリシーの実装:
---   public.fooは書き換えが許可されていません
---   その他のテーブルは100ブロック以下であれば、
---   午前1時から午前6時までの間だけ書き換えが許可されます
---
DECLARE
  table_oid oid := pg_event_trigger_table_rewrite_oid();
  current_hour integer := extract('hour' from current_time);
  pages integer;
  max_pages integer := 100;
BEGIN
  IF pg_event_trigger_table_rewrite_oid() = 'public.foo'::regclass
  THEN
        RAISE EXCEPTION 'you''re not allowed to rewrite the table %',
                        table_oid::regclass;
```

```
  END IF;

  SELECT INTO pages relpages FROM pg_class WHERE oid = table_oid;
  IF pages > max_pages
  THEN
        RAISE EXCEPTION 'rewrites only allowed for table with less than % pages',
                        max_pages;
  END IF;

  IF current_hour NOT BETWEEN 1 AND 6
  THEN
        RAISE EXCEPTION 'rewrites only allowed between 1am and 6am';
  END IF;
END;
$$;

CREATE EVENT TRIGGER no_rewrite_allowed
                  ON table_rewrite
   EXECUTE FUNCTION no_rewrite();
```

第40章 ルールシステム

本章ではPostgreSQLのルールシステムについて説明します。本番で稼働するルールシステムは概念としては単純ですが、実際に使ってみると、わかりにくいところが少なからずあります。

通常それらはストアドプロシージャとトリガですが、他のいくつかのデータベースシステムは能動的データベースルールを定義しています。PostgreSQLでは関数とトリガとして実装されています。

ルールシステム（より正確に言うと問い合わせ書き換えルールシステム）はストアドプロシージャとトリガとはまったく異なります。ルールシステムはルールを参照して問い合わせを修正し、修正した問い合わせを、計画作成と実行のために問い合わせプランナに渡します。これは非常に強力なため、問い合わせ言語プロシージャ、ビューあるいはバージョンなど多くのパターンで使用することができます。このルールシステムの基礎理論と能力は[ston90b]および[ong90]で解説されています。

40.1. 問い合わせツリーとは

どのようにルールシステムが機能するかを理解するためには、ルールがどのように起動され、その入力と結果は何かを理解しなければなりません。

ルールシステムは問い合わせパーサとプランナの中間に位置します。ルールシステムは、入力としてパーサの出力、単一の問い合わせツリー、および何らかの特別な情報を持つ問い合わせツリーでもあるユーザ定義の書き換えルールを取り、結果として0個以上の問い合わせツリーを生成します。ルールシステムの入力と出力は常にパーサ自体でも生成することができるもので、参照する対象は基本的にSQL文として表現できるものです。

では問い合わせツリーとは何でしょうか。それは、SQL文を構成する個々の部品を別々に記憶した、SQL文の内部表現です。debug_print_parse、debug_print_rewritten、もしくはdebug_print_plan設定パラメータを設定していれば、サーバログ内で問い合わせツリーを見ることができます。ルールアクションもpg_rewriteシステムカタログ内に問い合わせツリーとして格納されています。これはログ出力のように整形されていませんが、まったく同じ情報を持っています。

問い合わせツリーそのものを読むためにはある程度の経験が必要です。ルールシステムを理解するためには問い合わせツリーのSQL表現で十分ですので、ここではその読み方までは教えません。

本章の問い合わせツリーのSQL表現形式を読む時に必要なのは、問い合わせツリー構造の中に分解された、ある文の部品を識別できることです。問い合わせツリーには以下の部品があります。

コマンド種類

これはどのコマンド（SELECT、INSERT、UPDATE、DELETE）が構文解析ツリーを作ったかを示す単純な値です。

範囲テーブル

範囲テーブルは問い合わせで使われるリレーションのリストです。SELECT文ではこれはFROMキーワードの後で与えられるリレーションになります。

範囲テーブルのそれぞれの項目はテーブルもしくはビューを識別し、問い合わせの別の部品ではどんな名前で呼び出されるかを示します。問い合わせツリーでは範囲テーブルの項目は名前よりも番号で参照

されることが多いため、ここではSQL文とは違い、重複する名前があるかということは問題になりません。これはルールの範囲テーブルがマージされた後に起こる可能性があります。本章の例ではその状況を含んでいません。

結果リレーション

問い合わせの結果が格納されるリレーションを識別する範囲テーブルへのインデックスです。

SELECT問い合わせは結果リレーションを持ちません。(SELECT INTOの場合は特別ですが、INSERT ... SELECTが付いたCREATE TABLEとほぼ同じですので、ここでは個別には説明しません。)

INSERT、UPDATE、DELETEコマンドでは、結果リレーションは変更が有効になるテーブル(もしくはビュー)です。

目的リスト

目的リストは問い合わせの結果を定義する式のリストです。SELECTの場合、この式は問い合わせの最終結果を構築するものです。これらはSELECTとFROMキーワードの間にある式に対応します (*は単にリレーションの全ての列名の省略です。これはパーサによって個別の列に展開されますので、ルールシステムが見ることはありません)。

DELETEコマンドは結果を返しませんので、通常の目的リストは必要ありません。その代わり、プランナは空の目的リストに特別なCTID項目を追加し、エクゼキュータが削除すべき行を見つけられるようにします。(CTIDは結果リレーションが通常のテーブルの場合に追加されます。もしビューであれば40.2.4で述べるように、代わりに行全体の変数がルールシステムによって追加されます。)

INSERT問い合わせでは、目的リストは結果リレーションへ入る新規の行を示します。これはVALUES句かINSERT ... SELECTの中のSELECT句の式です。書き換え処理の最初のステップでは、元の問い合わせでは割り当てられず、デフォルト値となっている列の目的リストの項目を追加します。残った列(値が与えられていない列、かつデフォルト値を持たない列)は全て、プランナによって定数NULL式で埋められます。

UPDATEコマンドでは、目的リストは古いものを置き換えるべき新しい行を示します。ルールシステムではコマンド内のSET column = expression部分にある式だけを持っています。プランナは、古い行から新しい行へ値をコピーする式を挿入することにより、抜けている列を処理します。DELETEの場合と同様、エクゼキュータが更新すべき行を見つけられるように、CTIDもしくは行全体の変数が追加されます。

目的リストの各項目は、定数値、範囲テーブル内のリレーション中の1つの列を指し示す変数、パラメータ等の式を保持するか、または、関数呼び出し、定数、変数、演算子などにより作られた式のツリーを保持します。

条件

問い合わせの条件は目的リストの項目に含まれている式によく似た式です。この式の結果は、最終的な結果の行を得るための(INSERT、UPDATE、DELETEまたはSELECT)演算を実行すべきかどうかを示すブール値です。それはSQL文の中のWHERE句に対応します。

結合ツリー

問い合わせの結合ツリーはFROM句の構造を表します。SELECT ... FROM a, b, cのような単純な問い合わせでは、結合ツリーは単なるFROM項目のリストです。なぜならこれらはどんな順番で結合しても構

わないためです。しかしJOIN式、特に外部結合が使われた場合は、その結合が示す順番通りに結合しなければいけません。この場合結合ツリーはJOIN式の構造を表します。特定のJOIN句と関連付けられた制約（ONもしくはUSING式からのもの）はこれらの結合ツリーノードに付加された条件として格納されます。頂点レベルのWHERE式を頂点レベルの結合ツリー項目に付加された条件として格納することも便利です。ですから、結合ツリーはSELECTのFROM句とWHERE句の両方を表しているわけです。

その他

ORDER BY句のような、問い合わせツリーのその他の部品は、ここでは取り上げません。ルールシステムはルールを適用している時にそこで項目を入れ替えることもありますが、これはルールシステムの基本とはあまり関係しません。

40.2. ビューとルールシステム

PostgreSQLにおけるビューはルールシステムを使って実装されています。実際、

```
CREATE VIEW myview AS SELECT * FROM mytab;
```

と

```
CREATE TABLE myview (same column list as mytab);
CREATE RULE "_RETURN" AS ON SELECT TO myview DO INSTEAD
    SELECT * FROM mytab;
```

の2つのコマンドの間には基本的な違いはありません。と言うのは、CREATE VIEWコマンドによって内部的にまったく同じコマンドが行われるからです。これには副作用もあります。その1つはPostgreSQLシステムカタログのビューについての情報はテーブルの情報とまったく同一であるということです。そのため、パーサにとってはテーブルとビューの間に違いは一切ありません。これらは同じもの、つまりリレーションです。

40.2.1. SELECTルールの動き

たとえコマンドがINSERT、UPDATE、DELETEなどであっても、ON SELECTルールは全ての問い合わせに対し最後に適用されます。そして、このルールは他のコマンド種類のルールと異なるセマンティクスを持っていて、問い合わせツリーを新規に生成せずに、そこにあるものを修正します。したがってSELECTルールを一番初めに記述します。

現在のところ、ON SELECTルールでは1つのアクションしか許されず、それはINSTEADである無条件のSELECTアクションでなければいけません。この制約は、一般のユーザが何をしても、ルールシステムが堅牢であるために必要であり、ON SELECTのルールはビュー同様の動作に限定されます。

本章の例として挙げているのは、ちょっとした演算をする2つの結合のビューと、次にこれらの機能を利用するいくつかのビューです。初めの2つのビューのうちの1つは、INSERT、UPDATE、DELETE操作に対するルールを後で追加することでカスタマイズされ、最終結果は何らかの魔法の機能によりあたかも実テーブルのように振舞うビューになります。初めて学ぶための例としては決して簡単ではなく先に進むことを躊躇させるかもしれませんが、多くの別々の例を持ち出して頭の混乱を招くよりも、全ての論点をステップごとに追う1つの例を挙げる方が良いでしょう。

最初の2つのルールシステムの説明で必要とする実テーブルを以下に示します。

```
CREATE TABLE shoe_data (
    shoename   text,          -- 主キー
    sh_avail   integer,       -- 在庫
    slcolor    text,          -- 望ましい靴紐の色
    slminlen   real,          -- 靴紐の最短サイズ
    slmaxlen   real,          -- 靴紐の最長サイズ
    slunit     text           -- 長さの単位
);

CREATE TABLE shoelace_data (
    sl_name    text,          -- 主キー
    sl_avail   integer,       -- 在庫
    sl_color   text,          -- 靴紐の色
    sl_len     real,          -- 靴紐の長さ
    sl_unit    text           -- 長さの単位
);

CREATE TABLE unit (
    un_name    text,          -- 主キー
    un_fact    real           -- cmに変換するファクタ
);
```

これでわかるかもしれませんが、これらは靴屋のデータを表しています。

ビューを以下のように作成します。

```
CREATE VIEW shoe AS
    SELECT sh.shoename,
           sh.sh_avail,
           sh.slcolor,
           sh.slminlen,
           sh.slminlen * un.un_fact AS slminlen_cm,
           sh.slmaxlen,
           sh.slmaxlen * un.un_fact AS slmaxlen_cm,
           sh.slunit
      FROM shoe_data sh, unit un
     WHERE sh.slunit = un.un_name;

CREATE VIEW shoelace AS
    SELECT s.sl_name,
           s.sl_avail,
           s.sl_color,
           s.sl_len,
           s.sl_unit,
```

```
           s.sl_len * u.un_fact AS sl_len_cm
      FROM shoelace_data s, unit u
     WHERE s.sl_unit = u.un_name;

CREATE VIEW shoe_ready AS
    SELECT rsh.shoename,
           rsh.sh_avail,
           rsl.sl_name,
           rsl.sl_avail,
           least(rsh.sh_avail, rsl.sl_avail) AS total_avail
      FROM shoe rsh, shoelace rsl
     WHERE rsl.sl_color = rsh.slcolor
       AND rsl.sl_len_cm >= rsh.slminlen_cm
       AND rsl.sl_len_cm <= rsh.slmaxlen_cm;
```

shoelaceビュー（今ある一番簡単なビュー）用のCREATE VIEWコマンドは、shoelaceリレーションと、問い合わせ範囲テーブルの中でshoelaceリレーションが参照される時はいつでも、適用されるべき書き換えルールの存在を示す項目をpg_rewriteに作ります。ルールはルール条件（SELECTルールは現在持つことができませんので、非SELECTルールのところで取り上げます）を持たないINSTEADです。ルール条件は問い合わせ条件とは異なることに注意してください！ ルールアクションは問い合わせ条件を持っています。このルールアクションは、ビュー作成コマンド内のSELECTのコピーである、1つの問い合わせツリーです。

> **注記**
>
> pg_rewrite項目のNEWとOLDに対する2つの特別な範囲テーブル項目はSELECTルールには関係ありません。

ではここでunit、shoe_data、shoelace_dataにデータを入れ、ビューに簡単な問い合わせを行います。

```
INSERT INTO unit VALUES ('cm', 1.0);
INSERT INTO unit VALUES ('m', 100.0);
INSERT INTO unit VALUES ('inch', 2.54);

INSERT INTO shoe_data VALUES ('sh1', 2, 'black', 70.0, 90.0, 'cm');
INSERT INTO shoe_data VALUES ('sh2', 0, 'black', 30.0, 40.0, 'inch');
INSERT INTO shoe_data VALUES ('sh3', 4, 'brown', 50.0, 65.0, 'cm');
INSERT INTO shoe_data VALUES ('sh4', 3, 'brown', 40.0, 50.0, 'inch');

INSERT INTO shoelace_data VALUES ('sl1', 5, 'black', 80.0, 'cm');
INSERT INTO shoelace_data VALUES ('sl2', 6, 'black', 100.0, 'cm');
INSERT INTO shoelace_data VALUES ('sl3', 0, 'black', 35.0 , 'inch');
INSERT INTO shoelace_data VALUES ('sl4', 8, 'black', 40.0 , 'inch');
INSERT INTO shoelace_data VALUES ('sl5', 4, 'brown', 1.0 , 'm');
INSERT INTO shoelace_data VALUES ('sl6', 0, 'brown', 0.9 , 'm');
INSERT INTO shoelace_data VALUES ('sl7', 7, 'brown', 60 , 'cm');
INSERT INTO shoelace_data VALUES ('sl8', 1, 'brown', 40 , 'inch');
```

```
SELECT * FROM shoelace;

 sl_name   | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
-----------+----------+----------+--------+---------+-----------
 sl1       |        5 | black    |     80 | cm      |        80
 sl2       |        6 | black    |    100 | cm      |       100
 sl7       |        7 | brown    |     60 | cm      |        60
 sl3       |        0 | black    |     35 | inch    |      88.9
 sl4       |        8 | black    |     40 | inch    |     101.6
 sl8       |        1 | brown    |     40 | inch    |     101.6
 sl5       |        4 | brown    |      1 | m       |       100
 sl6       |        0 | brown    |    0.9 | m       |        90
(8 rows)
```

これは、ビューに対する最も簡単なSELECTですので、この機会にビュールールの基本を説明します。SELECT * FROM shoelaceはパーサによって処理され、次の問い合わせツリーが生成されます。

```
SELECT shoelace.sl_name, shoelace.sl_avail,
       shoelace.sl_color, shoelace.sl_len,
       shoelace.sl_unit, shoelace.sl_len_cm
  FROM shoelace shoelace;
```

このツリーがルールシステムに伝えられます。ルールシステムは範囲テーブルを参照し、何らかのリレーションに対してルールが存在するか調べます。shoelace(現時点では唯一のビュー)についての範囲テーブル項目を処理する際、問い合わせツリーで_RETURNルールを検出します。

```
SELECT s.sl_name, s.sl_avail,
       s.sl_color, s.sl_len, s.sl_unit,
       s.sl_len * u.un_fact AS sl_len_cm
  FROM shoelace old, shoelace new,
       shoelace_data s, unit u
 WHERE s.sl_unit = u.un_name;
```

ビューを展開するために、リライタは単純にルールのアクション問い合わせツリーを持つ副問い合わせ範囲テーブルの項目を作り、ビューを参照していた元の範囲テーブルを置き換えます。書き換えられた結果の問い合わせツリーは、以下のように入力した場合とほぼ同じです。

```
SELECT shoelace.sl_name, shoelace.sl_avail,
       shoelace.sl_color, shoelace.sl_len,
       shoelace.sl_unit, shoelace.sl_len_cm
  FROM (SELECT s.sl_name,
               s.sl_avail,
               s.sl_color,
               s.sl_len,
               s.sl_unit,
               s.sl_len * u.un_fact AS sl_len_cm
```

```
          FROM shoelace_data s, unit u
         WHERE s.sl_unit = u.un_name) shoelace;
```

しかし1つだけ違いがあります。副問い合わせの範囲テーブルが2つの余分な項目shoelace oldとshoelace newを持っていることです。これらの項目は副問い合わせの結合ツリーや目的リストで参照されませんので、直接問い合わせでは使われません。リライタはそれらを使用して、ビューを参照した範囲テーブルの項目に元々存在したアクセス権限確認情報を格納します。この方法で、書き換えられた問い合わせで直接ビューを使用していなくても、エクゼキュータはユーザがそのビューにアクセスするための正しい権限を持っているか確認します。

これが最初に適用されるルールです。ルールシステムは最上位の問い合わせの残り(この例ではこれ以上ありません)の範囲テーブルの項目をチェックし続けます。そしてルールシステムは、追加された副問い合わせの範囲テーブルの項目がビューを参照するかを再帰的に確認します(しかしoldやnewは展開しません。そうでなければ無限再帰になってしまいます!)。この例ではshoelace_dataやunit用の書き換えルールはありません。ですから書き換えは完結し、上記がプランナに渡される最終的な結果となります。

さて、店に置いてある靴紐(の色とサイズ)に一致する靴が店にあるか、完全に一致する靴紐の在庫数が2以上あるかどうかを把握する問い合わせを書いてみましょう。

```
SELECT * FROM shoe_ready WHERE total_avail >= 2;

 shoename | sh_avail | sl_name | sl_avail | total_avail
----------+----------+---------+----------+-------------
 sh1      |        2 | sl1     |        5 |           2
 sh3      |        4 | sl7     |        7 |           4
(2 rows)
```

今回のパーサの出力は以下の問い合わせツリーです。

```
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
  FROM shoe_ready shoe_ready
 WHERE shoe_ready.total_avail >= 2;
```

最初に適用されるルールはshoe_readyビュー用のもので、問い合わせツリーにおける結果は以下のようになります。

```
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
  FROM (SELECT rsh.shoename,
               rsh.sh_avail,
               rsl.sl_name,
               rsl.sl_avail,
               least(rsh.sh_avail, rsl.sl_avail) AS total_avail
          FROM shoe rsh, shoelace rsl
         WHERE rsl.sl_color = rsh.slcolor
```

```
           AND rsl.sl_len_cm >= rsh.slminlen_cm
           AND rsl.sl_len_cm <= rsh.slmaxlen_cm) shoe_ready
 WHERE shoe_ready.total_avail >= 2;
```

同じように、shoeとshoelace用のルールは副問い合わせの範囲テーブルとして代用され、3レベルの最終問い合わせツリーへと導きます。

```
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
  FROM (SELECT rsh.shoename,
               rsh.sh_avail,
               rsl.sl_name,
               rsl.sl_avail,
               least(rsh.sh_avail, rsl.sl_avail) AS total_avail
          FROM (SELECT sh.shoename,
                       sh.sh_avail,
                       sh.slcolor,
                       sh.slminlen,
                       sh.slminlen * un.un_fact AS slminlen_cm,
                       sh.slmaxlen,
                       sh.slmaxlen * un.un_fact AS slmaxlen_cm,
                       sh.slunit
                  FROM shoe_data sh, unit un
                 WHERE sh.slunit = un.un_name) rsh,
               (SELECT s.sl_name,
                       s.sl_avail,
                       s.sl_color,
                       s.sl_len,
                       s.sl_unit,
                       s.sl_len * u.un_fact AS sl_len_cm
                  FROM shoelace_data s, unit u
                 WHERE s.sl_unit = u.un_name) rsl
         WHERE rsl.sl_color = rsh.slcolor
           AND rsl.sl_len_cm >= rsh.slminlen_cm
           AND rsl.sl_len_cm <= rsh.slmaxlen_cm) shoe_ready
 WHERE shoe_ready.total_avail > 2;
```

これは非効率的に見えるかもしれませんが、プランナは副問い合わせを「引っ張り上げること」で、これを単一レベルの問い合わせツリーに縮めてから、手で書き出したかのように結合を計画します。そのため、問い合わせツリーを縮めるという最適化を、書き換えシステム自身で意識する必要はありません。

40.2.2. 非SELECT文のビュールール

これまでのビュールールの説明では問い合わせツリーの2つの詳細について触れませんでした。それらは、コマンドタイプと結果リレーションです。実際、コマンドタイプはビュールールでは必要とされませんが、結果リ

レーションがビューの場合には特別な考慮が必要ですので、結果リレーションは問い合わせリライタの動作に影響するかもしれません。

SELECTと他のコマンドに対する問い合わせツリーの間には大きな違いはありません。明らかに、それらは違うコマンドタイプを持っていて、SELECT以外のコマンドでは、結果リレーションは結果の格納先となる範囲テーブルの項目を指し示します。それ以外ではまったく同じです。ですから、aとbの列を持つテーブルt1、t2に対する以下の2つの文の問い合わせツリーは、ほとんど同じです。

```
SELECT t2.b FROM t1, t2 WHERE t1.a = t2.a;

UPDATE t1 SET b = t2.b FROM t2 WHERE t1.a = t2.a;
```

以下に、具体的に示します。

- 範囲テーブルには、テーブルt1とt2に対する項目があります。
- 目的リストにはテーブルt2に対する範囲テーブル項目のb列を指し示す1つの変数があります。
- 条件式は、範囲テーブルの両項目のa列の等価性を比較します。
- 結合ツリーはt1とt2の単純な結合を表しています。

結果として、両方の問い合わせツリーは似たような実行計画になります。それらはともに2つのテーブルの結合です。UPDATEではt1から抜けている列はプランナによって目的リストに追加され、最終の問い合わせツリーは、以下のようになります。

```
UPDATE t1 SET a = t1.a, b = t2.b FROM t2 WHERE t1.a = t2.a;
```

そして、結合を実行したエクゼキュータは、

```
SELECT t1.a, t2.b FROM t1, t2 WHERE t1.a = t2.a;
```

の結果集合とまったく同じ結果集合を作成します。とは言ってもUPDATEにはちょっとした問題があります。結合を行うエクゼキュータの計画の部分は、結合の結果が何に向けられているかに関与しません。エクゼキュータは単に結果となる行の集合を作成するだけです。1つはSELECTコマンドでもう1つはUPDATEコマンドであるという事実は、エクゼキュータの中のより上位で扱われます。そこでは、これがUPDATEであるとわかっていて、この結果がテーブルt1に入らなければいけないことを知っています。しかし、そこにあるどの行が新しい行によって置換されなければならないのでしょうか。

この問題を解決するため、UPDATE文（DELETE文の場合も同様）の目的リストに別の項目が付け加えられます。それは、現在のタプルID（CTID）です。これはその行のファイルブロック番号とブロック中の位置を持つシステム列です。テーブルがわかっている場合、CTIDを使用して、元のt1行を抽出して更新することができます。CTIDを目的リストに追加すると、問い合わせは以下のようになります。

```
SELECT t1.a, t2.b, t1.ctid FROM t1, t2 WHERE t1.a = t2.a;
```

では、PostgreSQLの別の詳細説明に入りましょう。テーブルの行は上書きされませんので、ROLLBACK処理は速いのです。UPDATEでは、（CTIDを取り除いた後）テーブルに新しい結果行が挿入され、CTIDが指し示す古

い行の行ヘッダ内のcmaxとxmax項目は現在のコマンドカウンタと現在のトランザクションIDに設定されます。このようにして、古い行は隠され、トランザクションがコミットされた後、vacuum掃除機が不必要になった行をそのうちに削除できます。

これらの詳細が全部理解できれば、どんなコマンドに対してもまったく同じようにしてビューのルールを簡単に適用することができます。そこには差異がありません。

40.2.3. PostgreSQLにおけるビューの能力

ここまでで、ルールシステムがどのようにビューの諸定義を元の問い合わせツリーに組み入れるかを解説しました。第2の例では、1つのビューからの単純なSELECTによって、最終的に4つのテーブルを結合する問い合わせツリーが生成されました(unitは違った名前で2回使われました)。

ビューをルールシステムで実装する利点は、どのテーブルをスキャンすべきか、それらのテーブル間の関連性、ビューからの制約条件、元の問い合わせ条件に関する情報を全て、プランナが1つの問い合わせツリーの中に持っていることです。元の問い合わせが既にビューに対する結合である時も同様です。プランナはここでどれが問い合わせ処理の最適経路かを決定しなければなりません。プランナは保持する情報が多ければ多いほど、より良い決定を下すことができます。そしてPostgreSQLに実装されているルールシステムはこれが現時点で、提供されている全ての情報であることを保証します。

40.2.4. ビューの更新について

ビューがINSERT、UPDATE、DELETEなどの目的リレーションとして名付けられた場合はどうなるのでしょうか?上で説明したような置換をすると、結果リレーションが副問い合わせの範囲テーブル項目を指す問い合わせツリーができてしまい、それは上手く機能しません。しかし、いくつかのケースではPostgreSQLはビューの更新をサポートする事ができます。

副問い合わせが単一のテーブルを参照しかつ十分に単純である時、リライタは副問い合わせを被参照テーブルに自動的に置き換え、したがって、INSERT、UPDATEあるいはDELETEを適切な方法で被参照テーブルに適用する事ができます。この場合の「十分に単純」であるとは自動的に更新可能ある事です。より詳細な自動的に更新可能なビューの情報については、CREATE VIEWを参照してください。

もう一つの方法として、ビューに対するユーザ定義のINSTEAD OFトリガによってこれらのコマンドを処理する事ができます。この場合、書き換えは少々違う形で行われます。INSERTに対しては、リライタはビューに全く何もせず、問い合わせの結果リレーションをそのままにします。UPDATEとDELETEに対しては、コマンドが更新もしくは削除しようとする「古い」行を生成するためにビュー問い合わせを展開する必要がまだあります。そのため、ビューは通常通り展開されますが、もう一つの展開されない範囲テーブル項目が結果リレーションとしてビューを表す問い合わせに追加されます。

ここで起こる問題はビューで更新される行をどのように特定するかということです。結果リレーションがテーブルの場合、更新する行の物理的な位置を特定するために特別なCTID項目が目的リストに追加されることを思い出して下さい。ビューの行には実際の物理的な位置がないため、ビューにはCTIDがありませんので、これは結果リレーションがビューの場合には上手くいきません。その代わり、UPDATEやDELETE操作では、特別な行全体の項目が目的リストに追加されていて、それはビューからすべての列を含むように展開されています。エクゼキュータはこの値を使って「古い」行をINSTEAD OFトリガに提供します。新旧の行の値に基づいて更新するものを計算するのはトリガの責任です。

別の方法としては、ビューに対するINSERT、UPDATE、DELETEコマンドに代替の動作を指定するINSTEADルールを定義する事です。これらのルールは、ビューではなくコマンドを、通常は1つもしくは複数のテーブルを更新するコマンドに書き換えます。それが40.4の論題になります。

ルールが最初に評価され、元の問い合わせが計画され実行される前にそれを書き換えることに注意して下さい。そのためビューにINSTEAD OFトリガとINSERTやUPDATEやDELETEに関するルールがあった場合、ルールが最初に評価され、その結果よってはトリガが全く使われないかもしれません。

単純なビューに対するINSERT、UPDATEあるいはDELETEコマンドの自動書き換えは常に最後に試みられます。したがって、ビューがルールもしくはトリガを持っていた場合、これらは更新可能ビューのデフォルト動作を上書きします。

ビューにINSTEADルールもINSTEAD OFトリガも定義されておらず、かつ、リライタが問い合わせを自動的に被参照テーブルへの更新に書き換える事ができなかった場合、エクゼキュータはビューを更新できませんのでエラーが発生します。

40.3. マテリアライズドビュー

PostgreSQLにおけるマテリアライズドビューはビューのようにルールシステムを使用しますが、あたかもテーブルであるかのような形態で結果を保持します。

```
CREATE MATERIALIZED VIEW mymatview AS SELECT * FROM mytab;
```

と

```
CREATE TABLE mymatview AS SELECT * FROM mytab;
```

の間の主な違いは、その後にマテリアライズドビューを直接更新できない事と、マテリアライズドビューを作成するために使われた問い合わせがビューと全く同様の方法で保持され、以下のコマンドを用いて最新のデータでマテリアライズドビューを再構築できる事です。

```
REFRESH MATERIALIZED VIEW mymatview;
```

マテリアライズドビューに関する情報はPostgreSQLシステムカタログでビューやテーブルに対するものと全く同様に保持されています。そのため、パーサにとってマテリアライズドビューはテーブルやビューと同じリレーションです。問い合わせでマテリアライズドビューが参照された時、あたかもテーブルのように、データはマテリアライズドビューから直接返されます。ルールはマテリアライズドビューにデータを投入する時にだけ使用されます。

多くの場合、マテリアライズドビューに格納されているデータの参照は、ビューを通して、あるいはビューから参照されているテーブルを直接参照するよりも高速ですが、データが常に最新であるとは限りません。ですが、時には最新のデータは必要でない事もあります。販売履歴を記録するテーブルの例を考えてみましょう。

```
CREATE TABLE invoice (
    invoice_no    integer        PRIMARY KEY,
```

```
    seller_no     integer,       -- 販売員のID
    invoice_date  date,          -- 販売日
    invoice_amt   numeric(13,2)  -- 販売量
);
```

もし利用者が過去の販売データを速やかにグラフ化可能であってほしいと考えるなら、彼らはデータの要約を望むのであって、最新のデータが不完全である事は気にしないでしょう。

```
CREATE MATERIALIZED VIEW sales_summary AS
  SELECT
      seller_no,
      invoice_date,
      sum(invoice_amt)::numeric(13,2) as sales_amt
    FROM invoice
    WHERE invoice_date < CURRENT_DATE
    GROUP BY
      seller_no,
      invoice_date
    ORDER BY
      seller_no,
      invoice_date;

CREATE UNIQUE INDEX sales_summary_seller
  ON sales_summary (seller_no, invoice_date);
```

このマテリアライズドビューは営業担当用に作成されるダッシュボードのグラフを表示するのにぴったりでしょう。以下のSQLを使った統計情報を更新するジョブを毎晩スケジュールしておくことができます。

```
REFRESH MATERIALIZED VIEW sales_summary;
```

それ以外のマテリアライズドビューの用途として、外部データラッパを通じてリモートシステムから取得したデータの高速化が挙げられます。以下の例はfile_fdwを用いた単純な例で、実行時間を含みますが、これはローカルシステムのキャッシュ機構を用いているため、リモートシステムへのアクセスと比較した違いの方がここで示したものより劇的です。マテリアライズドビューにはインデックスを設定することもできますが、file_fdwはインデックスをサポートしないことに注意してください。この有利な点は、他の種類の外部データアクセスには当てはまらないでしょう。

セットアップ:

```
CREATE EXTENSION file_fdw;
CREATE SERVER local_file FOREIGN DATA WRAPPER file_fdw;
CREATE FOREIGN TABLE words (word text NOT NULL)
  SERVER local_file
  OPTIONS (filename '/usr/share/dict/words');
CREATE MATERIALIZED VIEW wrd AS SELECT * FROM words;
CREATE UNIQUE INDEX wrd_word ON wrd (word);
CREATE EXTENSION pg_trgm;
```

```
CREATE INDEX wrd_trgm ON wrd USING gist (word gist_trgm_ops);
VACUUM ANALYZE wrd;
```

file_fdwを直接用いて字句のスペルチェックをしてみましょう。

```
SELECT count(*) FROM words WHERE word = 'caterpiler';

 count
-------
     0
(1 row)
```

EXPLAIN ANALYZEによれば以下の通りです:

```
Aggregate  (cost=21763.99..21764.00 rows=1 width=0) (actual time=188.180..188.181 rows=1 loops=1)
   ->  Foreign Scan on words  (cost=0.00..21761.41 rows=1032 width=0) (actual
 time=188.177..188.177 rows=0 loops=1)
         Filter: (word = 'caterpiler'::text)
         Rows Removed by Filter: 479829
         Foreign File: /usr/share/dict/words
         Foreign File Size: 4953699
 Planning time: 0.118 ms
 Execution time: 188.273 ms
```

代わりにマテリアライズドビューを使った場合、問い合わせは非常に速くなります。

```
Aggregate  (cost=4.44..4.45 rows=1 width=0) (actual time=0.042..0.042 rows=1 loops=1)
   ->  Index Only Scan using wrd_word on wrd  (cost=0.42..4.44 rows=1 width=0) (actual
 time=0.039..0.039 rows=0 loops=1)
         Index Cond: (word = 'caterpiler'::text)
         Heap Fetches: 0
 Planning time: 0.164 ms
 Execution time: 0.117 ms
```

どちらの場合でも、wordの綴りは間違っています。では、我々が望んでいたであろう結果を得るために、もう一度file_fdwとpg_trgmを使ってみます。（訳注：検索条件の正しい綴りは「caterpillar」）

```
SELECT word FROM words ORDER BY word <-> 'caterpiler' LIMIT 10;

     word
---------------
 cater
 caterpillar
 Caterpillar
 caterpillars
 caterpillar's
```

```
 Caterpillar's
 caterer
 caterer's
 caters
 catered
(10 rows)
```

```
Limit  (cost=11583.61..11583.64 rows=10 width=32) (actual time=1431.591..1431.594 rows=10
 loops=1)
   ->  Sort  (cost=11583.61..11804.76 rows=88459 width=32) (actual time=1431.589..1431.591
 rows=10 loops=1)
         Sort Key: ((word <-> 'caterpiler'::text))
         Sort Method: top-N heapsort  Memory: 25kB
         ->  Foreign Scan on words  (cost=0.00..9672.05 rows=88459 width=32) (actual
 time=0.057..1286.455 rows=479829 loops=1)
               Foreign File: /usr/share/dict/words
               Foreign File Size: 4953699
 Planning time: 0.128 ms
 Execution time: 1431.679 ms
```

マテリアライズドビューを使用した場合:

```
Limit  (cost=0.29..1.06 rows=10 width=10) (actual time=187.222..188.257 rows=10 loops=1)
   ->  Index Scan using wrd_trgm on wrd  (cost=0.29..37020.87 rows=479829 width=10) (actual
 time=187.219..188.252 rows=10 loops=1)
         Order By: (word <-> 'caterpiler'::text)
 Planning time: 0.196 ms
 Execution time: 198.640 ms
```

定期的にリモートのデータをローカルに更新せねばならない事を許容できるのであれば、代わりに性能上の便益を得られることでしょう。

40.4. INSERT、UPDATE、DELETEについてのルール

INSERT、UPDATE、DELETEに定義するルールは前節で解説したビューのルールとはまったく異なります。第一点として、これらのCREATE RULEコマンドでは以下を行うことができます。

- アクションがないルールも可能です。
- 複数のアクションを持てます。
- INSTEADもしくはALSO(デフォルト)を取ることができます。
- 疑似リレーションNEWとOLDが役立つようになります。
- ルール条件を持たせることができます。

第二点として、その場で問い合わせツリーを変更しません。その代わりに新規の0個以上の問い合わせツリーを生成して、オリジナルを破棄することができます。

> **注意**
>
> 多くの場合、INSERT/UPDATE/DELETEにおけるルールによって実行できるタスクは、トリガーで実行した方が良いでしょう。トリガーは概念としては少し複雑ですが、意味を理解するにはとても単純です。元の問い合わせにvolatile関数を含む場合、ルールは驚かせる結果を返すことがよくあります。（volatile関数はルールを遂行する過程で予期していた回数より多く実行されてしまうかもしれません）
>
> また、これらのタイプのルールが全くサポートしない場合もあります。特にWITH句を元の問い合わせに含む場合とUPDATE問い合わせのSETリストの中で複数列に代入するサブSELECTの場合です。これはルール問い合わせにこれらの構造を複製すると副問い合わせを複数回評価し、問い合わせの作者が表現したかった意図と異なる結果となるためです。

40.4.1. 更新ルールの動作

```
CREATE [ OR REPLACE ] RULE name AS ON event
    TO table [ WHERE condition ]
    DO [ ALSO | INSTEAD ] { NOTHING | command | ( command ; command ... ) }
```

上記の構文を覚えておいてください。以下では、*更新ルール*はINSERT、UPDATE、DELETEに定義されたルールを意味します。

更新ルールは、問い合わせツリーの結果リレーションとコマンド種類がCREATE RULEで与えられるオブジェクトとイベントと等しい場合にルールシステムによって適用されます。更新ルールに対してルールシステムは問い合わせツリーのリストを生成します。最初は問い合わせツリーリストは空です。0（NOTHINGキーワード）、1つまたは複数のアクションが有効です。簡単にするため、ここでは1つのアクションのルールを取り上げます。このルールは条件を持っていても持っていなくても構いませんし、またINSTEADかALSO（デフォルト）のいずれかを取ることができます。

ルール条件とはどんなものでしょうか。それはルールのアクションを行わなければならない時と、行ってはならない時を指定する条件です。基本的に（特別な意味合いを持った）オブジェクトとして与えられるリレーションであるNEW疑似リレーションかOLD疑似リレーション、または、その両者のみをこの条件は参照することができます。

1アクションのルールに対し、以下の問い合わせツリーを生成する3つの場合があります。

ALSOまたはINSTEADで条件がない場合。

　元の問い合わせツリーの条件が追加された、ルールアクションからの問い合わせツリー

条件付き、かつALSO

　ルール条件と元の問い合わせツリーの条件が追加された、ルールアクションからの問い合わせツリー

条件付き、かつINSTEAD

ルール条件と元の問い合わせツリーの条件が追加された、ルールアクションからの問い合わせツリーとルール条件の否定条件が追加された元の問い合わせツリー

最後に、もしルールがALSOの場合、変更されていない元の問い合わせツリーがリストに付け加えられます。条件付きのINSTEADルールのみが既に元の構文解析ツリーに追加をしているので、最後は1つのアクションを持つルールに対して1つもしくは2つの問い合わせツリーにたどり着きます。

ON INSERTルールでは、元の問い合わせは、(INSTEADにより止められていない限り)ルールによって追加されたアクションの前に実行されます。これにより、アクションは挿入された行(複数可)を参照することができます。しかし、ON UPDATEとON DELETEルールでは、元の問い合わせはルールによって追加されたアクションの後に実行されます。これは、アクションが更新される予定の、または削除される予定の行を参照できることを保証します。さもないと、条件に一致する行を見つけることができないためにアクションが作動しなくなる可能性が起こります。

ルールアクションで生成された問い合わせツリーは、再度書き換えシステムに渡され、より多くのルールの適用を受けてより多くのもしくは少ない問い合わせツリーになるかもしれません。ですから、ルールのアクションはルール自身とは異なるコマンド種類か、異なる結果リレーションを持っていなければなりません。さもないと、この再帰的処理により無限ループに陥ってしまいます。(ルールの再帰展開は検知され、エラーとして報告されます。)

pg_rewriteシステムカタログのアクションにある問い合わせツリーは単なるテンプレートです。これらはNEWとOLDに対する範囲テーブルの項目を参照することができるため、使用される前に何らかの置換措置がとられていなければなりません。NEWを参照する全てに対し、元の問い合わせの目的リストは対応する項目があるかどうか検索されます。項目が見つかった場合には、その項目式が参照を置き換えます。項目がなかった場合、NEWはOLDと同じ意味になる(UPDATEの場合)か、NULLによって置き換えられます(INSERTの場合)。OLDに対する参照は全て結果リレーションである範囲テーブルの項目への参照に置き換えられます。

更新ルールの適用が終わると、システムはそこで作られた構文解析ツリーにビュールールを適用します。ビューは、新しい更新アクションを挿入できないため、ビュー書き換えの結果に更新ルールを適用する必要はありません。

40.4.1.1. 最初のルール、ステップバイステップ

shoelace_dataリレーションのsl_avail列の変化を追跡してみたいと思います。そこでログ用テーブルと、shoelace_dataに対して行われるUPDATEをログに記録するルールを用意しました。

```
CREATE TABLE shoelace_log (
    sl_name    text,          -- 変更された靴紐
    sl_avail   integer,       -- 新しい現在値
    log_who    text,          -- 誰が行ったか
    log_when   timestamp      -- いつ行ったか
);

CREATE RULE log_shoelace AS ON UPDATE TO shoelace_data
    WHERE NEW.sl_avail <> OLD.sl_avail
```

```
    DO INSERT INTO shoelace_log VALUES (
                                    NEW.sl_name,
                                    NEW.sl_avail,
                                    current_user,
                                    current_timestamp
                                );
```

ここで誰かが以下を実行します。

```
UPDATE shoelace_data SET sl_avail = 6 WHERE sl_name = 'sl7';
```

ログテーブルを見てみましょう。

```
SELECT * FROM shoelace_log;

 sl_name | sl_avail | log_who | log_when
---------+----------+---------+----------------------------------
 sl7     |        6 | Al      | Tue Oct 20 16:14:45 1998 MET DST
(1 row)
```

思った通りの結果が出ました。以下に裏で何が起こったのかを説明します。パーサがまず以下の構文解析ツリーを生成しました。

```
UPDATE shoelace_data SET sl_avail = 6
  FROM shoelace_data shoelace_data
 WHERE shoelace_data.sl_name = 'sl7';
```

以下のルール条件式

```
NEW.sl_avail <> OLD.sl_avail
```

および、以下のアクションを持つON UPDATEのlog_shoelaceルールがあります。

```
INSERT INTO shoelace_log VALUES (
       new.sl_name, new.sl_avail,
       current_user, current_timestamp )
  FROM shoelace_data new, shoelace_data old;
```

（通常、INSERT ... VALUES ... FROMを書くことはできないのでちょっと奇妙に見えるかもしれません。ここのFROM句は単にnewとoldの問い合わせツリーの範囲テーブル項目があることを示しているだけです。これらは、INSERTコマンドの問い合わせツリー中の変数から参照されるために必要なのです。）

このルールは条件付きのALSOルールですので、ルールシステムは変更されたルールアクションと元の問い合わせツリーという2つの問い合わせツリーを返さなければなりません。第1の段階で元の問い合わせの範囲テーブルはルールアクション問い合わせツリーに取り込まれます。そして、次の結果を生みます。

```
INSERT INTO shoelace_log VALUES (
```

```
       new.sl_name, new.sl_avail,
       current_user, current_timestamp )
  FROM shoelace_data new, shoelace_data old,
       shoelace_data shoelace_data;
```

第2段階で、以下のようにルール条件が付け加えられます。これにより、この結果集合はsl_availが変更した行に限定されます。

```
INSERT INTO shoelace_log VALUES (
       new.sl_name, new.sl_avail,
       current_user, current_timestamp )
  FROM shoelace_data new, shoelace_data old,
       shoelace_data shoelace_data
 WHERE new.sl_avail <> old.sl_avail;
```

（INSERT ... VALUESはWHERE句を持たないため、これはさらに奇妙です。しかし、プランナとエクゼキュータには問題はありません。これらはどのみちINSERT ... SELECTのために同じ機能をサポートしなければなりません。）

第3段階で、以下のように元の問い合わせツリーの条件が付け加えられ、結果集合は元の問い合わせで変更された行のみにさらに限定されます。

```
INSERT INTO shoelace_log VALUES (
       new.sl_name, new.sl_avail,
       current_user, current_timestamp )
  FROM shoelace_data new, shoelace_data old,
       shoelace_data shoelace_data
 WHERE new.sl_avail <> old.sl_avail
   AND shoelace_data.sl_name = 'sl7';
```

第4段階では、以下のように元の問い合わせツリーの目的リスト項目、または結果リレーションの該当する変数参照で、NEWの参照を置換します。

```
INSERT INTO shoelace_log VALUES (
       shoelace_data.sl_name, 6,
       current_user, current_timestamp )
  FROM shoelace_data new, shoelace_data old,
       shoelace_data shoelace_data
 WHERE 6 <> old.sl_avail
   AND shoelace_data.sl_name = 'sl7';
```

第5段階は、以下のようにOLD参照を結果リレーション参照に置き換えます。

```
INSERT INTO shoelace_log VALUES (
       shoelace_data.sl_name, 6,
       current_user, current_timestamp )
  FROM shoelace_data new, shoelace_data old,
```

```
       shoelace_data shoelace_data
 WHERE 6 <> shoelace_data.sl_avail
   AND shoelace_data.sl_name = 'sl7';
```

これで終わりです。このルールはALSOのため、元の問い合わせツリーも出力します。まとめると、ルールシステムからの出力は以下の文に対応する2つの問い合わせツリーのリストです。

```
INSERT INTO shoelace_log VALUES (
       shoelace_data.sl_name, 6,
       current_user, current_timestamp )
  FROM shoelace_data
 WHERE 6 <> shoelace_data.sl_avail
   AND shoelace_data.sl_name = 'sl7';

UPDATE shoelace_data SET sl_avail = 6
 WHERE sl_name = 'sl7';
```

この2つは順番通りに処理され、正確にルールが定義した通りです。

元の問い合わせが例えば下記のような場合に、置換と追加された条件は、ログには何も書かれないことを確実にします。

```
UPDATE shoelace_data SET sl_color = 'green'
 WHERE sl_name = 'sl7';
```

この場合、元の問い合わせツリーの目的リストにはsl_availの項目がありませんので、NEW.sl_availがshoelace_data.sl_availに置き換えられます。その結果、このルールによって以下のような特別の問い合わせが生成されます。

```
INSERT INTO shoelace_log VALUES (
       shoelace_data.sl_name, shoelace_data.sl_avail,
       current_user, current_timestamp )
  FROM shoelace_data
 WHERE shoelace_data.sl_avail <> shoelace_data.sl_avail
   AND shoelace_data.sl_name = 'sl7';
```

そしてこの条件は決して真にはなりません。

もし元の問い合わせが複数の行を変更してもうまくいきます。ですから、誰かが下記のようなコマンドを実行したとします。

```
UPDATE shoelace_data SET sl_avail = 0
 WHERE sl_color = 'black';
```

この場合、実際には4行が更新されます（sl1、sl2、sl3およびsl4）。しかしsl3は既にsl_avail = 0を持っています。この場合、元の問い合わせツリーの条件を満たさず、その結果、以下のような特別の問い合わせツリーがルールによって生成されます。

```
INSERT INTO shoelace_log
SELECT shoelace_data.sl_name, 0,
       current_user, current_timestamp
  FROM shoelace_data
 WHERE 0 <> shoelace_data.sl_avail
   AND shoelace_data.sl_color = 'black';
```

この構文解析ツリーは確実に3つの新しいログ項目を挿入します。これはまったく正しい動作です [訳注：sl3行はWHERE 0 != shoelace_data.sl_avail条件を満たさない（0!=0）ので、実際に更新される4行-1の3行分のログ項目が挿入されます]。

ここで元の構文解析ツリーが最後に実行されるということが重要な理由がわかります。もしUPDATEが先に実行されたとしたら、全ての行は0にセットされ、0 <> shoelace_data.sl_availである行をログ書き込み時のINSERTの段階で見つけられなくなります。

40.4.2. ビューとの協調

誰かがビューに対してINSERT、UPDATE、DELETEを発行するといった、前述の可能性からビューリレーションを保護する簡単な方法は、それらの構文解析ツリーを破棄してしまうことです。このために以下のルールを作ることができます。

```
CREATE RULE shoe_ins_protect AS ON INSERT TO shoe
    DO INSTEAD NOTHING;
CREATE RULE shoe_upd_protect AS ON UPDATE TO shoe
    DO INSTEAD NOTHING;
CREATE RULE shoe_del_protect AS ON DELETE TO shoe
    DO INSTEAD NOTHING;
```

誰かがshoeビューリレーションに上記の操作を行おうとすると、ルールシステムはルールを適用します。ルールにはアクションがなく、かつ、INSTEADですから、結果の問い合わせツリーリストは空になります。ルールシステムの処理が完了した後に最適化されるものや実行されるべきものが何も残っていませんので、問い合わせ全体が無効になります。

より洗練されたルールシステムの使用方法は、実テーブルに適当な操作を行う問い合わせツリーへの書き換えを行うルールを作ることです。shoelaceビューにこれを適用するために以下のルールを作ります。

```
CREATE RULE shoelace_ins AS ON INSERT TO shoelace
    DO INSTEAD
    INSERT INTO shoelace_data VALUES (
           NEW.sl_name,
           NEW.sl_avail,
           NEW.sl_color,
           NEW.sl_len,
           NEW.sl_unit
    );

CREATE RULE shoelace_upd AS ON UPDATE TO shoelace
```

```
    DO INSTEAD
    UPDATE shoelace_data
       SET sl_name = NEW.sl_name,
           sl_avail = NEW.sl_avail,
           sl_color = NEW.sl_color,
           sl_len = NEW.sl_len,
           sl_unit = NEW.sl_unit
     WHERE sl_name = OLD.sl_name;

CREATE RULE shoelace_del AS ON DELETE TO shoelace
    DO INSTEAD
    DELETE FROM shoelace_data
     WHERE sl_name = OLD.sl_name;
```

このビュー上でRETURNING問い合わせをサポートしたい場合、ビューの行を計算するRETURNING句を含むルールを作成しなければなりません。これは通常、単一テーブルに対するビューでは非常に簡単ですが、shoelaceのような結合されたビューの場合は多少やっかいです。挿入する場合を例として以下に示します。

```
CREATE RULE shoelace_ins AS ON INSERT TO shoelace
    DO INSTEAD
    INSERT INTO shoelace_data VALUES (
           NEW.sl_name,
           NEW.sl_avail,
           NEW.sl_color,
           NEW.sl_len,
           NEW.sl_unit
    )
    RETURNING
           shoelace_data.*,
           (SELECT shoelace_data.sl_len * u.un_fact
            FROM unit u WHERE shoelace_data.sl_unit = u.un_name);
```

この1つのルールが、ビューに対するINSERT問い合わせとINSERT RETURNING問い合わせルールをサポートすることに注意してください。INSERTではRETURNING句が無視されるだけです。

ここで店には不定期に靴紐のケースが分厚い送り状とともに届けられると仮定します。しかし、毎回毎回手作業でshoelaceビューを更新したくはありません。代わりに、送り状から品目を挿入するテーブルと特殊な仕掛けを持つテーブルの2つの小さなテーブルを用意します。以下はそれらを作成するコマンドです。

```
CREATE TABLE shoelace_arrive (
    arr_name    text,
    arr_quant   integer
);

CREATE TABLE shoelace_ok (
    ok_name     text,
    ok_quant    integer
```

```
);

CREATE RULE shoelace_ok_ins AS ON INSERT TO shoelace_ok
    DO INSTEAD
    UPDATE shoelace
       SET sl_avail = sl_avail + NEW.ok_quant
     WHERE sl_name = NEW.ok_name;
```

これで、送り状のデータをshoelace_arriveテーブルに投入することができます。

```
SELECT * FROM shoelace_arrive;

 arr_name | arr_quant
----------+-----------
 sl3      |        10
 sl6      |        20
 sl8      |        20
(3 rows)
```

そして現在のデータをチェックします。

```
SELECT * FROM shoelace;

 sl_name  | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
----------+----------+----------+--------+---------+-----------
 sl1      |        5 | black    |     80 | cm      |        80
 sl2      |        6 | black    |    100 | cm      |       100
 sl7      |        6 | brown    |     60 | cm      |        60
 sl3      |        0 | black    |     35 | inch    |      88.9
 sl4      |        8 | black    |     40 | inch    |     101.6
 sl8      |        1 | brown    |     40 | inch    |     101.6
 sl5      |        4 | brown    |      1 | m       |       100
 sl6      |        0 | brown    |    0.9 | m       |        90
(8 rows)
```

さて、届いた靴紐を移動します。

```
INSERT INTO shoelace_ok SELECT * FROM shoelace_arrive;
```

そして結果を確認します。

```
SELECT * FROM shoelace ORDER BY sl_name;

 sl_name  | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
----------+----------+----------+--------+---------+-----------
 sl1      |        5 | black    |     80 | cm      |        80
 sl2      |        6 | black    |    100 | cm      |       100
```

```
 sl7      |        6 | brown    |     60 | cm      |        60
 sl4      |        8 | black    |     40 | inch    |     101.6
 sl3      |       10 | black    |     35 | inch    |      88.9
 sl8      |       21 | brown    |     40 | inch    |     101.6
 sl5      |        4 | brown    |      1 | m       |       100
 sl6      |       20 | brown    |    0.9 | m       |        90
(8 rows)

SELECT * FROM shoelace_log;

 sl_name | sl_avail | log_who| log_when
---------+----------+--------+----------------------------------
 sl7     |        6 | Al     | Tue Oct 20 19:14:45 1998 MET DST
 sl3     |       10 | Al     | Tue Oct 20 19:25:16 1998 MET DST
 sl6     |       20 | Al     | Tue Oct 20 19:25:16 1998 MET DST
 sl8     |       21 | Al     | Tue Oct 20 19:25:16 1998 MET DST
(4 rows)
```

1つのINSERT ... SELECTからこの結果になるには、長い道のりがあります。本章での問い合わせツリーの変形に関する説明はこれが最後です。まず、以下のようなパーサの出力があります。

```
INSERT INTO shoelace_ok
SELECT shoelace_arrive.arr_name, shoelace_arrive.arr_quant
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok;
```

最初のshoelace_ok_insルールが適用され、結果は以下のようになります。

```
UPDATE shoelace
   SET sl_avail = shoelace.sl_avail + shoelace_arrive.arr_quant
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok old, shoelace_ok new,
       shoelace shoelace
 WHERE shoelace.sl_name = shoelace_arrive.arr_name;
```

そして、shoelace_okに対する元のINSERTを破棄します。この書き換えられた問い合わせは再びルールシステムに渡されて、2番目に適用されるshoelace_updルールは以下を生成します。

```
UPDATE shoelace_data
   SET sl_name = shoelace.sl_name,
       sl_avail = shoelace.sl_avail + shoelace_arrive.arr_quant,
       sl_color = shoelace.sl_color,
       sl_len = shoelace.sl_len,
       sl_unit = shoelace.sl_unit
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok old, shoelace_ok new,
       shoelace shoelace, shoelace old,
```

```
       shoelace new, shoelace_data shoelace_data
 WHERE shoelace.sl_name = shoelace_arrive.arr_name
   AND shoelace_data.sl_name = shoelace.sl_name;
```

これは再びINSTEADルールですので、以前の問い合わせツリーは破棄されます。この問い合わせはshoelaceビューを引き続き使用していることに注意してください。しかし、この段階ではルールシステムは終了していないため、引き続き_RETURNルールが適用され、下記のようになります。

```
UPDATE shoelace_data
   SET sl_name = s.sl_name,
       sl_avail = s.sl_avail + shoelace_arrive.arr_quant,
       sl_color = s.sl_color,
       sl_len = s.sl_len,
       sl_unit = s.sl_unit
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok old, shoelace_ok new,
       shoelace shoelace, shoelace old,
       shoelace new, shoelace_data shoelace_data,
       shoelace old, shoelace new,
       shoelace_data s, unit u
 WHERE s.sl_name = shoelace_arrive.arr_name
   AND shoelace_data.sl_name = s.sl_name;
```

最後に、log_shoelaceルールが適用され、以下のような特別な問い合わせツリーが生成されます。

```
INSERT INTO shoelace_log
SELECT s.sl_name,
       s.sl_avail + shoelace_arrive.arr_quant,
       current_user,
       current_timestamp
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok old, shoelace_ok new,
       shoelace shoelace, shoelace old,
       shoelace new, shoelace_data shoelace_data,
       shoelace old, shoelace new,
       shoelace_data s, unit u,
       shoelace_data old, shoelace_data new
       shoelace_log shoelace_log
 WHERE s.sl_name = shoelace_arrive.arr_name
   AND shoelace_data.sl_name = s.sl_name
   AND (s.sl_avail + shoelace_arrive.arr_quant) <> s.sl_avail;
```

この後、ルールシステムはルールを使い切り、生成された問い合わせツリーを返します。

そして、以下のSQL文と等価となる2つの最終問い合わせツリーで終結します。

```
INSERT INTO shoelace_log
```

```
SELECT s.sl_name,
       s.sl_avail + shoelace_arrive.arr_quant,
       current_user,
       current_timestamp
  FROM shoelace_arrive shoelace_arrive, shoelace_data shoelace_data,
       shoelace_data s
 WHERE s.sl_name = shoelace_arrive.arr_name
   AND shoelace_data.sl_name = s.sl_name
   AND s.sl_avail + shoelace_arrive.arr_quant <> s.sl_avail;

UPDATE shoelace_data
   SET sl_avail = shoelace_data.sl_avail + shoelace_arrive.arr_quant
  FROM shoelace_arrive shoelace_arrive,
       shoelace_data shoelace_data,
       shoelace_data s
 WHERE s.sl_name = shoelace_arrive.sl_name
   AND shoelace_data.sl_name = s.sl_name;
```

結果は、1つのリレーションから来たデータが別のリレーションに挿入され、3つ目のリレーションの更新に変更され、4つ目の更新と5つ目への最終更新のログ記録に変更され、最終的に2つの問い合わせに縮小されます。

ちょっと見苦しい小さな事項があります。でき上がった2つの問い合わせを見ると、1つに縮小されたはずのshoelace_dataリレーションが範囲テーブルに二度出てきます。プランナは処理をしないので、INSERTのルールシステムの出力に対する実行計画は次のようになります。

```
Nested Loop
  ->  Merge Join
        ->  Seq Scan
              ->  Sort
                    ->  Seq Scan on s
        ->  Seq Scan
              ->  Sort
                    ->  Seq Scan on shoelace_arrive
  ->  Seq Scan on shoelace_data
```

一方、余計な範囲テーブル項目を省略することで、以下のようにログテーブルにまったく同じ項目が作られます。

```
Merge Join
  ->  Seq Scan
        ->  Sort
              ->  Seq Scan on s
  ->  Seq Scan
        ->  Sort
              ->  Seq Scan on shoelace_arrive
```

ですから、ルールシステムは、まったく必要のないshoelace_dataリレーションに対する余計なスキャンを一度行うことになります。そしてUPDATEでも同様な不要なスキャンが再度実行されます。しかしながら、これらを全て可能にするのは大変な仕事です。

最後にPostgreSQLのルールシステムとその効力を示しましょう。例えば、まったく売れそうもない靴紐をデータベースに追加してみます。

```
INSERT INTO shoelace VALUES ('sl9', 0, 'pink', 35.0, 'inch', 0.0);
INSERT INTO shoelace VALUES ('sl10', 1000, 'magenta', 40.0, 'inch', 0.0);
```

全ての靴に合わない色がshoelace項目にあるかどうかを検査するビューを作成したいと考えます。ビューは以下のようになります。

```
CREATE VIEW shoelace_mismatch AS
    SELECT * FROM shoelace WHERE NOT EXISTS
        (SELECT shoename FROM shoe WHERE slcolor = sl_color);
```

この出力は以下のようになります。

```
SELECT * FROM shoelace_mismatch;

 sl_name | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
---------+----------+----------+--------+---------+-----------
 sl9     |        0 | pink     |     35 | inch    |      88.9
 sl10    |     1000 | magenta  |     40 | inch    |     101.6
```

ここで、合う靴がない靴紐のうち、在庫がないものをデータベースから削除するように設定してみます。これはPostgreSQLでは困難な作業ですので、直接削除しません。代わりに、以下のようにもう1つビューを作成します。

```
CREATE VIEW shoelace_can_delete AS
    SELECT * FROM shoelace_mismatch WHERE sl_avail = 0;
```

そして、以下を行います。

```
DELETE FROM shoelace WHERE EXISTS
    (SELECT * FROM shoelace_can_delete
             WHERE sl_name = shoelace.sl_name);
```

さあできました。

```
SELECT * FROM shoelace;

 sl_name | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
---------+----------+----------+--------+---------+-----------
 sl1     |        5 | black    |     80 | cm      |        80
 sl2     |        6 | black    |    100 | cm      |       100
 sl7     |        6 | brown    |     60 | cm      |        60
```

```
 sl4     |        8 | black    |     40 | inch    |     101.6
 sl3     |       10 | black    |     35 | inch    |      88.9
 sl8     |       21 | brown    |     40 | inch    |     101.6
 sl10    |     1000 | magenta  |     40 | inch    |     101.6
 sl5     |        4 | brown    |      1 | m       |       100
 sl6     |       20 | brown    |    0.9 | m       |        90
(9 rows)
```

合計4つのネスト/結合されたビューを副問い合わせの条件として持ち、その中の1つはビューを含む副問い合わせ条件を持ち、かつ演算を施されたビューの列が使われる場合の、ビューに対するDELETEが、実テーブルから要求されたデータを削除する単一の問い合わせツリーに書き換えられます。

このような構造が必要な状況は実社会ではほとんどないと思われます。しかし、実際に動くことを確認できれば安心できます。

40.5. ルールと権限

PostgreSQLのルールシステムによる問い合わせの書き換えによって、オリジナルの問い合わせで使われたものではない他のテーブル/ビューがアクセスされます。更新ルールを使うことによってテーブルへの書き込みアクセスを包含することができます。

書き換えルールに別々の所有者はいません。リレーション（テーブルまたはビュー）の所有者は自動的にそれに定義された書き換えルールの所有者となります。PostgreSQLのルールシステムはデフォルトのアクセス制御システムの振舞いを変更します。ルールによって使用されるリレーションは、ルールを起動したユーザの権限ではなく、ルール所有者の権限でチェックされます。このことは、ユーザは問い合わせで明記するテーブル/ビューに対しての権限だけあればよいことを示しています。

例えば、以下のようにします。あるユーザが、いくつかは個人用の、その他は事務所で秘書が利用するための、電話番号のリストを持っていたとします。ユーザは次のようにして構築することができます。

```
CREATE TABLE phone_data (person text, phone text, private boolean);
CREATE VIEW phone_number AS
    SELECT person, CASE WHEN NOT private THEN phone END AS phone
    FROM phone_data;
GRANT SELECT ON phone_number TO assistant;
```

そのユーザ（とデータベースのスーパーユーザ）以外はphone_dataテーブルにアクセスできません。しかし、GRANTにより秘書はphone_numberビューに対しSELECTできます。ルールシステムはphone_numberからのSELECT操作をphone_dataからのSELECT操作に書き換えます。そのユーザはphone_numberの所有者、したがってルールの所有者ですから、phone_dataの読み込みに対するアクセスはそのユーザの権限に従ってチェックされ、問い合わせを受け付けてもよいことになります。phone_numberへのアクセスもチェックされますが、これは呼び出したユーザに対して行われますので、秘書とユーザ以外は使うことができません。

権限はルールごとにチェックされます。ですから秘書だけが今のところ公開の電話番号を参照することができます。しかし、秘書は別のビューを作成し、それにPUBLICに対するアクセス許可を与えることができます。こうすると秘書のビューを通して誰もがphone_numberデータを見ることができます。秘書ができないことはphone_dataに直接アクセスするビューを作ることです（実際には作成はできますが、アクセスは全て、権

限チェックで拒絶されます）。そして、秘書が独自のphone_numberビューを開いたことにユーザが気付いた時点で、秘書の権限を取り上げることができます。秘書のビューへのアクセスは即座に失敗に終わります。

このルールごとのチェックがセキュリティホールになると考える人がいるかもしれませんが、実際にはそうではありません。もしこのように機能しないとなると、秘書はphone_numberと同じ列を持ったテーブルを用意して、1日1回データをそこにコピーするかもしれません。そうなると、データは彼のものですから、誰にアクセス権を与えようが彼の自由です。GRANTは「あなたを信用しています」ということです。信用している誰かがこのようなことを行った場合は、考えを変えてREVOKEしてください。

上に示したような手法を使ってある列の内容を隠すのにビューは使えますが、security_barrierフラグが設定されていない限り、見えない行にあるデータを隠す信頼できる方法ではない事に注意してください。例えば、以下のビューは安全ではありません。

```
CREATE VIEW phone_number AS
    SELECT person, phone FROM phone_data WHERE phone NOT LIKE '412%';
```

ルールシステムがphone_numberからのSELECTをphone_dataからのSELECTに書き換え、phoneが412で始まらない項目のみが必要だという条件を追加しますので、このビューは安全に見えます。しかし、ユーザが自身の関数を作成できるのであれば、NOT LIKE式の前にユーザ定義の関数を実行するようプランナを説得することは難しくありません。例えば以下の通りです。

```
CREATE FUNCTION tricky(text, text) RETURNS bool AS $$
BEGIN
    RAISE NOTICE '% => %', $1, $2;
    RETURN true;
END;
$$ LANGUAGE plpgsql COST 0.00000000000000000001;

SELECT * FROM phone_number WHERE tricky(person, phone);
```

プランナはより高価なNOT LIKEの前に安価なtricky関数を実行することを選びますので、phone_dataテーブルの人と電話番号はすべてNOTICEとして表示されます。たとえユーザが新しい関数を定義できない場合でも、同様の攻撃で組み込み関数が使えます。（例えば、ほとんどの型変換関数は生成するエラーメッセージに入力値を含んでいます。）

同様の考慮は更新ルールにも適用できます。前節の例において、データベースのテーブルの所有者はshoelaceビューに対し、誰かにSELECT、INSERT、UPDATE、DELETE権限を与えることができます。しかし、shoelace_logに対してはSELECTだけです。ログ項目を書き込むルールアクションは支障なく実行され、また、他のユーザはログ項目を見ることができます。しかし、他のユーザは項目を捏造したり、既に存在する項目を操作する、あるいは削除することはできません。この場合、shoelace_logを参照しているルールは条件のないINSERTだけですので、操作の順序を変えるようにプランナを説得することでルールを破壊する可能性はありません。これはより複雑な状況では正しくないかもしれません。

ビューが行単位セキュリティを提供する場合には、そのビューにはsecurity_barrier属性を付与するべきです。これは、悪意を持って選ばれた関数や演算子が、ビューが適用されるより前に渡された行に対して実行されないようにします。例えば、上記のビューが次のように定義された場合、これは安全です。

```
CREATE VIEW phone_number WITH (security_barrier) AS
```

```
    SELECT person, phone FROM phone_data WHERE phone NOT LIKE '412%';
```

security_barrierを付けて定義されたビューは、このオプションなしのビューよりも性能の劣る問い合わせ実行プランを生成します。一般的に、最も高速だが、セキュリティ上の問題がある問い合わせ実行プランを破棄しなければならないという状況は不可避です。そのため、このオプションはデフォルトでは有効になっていません。

プランナは副作用が無い関数をもう少し柔軟に扱います。これらの関数はLEAKPROOF属性を持っており、等価演算子など、単純で広く用いられる演算子も多く含まれます。利用者に対して不可視な行に対するこれら関数の呼び出しはいかなる情報も漏えいさせないため、プランナはこれらの関数をどのような場所でも安全に実行させる事ができます。さらに、引数をとらなかったり、セキュリティバリアビューから引数を渡されない関数は、ビューからデータを渡されることは決して無いため、プッシュダウンされるためのLEAKPROOFをマークする必要はありません。一方、受け取った引数の値に応じてエラー（例えばオーバーフローやゼロ除算など）を発生させるかもしれない関数は leak-proof ではありません。これがもしセキュリティ・ビューの条件句でフィルタリングされるより前に適用されたなら、不可視行に関する何か重要な情報を与える事ができてしまいます。

ビューがsecurity_barrier属性付きで定義されたとしても、それは限定的な意味で安全である、つまり不可視な行の内容が潜在的に安全でない関数に渡される事がないという事を意図しているにすぎません。利用者には不可視な行に対して何らかの推測を行う他の手段があるかもしれません。例えば、EXPLAINを用いて問い合わせ実行プランを見たり、ビューに対する問い合わせ実行時間を計測したりすることです。悪意の攻撃者は不可視データの量を推測したり、分散や最頻値（なぜなら、これらは統計情報を通じて実行プランの選択、ひいては実行時間に影響するかもしれません）に関する何らかの情報を得る事ができるかもしれません。もし、この種の"隠れチャネル"に対する対策が必要であれば、とにかくデータに対するアクセスを許可するのは得策ではありません。

40.6. ルールおよびコマンドの状態

PostgreSQLサーバでは、受け取った問い合わせのそれぞれに対して、INSERT 149592 1のようなコマンド状態文字列を返します。これは簡単ですが、ルールが使用されていない場合には十分なものです。しかし、問い合わせがルールにより書き換えられた場合、どのようになるでしょうか。

ルールはコマンド状態に以下のように影響を与えます。

- 問い合わせに無条件のINSTEADルールが使用されていない場合、元々与えられていた問い合わせが実行され、そのコマンド状態は通常通り返されます（しかし、条件付きINSTEADルールが使用されている場合、その条件の否定が元の問い合わせに追加されることに注意してください。これにより、処理する行の数が減り、その結果報告される状態が影響を受けるかもしれません）。

- 問い合わせに無条件のINSTEADルールが使用されている場合、元の問い合わせはまったく実行されません。この場合、サーバでは、（条件付きもしくは無条件の）INSTEADルールによって挿入され、かつ、元の問い合わせと同じ種類（INSERT、UPDATEまたはDELETE）の最後の問い合わせについてコマンド状態を返します。この条件に合致する問い合わせがルールによって追加されない場合、返されるコマンド状態は、元の問い合わせの種類と行数およびOIDフィールドに0が表示されます。

後者の場合、プログラマは、有効ルールの中でアルファベット順の最後のルール名を与えることによって、必要なINSTEADルールを最後に実行することができます。そして、そのことによって、コマンド状態が確実にそのルールで設定されるようにできます。

40.7. ルール対トリガ

トリガによって行われる多くの操作はPostgreSQLのルールシステムで実装可能です。ルールで実装できないものの1つはある種の制約、特に外部キーに関してです。もし他のテーブルに列の値がなかった場合、条件ルールでコマンドをNOTHINGに書き換えてしまうことも可能ですが、これではデータがだまって消去されてしまい、良いアイディアとは言えません。有効な値かどうかのチェックが必要で、無効な値についてはエラーメッセージを表示する必要があるなら、このことは今のところトリガを使って行わなければなりません。

この章ではビューを更新するのにルールを使うことに焦点を当ててきました。この章の更新ルールの例はすべてビューのINSTEAD OFトリガを使っても実装できます。特に更新を実行するのに複雑な論理が要求される場合には、そのようなトリガを書くことはしばしばルールを書くよりも簡単です。

どちらでも実装できる事項に関してどちらがベストかはデータベースの使用方法によります。トリガは各行に対して一度起動します。ルールは問い合わせを修正するか追加の問い合わせを生成します。ですから、1つの文が多くの行に影響を与える場合、1つの行を処理する度に呼び出され、何をするかを何度も再決定しなければならないトリガよりも、追加の問い合わせを1つ発行するルールの方がほとんどの場合高速になります。しかし、トリガ方式は概念的にルールシステムよりかなり単純であり、初心者は簡単に正しく扱うことができます。

ここで、ある状況下でルールとトリガのどちらを選択するかを示す例を挙げます。例えば、2つのテーブルがあるとします。

```
CREATE TABLE computer (

    hostname        text,    -- インデックスあり
    manufacturer    text     -- インデックスあり
);

CREATE TABLE software (

    software        text,    -- インデックスあり
    hostname        text     -- インデックスあり
);
```

2つのテーブルにはともに数千の行があって、hostname上のインデックスは一意です。ルール/トリガは削除されたホストを参照する、softwareの行を削除する制限を実装しなければなりません。トリガの場合は以下のコマンドを使用します。

```
DELETE FROM software WHERE hostname = $1;
```

computerから削除された行1つひとつに対してこのトリガが呼び出されますので、このコマンドの準備を行い、計画を保存し、パラメータとしてhostnameを渡すことができます。ルールの場合は以下のように作成されます。

```
CREATE RULE computer_del AS ON DELETE TO computer
    DO DELETE FROM software WHERE hostname = OLD.hostname;
```

ここで別の類の削除を考えてみましょう。

```
DELETE FROM computer WHERE hostname = 'mypc.local.net';
```

上のような場合では、computerはインデックスにより（高速に）スキャンされます。トリガによってこのコマンドが発行された場合もインデックススキャンが使用されます（高速です）。ルールによる追加コマンドは以下のようになります。

```
DELETE FROM software WHERE computer.hostname = 'mypc.local.net'
                       AND software.hostname = computer.hostname;
```

適切なインデックスが設定されていますので、プランナは以下の計画を作成します。

```
Nestloop
  ->  Index Scan using comp_hostidx on computer
  ->  Index Scan using soft_hostidx on software
```

ですので、トリガとルールの実装間での速度差はあまりありません。

次の削除処理ではhostnameがoldで始まる2,000台全てのcomputerを削除しようと思います。方法として2つの有効な問い合わせがあって、1つは以下のようなものです。

```
DELETE FROM computer WHERE hostname >= 'old'
                       AND hostname <  'ole'
```

ルールによって追加されるコマンドは以下のようになります。

```
DELETE FROM software WHERE computer.hostname >= 'old' AND computer.hostname < 'ole'
                       AND software.hostname = computer.hostname;
```

これに対する計画は以下のようになります。

```
Hash Join
  ->  Seq Scan on software
  ->  Hash
    ->  Index Scan using comp_hostidx on computer
```

もう1つのコマンドは以下のようなものです。

```
DELETE FROM computer WHERE hostname ~ '^old';
```

これにより、ルールによって追加されるコマンド用の実行計画は以下のようになります。

```
Nestloop
  ->  Index Scan using comp_hostidx on computer
  ->  Index Scan using soft_hostidx on software
```

これが示していることは、ANDで結合された複数の検索条件が存在する場合、プランナは正規表現版のコマンドでは行っていることですが、computer上のhostnameに対する検索条件をsoftware上のインデックススキャンにも同様に使用できることを理解しないということです。トリガは削除されるべき2,000台の旧式コンピュータのそれぞれについて1回呼び出され、結果computer上で1回のインデックススキャンとsoftware上で2,000回のインデックススキャンが行われます。ルールによる実装ではインデックスを使用する2つの問い合わせによって実行されます。シーケンシャルスキャンの場合でもルールがより速いかどうかはsoftwareテーブルの大きさに依存します。参照する全てのインデックスブロックがすぐにキャッシュに現れるとしても、トリガによるSPIマネージャ経由の2,000回のコマンドの実行には時間を要します。

最後のコマンドを見てみましょう。

```
DELETE FROM computer WHERE manufacturer = 'bim';
```

この文でもcomputerから多くの行が削除される結果となります。ですので、ここでもトリガはエクゼキュータを通して多くのコマンドを実行することになります。ルールで作成されるコマンドは以下のようなものです。

```
DELETE FROM software WHERE computer.manufacturer = 'bim'
                       AND software.hostname = computer.hostname;
```

このコマンド用の計画もまた前回同様2つのインデックススキャンのネステッドループとなります。computerの別のインデックスを使用する点のみが異なります。

```
Nestloop
  ->  Index Scan using comp_manufidx on computer
  ->  Index Scan using soft_hostidx on software
```

いずれの場合においても、ルールシステムが生成する追加コマンドは影響を受ける行数からは多かれ少なかれ独立しています。

まとめると、問い合わせ結果が大きく、プランナがうまく結合条件を設定できないような状況下でのみルールはトリガに比べて明らかに遅くなります。

第41章 手続き言語

PostgreSQLでは、SQLやC言語以外の言語でユーザ定義の関数を作成することができます。これらの他の言語は一般に*手続き言語*（PL）と呼ばれます。手続き言語で関数が記述されていた場合、データベースサーバにはその関数のソースをどのように解釈すればよいかについての知識が組み込まれていません。代わりに、その処理はその言語を解釈する特別なハンドラに引き渡されます。そのハンドラは解析、構文分析、実行などすべてのことを行うこともできますし、PostgreSQLと存在するプログラミング言語の実装との「橋渡し」ともなり得ます。ハンドラそのものはC言語関数で、他のC言語関数と同様に、共有オブジェクトにコンパイルされ、要求に応じてロードされます。

現在PostgreSQLの標準配布物では、PL/pgSQL（第42章）、PL/Tcl（第43章）、PL/Perl（第44章）、PL/Python（第45章）という4つの手続き言語があります。さらに、コア配布物には含まれない手続き言語があります。付録Hでその見つけ方を説明します。ユーザは他の言語を定義することもできます。新しい手続き言語の開発について、その基礎を第55章で説明します。

41.1. 手続き言語のインストール

手続き言語は、それらが使用されるデータベースすべてに「インストール」されている必要があります。しかし、template1データベースにインストールされた手続き言語は、template1内の項目はCREATE DATABASEによってコピーされますので、その後に作成されたすべてのデータベースで自動的に使用できます。したがって、データベース管理者はどのデータベースにどの言語を使用するかを決定できますし、デフォルトで使用できる言語も決定できます。

標準配布物で提供される言語では、その言語を現在のデータベースにインストールするにはCREATE EXTENSION language_nameの実行のみが必要です。下記の手作業は、拡張機能としてパッケージ化されていない言語をインストールする場合にのみ行うことを推奨します。

手続き言語の手作業によるインストール方法

手続き言語を次の5段階でデータベースにインストールすることができます。この作業はデータベースのスーパーユーザで行う必要があります。ほとんどの場合、必要なSQLコマンドは「拡張機能」のインストールスクリプトとしてパッケージ化されていますので、この作業を実行するのにCREATE EXTENSIONが利用できます。

1. その言語ハンドラ用の共有オブジェクトがコンパイルされ、適切なライブラリディレクトリにインストールされている必要があります。これは、通常のユーザ定義のC関数を作成してインストールする時と同じです。37.10.5を参照してください。実際のプログラミング言語エンジンを提供する外部ライブラリに、言語ハンドラが依存していることがよくあります。この場合はそのライブラリもインストールしなければなりません。

2. ハンドラは下記のコマンドで宣言されなければなりません。

```
CREATE FUNCTION handler_function_name()
    RETURNS language_handler
    AS 'path-to-shared-object'
    LANGUAGE C;
```

language_handlerという特別な戻り値の型は、この関数が定義済みのSQLデータ型を返さず、SQL文では直接使用できないことをデータベースシステムに伝えます。

3. (Optional) 省略可能ですが、言語ハンドラは、この言語で書かれた無名コードブロック(DO コマンド)を実行する「インライン」ハンドラ関数を提供することができます。インラインハンドラ関数が言語により提供されるのであれば、以下のようなコマンドで宣言されます。

```
CREATE FUNCTION inline_function_name(internal)
    RETURNS void
    AS 'path-to-shared-object'
    LANGUAGE C;
```

4. (Optional) 省略可能ですが、言語ハンドラは、実際に実行することなく関数定義の正確性を検査する「有効性検査」関数を提供することができます。もし存在すれば、有効性検査関数はCREATE FUNCTIONで呼び出されます。有効性検査関数が言語により提供されるのであれば、以下のようなコマンドで宣言されます。

```
CREATE FUNCTION validator_function_name(oid)
    RETURNS void
    AS 'path-to-shared-object'
    LANGUAGE C STRICT;
```

5. 最終的に、PLは下記のコマンドで宣言されなければいけません。

```
CREATE [TRUSTED] LANGUAGE language_name
    HANDLER handler_function_name
    [INLINE inline_function_name]
    [VALIDATOR validator_function_name] ;
```

TRUSTEDというオプションキーワードは、ユーザがアクセス権を持たないデータに対して、その言語がアクセス権を持たないことを指定します。TRUSTEDである言語は(スーパーユーザ権限を持たない)一般ユーザ用に設計されており、安全に関数やプロシージャを作成できます。PL関数はデータベースサーバの内部で実行されますので、TRUSTEDフラグはデータベースサーバ内部やファイルシステムへのアクセスを持たない言語のみが使わなければなりません。PL/pgSQLとPL/Tcl、PL/Perl言語はTRUSTEDと考えられています。提供される機能が無制限に設計されているPL/TclU、PL/PerlU、PL/PythonU言語については、TRUSTEDを指定してはなりません。

例 41.1に、手作業によるインストール手順がPL/Perl言語でどのように動作するかを示します。

例41.1 PL/Perlの手作業によるインストール

以下のコマンドは、データベースサーバにPL/Perl言語の呼び出しハンドラ関数用の共有ライブラリの存在場所を通知します。

```
CREATE FUNCTION plperl_call_handler() RETURNS language_handler AS
    '$libdir/plperl' LANGUAGE C;
```

PL/Perlはインラインハンドラ関数と有効性検査関数を有していますので、以下のようにも宣言します。

```
CREATE FUNCTION plperl_inline_handler(internal) RETURNS void AS
    '$libdir/plperl' LANGUAGE C STRICT;

CREATE FUNCTION plperl_validator(oid) RETURNS void AS
    '$libdir/plperl' LANGUAGE C STRICT;
```

以下のコマンドは、直前に宣言された関数を、言語属性がplperlである関数やプロシージャ用に呼び出さなければならないことを定義します。

```
CREATE TRUSTED LANGUAGE plperl
    HANDLER plperl_call_handler
    INLINE plperl_inline_handler
    VALIDATOR plperl_validator;
```

デフォルトのPostgreSQLインストレーションでは、PL/pgSQL言語用のハンドラは構築され、「ライブラリ」ディレクトリにインストールされます。さらに、PL/pgSQL言語自体がデータベースすべてにインストールされます。Tclのサポート付きで構築した場合、PL/TclとPL/TclU用のハンドラも構築されライブラリディレクトリにインストールされますが、言語自体はデフォルトではどのデータベースにもインストールされません。同様に、Perlサポート付きで構築した場合はPL/PerlとPL/PerlUハンドラが、Pythonサポート付きで構築した場合はPL/PythonUハンドラが構築され、インストールされますが、言語自体はデフォルトではインストールされません。

第42章 PL/pgSQL — SQL手続き言語

42.1. 概要

PL/pgSQLは、PostgreSQLデータベースシステム用の読み込み可能な手続き言語です。PL/pgSQLの設計目的は、次のような読み込み可能な手続き言語でした。

- 関数とトリガを作成するために使用できること
- SQL言語に制御構造を追加すること
- 複雑な演算が可能であること
- 全てのユーザ定義型、関数、演算子を継承すること
- サーバによって信頼できるものと定義できること
- 使いやすいこと

PL/pgSQLで作成した関数は、組み込み関数が使えるところであれば、どこでも使用できます。例えば、複雑な条件のある演算処理関数の作成が可能ですし、作成した関数を使用して演算子を定義することも、インデックス式にその関数を使用することも可能です。

PostgreSQL 9.0以降ではPL/pgSQLはデフォルトでインストールされます。しかしこれはまだロード可能なモジュールですので、特にセキュリティに厳しい管理者は削除することもできます。

42.1.1. PL/pgSQLを使用することの利点

SQLはPostgreSQLおよびその他のほとんどのリレーショナルデータベースが問い合わせ言語として使用している言語です。移植性があり、習得が容易です。しかし、あらゆるSQL文はデータベースサーバによって個々に実行されなければいけません。

これはクライアントアプリケーションに対して以下のようなことを要求しています。まず、データベースサーバに問い合わせを送信します。次にそれが処理されるのを待ちます。次に、結果を取得して処理します。次に若干の計算を行います。そして、サーバに次の問い合わせを送信します。クライアントがデータベースサーバマシンと異なるマシンの場合、これが招いたプロセス間通信により、ネットワークオーバーヘッドを起こすでしょう。

PL/pgSQLを使うことで、計算と複数の問い合わせをデータベースサーバ内部でひとまとめに実行することができます。このように、手続き言語の能力とSQLの使いやすさを持ち合わせているにもかかわらず、クライアント/サーバ通信のオーバーヘッドをかなり節約できます。

- クライアント・サーバ間の余計なやり取りを排除する。
- クライアントサーバ間において、クライアントに不必要な中間結果の整理と転送を不要とする。
- 一連の問い合わせに、複数の解析が不要である。

これにより、ストアドプロシージャを使用しないアプリケーションに比較して、かなり性能を向上させることができます。

また、PL/pgSQLではSQL全てのデータ型、演算子、関数を使用することができます。

42.1.2. 引数と結果データ型のサポート

PL/pgSQLで作成された関数は、サーバでサポートされる任意のスカラデータ型や配列データ型を引数として受け付けることができ、また、これらの型を結果として返すことができます。また、任意の、名前で指定された複合型（行型）を受け付けたり、返したりすることもできます。さらに、7.2.1.4で説明されているように、PL/pgSQL関数がrecordを受け入れるように、すなわち、任意の複合型を入力としたりrecordを返すように宣言することも可能です。recordを返す場合の結果は、その各列が呼び出す問い合わせの中での指定で決まる行型です。

PL/pgSQL関数はVARIADIC記号を使用して可変長の引数を受け付けられるように宣言することができます。これは37.5.5で論議したように、SQL関数と全く同じ方法で動作します。

また、PL/pgSQL関数を、37.2.5で説明されている多様型を受け付けたり、返したりするように宣言することもできます。これにより、関数によって処理される実際のデータ型は呼び出しごとに変動することができます。例を42.3.1に示します。

PL/pgSQL関数は、1つのインスタンスとして返すことができる任意のデータ型の「集合」（テーブル）を返すように宣言できます。こうした関数は、結果集合の必要な要素に対してRETURN NEXTを実行すること、または問い合わせの評価結果を得るためにRETURN QUERYを使用することで、その出力を生成します。

最後に、有用な戻り値を持たない場合、PL/pgSQL関数は、voidを返すように宣言することができます。（あるいは、この場合はプロシージャとして書くこともできます）

PL/pgSQL関数は戻り値の型を明確に指定する代わりに、出力パラメータと共に宣言することもできます。これは言語に対して基本的な能力を追加するものではありませんが、特に複数の値を返す時にしばしば便利です。RETURNS TABLE表記はRETURNS SETOFの代わりとして使用できます。

関連する例は42.3.1および42.6.1にあります。

42.2. PL/pgSQLの構造

PL/pgSQLで書かれた関数はCREATE FUNCTIONコマンドを実行することでサーバに定義されます。そのようなコマンドは通常、例えば次のようになります。

```
CREATE FUNCTION somefunc(integer, text) RETURNS integer
AS 'function body text'
LANGUAGE plpgsql;
```

関数本体はCREATE FUNCTIONにとっては単なる文字列リテラルです。関数本体を書くのには、普通の単一引用符構文よりは、ドル引用符(4.1.2.4を参照)を使うのが、多くの場合役に立ちます。ドル引用符でなければ、関数本体内の単一引用符やバックスラッシュをすべて二重化してエスケープしなければなりません。この章のほぼすべての例では、関数本体にドル記号で括られたリテラルを使っています。

PL/pgSQLはブロック構造の言語です。関数本体のテキスト全体は*ブロック*でなければなりません。ブロックは以下のように定義されます。

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements
END [ label ];
```

ブロック内の宣言や文はそれぞれ、セミコロンで終わります。上に示したように、他のブロック内に出現するブロックのENDの後にはセミコロンが必要ですが、関数本体を完結する最後のENDには不要です。

> **ヒント**
>
> BEGINの直後にセミコロンを書くことも、同じように間違いです。これは不正であり、構文エラーとなります。

*label*が必要となるのは、EXIT文が使用されるブロックを特定したい場合、またはブロック内で宣言された変数名を修飾したい場合だけです。ENDの後にラベルを配置する時は、そのブロックの先頭ラベルと一致させなければなりません。

全てのキーワードは大文字と小文字を区別しません。識別子は二重引用符でくくられていない限り、通常のSQLコマンドと同様に、暗黙的に小文字に変換されます。

PL/pgSQLコード内では、通常のSQLと同じ方法のコメントが動作します。二重のダッシュ(--)はその行末までをコメントとするコメントを開始します。/*はコメントブロックの始まりを意味し、次に*/が現れるまでをコメントとします。ブロックコメントは入れ子になります。

ブロックの文節内の全ての文は*副ブロック*になることができます。副ブロックは論理的なグループ分けや変数を文の小さな集まりに局所化するのに使用できます。副ブロックにおいて宣言された変数は、副ブロック内部では外側のブロックにおける同名の変数を遮蔽しますが、外側のラベルを変数名に付加して指定すればアクセスできます。以下に例を示します。

```
CREATE FUNCTION somefunc() RETURNS integer AS $$
<< outerblock >>
DECLARE
    quantity integer := 30;
BEGIN

    RAISE NOTICE 'Quantity here is %', quantity;  -- Quantity here is 30と表示
    quantity := 50;
    --

    -- 副ブロックの作成
    --
    DECLARE
```

```
        quantity integer := 80;
    BEGIN

        RAISE NOTICE 'Quantity here is %', quantity;  -- Quantity here is 80と表示
        RAISE NOTICE 'Outer quantity here is %', outerblock.quantity;  -- Quantity here is 50と表
示
    END;


    RAISE NOTICE 'Quantity here is %', quantity;  -- Quantity here is 50と表示

    RETURN quantity;
END;
$$ LANGUAGE plpgsql;
```

注記

PL/pgSQL関数の本体を囲む、隠れた「外側のブロック」が存在します。この隠れたブロックにおいて、関数のパラメータがあれば宣言をして、同様にFOUNDのような特殊な変数（42.5.5を参照）を提供します。この外側のブロックのラベルは関数名となります。つまりパラメータと特殊な変数は関数名によって修飾することを意味します。

PL/pgSQLにおける文をまとめるためのBEGIN/ENDとトランザクション制御用の同名のSQLコマンドとを取り違えないようにすることが重要です。PL/pgSQLのBEGIN/ENDは単にまとめるためのもので、トランザクションを始めたり終わらせたりしません。トランザクションをPL/pgSQL内で制御するための情報に関しては、42.8を参照してください。また、EXCEPTION句を含むブロックは外側のトランザクションに影響しないでロールバックできるサブトランザクションを、実質的に作成できます。これについては42.6.8を参照してください。

42.3. 宣言

ブロック内で使用される全ての変数はそのブロックの宣言部で宣言されなければなりません。（唯一の例外は、FORループである整数値の範囲に渡って繰り返されるループ変数で、これは、自動的に整数型変数として宣言されます。同様に、カーソルの結果に対して繰り返し適用されるFORループのループ変数はレコード変数として自動的に宣言されます。）

PL/pgSQL変数は、integer、varchar、charといった、任意のSQLデータ型を持つことができます。

変数宣言の例を以下に示します。

```
user_id integer;
quantity numeric(5);
url varchar;
myrow tablename%ROWTYPE;
myfield tablename.columnname%TYPE;
arow RECORD;
```

変数宣言の一般的な構文は以下の通りです。

```
name [ CONSTANT ] type [ COLLATE collation_name ] [ NOT NULL ] [ { DEFAULT | := | = } expression
 ];
```

DEFAULT句が指定された場合、ブロックに入った時に変数に代入される初期値を指定します。DEFAULT句が指定されない場合、変数はSQLのNULL値に初期化されます。CONSTANTオプションにより、そのブロック内でその値が不変になるように、その変数への初期化後の代入は禁止されます。COLLATEオプションは、変数として使用するための照合を指定します（42.3.6を参照してください）。NOT NULLが指定された場合、NULL値の代入は実行時エラーになります。NOT NULLとして宣言した変数は全て、非NULLのデフォルト値を指定しなければなりません。等号（=）がPL/SQLにおける代入記号（:=）の代わりに使用できます。

変数のデフォルト値はブロックに入る度に評価され、変数に代入されます（関数を呼び出す時に一度だけではありません）。ですから、例えばnow()をtimestamp型の変数に代入することで、その変数には関数をプリコンパイルした時刻ではなく、関数呼び出し時の現在時刻が格納されます。

例：

```
quantity integer DEFAULT 32;
url varchar := 'http://mysite.com';
user_id CONSTANT integer := 10;
```

42.3.1. 関数引数の宣言

関数に渡されるパラメータの名前には$1、$2という識別子が付けられます。省略することもできますが、$nというパラメータ名に別名を宣言することができ、可読性が向上します。別名、数字による識別子の両方とも引数の値を参照する時に使用することができます。

別名を作成する方法は2つあり、望ましい方法はCREATE FUNCTIONコマンドの中でパラメータを命名するものです。以下に例を示します。

```
CREATE FUNCTION sales_tax(subtotal real) RETURNS real AS $$
BEGIN
    RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

他の方法は、宣言構文を用いて別名を明確に宣言するものです。

```
name ALIAS FOR $n;
```

以下にこの方法による例を示します。

```
CREATE FUNCTION sales_tax(real) RETURNS real AS $$
DECLARE
    subtotal ALIAS FOR $1;
BEGIN
```

```
    RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

> **注記**
>
> この二例は完全に同等ではありません。最初の例では、subtotalをsales_tax.subtotalで参照できますが、次の例ではできません（その代わり、内部ブロックにラベルを付与すれば、subtotalをラベルで修飾することができます）。

さらに数例を示します。

```
CREATE FUNCTION instr(varchar, integer) RETURNS integer AS $$
DECLARE
    v_string ALIAS FOR $1;
    index ALIAS FOR $2;
BEGIN

    -- v_string とインデックスを使用した何らかの演算を行なう
END;
$$ LANGUAGE plpgsql;



CREATE FUNCTION concat_selected_fields(in_t sometablename) RETURNS text AS $$
BEGIN
    RETURN in_t.f1 || in_t.f3 || in_t.f5 || in_t.f7;
END;
$$ LANGUAGE plpgsql;
```

PL/pgSQL関数が出力パラメータと共に宣言されると、通常の入力パラメータと同様に、出力パラメータには$nというパラメータ名と任意の別名が与えられます。出力パラメータは実質的には最初がNULL値の変数であり、関数の実行中に値が指定されるはずです。出力パラメータの最後の値は戻り値です。例えば、消費税の例題は、次のようにすることもできます。

```
CREATE FUNCTION sales_tax(subtotal real, OUT tax real) AS $$
BEGIN
    tax := subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

RETURNS realを省略したことに注意してください。含めることもできますが、冗長になります。

出力パラメータは複数の値を返す時に最も有用になります。簡単な例題を示します。

```
CREATE FUNCTION sum_n_product(x int, y int, OUT sum int, OUT prod int) AS $$
BEGIN
```

```
    sum := x + y;
    prod := x * y;
END;
$$ LANGUAGE plpgsql;
```

37.5.4で述べたように、この方法は関数の結果に対する匿名のレコード型を実質的に作成します。RETURNS句が与えられた時は、RETURNS recordと言わなければなりません。

PL/pgSQL関数を宣言する他の方法として、RETURNS TABLEを伴うことが挙げられます。以下に例を示します。

```
CREATE FUNCTION extended_sales(p_itemno int)
RETURNS TABLE(quantity int, total numeric) AS $$
BEGIN
    RETURN QUERY SELECT s.quantity, s.quantity * s.price FROM sales AS s
                 WHERE s.itemno = p_itemno;
END;
$$ LANGUAGE plpgsql;
```

これは、1つ、またはそれ以上のOUTパラメータを宣言すること、およびRETURNS SETOF 何らかのデータ型を指定することと全く等価です。

PL/pgSQL関数の戻り値が多様型（37.2.5を参照）として宣言されると、特別な$0パラメータが作成されます。このデータ型が、実際の入力型から推定された関数の実際の戻り値の型です。これにより、関数は42.3.3に示すように、実際の戻り値の型にアクセスできます。$0はNULLで初期化され、関数内で変更することができます。ですので、必須ではありませんが、これを戻り値を保持するために使用しても構いません。また$0に別名を付与することもできます。例えば、以下の関数は+演算子を持つ任意のデータ型に対して稼働します。

```
CREATE FUNCTION add_three_values(v1 anyelement, v2 anyelement, v3 anyelement)
RETURNS anyelement AS $$
DECLARE
    result ALIAS FOR $0;
BEGIN
    result := v1 + v2 + v3;
    RETURN result;
END;
$$ LANGUAGE plpgsql;
```

1つ以上の出力パラメータを多様型として宣言することにより、同様の結果を得ることができます。この場合、特殊な$0パラメータは使用されません。出力パラメータ自身が同じ目的を果たします。以下に例を示します。

```
CREATE FUNCTION add_three_values(v1 anyelement, v2 anyelement, v3 anyelement,
                                 OUT sum anyelement)
AS $$
BEGIN
```

```
    sum := v1 + v2 + v3;
END;
$$ LANGUAGE plpgsql;
```

実際には型にanycompatible族を使用して多様関数を宣言する方が有用である可能性があります。そうすれば、入力引数が共通の型に自動的に昇格されます。以下に例を示します。

```
CREATE FUNCTION add_three_values(v1 anycompatible, v2 anycompatible, v3 anycompatible)
RETURNS anycompatible AS $$
BEGIN
    RETURN v1 + v2 + v3;
END;
$$ LANGUAGE plpgsql;
```

この例は

```
SELECT add_three_values(1, 2, 4.7);
```

自動的に整数の入力を数値データに昇格して呼び出しが動作します。anyelementを使用する関数では、3つの入力を同じ型に手動でキャストする必要があります。

42.3.2. ALIAS

```
newname ALIAS FOR oldname;
```

ALIAS構文は前節で示したものより一般的です。関数の引数だけではなく、任意の変数に別名を宣言することができます。この現実的な使用は主に、トリガ関数におけるNEWやOLDなど、前もって決まった名前の変数に別の名前を割り当てることです。

以下に例を示します。

```
DECLARE
  prior ALIAS FOR old;
  updated ALIAS FOR new;
```

ALIASは同じオブジェクトを命名する2つの異なる手段を提供しますので、無制限に使用すると混乱を招くかもしれません。前もって決まっている名前を上書きする目的に限定して使用することが最善です。

42.3.3. 型のコピー

```
variable%TYPE
```

%TYPEは変数やテーブル列のデータ型を提供します。これを使用してデータベース値を保持する変数を宣言することができます。例えば、usersテーブルにuser_idという列があるものとします。users.user_idと同じデータ型の変数を宣言するには、以下のように記述します。

```
user_id users.user_id%TYPE;
```

%TYPEを使用することで、参照する構造のデータ型を把握する必要がなくなります。また、これが最も重要なことですが、参照される項目のデータ型が将来変更された（例えば、user_idのテーブル定義をintegerからrealに変更した）場合でも、関数定義を変更する必要をなくすことができます。

内部変数用のデータ型は呼び出す度に変わるかもしれませんので%TYPEは特に多様関数で有用です。関数の引数や結果用のプレースホルダに%TYPEを適用することで、適切な変数を作成することができます。

42.3.4. 行型

```
name table_name%ROWTYPE;
name composite_type_name;
```

複合型の変数は、行変数（または行型変数）と呼ばれます。こういった変数には、問い合わせの列集合が変数の型宣言と一致する限り、SELECTやFOR問い合わせの結果の行全体を保持することができます。行変数の個々のフィールド値には、例えば、rowvar.fieldといったドット記法を使用してアクセスすることができます。

table_name%ROWTYPEという記法を使用して、既存のテーブルやビューの行と同じ型を持つ行変数を宣言することができます。もしくは、複合型の名前を付与して宣言することができます。（全てのテーブルは、同じ名前の関連する複合型を持ちますので、実際のところPostgreSQLでは、%ROWTYPEと書いても書かなくても問題にはなりません。しかし、%ROWTYPEの方がより移植性が高まります。）

関数へのパラメータとして複合型（テーブル行全体）を取ることができます。その場合、対応する識別子$nは行変数であり、そのフィールドを、例えば、$1.user_idで選択することができます。

以下に複合型を使用する例を示します。table1及びtable2は、少なくとも言及するフィールドを有する既存のテーブルです。

```
CREATE FUNCTION merge_fields(t_row table1) RETURNS text AS $$
DECLARE
    t2_row table2%ROWTYPE;
BEGIN
    SELECT * INTO t2_row FROM table2 WHERE ... ;
    RETURN t_row.f1 || t2_row.f3 || t_row.f5 || t2_row.f7;
END;
$$ LANGUAGE plpgsql;

SELECT merge_fields(t.*) FROM table1 t WHERE ... ;
```

42.3.5. レコード型

```
name RECORD;
```

レコード変数は行型変数と似ていますが、事前に定義された構造を持っていません。これはSELECTやFORコマンドの間で代入された行の実際の行構造を取ります。レコード変数の副構造は、代入を行う度に変更できます。つまり、レコード変数は、最初に代入されるまで副構造を持たず、したがって、フィールドへのアクセスを試みると実行時エラーが発生します。

RECORDは本当のデータ型ではなく、単なるプレースホルダであることに注意してください。PL/pgSQL関数がrecord型を返す時、この関数ではレコード変数を使用してその結果を保持することができますが、これはレコード変数としての概念とはまったく異なることを認識すべきです。両方とも、関数の作成段階では実際の行構造は不明です。しかし、レコード変数はその場その場でその行構造を変更できるにもかかわらず、recordを返す関数では呼び出し元の問い合わせが解析された時点で実際の構造は決定されます。

42.3.6. PL/pgSQL変数の照合

PL/pgSQL関数が照合可能なデータ型のパラメータを 1つ以上保有する場合、23.2に記述したように、実際の引数に割り当てられた照合に従って、関数呼び出し毎に照合が識別されます。照合の識別に成功した場合（すなわち、引数の間に事実上の照合における衝突がない場合）、照合可能な全てのパラメータは暗黙の照合を有するとして扱われます。これは関数内部において、照合に依存する操作の作用に影響します。以下の例を考えてください。

```
CREATE FUNCTION less_than(a text, b text) RETURNS boolean AS $$
BEGIN
    RETURN a < b;
END;
$$ LANGUAGE plpgsql;

SELECT less_than(text_field_1, text_field_2) FROM table1;
SELECT less_than(text_field_1, text_field_2 COLLATE "C") FROM table1;
```

第一の使用方法においてless_thanは、text_field_1とtext_field_2の比較のための通常の照合として用いられます。第二の使用方法においては、C照合として用いられます。

さらに、識別された照合は、照合可能なデータ型の全ての局所変数の照合としても仮定されます。したがって、この関数は下に記述する関数と差異なく作動します。

```
CREATE FUNCTION less_than(a text, b text) RETURNS boolean AS $$
DECLARE
    local_a text := a;
    local_b text := b;
BEGIN
    RETURN local_a < local_b;
END;
$$ LANGUAGE plpgsql;
```

照合可能なデータ型のパラメータが存在しない場合、または、それらで共通する照合順序を識別できない場合、パラメータと局所変数は自身のデータ型のデフォルトの照合順序（通常これはデータベースのデフォルトの照合順序ですが、ドメイン型の変数の場合は異なるかもしれません）を使用します。

照合可能なデータ型の局所変数は、宣言内でCOLLATEオプションを含めることにより、別の照合と関連づけることができます。例を示します。

```
DECLARE
    local_a text COLLATE "en_US";
```

このオプションは上記ルールにより、変数に他の方法で付与されるはずであった照合を上書きします。

また当然ながら、強制的に特定の操作において特定の照合順序を使用したい場合、明示的なCOLLATE句を関数内部に記述することができます。例を示します。

```
CREATE FUNCTION less_than_c(a text, b text) RETURNS boolean AS $$
BEGIN
    RETURN a < b COLLATE "C";
END;
$$ LANGUAGE plpgsql;
```

単純な SQL コマンドで起こるように、これはテーブルの列、パラメータ、または式の中の局所変数に関連づけられた照合を上書きします

42.4. 式

PL/pgSQL文で使用される式は全て、サーバの主SQLエクゼキュータを使用して処理されます。例えば、以下のPL/pgSQL文

```
IF expression THEN ...
```

が記述された時、PL/pgSQLは

```
SELECT expression
```

を主SQLエンジンに供給して、上式を評価します。42.11.1において詳細を説明したように、SELECTコマンドの形成においてPL/pgSQL変数名は、その都度パラメータによって置換されます。これにより、SELECTの問い合わせ計画は一度だけ準備することができ、その後の評価で異なった変数値を代入して再利用されます。すなわち、式の最初の使用においては、実質的にPREPAREコマンドと同等です。例えば、2つの整数変数xとyを宣言して、

```
IF x < y THEN ...
```

という条件式を記述すると背後では

```
PREPARE statement_name(integer, integer) AS SELECT $1 < $2;
```

と同等なプリペアドステートメントが作成されます。そしてIF文を実行する度にPL/pgSQLの最新の変数値をパラメータ値として供給して、このプリペアドステートメントに対してEXECUTEを行います。通常この詳細は、

PL/pgSQLユーザにとって重要ではありませんが、この知識は問題点の解析に有用です。それ以外の情報は、42.11.2に記述されています。

42.5. 基本的な文

本節および次節では、明示的にPL/pgSQLで解釈される、全ての種類の文について説明します。これらの種類の文として認められないものは全て、SQLコマンドであると仮定され、42.5.2および42.5.3において記述したように、メインデータベースエンジンに送信され実行されます。

42.5.1. 代入

値をPL/pgSQL変数に代入する場合は以下のように記述します。

```
variable { := | = } expression;
```

上述した通り、このような文中にある式は、メインデータベースエンジンに送信されるSELECT SQLコマンドによって評価されます。式は1つの値を生成しなければなりません (変数が行変数またはレコード変数の場合は行値となるかもしれません)。対象の変数は単純な変数(ブロック名で修飾可能)、行変数またはレコード変数のフィールド、または単純な変数またはフィールドとなる配列要素とすることができます。等号(=)がPL/SQLにおける代入記号(:=)の代わりに使用できます。

式の結果データ型が変数のデータ型に一致しない場合、値は代入キャスト(10.4を参照)と同様に変換されます。関係する二つのデータ型のための代入キャストが無いときには、PL/pgSQLインタプリタは結果値を、変数のデータ型の入力関数に続けて結果データ型の出力関数を適用することで、テキストとして変換しようとします。結果値の文字列形式が入力関数で受け付けることができない場合に、入力関数において実行時エラーが発生するかもしれないことに注意してください。

例:

```
tax := subtotal * 0.06;
my_record.user_id := 20;
```

42.5.2. 結果を伴わないコマンドの実行

例えば、RETURNING句のないINSERTのように、行を返さない任意のSQLのコマンドについては、単にそのコマンドを記述することによってPL/pgSQL関数の内部でコマンドを実行できます。

コマンドテキストに現れる全てのPL/pgSQL変数名は、パラメータとして扱われます。その後、実行時のパラメータ値として、その時点の変数値が提供されます。これは以前に述べた式に関する処理と同じです。42.11.1を参照してください。

SQLコマンドがこのように実行されると、42.11.2に記述したように、PL/pgSQLはコマンドのために、実行計画をキャッシュして再利用します。

式またはSELECT問い合わせを評価して結果を破棄することが、役に立つ場合があります。例えば、関数の呼び出しにおいて、副次的な成果を取得できるが、結果は無用である場合です。このような時PL/pgSQLでは、PERFORM文を使用してください。

```
PERFORM query;
```

これはqueryを実行し、その結果を破棄します。SQLのSELECT文と同じ方法でqueryを記述しますが、最初のキーワードSELECTをPERFORMに置き換えてください。WITH問い合わせに対しては、PERFORM を使用して、問い合わせをカッコ内に配置してください。(この場合、問い合わせは1行だけ返すことができます。) 結果を返さないコマンドと同様に、PL/pgSQL変数は問い合わせ内に置き換えられ、計画は同様にキャッシュされます。また、特殊な変数であるFOUNDは問い合わせ結果が1行でも生成された場合は真に設定され、生成されない場合は偽に設定されます(42.5.5を参照してください)。

> **注記**
>
> 直接SELECTを記述すれば、この結果を得ることができると考えるかもしれませんが、現時点でこれを行う方法はPERFORMしかありません。SELECTのように行を返すSQLコマンドは、エラーとして拒絶されます。なお、INTO句を有する時は例外であり、次節で説明します。

以下に例を示します。

```
PERFORM create_mv('cs_session_page_requests_mv', my_query);
```

42.5.3. 1行の結果を返す問い合わせの実行

(多分、複数列の)1行を返すSQLコマンドの結果は、レコード変数、行型の変数、スカラ変数のリストに代入することができます。これは、基本的なSQLコマンドを記述して、それにINTO句を追加することによって行われます。以下に例を示します。

```
SELECT select_expressions INTO [STRICT] target FROM ...;
INSERT ... RETURNING expressions INTO [STRICT] target;
UPDATE ... RETURNING expressions INTO [STRICT] target;
DELETE ... RETURNING expressions INTO [STRICT] target;
```

ここで、targetはレコード変数、行変数、あるいは、単純な変数とレコード/行変数のフィールドをカンマで区切ったリストです。PL/pgSQL変数により残りの問い合わせが置換され、行を返さないコマンドにおいて述べたように計画がキャッシュされます。このように作動するのは、RETURNINGを伴ったINSERT/UPDATE/DELETEとSELECTおよび行セットの結果を返すユーティリティコマンド(例えば、EXPLAIN)です。INTO句以外では、SQLコマンドはPL/pgSQLの外部に記述したものと同じです。

> **ヒント**
>
> 通常のPostgreSQLのSELECT INTO文では、INTOの対象は新しく作成されるテーブルです。しかし、INTOを伴ったSELECTでは、この解釈が通常と大きく異なることに注意してください。PL/pgSQL関数内

> 部でSELECTの結果からテーブルを作成したい場合は、CREATE TABLE ... AS SELECT構文を使用してください。

行または変数リストが対象に使用された場合、列数とデータ型において問い合わせの結果と対象の構造が正確に一致しなければなりません。さもないと、実行時エラーが発生します。レコード変数が対象の場合は、問い合わせ結果の列の行型に自身を自動的に設定します。

INTO句はSQLコマンドのほとんど任意の場所に記述することができます。習慣的には、SELECT文においてはselect_expressionsの直前または直後に、他のコマンドにおいては文の終わりに記述されます。将来のバージョンでPL/pgSQLのパーサがより厳格になる場合に備えて、この習慣に従うことを推奨します。

INTO句においてSTRICTが指定されない場合、targetは問い合わせが返す最初の行となり、行を返さない時はNULLとなります。(「最初の行」とはORDER BYを使用しないと定義できないことに注意してください。) 2行目以降の行の結果は、全て破棄されます。以下のように、特殊なFOUND変数(42.5.5を参照してください)を調べて、行が返されたかどうかを検査することができます。

```
SELECT * INTO myrec FROM emp WHERE empname = myname;
IF NOT FOUND THEN
    RAISE EXCEPTION 'employee % not found', myname;
END IF;
```

STRICTオプションが指定された場合、問い合わせは正確に1行を返さなければなりません。さもないと、行がない時はNO_DATA_FOUND、2行以上が返った時はTOO_MANY_ROWSという実行時エラーが生じます。エラーを捕捉したい時は、例外ブロックを使用できます。以下に例を示します。

```
BEGIN
    SELECT * INTO STRICT myrec FROM emp WHERE empname = myname;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            RAISE EXCEPTION 'employee % not found', myname;
        WHEN TOO_MANY_ROWS THEN
            RAISE EXCEPTION 'employee % not unique', myname;
END;
```

STRICTを指定したコマンドが成功すると、FOUND変数は常に真に設定されます。

PL/pgSQLはSTRICTが指定されない場合でも、RETURNINGを伴ったINSERT/UPDATE/DELETEが2行以上を返した時は、エラーとなります。なぜなら、どの1行を返すか決定するORDER BYのようなオプションが存在しないからです。

print_strict_paramsが関数に利用可能であり、かつ要求がSTRICTでないためにエラーが発生した場合、エラーメッセージのSTRICT部分は問い合わせに渡したパラメータに関する情報を含みます。plpgsql.print_strict_paramsを指定することにより、全ての関数のprint_strict_params設定を変更できます。しかし、変更後にコンパイルした関数にだけ有効です。コンパイルオプションを使用すれば、個々の関数を基準とした設定変更もできます。例を示します。

```
CREATE FUNCTION get_userid(username text) RETURNS int
```

```
AS $$
#print_strict_params on
DECLARE
userid int;
BEGIN
    SELECT users.userid INTO STRICT userid
        FROM users WHERE users.username = get_userid.username;
    RETURN userid;
END;
$$ LANGUAGE plpgsql;
```

失敗したとき、この関数は次のようなエラーメッセージを生成します。

```
ERROR:  query returned no rows
DETAIL:  parameters: $1 = 'nosuchuser'
CONTEXT:  PL/pgSQL function get_userid(text) line 6 at SQL statement
```

注記

STRICTオプションは、OracleのPL/SQLのSELECT INTOおよび関連した文に対応します。

SQLの問い合わせが返す複数行の結果を処理したい場合は、42.6.6を参照してください。

42.5.4. 動的コマンドの実行

PL/pgSQL関数の内部で、動的コマンド、つまり実行する度に別のテーブルや別のデータ型を使用するコマンドを生成したいということがよくあるでしょう。PL/pgSQLが通常行うコマンドの計画のキャッシュは(42.11.2で述べたように)このような状況では動作しません。この種の問題を扱うために、以下のEXECUTE文が用意されています。

```
EXECUTE command-string [ INTO [STRICT] target ] [ USING expression [, ... ] ];
```

ここで、command-stringは実行されるコマンドを含む(text型の)文字列を生成する式です。オプションのtargetはレコード変数、行変数、あるいは、単純な変数とレコード/行変数のフィールドをカンマで区切ったリストで、その中にコマンドの結果が格納されます。オプションのUSING式は コマンドに挿入される値を与えます。

PL/pgSQL変数は、この演算用のコマンド文字列へ置換されません。必要な変数の値はすべてコマンド文字列を作成する時に埋め込まなければなりません。もしくは、以下に説明するパラメータを使用することもできます。

また、EXECUTEを介して実行されるコマンド計画をキャッシュすることはありません。代わりに、コマンドは文が実行されるとき常に計画されます。したがって、異なるテーブルと列に対する操作を実行できるように、コマンド文字列を関数内部で動的に作成することができます。

INTO句は、行を返すSQLコマンドの結果を代入するべき場所を指定します。行または変数リストが用いられる時、それは問い合わせの結果の構造と正確に一致しなければなりません (レコード変数が使用される時、自動的に結果の構造と一致するように自身を構築させます)。複数の行が返された時、最初の行だけがINTO変数に代入されます。1行も返されない時、NULL がINTO変数に代入されます。INTO句が指定されない時、問い合わせの結果は捨てられます。

STRICTオプションが指定された時、問い合わせの結果が正確に1行の場合を除き、エラーとなります。

コマンド文字列はパラメータ値を使用可能で、それらは$1、$2等としてコマンドの中で参照されます。これらの記号はUSINGで与えられる値を参照します。この方式はデータの値をテキストとしてコマンド文字列の中に挿入する際、よく好まれます。それは値をテキストに変換、そしてその逆を行う場合の実行時オーバーヘッドを防止するとともに、引用符付けするとか、エスケープをする必要がないため、SQLインジェクション攻撃に対してより襲われにくくなります。以下に例を示します。

```
EXECUTE 'SELECT count(*) FROM mytable WHERE inserted_by = $1 AND inserted <= $2'
   INTO c
   USING checked_user, checked_date;
```

パラメータ記号はデータ値のみ使用可能です。もし動的に決定されるテーブルや列名を使用したい場合、テキストでコマンド文字列にそれらを挿入する必要があります。例えば、先行する問い合わせが、動的に選択されたテーブルに対して処理される必要がある時は、次のようにします。

```
EXECUTE 'SELECT count(*) FROM '
    || quote_ident(tabname)
    || ' WHERE inserted_by = $1 AND inserted <= $2'
   INTO c
   USING checked_user, checked_date;
```

よりきれいな方法はformat()の%I指定をテーブル名または列名に使うことです（改行で分かれた文字列は連結されます）。

```
EXECUTE format('SELECT count(*) FROM %I '
   'WHERE inserted_by = $1 AND inserted <= $2', tabname)
   INTO c
   USING checked_user, checked_date;
```

他にもパラメータ記号はSELECT、INSERT、UPDATE、DELETEコマンドでしか動作しない、という制限があります。他の種類の文(一般的にユーティリティ文と呼ばれます)では、単なるデータ値であったとしてもテキストの値として埋め込まなければなりません。

最初の例のように、単純な定数コマンドとUSINGパラメータを使ったEXECUTEは、コマンドを直接PL/pgSQLで書いて、PL/pgSQL変数を自動的に置換したものと機能的に同じです。重要な差異として、EXECUTEが現在のパラメータ値に特化した計画を生成し、コマンドを実行する度に計画を再作成することです。一方、PL/pgSQLはその他に汎用的な計画を作成し、再使用に備えキャッシュします。最適な計画がパラメータ値に大きく依存する場合、汎用的な計画が選択されないことを確保するために、EXECUTEの使用は助けになります。

SELECT INTOはEXECUTEでは現在サポートされません。代わりに、普通のSELECTコマンドを実行し、EXECUTEの一部としてINTOを記述してください。

> **注記**
>
> PL/pgSQL EXECUTE文はPostgreSQLサーバでサポートされているEXECUTESQL文とは関連がありません。サーバのEXECUTE文はPL/pgSQL関数内で使用することはできません(使用する必要もありません)。

例42.1 動的問い合わせの中の値の引用符付け

動的コマンドを使用する時、しばしば単一引用符をエスケープしなければなりません。関数本体における固定のテキストを引用符付けする推奨方法は、ドル引用符を使用する方法です。(ドル引用符を用いない旧式のコードを保有している場合は、42.12.1の概要を参照することが、理解しやすいコードへの変換作業の手助けになります)。

動的な値は引用符を含んでいる可能性があるので注意深い取り扱いが必要です。以下にformat()を使う例を示します(ここでは関数にドル引用符を用いる方法を使用すると仮定しているので、引用符を二重化する必要はありません)。

```
EXECUTE format('UPDATE tbl SET %I = $1 '
   'WHERE key = $2', colname) USING newvalue, keyvalue;
```

クオート関数を直接呼び出すことも可能です。

```
EXECUTE 'UPDATE tbl SET '
        || quote_ident(colname)
        || ' = '
        || quote_literal(newvalue)
        || ' WHERE key = '
        || quote_literal(keyvalue);
```

この例は、quote_identとquote_literal関数(9.4を参照)の使用方法を示しています。安全のため、列またはテーブル識別子を含む式は動的問い合わせに挿入する前にquote_identを介して渡されなくてはなりません。組み立てられるコマンドの中のリテラル文字列となるはずの値を含む式は、quote_literalを介して渡されなければなりません。これらの関数は、すべての特殊文字を適切にエスケープして埋め込んだ、二重引用符または単一引用符で囲まれた入力テキストを返すために、適切な手順を踏みます。

quote_literalはSTRICTラベル付けされているため、NULL引数で呼び出された場合、常にNULLを返します。上記の例で、newvalueまたはkeyvalueがNULLの場合、動的問合せ文字列全体がNULLとなり、EXECUTEからのエラーを導きます。quote_nullable関数を使用することで、この問題を回避することができます。その動作は、NULL引数付きで呼び出された場合に文字列NULLを返すことを除いてquote_literalと同一です。以下に例を示します。

```
EXECUTE 'UPDATE tbl SET '
        || quote_ident(colname)
        || ' = '
        || quote_nullable(newvalue)
```

```
        || ' WHERE key = '
        || quote_nullable(keyvalue);
```

NULLの可能性のある値を処理するのであれば、通常quote_literalの代わりにquote_nullableを使用しなければなりません。

いつものように、問い合わせの中のNULL値は意図しない結果を確実にもたらさないよう配慮をしなければなりません。例えば次のようなWHERE句の結果はどうなるのでしょう。

```
'WHERE key = ' || quote_nullable(keyvalue)
```

これはkeyvalueがNULLである限り成功しません。その理由は、等価演算子=をNULLオペランドで使用するとその結果は常にNULLとなるからです。NULLを通常のキーの値と同じように動作させたい場合、上記を、 以下のように書き換えなければなりません。

```
'WHERE key IS NOT DISTINCT FROM ' || quote_nullable(keyvalue)
```

（現時点では、IS NOT DISTINCT FROMは=よりもより効率性が少なく扱われますので、必要に迫られた場合以外は行わないようにしてください。NULLとIS DISTINCTについての更なる情報は9.2を参照してください。）

ドル引用符は固定のテキストを引用符付けする場合のみ有用であるということに注意してください。この例を次のように記述するのは非常に悪い考えです。

```
EXECUTE 'UPDATE tbl SET '
        || quote_ident(colname)
        || ' = $$'
        || newvalue
        || '$$ WHERE key = '
        || quote_literal(keyvalue);
```

なぜなら、newvalueの内容がたまたま$$を含む時は、途中で次の処理へ移ってしまうからです。同様の不測事態は、ドル引用符の他の区切り文字を選んだ時も起こります。したがって、テキストの内容を把握していない時は、安全にテキストを引用符付けするために、quote_literal、quote_nullable、またはquote_ident関数を適切に使用しなければなりません。

動的なSQL文もformat関数（9.4.1を参照）を使って安全に作ることができます。例を示します。

```
EXECUTE format('UPDATE tbl SET %I = %L '
   'WHERE key = %L', colname, newvalue, keyvalue);
```

%Iはquote_identと同等で、%Lはquote_nullableと同等です。format関数はUSING句と共に使用できます。

```
EXECUTE format('UPDATE tbl SET %I = $1 WHERE key = $2', colname)
   USING newvalue, keyvalue;
```

変数が、無条件にテキストに変換されて%Lで引用符付けされることなく、固有のデータ形式で処理されるため、この形式はより優れています。

動的問い合わせとEXECUTEの長大な例は例 42.10にあります。それは新しい関数を定義するためにCREATE FUNCTIONコマンドを組み立て実行するものです。

42.5.5. 結果ステータスの取得

コマンドの効果を判断するにはいくつか方法があります。最初の方法は以下のような形式のGET DIAGNOSTICSを使用する方法です。

```
GET [ CURRENT ] DIAGNOSTICS variable { = | := } item [ , ... ];
```

このコマンドによってシステムステータスインジケータを取り出すことができます。CURRENTは無意味な単語です(しかし42.6.8.1のGET STACKED DIAGNOSTICSも参照してください)。各itemは、指定されたvariable(これは受け取るために正しいデータ型でなければなりません)に代入されるステータス値を識別するキーワードです。現在使用可能なステータス項目は、表 42.1で示されています。代入記号(:=)が標準SQLにおける等号(=)の代わりに使用できます。以下に例を示します。

```
GET DIAGNOSTICS integer_var = ROW_COUNT;
```

表42.1 使用できるステータス項目

名前	型	説明
ROW_COUNT	bigint	最後のSQLコマンドにより処理された行数
PG_CONTEXT	text	現在の呼び出しスタックを記述したテキストの行 (42.6.9を参照)

コマンドの効果を判断する2番目の方法は、FOUNDというboolean型の特殊な変数を検査することです。PL/pgSQLの各関数呼び出しで使用される際、FOUNDは最初は偽に設定されています。以下のように、それぞれの文の種類によって設定が変更されます。

- SELECT INTO文は、行が代入された場合は真、返されなかった場合は偽をFOUNDに設定します。
- PERFORM文は、1つ以上の行が生成(破棄)された場合は真、まったく生成されなかった場合は偽をFOUNDに設定します。
- UPDATE、INSERT、およびDELETE文は、少なくとも1行が影響を受けた場合は真、まったく影響を受けなかった場合は偽をFOUNDに設定します。
- FETCH文は、行が返された場合は真、まったく返されなかった場合は偽をFOUNDに設定します。
- MOVE文は、カーソルの移動が成功した場合は真、失敗した場合は偽をFOUNDに設定します。
- FOR文またはFOREACH文は、1回以上繰り返しが行われた場合は真、行われなかった場合は偽をFOUNDに設定します。FOUNDはループが終了した際、このように設定されます。ループ実行中はループ文によるFOUNDの変更はありません。ただし、ループ本体内の他種類の文を実行することによって、変更されるかもしれません。
- RETURN QUERYとRETURN QUERY EXECUTE文は、問い合わせが行を1つでも返せば真、行が返されなければ偽をFOUNDに設定します。

他のPL/pgSQL文はFOUNDの状態を変更しません。特に、EXECUTEはGET DIAGNOSTICSの出力を変更しますが、FOUNDを変更しないことに注意してください。

FOUNDはそれぞれのPL/pgSQL関数内部のローカル変数です。FOUNDに対して行われた全ての変更は、現在の関数にのみ影響します。

42.5.6. まったく何もしない

何もしないプレースホルダ文が有用になることがあります。例えば、IF/THEN/ELSE文の一部が空文であることを明示したい時です。このような目的にはNULL文を使用します。

```
NULL;
```

例えば、次の2つのコードは同等です。

```
BEGIN
    y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN

        NULL;  -- 誤りを無視する
END;
```

```
BEGIN
    y := x / 0;
EXCEPTION

    WHEN division_by_zero THEN  -- 誤りを無視する
END;
```

どちらが望ましいと思うかは、好みの問題です。

> **注記**
>
> OracleのPL/SQLでは無記述の文は許されませんので、こうした状況ではNULL文が*必須*です。しかしPL/pgSQLでは無記述の文が許可されています。

42.6. 制御構造

制御構造はおそらくPL/pgSQLの最も有用（かつ重要）な部分です。PL/pgSQLの制御構造を使用して、PostgreSQLのデータを非常に柔軟、強力に操作することができます。

42.6.1. 関数からの復帰

関数からデータを返すために使用できるコマンドが2つあります。RETURNおよびRETURN NEXTです。

42.6.1.1. RETURN

```
RETURN expression;
```

式を持つRETURNは関数を終了し、expressionの値を呼び出し元に返します。この形式は集合を返さないPL/pgSQL関数で使用されます。

スカラ型を返す関数において、代入のところで説明したように、式の結果は自動的に関数の戻り値の型にキャストされます。しかし、複合(行)値を返すためには、要求された列集合を正確に導出する式を記述しなければなりません。これにより、明示的なキャストの使用が必要となることがあります。

出力パラメータを持った関数を宣言した時は、式の無いRETURNを記述してください。その時点における出力パラメータの値が返されます。

voidを返すように関数を宣言した場合でも、関数を直ちに抜け出すためにRETURNを使用できますが、RETURNの後に式を記述しないでください。

関数の戻り値は未定義とさせたままにすることはできません。制御が、RETURN文が見つからない状態で関数の最上位のブロックの終わりまで達した時、実行時エラーが発生します。しかし、この制限は出力パラメータを持った関数及びvoidを返す関数には当てはまりません。このような場合は最上位のブロックが終わった時、RETURN文が自動的に実行されます。

例を示します。

```
-- スカラ型を返す関数
RETURN 1 + 2;
RETURN scalar_var;


-- 複合型を返す関数
RETURN composite_type_var;

RETURN (1, 2, 'three'::text);  -- 正しい型の列にキャストしなければなりません
```

42.6.1.2. RETURN NEXTおよびRETURN QUERY

```
RETURN NEXT expression;
RETURN QUERY query;
RETURN QUERY EXECUTE command-string [ USING expression [, ... ] ];
```

PL/pgSQL関数がSETOF sometypeを返すように宣言した場合、後続の処理が多少違います。この場合、戻り値の個々の項目は、RETURN NEXTコマンドまたはRETURN QUERYコマンドで指定されます。そして、引数のない最後のRETURNコマンドにより、関数が実行を終了したことが示されます。RETURN NEXTは、スカラ型および複合型の両方で使用することができます。複合型の場合、結果の「テーブル」全体が返されます。RETURN

QUERYは、問い合わせを実行した結果を関数の結果集合に追加します。RETURN NEXTとRETURN QUERYは、単一の集合を返す関数の中で自由に混合できます。この場合、連結されたものが結果となります。

実際には、RETURN NEXTおよびRETURN QUERYは関数から戻りません。単に関数の結果集合に行を追加しているだけです。そして、その実行はPL/pgSQL関数内の次の文に継続します。RETURN NEXTまたはRETURN QUERYコマンドが連続して実行されると、結果集合が作成されます。最後の、引数を持ってはならないRETURNにより、関数の終了を制御します (または制御を関数の最後に移すことができます)。

RETURN QUERYにはRETURN QUERY EXECUTEという亜種があり、それは問い合わせが動的に実行されることを指定します。パラメータ式を、EXECUTEコマンド内と全く同じように、USINGによって演算された問い合わせ文字列に挿入することができます。

出力パラメータを持つ関数を宣言した時は、式の無いRETURN NEXTだけを記述してください。実行の度に、その時点における出力パラメータの値が、関数からの戻り値のために結果の行として保存されます。出力パラメータを持つ集合を返す関数を作成するためには、出力パラメータが複数の時はSETOF recordを返すように関数を宣言し、単一のsometype型の出力パラメータの時はSETOF sometypeを返すように関数を宣言しなければならないことに注意してください。

RETURN NEXTを使用する関数の例を以下に示します。

```
CREATE TABLE foo (fooid INT, foosubid INT, fooname TEXT);
INSERT INTO foo VALUES (1, 2, 'three');
INSERT INTO foo VALUES (4, 5, 'six');

CREATE OR REPLACE FUNCTION get_all_foo() RETURNS SETOF foo AS
$BODY$
DECLARE
    r foo%rowtype;
BEGIN
    FOR r IN
        SELECT * FROM foo WHERE fooid > 0
    LOOP

        -- ここで処理を実行できます
        RETURN NEXT r; -- SELECTの現在の行を返します
    END LOOP;
    RETURN;
END;
$BODY$
LANGUAGE plpgsql;

SELECT * FROM get_all_foo();
```

RETURN QUERYを使用する関数の例を以下に示します。

```
CREATE FUNCTION get_available_flightid(date) RETURNS SETOF integer AS
$BODY$
BEGIN
```

```
    RETURN QUERY SELECT flightid
                   FROM flight
                  WHERE flightdate >= $1
                    AND flightdate < ($1 + 1);


-- 実行が終わっていないので、
      行が返されたか検査して、
      行がなければ例外を発生させます。
    IF NOT FOUND THEN
        RAISE EXCEPTION 'No flight at %.', $1;
    END IF;

    RETURN;
 END;
$BODY$
LANGUAGE plpgsql;


-- 利用できるフライトを返し、フライトがない場合は例外を発生させます。
SELECT * FROM get_available_flightid(CURRENT_DATE);
```

注記

上記のように、RETURN NEXTおよびRETURN QUERYの現在の実装では、関数から返される前に結果集合全体を保管します。これにより、PL/pgSQL関数が非常に大量の結果集合を返した場合、性能が低下する可能性があります。メモリの枯渇を避けるため、データはディスクに書き込まれます。しかし、関数自体は結果集合全体が生成されるまでは戻りません。将来のPL/pgSQLのバージョンでは、この制限を受けずに集合を返す関数をユーザが定義できるようになるかもしれません。現在、データがディスクに書き込まれ始まる時点はwork_mem設定変数によって制御されています。大量の結果集合を保管するのに十分なメモリがある場合、管理者はこのパラメータの値を大きくすることを考慮すべきです。

42.6.2. プロシージャからの戻り

プロシージャは戻り値を持ちません。したがって、プロシージャはRETURN文なしで終了できます。早期にコードを抜けるためにRETURN文を使いたいときには、式を伴わないRETURNだけを書いてください。

プロシージャが出力パラメータを持っている場合、出力パラメータ変数の最終の値が呼び出し元に返されます。

42.6.3. プロシージャを呼び出す

PL/pgSQLの関数、プロシージャ、DOブロックは、CALLを使ってプロシージャを呼び出しできます。CALLを普通のSQLで実行する場合とは、出力パラメータの扱いが異なります。プロシージャの各INOUTパラメータ

はCALL文の変数と対応しなければならず、プロシージャが返すものは全て、CALL文が返った後にこの変数に書き戻されます。以下に例を示します。

```
CREATE PROCEDURE triple(INOUT x int)
LANGUAGE plpgsql
AS $$
BEGIN
    x := x * 3;
END;
$$;

DO $$
DECLARE myvar int := 5;
BEGIN
  CALL triple(myvar);
  RAISE NOTICE 'myvar = %', myvar;  -- prints 15
END;
$$;
```

42.6.4. 条件分岐

IFとCASE文はある条件に基づいて代わりのコマンドを実行させます。PL/pgSQLには、以下のような3つのIFの形式があります。

- IF ... THEN ... END IF
- IF ... THEN ... ELSE ... END IF
- IF ... THEN ... ELSIF ... THEN ... ELSE ... END IF

また、以下のような2つのCASEの形式があります。

- CASE ... WHEN ... THEN ... ELSE ... END CASE
- CASE WHEN ... THEN ... ELSE ... END CASE

42.6.4.1. IF-THEN

```
IF boolean-expression THEN
    statements
END IF;
```

IF-THEN文は最も単純なIFの形式です。THENとEND IFの間の文が条件が真の場合に実行されます。さもなければそれらは飛ばされます。

例:

```
IF v_user_id <> 0 THEN
    UPDATE users SET email = v_email WHERE user_id = v_user_id;
END IF;
```

42.6.4.2. IF-THEN-ELSE

```
IF boolean-expression THEN
    statements
ELSE
    statements
END IF;
```

IF-THEN-ELSE文はIF-THENに加え、条件評価が偽の場合に実行すべき代替となる文の集合を指定することができます。(これには条件がNULLと評価した場合も含まれることに注意してください。)

例:

```
IF parentid IS NULL OR parentid = ''
THEN
    RETURN fullname;
ELSE
    RETURN hp_true_filename(parentid) || '/' || fullname;
END IF;
```

```
IF v_count > 0 THEN
    INSERT INTO users_count (count) VALUES (v_count);
    RETURN 't';
ELSE
    RETURN 'f';
END IF;
```

42.6.4.3. IF-THEN-ELSIF

```
IF boolean-expression THEN
    statements
[ ELSIF boolean-expression THEN
    statements
[ ELSIF boolean-expression THEN
    statements
    ...
]
]
[ ELSE
```

```
    statements ]
END IF;
```

選択肢が2つだけではなくより多くになる場合があります。IF-THEN-ELSIFは、順番に複数の代替手段を検査する、より便利な方法を提供します。IF条件は最初の真である結果が見つかるまで連続して検査されます。そして関連した文が実行され、その後END IF以降の次の文に制御が渡されます。(以降にあるIF条件の検査はすべて実行されません。) 全てのIF条件が真でない場合、ELSEブロックが(もし存在すれば)実行されます。機能的には、IF-THEN-ELSE-IF-THENコマンドを入れ子にしたものと同じですが、必要なEND IFは1つだけです。

以下に例を示します。

```
IF number = 0 THEN
    result := 'zero';
ELSIF number > 0 THEN
    result := 'positive';
ELSIF number < 0 THEN
    result := 'negative';
ELSE

    -- ふうむ、残る唯一の可能性はその値がNULLであることだ
    result := 'NULL';
END IF;
```

ELSIFキーワードはELSEIFのように書くことができます。

同じ作業を遂行する別の方法は、以下の例のようにIF-THEN-ELSE文を入れ子にすることです。

```
IF demo_row.sex = 'm' THEN
    pretty_sex := 'man';
ELSE
    IF demo_row.sex = 'f' THEN
        pretty_sex := 'woman';
    END IF;
END IF;
```

しかし、この方法はそれぞれのIFに対応するEND IFの記述が必要です。従って、多くの選択肢がある場合ELSIFを使用するよりも厄介です。

42.6.4.4. 単純なCASE

```
CASE search-expression
    WHEN expression [, expression [ ... ]] THEN
      statements
  [ WHEN expression [, expression [ ... ]] THEN
      statements
```

```
    ... ]
  [ ELSE
      statements ]
END CASE;
```

CASEの単純な形式はオペランドの等価性にもとづく条件的実行を提供します。search-expressionは（一度だけ）評価され、その後WHEN句内のそれぞれのexpressionと比較されます。一致するものが見つかると、関連したstatementsが実行され、END CASEの次の文に制御が渡されます。（以降のWHEN式は評価されません。）一致するものが見つからない場合、ELSE statementsが実行されますが、ELSEが無いときはCASE_NOT_FOUND例外を引き起こします。

以下は簡単な例です。

```
CASE x
    WHEN 1, 2 THEN
        msg := 'one or two';
    ELSE
        msg := 'other value than one or two';
END CASE;
```

42.6.4.5. 検索付きCASE

```
CASE
    WHEN boolean-expression THEN
      statements
  [ WHEN boolean-expression THEN
      statements
    ... ]
  [ ELSE
      statements ]
END CASE;
```

CASEの検索された形式は論理値式の真の結果に基づく条件付き実行を提供します。それぞれのWHEN句のboolean-expressionはtrueとなる1つが見つかるまで順番に評価されます。その後、関連するstatementsが実行され、その結果END CASEの次の文に制御が渡されます。（以降のWHEN式は評価されません。）真となる結果が見つからない場合、ELSE statementsが実行されますが、ELSEが存在しないときはCASE_NOT_FOUND例外を引き起こします。

以下は簡単な例です。

```
CASE
    WHEN x BETWEEN 0 AND 10 THEN
        msg := 'value is between zero and ten';
    WHEN x BETWEEN 11 AND 20 THEN
        msg := 'value is between eleven and twenty';
```

```
END CASE;
```

この形式のCASEは、判定基準が省略されたELSE句に達した場合に何もしないのではなくエラーなる点を除き、IF-THEN-ELSIFと全く同一です。

42.6.5. 単純なループ

LOOP、EXIT、CONTINUE、WHILE、FOR、FOREACH文を使用して、PL/pgSQL関数で、一連のコマンドを繰り返すことができます。

42.6.5.1. LOOP

```
[ <<label>> ]
LOOP
    statements
END LOOP [ label ];
```

LOOPは、EXIT文またはRETURN文によって終了されるまで無限に繰り返される、条件なしのループを定義します。省略可能なlabelは、ネステッドループにおいてEXITおよびCONTINUE文がどのレベルの入れ子を参照するかを指定するために使用されます。

42.6.5.2. EXIT

```
EXIT [ label ] [ WHEN boolean-expression ];
```

labelが指定されない場合、最も内側のループを終わらせ、END LOOPの次の文がその後に実行されます。labelが指定された場合、それは現在またはその上位のネステッドループやブロックのラベルである必要があります。その後、指名されたループまたはブロックを終わらせ、そのループまたはブロックの対応するENDの次の文に制御を移します。

WHENが指定された場合、boolean-expressionが真の場合のみループの終了が起こります。さもなければ、EXITの後の行に制御が移ります。

EXITは、すべての種類のループと共に使用できます。条件なしのループでの使用に限定されません。

BEGINブロックと共に使用した時、EXITによりブロックの次の文に制御が移ります。この目的のためにラベルが使用されなければならないことに注意してください。ラベル無しのEXITはBEGINブロックに対応するとは決して考えられません。（これは、ラベル無しのEXITがBEGINブロックに対応することを許容するPostgreSQLの8.4より前のリリースからの変更です。）

例：

```
LOOP
```

```
    -- 何らかの演算
    IF count > 0 THEN

        EXIT;  -- ループを抜け出す
    END IF;
END LOOP;

LOOP

-- 何らかの演算
    EXIT WHEN count > 0;  -- 上例と同じ結果
END LOOP;

<<ablock>>
BEGIN

    -- 何らかの演算
    IF stocks > 100000 THEN

        EXIT ablock;  -- これによりBEGINブロックを抜け出す
    END IF;

    -- stokcs > 100000 であればここでの演算は省略
END;
```

42.6.5.3. CONTINUE

```
CONTINUE [ label ] [ WHEN boolean-expression ];
```

labelが無い場合、すぐ外側のループの次の繰り返しが開始されます。すなわち、ループ本体の残りの文は飛ばされて、他のループの繰り返しが必要かどうかを決めるため、制御がループ制御式(もし存在すれば)に戻ります。labelが存在する場合、実行を継続するループのラベルを指定します。

WHENが指定された場合、boolean-expressionが真の場合のみループにおける次の繰り返しが始まります。さもなければ、CONTINUEの後の行に制御が移ります。

CONTINUEは全ての種類のループで使用可能です。条件なしのループに限定されません。

例

```
LOOP

    -- 何らかの演算
    EXIT WHEN count > 100;
    CONTINUE WHEN count < 50;
```

```
    -- 50から100を数える、何らかの演算
END LOOP;
```

42.6.5.4. WHILE

```
[ <<label>> ]
WHILE boolean-expression LOOP
    statements
END LOOP [ label ];
```

WHILE文はboolean-expressionの評価が真である間、一連の文を繰り返します。条件式は、ループ本体に入る前にのみ検査されます。

以下に例を示します。

```
WHILE amount_owed > 0 AND gift_certificate_balance > 0 LOOP

    -- ここで演算をいくつか行います。
END LOOP;

WHILE NOT done LOOP

    -- ここで演算をいくつか行います。
END LOOP;
```

42.6.5.5. 整数FORループ

```
[ <<label>> ]
FOR name IN [ REVERSE ] expression .. expression [ BY expression ] LOOP
    statements
END LOOP [ label ];
```

この形式のFORは整数値の範囲内で繰り返すループを生成します。name変数はinteger型として自動的に定義され、ループ内部のみで存在します（ループ外部で定義しても、ループ内部では全て無視されます）。範囲の下限、上限として与えられる2つの式はループに入った時に一度だけ評価されます。BY句を指定しない時の繰り返し刻みは1ですが、BY句を用いて指定でき、ループに入った時に一度だけ評価されます。REVERSEが指定された場合、繰り返し刻みの値は加算されるのではなく、繰り返しごとに減算されます。

整数FORループの例を以下に示します。

```
FOR i IN 1..10 LOOP

    -- i はループ内で 1、
      2、
```

```
      3、
      4、
      5、
      6、
      7、
      8、
      9、
      10 の値を取ります。
END LOOP;

FOR i IN REVERSE 10..1 LOOP

    -- i はループ内で 10、
      9、
      8、
      7、
      6、
      5、
      4、
      3、
      2、
      1 の値を取ります。
END LOOP;

FOR i IN REVERSE 10..1 BY 2 LOOP

    -- i はループ内で 10、8、6、4、2 の値を取ります。
END LOOP;
```

下限が上限よりも大きい(REVERSEの場合はより小さい)場合、ループ本体はまったく実行されません。エラーは発生しません。

labelをFORループに付加することにより、labelを用いて修飾した名前の整数ループ変数を参照できます。

42.6.6. 問い合わせ結果による繰り返し

別の種類のFORループを使用して、問い合わせの結果を繰り返し、そのデータを扱うことができます。以下に構文を示します。

```
[ <<label>> ]
FOR target IN query LOOP
    statements
END LOOP [ label ];
```

targetは、レコード変数、行変数またはカンマで区切ったスカラ変数のリストです。targetには順次、queryの結果の全ての行が代入され、各行に対してループ本体が実行されます。以下に例を示します。

```
CREATE FUNCTION refresh_mviews() RETURNS integer AS $$
DECLARE
    mviews RECORD;
BEGIN
    RAISE NOTICE 'Refreshing all materialized views...';

    FOR mviews IN
       SELECT n.nspname AS mv_schema,
              c.relname AS mv_name,
              pg_catalog.pg_get_userbyid(c.relowner) AS owner
         FROM pg_catalog.pg_class c
    LEFT JOIN pg_catalog.pg_namespace n ON (n.oid = c.relnamespace)
        WHERE c.relkind = 'm'
     ORDER BY 1
    LOOP


        -- ここで"mviews"はcs_materialized_viewsの1つのレコードを持ちます

        RAISE NOTICE 'Refreshing materialized view %.% (owner: %)...',
                     quote_ident(mviews.mv_schema),
                     quote_ident(mviews.mv_name),
                     quote_ident(mviews.owner);
        EXECUTE format('REFRESH MATERIALIZED VIEW %I.%I', mviews.mv_schema, mviews.mv_name);
    END LOOP;

    RAISE NOTICE 'Done refreshing materialized views.';
    RETURN 1;
END;
$$ LANGUAGE plpgsql;
```

このループがEXIT文で終了した場合、最後に割り当てられた行の値はループを抜けた後でもアクセスすることができます。

この種類のFOR文のqueryとしては、呼び出し元に行を返すSQLコマンドをすべて使用できます。通常はSELECTですが、RETURNING句を持つINSERT、UPDATEまたはDELETEも使用できます。EXPLAINなどのユーティリティコマンドも作動します。

PL/pgSQL変数は問い合わせテキストに置き換えられます。問い合わせ計画は、42.11.1および42.11.2で述べたように、再利用のためにキャッシュされます。

FOR-IN-EXECUTE文は行を繰り返すもう1つの方法です。

```
[ <<label>> ]
FOR target IN EXECUTE text_expression [ USING expression [, ... ] ] LOOP
    statements
END LOOP [ label ];
```

この方法は、問い合わせのソースが文字列式で指定される点を除き、前の形式と似ています。この式はFORループの各項目で評価され、再計画が行われます。これにより、プログラマは、通常のEXECUTE文と同じように事前に計画された問い合わせによる高速性と、動的な問い合わせの持つ柔軟性を選択することができます。EXECUTEの場合と同様、パラメータ値はUSINGにより動的コマンドに挿入できます。

結果を通して繰り返さなければならない問い合わせを指定するもう1つの方法として、カーソルの宣言があります。これは42.7.4で説明します。

42.6.7. 配列を巡回

FOREACHループはFORループにとてもよく似ています。しかし、SQL 問い合わせが抽出した行を繰り返す代わりに、配列の要素を繰り返します。(一般的にFOREACHは、複合値で表現される構成要素の巡回を意味しますが、配列でない複合値も巡回する亜種が将来は追加されるかもしれません。) 配列を巡回するFOREACH文を示します。

```
[ <<label>> ]
FOREACH target [ SLICE number ] IN ARRAY expression LOOP
    statements
END LOOP [ label ];
```

SLICEがない、またはSLICE 0が指定された場合、ループはexpressionによって評価されて作成された配列の各要素を繰り返します。target変数が各要素の値に順次割り当てられ、各要素に対してループ本体が実行されます。整数配列の要素を巡回する例を示します。

```
CREATE FUNCTION sum(int[]) RETURNS int8 AS $$
DECLARE
  s int8 := 0;
  x int;
BEGIN
  FOREACH x IN ARRAY $1
  LOOP
    s := s + x;
  END LOOP;
  RETURN s;
END;
$$ LANGUAGE plpgsql;
```

配列の次元数に関係なく、要素は格納した順番で処理されます。通常targetは単一の変数ですが、複合値(レコード)の配列を巡回するときは、変数のリストも可能です。その場合、配列の各要素に対して、変数は複合値(レコード)の列から連続的に割り当てられます。

正のSLICE値を持つ場合、FOREACHは単一の要素ではなく多次元配列の低次元部分配列を通して繰り返します。SLICE値は、配列の次元数より小さい整数定数でなければなりません。target変数は配列でなければなりません。この変数は、配列値から連続した部分配列を受けとります ここで部分配列はSLICEで指定した次数となります。以下に1次元の部分配列を通した繰り返しの例を示します。

```
CREATE FUNCTION scan_rows(int[]) RETURNS void AS $$
```

```
DECLARE
  x int[];
BEGIN
  FOREACH x SLICE 1 IN ARRAY $1
  LOOP
    RAISE NOTICE 'row = %', x;
  END LOOP;
END;
$$ LANGUAGE plpgsql;

SELECT scan_rows(ARRAY[[1,2,3],[4,5,6],[7,8,9],[10,11,12]]);

NOTICE:  row = {1,2,3}
NOTICE:  row = {4,5,6}
NOTICE:  row = {7,8,9}
NOTICE:  row = {10,11,12}
```

42.6.8. エラーの捕捉

デフォルトでは、PL/pgSQL関数の内部でエラーが発生すると関数とそれを囲むトランザクションをアボートします。BEGINブロックおよびEXCEPTION句を使用すれば、エラーを捕捉してその状態から回復できます。その構文は通常のBEGINブロックの構文を拡張したものです。

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements
EXCEPTION
    WHEN condition [ OR condition ... ] THEN
        handler_statements
    [ WHEN condition [ OR condition ... ] THEN
          handler_statements
      ... ]
END;
```

エラーが発生しない時、この形式のブロックは単に全てのstatementsを実行し、ENDの次の文に制御が移ります。しかし、statementsの内部でエラーが発生すると、それ以後のstatementsの処理は中断され、EXCEPTIONリストに制御が移ります。そしてリストの中から、発生したエラーと合致する最初のconditionを探します。合致するものがあれば、対応するhandler_statementsを実行し、ENDの次の文に制御が移ります。合致するものがなければ、EXCEPTION句が存在しないのと同じで、エラーは外側に伝播します。EXCEPTIONを含んだ外側のブロックはエラーを捕捉できますが、失敗すると関数の処理は中断されます。

全てのconditionの名前は付録Aに示したもののいずれかを取ることができます。分類名はそこに分類される全てのエラーに合致します。OTHERSという特別の状態名はQUERY_CANCELEDとASSERT_FAILUREを除く全てのエラーに合致します。(QUERY_CANCELEDとASSERT_FAILUREを名前で捕捉することは可能ですが、賢明ではあ

りません。）状態名は大文字小文字を区別しません。同時に、エラー状態はSQLSTATEコードで指定可能です。例えば以下は等価です。

```
WHEN division_by_zero THEN ...
WHEN SQLSTATE '22012' THEN ...
```

エラーが該当するhandler_statements内部で新たに発生した時、EXCEPTION句はそのエラーを捕捉できずエラーは外側に伝播します。なお、上位のEXCEPTION句はそのエラーを捕捉できます。

EXCEPTION句がエラーを捕捉した時、PL/pgSQL関数のローカル変数はエラーが起こった後の状態を保ちます。しかし、ブロック内部における永続的なデータベースの状態は、ロールバックされます。そのような例を以下に示します。

```
INSERT INTO mytab(firstname, lastname) VALUES('Tom', 'Jones');
BEGIN
    UPDATE mytab SET firstname = 'Joe' WHERE lastname = 'Jones';
    x := x + 1;
    y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN
        RAISE NOTICE 'caught division_by_zero';
        RETURN x;
END;
```

制御が変数yの代入に移ると、division_by_zeroエラーとなり、EXCEPTION句がそのエラーを捕捉します。RETURN文による関数の戻り値は、1を加算した後のxの値となりますが、UPDATEコマンドによる結果はロールバックされます。しかし、前のブロックのINSERTコマンドはロールバックされません。したがって、データベースの内容の最終結果はTom Jonesであり、Joe Jonesではありません。

> **ヒント**
>
> EXCEPTION句を含んだブロックの実行に要する時間は、含まないブロックに比べてとても長くなります。したがって、必要のない時にEXCEPTIONを使用してはいけません。

例42.2 UPDATE/INSERTの例外

これはUPDATEまたはINSERTの実行における例外処理を使用した適当な例題です。アプリケーションでは実際にこの方式を使うよりも、ON CONFLICT DO UPDATEを伴ったINSERTを使うことが推奨されます。本例は主としてPL/pgSQLの制御構造の使い方を示すものです。

```
CREATE TABLE db (a INT PRIMARY KEY, b TEXT);

CREATE FUNCTION merge_db(key INT, data TEXT) RETURNS VOID AS
$$
BEGIN
    LOOP
```

```
        -- 最初にキーを更新する
        UPDATE db SET b = data WHERE a = key;
        IF found THEN
            RETURN;
        END IF;

        -- キーが存在しないので、
      キーの挿入を試行する
        -- 他者がすでに同一のキーを挿入していたならば
        -- 一意性に違反する欠陥となります
        BEGIN
            INSERT INTO db(a,b) VALUES (key, data);
            RETURN;
        EXCEPTION WHEN unique_violation THEN

            -- 何もしないで、更新を再試行します
        END;
    END LOOP;
END;
$$
LANGUAGE plpgsql;

SELECT merge_db(1, 'david');
SELECT merge_db(1, 'dennis');
```

このコーディングではunique_violationエラーの原因がINSERTによるものであり、テーブルのトリガ関数内部のINSERTによるものでないと仮定します。また、テーブルに2つ以上の一意インデックスが存在した場合、どちらのインデックスがエラーの原因になろうと操作を再試行するので、誤作動となります。捕捉したエラーが予測したものであるか検証するために、次節で記述するエラー情報を利用すれば、より安全となります。

42.6.8.1. エラーに関する情報の取得

例外ハンドラはしばしば、起こった特定のエラーを識別する必要があります。PL/pgSQLで現在の例外に関する情報を取得する方法は2つあります。特殊な変数とGET STACKED DIAGNOSTICSコマンドです。

例外ハンドラの内部では、特殊な変数SQLSTATE変数が起こった例外に対応したエラーコード（表 A.1のエラーコード表を参照してください）を保有します。特殊な変数SQLERRMは例外に関連したエラーメッセージを保有します。これらの変数は、例外ハンドラの外部では定義されていません。

例外ハンドラの内部では、GET STACKED DIAGNOSTICSコマンドを使用して、現在の例外に関する情報を取り出すこともできます。次のようなやり方となります。

```
GET STACKED DIAGNOSTICS variable { = | := } item [ , ... ];
```

各itemは、指定された変数（これは受け取るために正しいデータ型でなければなりません）に代入される状態値を識別するキーワードです。現在使用可能なステータス項目は表 42.2に表示されています。

表42.2 エラーの診断値

名前	型	説明
RETURNED_SQLSTATE	text	例外のSQLSTATEエラーコード
COLUMN_NAME	text	例外に関する列名
CONSTRAINT_NAME	text	例外に関する制約名
PG_DATATYPE_NAME	text	例外に関するデータ型名
MESSAGE_TEXT	text	例外の主要なメッセージのテキスト
TABLE_NAME	text	例外に関するテーブル名
SCHEMA_NAME	text	例外に関するスキーマ名
PG_EXCEPTION_DETAIL	text	例外の詳細なメッセージのテキスト、存在する場合
PG_EXCEPTION_HINT	text	例外のヒントとなるメッセージのテキスト、存在する場合
PG_EXCEPTION_CONTEXT	text	例外時における呼び出しスタックを記述するテキストの行（42.6.9を参照）

例外が項目の値を設定しない場合、空文字列が返されます。

以下に例を示します。

```
DECLARE
  text_var1 text;
  text_var2 text;
  text_var3 text;
BEGIN

  -- 例外を引き起こす処理
  ...
EXCEPTION WHEN OTHERS THEN
  GET STACKED DIAGNOSTICS text_var1 = MESSAGE_TEXT,
                          text_var2 = PG_EXCEPTION_DETAIL,
                          text_var3 = PG_EXCEPTION_HINT;
END;
```

42.6.9. 実行位置情報の取得

以前、42.5.5に記載されていたGET DIAGNOSTICSコマンドは、現在の実行状態に関する情報を取得します（対して、前述のGET STACKED DIAGNOSTICSコマンドは一つ前のエラー時点の実行状態を報告します）。これのPG_CONTEXTステータス項目は現在の実行位置を識別するのに役立ちます。PG_CONTEXTは呼び出しスタックを記述したテキスト行を含むテキスト文字列を返します。最初の行は現在の関数と現在実行中のGET DIAGNOSTICSコマンドを参照します。次行および後の行は、呼び出しスタック上の呼び出し関数を参照します。例を示します。

```
CREATE OR REPLACE FUNCTION outer_func() RETURNS integer AS $$
```

```
BEGIN
  RETURN inner_func();
END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION inner_func() RETURNS integer AS $$
DECLARE
  stack text;
BEGIN
  GET DIAGNOSTICS stack = PG_CONTEXT;
  RAISE NOTICE E'--- Call Stack ---\n%', stack;
  RETURN 1;
END;
$$ LANGUAGE plpgsql;

SELECT outer_func();

NOTICE:  --- Call Stack ---
PL/pgSQL function inner_func() line 5 at GET DIAGNOSTICS
PL/pgSQL function outer_func() line 3 at RETURN
CONTEXT:  PL/pgSQL function outer_func() line 3 at RETURN
 outer_func
 ------------
           1
(1 row)
```

GET STACKED DIAGNOSTICS ... PG_EXCEPTION_CONTEXTは同種のスタックトレースを返しますが、現在の位置ではなく、エラーが検出されたところの位置を記述します。

42.7. カーソル

問い合わせ全体を一度に実行するのではなく、カーソルを設定して、問い合わせをカプセル化し、問い合わせの結果を一度に数行ずつ読み取ることができます。これを行う理由の1つは、結果内に多数の行がある場合のメモリの枯渇を防ぐことです。(しかし、PL/pgSQLユーザは通常これを心配する必要はありません。FORループは自動的にカーソルを内部的に使用してメモリの問題を防ぐからです。) より興味深い使用方法として、呼び出し元が行を読み取ることをできるように、作成されたカーソルへの参照を返す方法があります。これにより、関数から大量の行集合を返す際の効率が向上します。

42.7.1. カーソル変数の宣言

PL/pgSQLにおけるカーソルへのアクセスは全て、カーソル変数を経由します。カーソル変数は、常に特殊なrefcursorデータ型です。カーソル変数を作成する1つの方法は、単にrefcursor型の変数として宣言することです。他の方法は、カーソル宣言構文を使用することです。以下にその一般形を示します。

```
name [ [ NO ] SCROLL ] CURSOR [ ( arguments ) ] FOR query;
```

(Oracleとの互換性のため、FORはISに置き換えることができます。) もしSCROLLを指定すれば、カーソルは逆方向に移動できます。もしNO SCROLLを指定すれば、逆方向の行の取り出しはできません。どちらも指定しない時、逆方向に取り出しできるかは問い合わせに依存します。もしargumentsがあれば、name datatypeをカンマで区切ったリストで、与えられた問い合わせ内のパラメータ値として置換される名前を定義します。その名前に実際に置換される値は、カーソルを開いた時点より後に指定されます。

以下に例を示します。

```
DECLARE
    curs1 refcursor;
    curs2 CURSOR FOR SELECT * FROM tenk1;
    curs3 CURSOR (key integer) FOR SELECT * FROM tenk1 WHERE unique1 = key;
```

これら3つの変数は全てrefcursorデータ型を持ちますが、最初のものは全ての問い合わせに使用でき、2番目には完全な問い合わせが既にバウンドされています(結び付けられています)。また、最後のものには、パラメータ付きの問い合わせがバウンドされています (keyはカーソルが開いた時に整数パラメータ値に置き換えられます)。curs1変数は、特定の問い合わせに結び付けられていませんので、アンバウンドであると呼ばれます。

42.7.2. カーソルを開く

カーソルを使用して行を取り出す前に、開かれる必要があります。(これはDECLARE CURSOR SQLコマンドの動作と同じです。) PL/pgSQLには3種類のOPEN文があり、そのうちの2つはアンバウンドカーソル変数を使用し、残りの1つはバウンドカーソル変数を使用します。

> **注記**
>
> バウンドカーソル変数は42.7.4で説明されているFOR文で、明示的にカーソルを開かなくても使用することができます。

42.7.2.1. OPEN FOR query

```
OPEN unbound_cursorvar [ [ NO ] SCROLL ] FOR query;
```

カーソル変数は開かれ、実行するよう指定した問い合わせが付与されます。既に開いたカーソルを開くことはできず、また、アンバウンドカーソル変数として(つまり、単なるrefcursor変数として)宣言されていなければなりません。この問い合わせはSELECT文であるか、または(EXPLAINのように)何らかの行を返すものでなければなりません。この問い合わせは、他のPL/pgSQLのSQL文と同様の方法で扱われます。PL/pgSQLの変数名は置き換えられ、問い合わせ計画は再利用できるようにキャッシュされます。PL/pgSQL変数がカーソルを使用する問い合わせに代入された時、変数はOPEN時の値となり、その後の変更はカーソルの動きに影響しません。SCROLLおよびNO SCROLLオプションの意味はバウンドカーソルと同様です。

以下に例を示します。

```
OPEN curs1 FOR SELECT * FROM foo WHERE key = mykey;
```

42.7.2.2. OPEN FOR EXECUTE

```
OPEN unbound_cursorvar [ [ NO ] SCROLL ] FOR EXECUTE query_string
                                    [ USING expression [, ... ] ];
```

カーソル変数は開かれ、実行するよう指定した問い合わせが付与されます。既に開いたカーソルを開くことはできず、また、アンバウンドカーソル変数として（つまり、単なるrefcursor変数として）宣言されていなければなりません。問い合わせは、EXECUTEコマンドと同じ方法による文字列式として指定されます。通常と同様に、これにより、次回に実行する際に違った問い合わせを計画できる柔軟性が得られます(42.11.2参照)。また、変数置換がコマンド文字列上で行われないことも意味します。EXECUTEと同様にformat()とUSINGを介して動的コマンドにパラメータ値を挿入することができます。SCROLLおよびNO SCROLLオプションの意味はバウンドカーソルと同様です。

以下に例を示します。

```
OPEN curs1 FOR EXECUTE format('SELECT * FROM %I WHERE col1 = $1',tabname) USING keyvalue;
```

この例では、テーブル名は問い合わせにformat()で挿入されています。col1との比較値はUSING経由で埋め込まれますので、引用符を付ける必要がありません。

42.7.2.3. バウンドカーソルを開く

```
OPEN bound_cursorvar [ ( [ argument_name := ] argument_value [, ...] ) ];
```

宣言時に問い合わせが結び付いたカーソル変数を開くために使用されるOPENの形式です。既に開いたカーソルを開くことはできません。実引数の式のリストはカーソルが引数を取るものと宣言された場合にのみ現れます。これらの値は問い合わせの中で置き換えられます。

バウンドカーソルの問い合わせ計画は常にキャッシュ可能とみなされます。この場合、EXECUTEと等価なものはありません。SCROLLおよびNO SCROLLをOPENにおいて指定できないことに注意してください。カーソル移動の仕様はすでに決まっているからです。

位置的表記または記名的表記を使用して、引数の値を渡すことができます。位置的表記では、全ての引数が順番に指定されます。記名的表記では、引数の式と区別するために:=を使用して、各々の引数の名前が指定されます。4.3に記述した関数呼び出しと同様に、位置的表記と記名的表記を混用できます。

例を示します（ここでは上例のカーソル宣言を使用します）。

```
OPEN curs2;
OPEN curs3(42);
OPEN curs3(key := 42);
```

変数の代入はバウンドカーソルの問い合わせで行われるため、カーソルへ値を渡す方法が2つあります。OPENコマンドの明確な引数とするものと、問い合わせにおけるPL/pgSQL変数として暗黙的に参照するものです。しかし、バウンドカーソルの宣言より前に宣言した変数だけが代入されます。どちらの場合も、OPENの実行時に変数値が決まります。例えば、上例のcurs3と同じ結果を取得する方法を、以下に示します。

```
DECLARE
    key integer;
    curs4 CURSOR FOR SELECT * FROM tenk1 WHERE unique1 = key;
BEGIN
    key := 42;
    OPEN curs4;
```

42.7.3. カーソルの使用

カーソルを開いてから、ここで説明する文を使用してカーソルを扱うことができます。

これらの操作は、カーソルを開始するために開いた関数内で行う必要はありません。関数からrefcursor値を返し、呼び出し元でそのカーソルの操作をさせることもできます。(内部的にはrefcursor値は、カーソルへの有効な問い合わせを持つポータルの名前を示す単なる文字列です。この名前は、ポータルを壊すことなく、他のrefcursor型の変数に代入することで、他に渡すことができます。)

全てのポータルは、暗黙的にトランザクションの終わりで閉ざされます。したがって、refcursor値はそのトランザクションの終わりまでの間のみ開いたカーソルへの参照として有効です。

42.7.3.1. FETCH

```
FETCH [ direction { FROM | IN } ] cursor INTO target;
```

FETCHはSELECT INTOと同様に、カーソルから次の行を抽出し、対象に格納します。対象とは、行変数、レコード変数、または単純な変数をカンマで区切ったリストです。SELECT INTOの場合と同様、特殊なFOUND変数を検査することで、行が取得できたかどうかを確認することができます。

direction句は複数行を取り出すことができるコマンドを除き、SQL FETCHで許可されたどのようなコマンドも可能です。すなわち、以下のものです。NEXT, PRIOR, FIRST, LAST, ABSOLUTE count, RELATIVE count, FORWARDまたは BACKWARD. direction句の省略は、NEXTの指定と同じです。countを使う形式では、countはいかなる整数値の式も可能です。(SQL FETCHコマンドとは異なります。あちらは整数定数のみを受け付けます。) SCROLLオプションを用いてカーソルを宣言または開かないと、directionの値による逆方向への移動の要求は失敗します。

cursor名は、開いているカーソルのポータルを参照するrefcursor変数名でなければなりません。

例:

```
FETCH curs1 INTO rowvar;
FETCH curs2 INTO foo, bar, baz;
```

```
FETCH LAST FROM curs3 INTO x, y;
FETCH RELATIVE -2 FROM curs4 INTO x;
```

42.7.3.2. MOVE

```
MOVE [ direction { FROM | IN } ] cursor;
```

MOVEコマンドは、データを取り出さないでカーソルの位置を変更します。移動先の行を返さないでカーソルの位置だけを変更することを除けば、FETCHコマンドと同一の働きをします。SELECT INTOと同様に、特殊な変数FOUNDを用いて、移動先に行が存在するかどうかを検査できます。

例:

```
MOVE curs1;
MOVE LAST FROM curs3;
MOVE RELATIVE -2 FROM curs4;
MOVE FORWARD 2 FROM curs4;
```

42.7.3.3. UPDATE/DELETE WHERE CURRENT OF

```
UPDATE table SET ... WHERE CURRENT OF cursor;
DELETE FROM table WHERE CURRENT OF cursor;
```

カーソルの位置をテーブルの行に変更すれば、カーソルによって特定した行を更新または消去できます。カーソル問い合わせは何が許されているのか（特にグループ化しないとき）についての制限があり、それはカーソル内でFOR UPDATEを使用することが最善です。より詳細についてはDECLAREマニュアルページを参照下さい。

以下に例を示します。

```
UPDATE foo SET dataval = myval WHERE CURRENT OF curs1;
```

42.7.3.4. CLOSE

```
CLOSE cursor;
```

CLOSEはポータルの背後にあるカーソルを閉じます。これを使用してトランザクションの終わりよりも前にリソースを解放することができ、また、カーソル変数を解放し、再度開くことができます。

例：

```
CLOSE curs1;
```

42.7.3.5. カーソルを返す

PL/pgSQL関数では、呼び出し元にカーソルを返すことができます。この方法は、関数から複数行または複数列を返す場合、特にその結果集合が非常に大きい場合に有用です。これを行うには、関数はカーソルを開き、呼び出し元にカーソル名を返します（もしくは、もし呼び出し元でポータル名がわかっていれば、単純に指定されたポータル名を使用してカーソルを開きます）。これにより、呼び出し元はカーソルから行を取り出すことができるようになります。カーソルは呼び出し元で閉じることができます。または、トランザクションが終了した際に自動的に閉じられます。

カーソル用のポータル名は、プログラマが指定するか、または自動的に生成されます。ポータル名を指定するには、開く前に、単にrefcursor変数に文字列を代入します。refcursor変数の文字列値はOPENによって、背後のポータル名として使用されます。しかし、refcursor変数がNULLの場合、OPENは自動的に既存のポータルと競合しない名前を生成し、それをrefcursor変数に代入します。

> **注記**
>
> バウンドカーソル変数は、その名前を表現する文字列値で初期化されます。そのため、プログラマがカーソルを開く前に代入により上書きしない限り、ポータル名はカーソル変数と同じになります。しかし、アンバウンドカーソル変数の初期値はデフォルトでNULLです。そのため、上書きされていない場合に自動的に生成される一意な名前を受け取ります。

以下の例は、呼び出し元でカーソル名を指定する方法を示しています。

```
CREATE TABLE test (col text);
INSERT INTO test VALUES ('123');

CREATE FUNCTION reffunc(refcursor) RETURNS refcursor AS '
BEGIN
    OPEN $1 FOR SELECT col FROM test;
    RETURN $1;
END;
' LANGUAGE plpgsql;

BEGIN;
SELECT reffunc('funccursor');
FETCH ALL IN funccursor;
COMMIT;
```

以下の例では、自動的に生成されたカーソル名を使用しています。

```
CREATE FUNCTION reffunc2() RETURNS refcursor AS '
DECLARE
    ref refcursor;
BEGIN
    OPEN ref FOR SELECT col FROM test;
```

```
    RETURN ref;
END;
' LANGUAGE plpgsql;


-- カーソルを使用するには、トランザクション内部である必要があります。
BEGIN;
SELECT reffunc2();

      reffunc2
--------------------
 <unnamed cursor 1>
(1 row)

FETCH ALL IN "<unnamed cursor 1>";
COMMIT;
```

以下の例は単一関数から複数のカーソルを返す方法を示しています。

```
CREATE FUNCTION myfunc(refcursor, refcursor) RETURNS SETOF refcursor AS $$
BEGIN
    OPEN $1 FOR SELECT * FROM table_1;
    RETURN NEXT $1;
    OPEN $2 FOR SELECT * FROM table_2;
    RETURN NEXT $2;
END;
$$ LANGUAGE plpgsql;


-- カーソルを使用するには、トランザクション内部である必要があります。
BEGIN;

SELECT * FROM myfunc('a', 'b');

FETCH ALL FROM a;
FETCH ALL FROM b;
COMMIT;
```

42.7.4. カーソル結果に対するループ

カーソルで返される行に対して反復することができるFOR文の亜種があります。構文は以下のようになります。

```
[ <<label>> ]
FOR recordvar IN bound_cursorvar [ ( [ argument_name := ] argument_value [, ...] ) ] LOOP
    statements
```

```
END LOOP [ label ];
```

カーソル変数は宣言されたとき、何らかの問い合わせとバウンドされていなければならず、また既に開かれていてはなりません。FOR文は自動的にカーソルを開き、ループから抜けたときに再度閉じます。実際の引数値式のリストは、カーソルが引数を取ることを宣言された場合に限ってのみ出現できます。これらの値は、OPEN過程と同じ方法で、問い合わせの中で置換されます（42.7.2.3を参照してください）。

recordvar変数は、record型として自動的に定義され、ループ内でのみ存在します（存在するいかなる変数名の定義もループ内では無視されます）。カーソルによって返されたそれぞれの行はこのレコード変数に引き続いて割り当てられ、ループ本体が実行されます。

42.8. トランザクション制御

CALLコマンドで呼び出されたプロシージャ、また同様に無名コードブロック（DOコマンド）では、COMMITおよびROLLBACKコマンドを使ってトランザクションを終えることができます。トランザクションをこれらコマンドで終了した後、新たなトランザクションが自動的に開始されます。そのため、別途のSTART TRANSACTIONはありません。（PL/pgSQLではBEGINとENDは別の意味を持つことに注意してください。）

以下に例を示します。

```
CREATE PROCEDURE transaction_test1()
LANGUAGE plpgsql
AS $$
BEGIN
    FOR i IN 0..9 LOOP
        INSERT INTO test1 (a) VALUES (i);
        IF i % 2 = 0 THEN
            COMMIT;
        ELSE
            ROLLBACK;
        END IF;
    END LOOP;
END;
$$;

CALL transaction_test1();
```

新しいトランザクションは、トランザクション分離レベル等のデフォルトのトランザクションの特性で開始します。トランザクションがループ内でコミットされた場合、新しいトランザクションは前のトランザクションと同じ特性で自動的に開始するのが好ましいかもしれません。コマンドCOMMIT AND CHAINとROLLBACK AND CHAINはそのように動作します。

トランザクション制御は、トップレベル、または、他の干渉するコマンドを伴わない入れ子のCALLまたはDO呼び出しからの、CALLまたはDOによる呼び出しのみで可能です。例えば、呼び出しスタックがCALL proc1() → CALL proc2() → CALL proc3()である場合、二番目と三番目のプロシージャはトランザクション制御を実行

できます。しかし、呼び出しスタックがCALL proc1() → SELECT func2() → CALL proc3()である場合、間のSELECTのため、最後のプロシージャはトランザクション制御を実行できません。

カーソルループには特別な考慮事項が当てはまります。以下の例をよく確認してください。

```
CREATE PROCEDURE transaction_test2()
LANGUAGE plpgsql
AS $$
DECLARE
    r RECORD;
BEGIN
    FOR r IN SELECT * FROM test2 ORDER BY x LOOP
        INSERT INTO test1 (a) VALUES (r.x);
        COMMIT;
    END LOOP;
END;
$$;

CALL transaction_test2();
```

通常、カーソルはトランザクションのコミット時に自動的に閉じられます。しかしながら、このようにループの一部として作られたカーソルは、最初のCOMMITまたはROLLBACKから自動的にホールドカーソルに変換されます。このことは、今や、最初のCOMMITやROLLBACKの時点でカーソルが行ごとではなく完全に評価されることを意味します。従来通りカーソルはループ後に自動で削除されるので、このことはユーザにほとんど認識されません。

トランザクションコマンドは、読み込み専用でないコマンド（例えばUPDATE ... RETURNING）で駆動されるカーソルループ内では許可されません。

例外ハンドラを伴うブロック内ではトランザクションを終了できません。

42.9. エラーとメッセージ

42.9.1. エラーとメッセージの報告

RAISE文を使用してメッセージを報告し、エラーを発生することができます。

```
RAISE [ level ] 'format' [, expression [, ... ]] [ USING option = expression [, ... ] ];
RAISE [ level ] condition_name [ USING option = expression [, ... ] ];
RAISE [ level ] SQLSTATE 'sqlstate' [ USING option = expression [, ... ] ];
RAISE [ level ] USING option = expression [, ... ];
RAISE ;
```

levelオプションはエラーの深刻度を指定します。使用可能なレベルはDEBUG、LOG、INFO、NOTICE、WARNINGおよびEXCEPTIONで、EXCEPTIONがデフォルトです。EXCEPTIONはエラーを発生させ、現在のトラ

ンザクションをアボートします。他のレベルは異なる優先度レベルのメッセージを生成するだけです。特定の優先度のエラーメッセージがクライアントに報告するか、サーバログに書き込むか、またはその両方はlog_min_messagesおよびclient_min_messages設定変数によって制御されます。詳細については、第19章を参照してください。

もしあればlevelの後にformatを記述することができます（これは評価式ではなく、単純文字列リテラルでなければなりません）。書式文字列は報告されるエラーメッセージテキストを指定します。書式文字列内では、%は次の省略可能な引数の値の文字列表現で書き換えられます。%%と記述することで%リテラルを表すことができます。引数の数は書式文字列のプレースホルダ%の数と一致しなければいけません。さもなくば、関数のコンパイル時にエラーが起きます。

以下の例では、v_job_idの値は文字列内の%を置き換えます。

```
RAISE NOTICE 'Calling cs_create_job(%)', v_job_id;
```

USINGに続いて、option = expression項目を記載することで、エラー報告に追加の情報を加えることができます。各expressionは、どのような文字列による式も可能です。使用可能なoptionキーワードは以下です。

MESSAGE

エラーメッセージテキストを設定します。このオプションはUSINGの前に書式文字列を含むRAISE形式では使用できません。

DETAIL

エラー詳細メッセージを出力します。

HINT

ヒントメッセージを出力します。

ERRCODE

付録Aで示されている状況名、または直接的に5文字によるSQLSTATEコードのいずれかで、報告すべきエラーコード（SQLSTATE）を指定します。

COLUMN
CONSTRAINT
DATATYPE
TABLE
SCHEMA

関連するオブジェクト名を出力します。

以下の例は、与えられたエラーメッセージとヒントを付けてトランザクションをアボートします。

```
RAISE EXCEPTION 'Nonexistent ID --> %', user_id
      USING HINT = 'Please check your user ID';
```

以下の2つの例は、SQLSTATEを設定する等価な方法を示しています。

```
RAISE 'Duplicate user ID: %', user_id USING ERRCODE = 'unique_violation';
RAISE 'Duplicate user ID: %', user_id USING ERRCODE = '23505';
```

主引数が報告されるべき状況名、またはSQLSTATEである場合、2番目のRAISE構文があります。例を示します。

```
RAISE division_by_zero;
RAISE SQLSTATE '22012';
```

この構文において、USINGは独自のエラーメッセージ、詳細、またはヒントを供給するように使用できます。先の例と同じことを行う別の方法は次のようになります。

```
RAISE unique_violation USING MESSAGE = 'Duplicate user ID: ' || user_id;
```

他にも亜種があり、RAISE USING または RAISE level USINGと記述して、全て一括してUSINGリスト内に書き加えます。

最後のRAISE亜種はパラメータを全く取りません。この形式はBEGINブロックのEXCEPTION句で使用されるのみです。これは、現在処理中のエラーを再発生させます。

> **注記**
>
> PostgreSQL9.1より前のバージョンでは、パラメータのないRAISEは稼動している例外ハンドラを含むブロックからのエラーの再発生と解釈されました。したがって、例外ハンドラの中で入れ子となったEXCEPTION句は、RAISEが入れ子となったEXCEPTION句のブロック内にあるときでも、エラーを捕捉できないことになりました。これは驚くべきことであり、オラクルの PL/SQLと非互換でした。

RAISE EXCEPTIONコマンド内で状況名もSQLSTATEも指定されない場合、デフォルトはERRCODE_RAISE_EXCEPTION (P0001)を使用します。メッセージテキストが指定されない場合、デフォルトは状況名、またはSQLSTATEをメッセージテキストとして使用します。

> **注記**
>
> SQLSTATEコードでエラーコードを指定する場合、事前に定義されたエラーコードに制約されることはありません。00000以外の5桁の数字かASCIIの大文字からなるどんなエラーコードも選択できます。3つのゼロで終わるエラーコードの出力を避けるように推奨されています。と言うのは、そこには分類コードがあり、それらは全ての分類から捕捉することによってのみ補足可能だからです。

42.9.2. アサート検査

ASSERT文は、PL/pgSQL関数にデバッグ用検査を差し込むための便利な省略形です。

```
ASSERT condition [ , message ];
```

conditionは常に真と評価されると想定される論理値式で、結果が真ならASSERT文がさらに何かすることはありません。結果が偽かNULLなら、ASSERT_FAILURE例外が発生します。(もし、conditionを評価する間にエラーが生じた場合、それは通常のエラーと同様に報告されます。)

省略可能なmessageが与えられた場合、その式の結果で(NULLでないなら)、conditionに失敗した際のデフォルトエラーメッセージ文「assertion failed」が置き換えられます。message式はアサートに成功する通常の場合には評価されません。

アサート検査は、設定パラメータplpgsql.check_assertsで有効または無効にできます。設定値は真偽値で、デフォルトはonです。offのときには、ASSERT文は何もしません。

ASSERTはプログラムのバグを見つけるためのものであって、通常のエラー条件を報告するものではないことに注意してください。そのためには前述のRAISEを使ってください。

42.10. トリガ関数

PL/pgSQLはデータ変更やデータベースのイベントによるトリガ関数の定義に使用できます。トリガ関数は、CREATE FUNCTIONコマンドを使って、(データ変更トリガには)trigger、(データベースイベントトリガには)event_triggerという戻り値の型を持った引数のない関数として作成されます。その呼出しのトリガの原因となった条件を記述するため、TG_somethingという名前の特別な局所変数が自動的に定義されます。

42.10.1. データ変更によるトリガ

データ変更トリガはtriggerという戻り値の型を持った引数のない関数として宣言されます。その関数は、たとえ、CREATE TRIGGERにて引数を取るものとしていたとしても、引数を持たないものと宣言しなければならないことに注意してください。トリガの引数は、後述する通り、TG_ARGV経由で渡されます。

PL/pgSQL関数がトリガとして呼び出された場合、いくつかの特殊な変数が自動的に最上位レベルのブロックで作成されます。それらを以下に示します。

NEW

RECORDデータ型。この変数は行レベルのトリガでのINSERT/UPDATE操作によって更新された、新しいデータベースの行を保持します。文レベルのトリガおよびDELETE操作では、この変数はnullです。

OLD

RECORDデータ型。この変数は、行レベルのトリガでのUPDATE/DELETE操作によって更新される前のデータベースの行を保持します。文レベルのトリガおよびINSERT操作では、この変数はnullです。

TG_NAME

nameデータ型。実際に発行されたトリガの名前を持つ変数。

TG_WHEN

textデータ型。トリガの定義に依存したBEFORE、AFTER、またはINSTEAD OFという文字列。

TG_LEVEL

textデータ型。トリガの定義に依存したROWまたは STATEMENTという文字列。

TG_OP

textデータ型。トリガを起動した操作を示す、INSERT、UPDATE、DELETE、またはTRUNCATEという文字列。

TG_RELID

oidデータ型。このトリガの呼び出し元になるテーブルのオブジェクトID。

TG_RELNAME

nameデータ型。このトリガの呼び出し元になるテーブルの名前。将来これは廃止されそうです。代わりにTG_TABLE_NAMEを使用してください。

TG_TABLE_NAME

nameデータ型。このトリガの呼び出し元になるテーブルの名前。

TG_TABLE_SCHEMA

nameデータ型。このトリガの呼び出し元になるテーブルのスキーマ名。

TG_NARGS

integer型。CREATE TRIGGER文におけるトリガ関数に与えられる引数の数。

TG_ARGV[]

text型の配列型。CREATE TRIGGER文での引数。このインデックスは0から始まります。無効なインデックス（0未満やtg_nargs以上）は、NULL値という結果になります。

トリガ関数はNULLまたは、トリガの発行元になったテーブルの構造を正確に持ったレコード/行を返さなければなりません。

BEFOREとして発行された行レベルトリガがNULLを返す場合には、トリガマネージャに実際の行への操作を取りやめるように通知します（つまり、その後にトリガが発行されず、そのINSERT/UPDATE/DELETEはその行に対して実行されません）。非NULL値を返す場合には、その操作はその行値で処理されます。元のNEWの値と異なる行値を返すことは、挿入、更新される値を変更します。したがってトリガ関数が行値を変更せずにトリガ処理を普通に成功させたい場合は、NEW(またはその等価な値)を返さなければなりません。格納する行を変更するために、NEWの個々の値を直接置き換え、変更したNEWを返すことも、新しいレコード/行を完全に作成して返すことも可能です。DELETEに対するBEFOREトリガの場合、返される値は直接的な影響を与えませんが、トリガ動作を継続させるためには非NULLを返さなければなりません。DELETEトリガではNEWがNULLであり、これを返すことは通常無意味であることに注意して下さい。DELETEトリガにおける通常の慣例はOLDを返すことです。

INSTEAD OFトリガ（これは常に行レベルトリガであり、ビューに対してのみ使用可能です）は、まったく更新を行わなかったためにこの行に対する残りの操作を飛ばさなければならない（つまり後続のトリガは発行されず、トリガの発生元のINSERT/UPDATE/DELETEにおいて影響を受けた行数として数えられない）ことを通知するNULLを返すことができます。この他の場合は、トリガが要求された操作を実行したことを通知するために、非NULLの値を返さなければなりません。INSERTおよびUPDATE操作では、戻り値は、トリガ関数がINSERT RETURNINGおよびUPDATE RETURNINGをサポートするために変更しているかもしれない、NEWとなるはずです

(これは後続のトリガ、または、ON CONFLICT DO UPDATE句を伴うINSERT文の中で特別なEXCLUDED別名参照に渡される行値にも影響します)。DELETE操作では、戻り値はOLDとなるはずです。

行レベルのAFTERトリガ、文レベルのBEFOREまたはAFTERトリガの戻り値は常に無視されます。NULLとしても構いません。しかし、これらの種類のトリガでも、エラーを発生させることで操作全体を中断させることが可能です。

例 42.3にPL/pgSQLのトリガ関数の例を示します。

例42.3 PL/pgSQLトリガ関数

このトリガの例では、テーブルの行が挿入または更新された時には必ず、現在のユーザ名と時刻がその行に入っていることを確実にします。そして、従業員名が与えられていることとその給料が正の値であることを確認します。

```
CREATE TABLE emp (
    empname text,
    salary integer,
    last_date timestamp,
    last_user text
);

CREATE FUNCTION emp_stamp() RETURNS trigger AS $emp_stamp$
    BEGIN

        -- empnameとsalaryが与えられていることをチェック
        IF NEW.empname IS NULL THEN
            RAISE EXCEPTION 'empname cannot be null';
        END IF;
        IF NEW.salary IS NULL THEN
            RAISE EXCEPTION '% cannot have null salary', NEW.empname;
        END IF;


        -- 支払時に問題が起こらないように
        IF NEW.salary < 0 THEN
            RAISE EXCEPTION '% cannot have a negative salary', NEW.empname;
        END IF;


        -- 誰がいつ変更したかを記録
        NEW.last_date := current_timestamp;
        NEW.last_user := current_user;
        RETURN NEW;
    END;
$emp_stamp$ LANGUAGE plpgsql;

CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON emp
```

```
    FOR EACH ROW EXECUTE FUNCTION emp_stamp();
```

テーブルにおける変更のログを取る他の方法は、挿入、更新または削除の各々に対する行を保有する新テーブルを作成することです。この方法はテーブルにおける変更の監査と考えることができます。例 42.4はPL/pgSQLによる監査用トリガ関数の一例を示します。

例42.4 PL/pgSQLによる監査用のトリガ関数

このトリガの例では、empテーブルにおける行の挿入、更新または削除のどれもがemp_auditテーブルの中へ確実に記録（すなわち監査）されます。現在時刻とユーザ名は、行った操作の種類とともにemp_auditの行の中に記録されます。

```
CREATE TABLE emp (
    empname           text NOT NULL,
    salary            integer
);

CREATE TABLE emp_audit(
    operation         char(1)   NOT NULL,
    stamp             timestamp NOT NULL,
    userid            text      NOT NULL,
    empname           text      NOT NULL,
    salary integer
);

CREATE OR REPLACE FUNCTION process_emp_audit() RETURNS TRIGGER AS $emp_audit$
    BEGIN
        --

        -- empで行った操作を反映する行をemp_auditに作成
        -- 操作の種類を決定するために、
      特殊な変数TG_OPを活用
        --
        IF (TG_OP = 'DELETE') THEN
            INSERT INTO emp_audit SELECT 'D', now(), user, OLD.*;
        ELSIF (TG_OP = 'UPDATE') THEN
            INSERT INTO emp_audit SELECT 'U', now(), user, NEW.*;
        ELSIF (TG_OP = 'INSERT') THEN
            INSERT INTO emp_audit SELECT 'I', now(), user, NEW.*;
        END IF;

        RETURN NULL; -- AFTERトリガですので、結果は無視されます
    END;
$emp_audit$ LANGUAGE plpgsql;

CREATE TRIGGER emp_audit
AFTER INSERT OR UPDATE OR DELETE ON emp
```

```
    FOR EACH ROW EXECUTE FUNCTION process_emp_audit();
```

前例の変形では、各エントリが最終修正された時を表示するため、主テーブルを監査テーブルに結合したビューを使用します。この方法でもテーブルの変化の監査証跡を全て記録できますが、監査証跡から抽出した各エントリの最終修正のタイムスタンプ表示することにより、監査証跡の簡単なビューを表示することにもなります。例 42.5で示すものは、PL/pgSQLを用いたビューの監査トリガの例です。

例42.5 監査のためのPL/pgSQLビュートリガ関数

この例では、ビューを更新可能とし、その行の挿入、更新、削除をemp_auditテーブルに確実に記録（つまり監査）するためにビューに対するトリガを使用します。現在時刻とユーザ名が実行された操作種類と一緒に記録されます。ビューは各行の最終更新時間を表示します。

```
CREATE TABLE emp (
    empname           text PRIMARY KEY,
    salary            integer
);

CREATE TABLE emp_audit(
    operation         char(1)   NOT NULL,
    userid            text      NOT NULL,
    empname           text      NOT NULL,
    salary            integer,
    stamp             timestamp NOT NULL
);

CREATE VIEW emp_view AS
    SELECT e.empname,
           e.salary,
           max(ea.stamp) AS last_updated
      FROM emp e
      LEFT JOIN emp_audit ea ON ea.empname = e.empname
     GROUP BY 1, 2;

CREATE OR REPLACE FUNCTION update_emp_view() RETURNS TRIGGER AS $$
    BEGIN
        --

        -- 要求された操作を emp に実行し
        -- emp_audit に行を作成し
        -- emp の変化を反映する
        --
        IF (TG_OP = 'DELETE') THEN
            DELETE FROM emp WHERE empname = OLD.empname;
            IF NOT FOUND THEN RETURN NULL; END IF;

            OLD.last_updated = now();
```

```
            INSERT INTO emp_audit VALUES('D', user, OLD.*);
            RETURN OLD;
        ELSIF (TG_OP = 'UPDATE') THEN
            UPDATE emp SET salary = NEW.salary WHERE empname = OLD.empname;
            IF NOT FOUND THEN RETURN NULL; END IF;

            NEW.last_updated = now();
            INSERT INTO emp_audit VALUES('U', user, NEW.*);
            RETURN NEW;
        ELSIF (TG_OP = 'INSERT') THEN
            INSERT INTO emp VALUES(NEW.empname, NEW.salary);

            NEW.last_updated = now();
            INSERT INTO emp_audit VALUES('I', user, NEW.*);
            RETURN NEW;
        END IF;
    END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER emp_audit
INSTEAD OF INSERT OR UPDATE OR DELETE ON emp_view
    FOR EACH ROW EXECUTE FUNCTION update_emp_view();
```

トリガの使用目的の1つは、あるテーブルのサマリテーブルを維持することです。結果のサマリテーブルは、元のテーブルに代わって、ある種の問い合わせに対して使用でき、しばしば実行時間を大幅に縮小します。通常この手法は、計測または観測データ（ファクトテーブルと言います）が非常に大きくなるかもしれない、データウェアハウスに使用されます。データウェアハウス内のファクトテーブルに対してサマリテーブルを維持するPL/pgSQLのトリガ関数の例を例 42.6に示します。

例42.6 サマリテーブルを維持するためのPL/pgSQLトリガ関数

ここに述べるスキーマの一部はRalph Kimballによる*The Data Warehouse Toolkit*の*Grocery Store*の例に基づいています。

```
--

-- time dimensionとsales factの主テーブル
--
CREATE TABLE time_dimension (
    time_key                    integer NOT NULL,
    day_of_week                 integer NOT NULL,
    day_of_month                integer NOT NULL,
    month                       integer NOT NULL,
    quarter                     integer NOT NULL,
    year                        integer NOT NULL
);
CREATE UNIQUE INDEX time_dimension_key ON time_dimension(time_key);
```

```
CREATE TABLE sales_fact (
    time_key                    integer NOT NULL,
    product_key                 integer NOT NULL,
    store_key                   integer NOT NULL,
    amount_sold                 numeric(12,2) NOT NULL,
    units_sold                  integer NOT NULL,
    amount_cost                 numeric(12,2) NOT NULL
);
CREATE INDEX sales_fact_time ON sales_fact(time_key);

--

-- sales by timeのサマリテーブル
--
CREATE TABLE sales_summary_bytime (
    time_key                    integer NOT NULL,
    amount_sold                 numeric(15,2) NOT NULL,
    units_sold                  numeric(12) NOT NULL,
    amount_cost                 numeric(15,2) NOT NULL
);
CREATE UNIQUE INDEX sales_summary_bytime_key ON sales_summary_bytime(time_key);

--

-- 更新、
      挿入および削除によりサマリテーブルの列を修正する関数とトリガ
--
CREATE OR REPLACE FUNCTION maint_sales_summary_bytime() RETURNS TRIGGER
AS $maint_sales_summary_bytime$
    DECLARE
        delta_time_key          integer;
        delta_amount_sold       numeric(15,2);
        delta_units_sold        numeric(12);
        delta_amount_cost       numeric(15,2);
    BEGIN


        -- 増加または減少量を算出
        IF (TG_OP = 'DELETE') THEN

            delta_time_key = OLD.time_key;
            delta_amount_sold = -1 * OLD.amount_sold;
            delta_units_sold = -1 * OLD.units_sold;
            delta_amount_cost = -1 * OLD.amount_cost;
```

```
        ELSIF (TG_OP = 'UPDATE') THEN


            -- time_keyを変更する更新を禁止します
            -- （削除 + 挿入の方法により大部分の変更を行うため
            -- それほど厄介ではありません）。
            IF ( OLD.time_key != NEW.time_key) THEN
                RAISE EXCEPTION 'Update of time_key : % -> % not allowed',
                                                    OLD.time_key, NEW.time_key;
            END IF;

            delta_time_key = OLD.time_key;
            delta_amount_sold = NEW.amount_sold - OLD.amount_sold;
            delta_units_sold = NEW.units_sold - OLD.units_sold;
            delta_amount_cost = NEW.amount_cost - OLD.amount_cost;

        ELSIF (TG_OP = 'INSERT') THEN

            delta_time_key = NEW.time_key;
            delta_amount_sold = NEW.amount_sold;
            delta_units_sold = NEW.units_sold;
            delta_amount_cost = NEW.amount_cost;

        END IF;



        -- サマリテーブルの行を挿入または新しい値で更新します。
        <<insert_update>>
        LOOP
            UPDATE sales_summary_bytime
                SET amount_sold = amount_sold + delta_amount_sold,
                    units_sold = units_sold + delta_units_sold,
                    amount_cost = amount_cost + delta_amount_cost
                WHERE time_key = delta_time_key;

            EXIT insert_update WHEN found;

            BEGIN
                INSERT INTO sales_summary_bytime (
                            time_key,
                            amount_sold,
                            units_sold,
                            amount_cost)
                    VALUES (
                            delta_time_key,
```

```
                            delta_amount_sold,
                            delta_units_sold,
                            delta_amount_cost
                           );

                EXIT insert_update;

            EXCEPTION
                WHEN UNIQUE_VIOLATION THEN

                    -- 何もしません
            END;
        END LOOP insert_update;

        RETURN NULL;

    END;
$maint_sales_summary_bytime$ LANGUAGE plpgsql;

CREATE TRIGGER maint_sales_summary_bytime
AFTER INSERT OR UPDATE OR DELETE ON sales_fact
    FOR EACH ROW EXECUTE FUNCTION maint_sales_summary_bytime();

INSERT INTO sales_fact VALUES(1,1,1,10,3,15);
INSERT INTO sales_fact VALUES(1,2,1,20,5,35);
INSERT INTO sales_fact VALUES(2,2,1,40,15,135);
INSERT INTO sales_fact VALUES(2,3,1,10,1,13);
SELECT * FROM sales_summary_bytime;
DELETE FROM sales_fact WHERE product_key = 1;
SELECT * FROM sales_summary_bytime;
UPDATE sales_fact SET units_sold = units_sold * 2;
SELECT * FROM sales_summary_bytime;
```

AFTERトリガは、トリガ文により変更された行の集合全体を調べるために遷移テーブルを使うこともできます。CREATE TRIGGERコマンドで名前を1つまたは2つの遷移テーブルに割り当てると、関数はその名前を読み込み専用の一時テーブルであるかのように参照できます。例 42.7に例を示します。

例42.7 遷移テーブルでの監査

この例は例 42.4と同じ結果になりますが、行毎に起動するトリガを使う代わりに、関係のある情報を遷移テーブルに集めた後に文毎に1回起動するトリガを使っています。これは、呼び出された文が多くの行を変更する場合には行トリガの方法よりとても速くなる場合があります。REFERENCING句はそれぞれの場合で異ならなければなりませんので、それぞれの種類のイベントに対して別々のトリガ宣言をしなければならないことに注意してください。ですが、もし選ぶのなら、このために単一のトリガ関数が使えなくなることはありません。(実際には、3つに別れた関数を使い、実行時のTG_OPの確認を避ける方が良いでしょう。)

```
CREATE TABLE emp (
```

```
    empname           text NOT NULL,
    salary            integer
);

CREATE TABLE emp_audit(
    operation         char(1)   NOT NULL,
    stamp             timestamp NOT NULL,
    userid            text      NOT NULL,
    empname           text      NOT NULL,
    salary integer
);

CREATE OR REPLACE FUNCTION process_emp_audit() RETURNS TRIGGER AS $emp_audit$
    BEGIN
        --

        -- empで実行された操作を反映するためにemp_auditに行を作り、

        -- 操作を完了するために特殊な変数TG_OPを使う。
        --
        IF (TG_OP = 'DELETE') THEN
            INSERT INTO emp_audit
                SELECT 'D', now(), user, o.* FROM old_table o;
        ELSIF (TG_OP = 'UPDATE') THEN
            INSERT INTO emp_audit
                SELECT 'U', now(), user, n.* FROM new_table n;
        ELSIF (TG_OP = 'INSERT') THEN
            INSERT INTO emp_audit
                SELECT 'I', now(), user, n.* FROM new_table n;
        END IF;

        RETURN NULL; -- これはAFTERトリガなので結果は無視される
    END;
$emp_audit$ LANGUAGE plpgsql;

CREATE TRIGGER emp_audit_ins
    AFTER INSERT ON emp
    REFERENCING NEW TABLE AS new_table
    FOR EACH STATEMENT EXECUTE FUNCTION process_emp_audit();
CREATE TRIGGER emp_audit_upd
    AFTER UPDATE ON emp
    REFERENCING OLD TABLE AS old_table NEW TABLE AS new_table
    FOR EACH STATEMENT EXECUTE FUNCTION process_emp_audit();
CREATE TRIGGER emp_audit_del
    AFTER DELETE ON emp
    REFERENCING OLD TABLE AS old_table
```

```
    FOR EACH STATEMENT EXECUTE FUNCTION process_emp_audit();
```

42.10.2. イベントによるトリガ

PL/pgSQLはイベントトリガの定義に使用できます。イベントトリガとして呼び出される関数は、引数のない関数として宣言され、戻り値の型はevent_triggerとなることがPostgreSQLでは必須です。

PL/pgSQL関数がイベントトリガとして呼び出された場合、数個の特別な変数が最高レベルのブロックで自動的に作成されます。以下に示します。

TG_EVENT

textデータ型。トリガが発行されたイベントを示す文字列。

TG_TAG

textデータ型。トリガが発行されたコマンドタグを含む変数。

例 42.8はPL/pgSQLにおけるイベントトリガ関数の一例を示します。

例42.8 PL/pgSQLイベントトリガ関数

以下の例では、サポートされたコマンドが実行されたとき、トリガはNOTICEを発生させるだけです。

```
CREATE OR REPLACE FUNCTION snitch() RETURNS event_trigger AS $$
BEGIN
    RAISE NOTICE 'snitch: % %', tg_event, tg_tag;
END;
$$ LANGUAGE plpgsql;

CREATE EVENT TRIGGER snitch ON ddl_command_start EXECUTE FUNCTION snitch();
```

42.11. PL/pgSQLの秘訣

本節では、PL/pgSQL利用者の知識として重要な、実装の詳細を述べます。

42.11.1. 変数置換

PL/pgSQL関数内のSQL文および式は変数および関数のパラメータを参照することができます。背後では、PL/pgSQLはこうした参照を問い合わせパラメータに置き換えます。パラメータまたは列参照が文法的に許されているところでのみパラメータは置換されます。極端な場合として、以下のよろしくないプログラミングスタイルの例を考えてみましょう。

```
INSERT INTO foo (foo) VALUES (foo);
```

最初に現れるfooの場所は文法的にはテーブル名でなければなりません。このため関数がfooという名前の変数を持っていたとしても、置換されません。2番目の場所はテーブルの列名でなければなりません。このためこれも置換されません。3番目の場所のみが関数の変数参照の候補です。

> **注記**
>
> 9.0より前のPostgreSQLでは、3つの場合すべてにおいて変数を置換しようとし、構文エラーを引き起こしました。

変数名は文法的にはテーブル列名と違いがありませんので、テーブルを参照する文の中であいまいさが出る可能性があります。与えられた名前はテーブル列を意味するのでしょうか、それとも変数なのでしょうか。前の例を次のように変えてみましょう。

```
INSERT INTO dest (col) SELECT foo + bar FROM src;
```

ここでは、destおよびsrcはテーブル名でなければなりません。また、colはdestの列でなければなりませんが、fooおよびbarは理論上関数の変数かもしれませんし、srcの列かもしれません。

デフォルトでPL/pgSQLはSQL文における名前が変数かテーブル列のいずれかを参照可能な場合にエラーを報告します。変数または列の名前を変更することやあいまいな参照を修飾すること、PL/pgSQLにどちらを優先して解釈するかを通知することで、こうした問題を解消することができます。

最も簡単な解法は変数名または列名を変更することです。一般的なコーディング法として、列の命名とPL/pgSQL変数の命名とで規約を分ける方法があります。例えば、一貫して関数の変数はv_somethingという名前とし、列名はv_で始まらないようにすれば、競合は起こりません。

その他、あいまいな参照を明確にするために修飾することができます。上の例では、src.fooによりテーブル列への参照についてあいまいさが解消します。あいまい性のない変数参照を行うためには、ラベル付けしたブロック内で変数を宣言し、そのブロックのラベルを使用します(42.2参照)。以下に例を示します。

```
<<block>>
DECLARE
    foo int;
BEGIN
    foo := ...;
    INSERT INTO dest (col) SELECT block.foo + bar FROM src;
```

ここでblock.fooはsrcにfoo列があったとしても、変数を意味することになります。FOUNDなどの特別な変数を含め、関数パラメータを関数名で修飾することができます。これらは暗黙的に関数名をラベル名とした上位ブロック内で宣言されているためです。

PL/pgSQLの大規模な本体コードにおける、すべてのあいまいな参照を修正することが現実的ではない場合があります。こうした場合、PL/pgSQLにあいまいな参照を変数として解決すべき(この動作はPostgreSQL 9.0より前のPL/pgSQLの動作と互換性を持ちます)、または、テーブル列参照として解決すべき(Oracleなどの他のシステムと互換性を持ちます)と指定することができます。

システム全体に対してこの動作を変更するためにはplpgsql.variable_conflict設定パラメータをerror、use_variable、use_columnのいずれかに設定します(errorが標準配布におけるデフォルトです)。このパラ

メータは以降のPL/pgSQL関数の文のコンパイルに影響しますが、現在のセッションでコンパイル済みの文には影響を与えません。この設定を変更することで、PL/pgSQLの動作において予期できない変化が発生することがありますので、これはスーパーユーザのみが変更することができます。

また、関数テキストの先頭に以下の特殊なコマンドの1つをいれることで、関数単位で動作を設定することもできます。

```
#variable_conflict error
#variable_conflict use_variable
#variable_conflict use_column
```

これらのコマンドを記述した関数に対してのみ、コマンドは影響を与え、plpgsql.variable_conflictの設定を上書きします。以下に例を示します。

```
CREATE FUNCTION stamp_user(id int, comment text) RETURNS void AS $$
    #variable_conflict use_variable
    DECLARE
        curtime timestamp := now();
    BEGIN
        UPDATE users SET last_modified = curtime, comment = comment
          WHERE users.id = id;
    END;
$$ LANGUAGE plpgsql;
```

UPDATEコマンドにおいて、curtime、commentおよびidは、usersに同名の列があるか否かに関わらず、関数の変数またはパラメータを参照します。テーブル列を参照させるためにWHERE句においてusers.idと参照を修飾する必要があったことに注意して下さい。しかしUPDATEリストの対象としてのcommentへの参照は修飾させる必要がありませんでした。これは文法的にusersの列でなければならないためです。以下のようにvariable_conflictの設定に依存せずに同じ関数を作成することもできます。

```
CREATE FUNCTION stamp_user(id int, comment text) RETURNS void AS $$
    <<fn>>
    DECLARE
        curtime timestamp := now();
    BEGIN
        UPDATE users SET last_modified = fn.curtime, comment = stamp_user.comment
          WHERE users.id = stamp_user.id;
    END;
$$ LANGUAGE plpgsql;
```

変数置換はEXECUTEコマンドまたはその亜種におけるコマンド文字列の中では起こりません。そのようなコマンドに可変値を挿入する時は、42.5.4に述べたように、文字列の値を構成するものの一部とするかUSINGを使用してください。

今のところ変数置換は、SELECTとINSERTとUPDATEとDELETEコマンドの中だけで作動します。メインSQLエンジンが問い合わせパラメータをこれらのコマンドでしか許可しないからです。他の種類の文（通常ユーティリティ文といいます）において可変名または可変値を使用するには、文字列としてユーティリティ文を構成しEXECUTEしてください。

42.11.2. 計画のキャッシュ

PL/pgSQLインタプリタは、初めてその関数が（各セッションで）呼び出された時に、関数のソーステキストを解析し、バイナリ形式の命令ツリーを内部で作成します。この命令ツリーは完全にPL/pgSQL文構造に変換されますが、関数内部の個々のSQL式とSQLコマンドは即座に変換されません。

各式やSQLコマンドが初めてその関数で実行される時に、PL/pgSQLインタプリタはSPIマネージャのSPI_prepare関数を使用して、プリペアドステートメントを作成するためにコマンドを解析します。その後にその式やコマンドが行われる時には、そのプリペアドステートメントを再利用します。こうして、めったに分岐されない条件付きコードパスを持つ関数では、現在のセッションで実行されないそれらのコマンドの解析によるオーバーヘッドを背負いこむことはありません。欠点は特定の式や問い合わせのエラーが、関数の該当部分が実行されるまで検出されないことです。（典型的な構文エラーは、最初の解釈において検出されますが、それより深いエラーは、実行の時まで検出されません）。

PL/pgSQLは（正確にはSPIマネージャは）さらに特定のプリペアドステートメントに関する実行計画のキャッシュを試行できます。キャッシュした実行計画が使用されなかった場合、プリペアドステートメントが呼び出される度に新しい実行計画が作成され、選択した実行計画を最適にするために、最新のパラメータ値（すなわちPL/pgSQLの変数値）が使用されます。プリペアドステートメントがパラメータを持たないか何回も使用される場合、SPIマネージャは特定のパラメータ値に依存しない一般的な実行計画の作成を考え、再使用のためにキャッシュします。典型的には、これは参照したPL/pgSQLの変数値が、実行計画にさほど影響しない場合にだけ起こります。それならば、毎回の実行計画の作成の方が優れています。プリペアドステートメントに関する詳細はPREPAREを参照してください。

このようにPL/pgSQLはプリペアドステートメントおよび時には実行計画を保存しますので、PL/pgSQL関数内に直接現れるSQLコマンドは実行の度に同じテーブルとフィールドを参照しなければなりません。つまり、SQLコマンドにて、テーブルやフィールドの名前としてパラメータを使用することができません。実行の度に新しく実行計画を作成して解析する無駄を覚悟で、PL/pgSQLのEXECUTE文を使った動的問い合わせを構成することで、この制限を回避できます。

レコード変数の変わりやすいという性質はこの接続において別の問題となります。レコード変数のフィールドが式や文の中で使用される場合、そのフィールドのデータ型を関数を呼び出す度に変更してはいけません。それぞれの式が最初に実行された時のデータ型を使用して、その式が解析されているからです。必要な場合EXECUTEを使用してこの問題を回避することができます。

同一の関数が2つ以上のテーブルのトリガとして使用される場合、PL/pgSQLはテーブルごとのプリペアドステートメントをキャッシュします。すなわち、各々のトリガ関数とテーブルの組ごとにキャッシュするのであり、トリガ関数ごとではありません。このため、データ型の変更に伴う問題の一部を軽減します。例えば、別のテーブルにある異なったデータ型であっても、keyと命名した列に対してトリガ関数は有効に作動します。

同様に、多様型の引数を持った関数は、実際に呼び出す引数の型の組み合わせごとに別々のプリペアドステートメントをキャッシュします。そのため、データ型の差異が原因で予期しない失敗が起こることはありません。

プリペアドステートメントのキャッシュにより、時間に依存する値の解釈の結果に違いが現れることがあります。例えば、以下の2つの関数の結果は異なります。

```
CREATE FUNCTION logfunc1(logtxt text) RETURNS void AS $$
    BEGIN
        INSERT INTO logtable VALUES (logtxt, 'now');
```

```
    END;
$$ LANGUAGE plpgsql;
```

および

```
CREATE FUNCTION logfunc2(logtxt text) RETURNS void AS $$
    DECLARE
        curtime timestamp;
    BEGIN
        curtime := 'now';
        INSERT INTO logtable VALUES (logtxt, curtime);
    END;
$$ LANGUAGE plpgsql;
```

logfunc1の場合では、PostgreSQLのメインパーサは、INSERTを解析する時に、logtableの対象列の型から'now'をtimestampと解釈しなければならないことを把握しています。こうして、パーサはINSERTが解析された時点で'now'をtimestamp定数に変換し、その定数値をその後のセッションの有効期間におけるlogfunc1の全ての呼び出しで使用します。言うまでもありませんが、これはプログラマが意図した動作ではありません。now()またはcurrent_timestamp関数の使用が優れています。

logfunc2の場合では、PostgreSQLのメインパーサは'now'の型を決定することができません。そのため、nowという文字列を持つtext型のデータ値を返します。curtimeローカル変数に代入する時に、PL/pgSQLインタプリタはこの文字列をtextoutとtimestamp_in関数を変換に使用してtimestamp型にキャストします。ですから、演算されたタイムスタンプは、プログラマが意図した通り、実行の度に更新されます。この方法でたまたま意図した通り動くけれど、それほど効率的ではありません。ですから、now()関数の使用の方が優れています。

42.12. PL/pgSQLによる開発向けのヒント

PL/pgSQLで開発する1つの良い方法は、関数を作成するのに自分の好きなテキストエディタを使い、もう1つのウィンドウでpsqlを使用して関数を読み込ませて試験を行うことです。この方法で行う場合にはCREATE OR REPLACE FUNCTIONを使用して関数を作成する方が良いでしょう。この方法でファイルを再読み込みすると、関数定義を更新することができます。例えば以下のようにします。

```
CREATE OR REPLACE FUNCTION testfunc(integer) RETURNS integer AS $$
          ....
$$ LANGUAGE plpgsql;
```

psqlを実行し、以下のように関数定義ファイルを読み込み、または再読み込みすることができます。

```
\i filename.sql
```

その後すぐに、関数を試験するためにSQLコマンドを発行することができます。

PL/pgSQLにおける開発のもう1つの良い方法は、手続き言語の開発機能を持つGUIデータベースアクセスツールを使用することです。他にもありますが、pgAdminがこうしたツールの一例です。こうしたツールは、

単一引用符をエスケープさせたり、関数の作り直しやデバッグが容易に行えたりする便利な機能をよく持っています。

42.12.1. 引用符の扱い

PL/pgSQL関数のコードはCREATE FUNCTION内で文字列リテラルとして指定されます。単一引用符で囲む通常のやり方で文字列リテラルを記述する時、関数本体内部の全ての単一引用符を二重化しなければなりません。同様に、全てのバックスラッシュを二重化しなければなりません。なお、文字列としてエスケープする構文が使用されると仮定します。引用符を単に重ねるやり方は最も冗長であり、簡単に想像できると思いますが、複雑な状態では数個以上の隣接した引用符が必要となるため、コードを率直には理解しにくくなります。それに代わって推奨されるのは、関数本体を「ドル引用符」の文字列リテラルとして記述することです（4.1.2.4を見てください）。ドル引用符を用いるやり方では他の引用符を二重化する必要はありませんが、それぞれの入れ子になった階層ごとに異なったドル引用符による区切り符号を用いなければなりません。例えば、CREATE FUNCTIONコマンドを以下のように記述しても構いません。

```
CREATE OR REPLACE FUNCTION testfunc(integer) RETURNS integer AS $PROC$
          ....
$PROC$ LANGUAGE plpgsql;
```

このやり方では、SQLコマンドの中で単純なリテラル文字列に対して引用符を使用でき、文字列として集積したSQLコマンドの断片を区切るために$$を使用できます。もし$$を含んだテキストを引用符で囲む時は、Qのような記述を使用できます。

以下の表はドル引用符を用いない時の引用符の記述法を示したものです。ドル引用符を用いる以前における引用符の記述を理解するのに、この表は役立つと思われます。

1つの引用符

関数本体の先頭と末尾。以下に例を示します。

```
CREATE FUNCTION foo() RETURNS integer AS '
          ....
' LANGUAGE plpgsql;
```

関数本体内部では引用符は必ずペアで現れます。

2つの引用符

関数本体内部の文字列リテラル用。以下に例を示します。

```
a_output := ''Blah'';
SELECT * FROM users WHERE f_name=''foobar'';
```

ドル引用符を用いる時は、次のように記述します。

```
a_output := 'Blah';
```

```
SELECT * FROM users WHERE f_name='foobar';
```

どちらもPL/pgSQLパーサから見ると同一となります。

4つの引用符

関数本体内部の文字列リテラル内の単一引用符がある場合。以下に例を示します。

```
a_output := a_output || '' AND name LIKE ''''foobar'''' AND xyz''
```

実際にa_outputに追加される値は、 AND name LIKE 'foobar' AND xyzです。

ドル引用符を用いる時は、次のように記述します。

```
a_output := a_output || $$ AND name LIKE 'foobar' AND xyz$$
```

なお、ドル引用符の区切り文字は$$だけとは限らないことに注意してください。

6つの引用符

関数本体内部の文字列内の単一引用符が、文字列定数の末尾にある場合。以下に例を示します。

```
a_output := a_output || '' AND name LIKE ''''foobar''''''
```

実際にa_outputに追加される値は、 AND name LIKE 'foobar'です。

ドル引用符を用いる時は、次のようになります。

```
a_output := a_output || $$ AND name LIKE 'foobar'$$
```

10個の引用符

文字列定数内に2つの単一引用符を持たせたい場合（これで8個の単一引用符になり）、かつ、この文字列定数の末尾にある場合（これで2個追加されます）。おそらく、他の関数を生成する関数を作成する場合（例 42.10）のみにこれが必要になるでしょう。以下に例を示します。

```
a_output := a_output || '' if v_'' ||
    referrer_keys.kind || '' like ''''''''''
    || referrer_keys.key_string || ''''''''''
    then return ''''''  || referrer_keys.referrer_type
    || ''''''; end if;'';
```

a_outputの値は以下のようになります。

```
if v_... like ''...'' then return ''...''; end if;
```

ドル引用符を用いる時は、次のようになります。

```
a_output := a_output || $$ if v_$$ || referrer_keys.kind || $$ like '$$
```

```
    || referrer_keys.key_string || $$'
    then return '$$  || referrer_keys.referrer_type
    || $$'; end if;$$;
```

ここで単一引用符は使用前に再評価されるため、a_output内部だけで必要であると仮定します。

42.12.2. コンパイル時と実行時の付加的チェック

単純でありふれた問題が有害となる前の実例を発見するユーザを助けるためPL/PgSQLは付加的checksを提供します。可能かどうかは設定に依存しますが、関数のコンパイルのときWARNINGまたはERRORを省略して使用できます。WARNINGを指定された関数は、それ以上のメッセージを生成しないで実行できます。したがって、分離された開発環境でテストを実行できます。

開発環境やテスト環境では、plpgsql.extra_warningsやplpgsql.extra_errorsを適切に"all"に設定することを勧めます。

この付加的チェックでは、設定変数plpgsql.extra_warningsを警告のためにplpgsql.extra_errorsをエラーのために使用できます。どちらも、カンマで区切ったチェックリストまたは"none"または"all"と設定できます。デフォルトは"none"です。現在指定できるチェックの一覧は以下の通りです。

shadowed_variables

宣言が以前に定義した変数を隠すかどうかチェックする。

strict_multi_assignment

PL/PgSQLコマンドのいくつかは、SELECT INTOのように、一度に2つ以上の変数に値を割り当てることを許しています。PL/PgSQLは、ない値に対してはNULLを使い、余分な変数は無視しますが、通常は対象の変数の数と元の変数の数は一致するべきです。このチェックを有効にすると、PL/PgSQLは対象の変数の数と元の変数の数が異なる場合には必ずWARNINGまたはERRORを発生するようになります。

too_many_rows

このチェックを有効にすると、PL/PgSQLはINTO句が使われている場合、与えられた問い合わせが2行以上の行を返すかどうか確認します。INTO文は必ず1行に対してのみ使われますので、複数の行を返す問い合わせがあるということは一般に非効率かつ/または非決定論的であり、そのためおそらくエラーです。

以下の例では、plpgsql.extra_warningsをshadowed_variablesに設定した結果を示します。

```
SET plpgsql.extra_warnings TO 'shadowed_variables';

CREATE FUNCTION foo(f1 int) RETURNS int AS $$
DECLARE
f1 int;
BEGIN
RETURN f1;
END;
```

```
$$ LANGUAGE plpgsql;
WARNING:  variable "f1" shadows a previously defined variable
LINE 3: f1 int;
        ^
CREATE FUNCTION
```

以下の例では、plpgsql.extra_warningsをstrict_multi_assignmentに設定した結果を示します。

```
SET plpgsql.extra_warnings TO 'strict_multi_assignment';

CREATE OR REPLACE FUNCTION public.foo()
 RETURNS void
 LANGUAGE plpgsql
AS $$
DECLARE
  x int;
  y int;
BEGIN
  SELECT 1 INTO x, y;
  SELECT 1, 2 INTO x, y;
  SELECT 1, 2, 3 INTO x, y;
END;
$$;

SELECT foo();
WARNING:  number of source and target fields in assignment does not match
DETAIL:  strict_multi_assignment check of extra_warnings is active.
HINT:  Make sure the query returns the exact list of columns.
WARNING:  number of source and target fields in assignment does not match
DETAIL:  strict_multi_assignment check of extra_warnings is active.
HINT:  Make sure the query returns the exact list of columns.

 foo
-----

(1 row)
```

42.13. Oracle PL/SQLからの移植

本節ではOracle®からPostgreSQLへアプリケーションを移植する開発者の手助けとなるように、PostgreSQLのPL/pgSQL言語とOracleのPL/SQL言語の違いについて説明します。

PL/pgSQLは多くの点でPL/SQLに似ています。それはブロックで構成されていて、厳格な言語であり、全ての変数は宣言されなければならない点です。代入やループ、条件分岐も同様です。PL/SQLからPL/pgSQLに移植する際に注意しなければならない、主な違いを以下に示します。

- SQLコマンド内に使用された名前が、テーブルの列名または関数の変数への参照のどちらにもなり得る場合、PL/SQLは列名として処理します。これはPL/pgSQLにおけるplpgsql.variable_conflict = use_column時の動作に対応しますが、42.11.1の説明通り、これはデフォルトではありません。初期段階において、そのようなあいまいさを避けることが最善です。しかしこの動作に依存するコードの量が多いものを移植しなければならない場合、variable_conflictを使用することが最善の解法かもしれません。

- PostgreSQLの関数本体は文字列リテラルとして書かなければなりません。したがって、関数本体内部でドル引用符を使用するか、単一引用符をエスケープする必要があります。(42.12.1を参照してください)。

- データ型名はしばしば翻訳が必要です。たとえば、Oracleでは文字列の値はよくvarchar2型と宣言されますが、それは非標準SQL型です。PostgreSQLでは、その代わりにvarchar型またはtext型を使ってください。同様に、number型はnumeric型で置き換えるか、より適切なものがあるなら他の数値データ型を使ってください。

- パッケージの代わりに、スキーマを使用して関数群をグループにまとめてください。

- パッケージがないため、パッケージレベルの変数もありません。これは幾分か厄介なことです。代わって、セッションごとの状態を一時テーブル内部に保存できます。

- REVERSEを付けた整数FORループの処理は異なります。PL/SQLでは最後の数から最初の数へ減少しながら処理しますが、PL/pgSQLでは最初の数から最後の数へ減少しながら処理します。移植において、ループの両端となる最初の数と最後の数を交換する必要があります。この非互換性は不幸なことですが、変わりそうもありません。(42.6.5.5を見てください。)

- 問い合わせ上のFORループも(カーソルを除いて)異なって処理されます。対象の変数は宣言されなければなりませんが、PL/SQLは常にそれらを暗黙的に宣言します。この優位点は変数値をループを抜けてからでも依然としてアクセスできることです。

- カーソル変数の使用に対する様々な表記上の違いがあります。

42.13.1. 移植例

例 42.9に簡単な関数のPL/SQLからPL/pgSQLへの移植方法を示します。

例42.9 簡単な関数のPL/SQLからPL/pgSQLへの移植

以下はOracle PL/SQLの関数です。

```
CREATE OR REPLACE FUNCTION cs_fmt_browser_version(v_name varchar2,
                                                  v_version varchar2)
RETURN varchar2 IS
BEGIN
    IF v_version IS NULL THEN
        RETURN v_name;
    END IF;
    RETURN v_name || '/' || v_version;
END;
/
```

```
show errors;
```

この関数を通じて、PL/pgSQLとの違いを見てみましょう。

- 型名varchar2は、varcharまたはtextに変えなければなりません。この節の例ではvarcharを使いますが、文字列を特定の長さに制限する必要がないのであればtextの方がたいていは良い選択です。
- 関数プロトタイプ内のRETURNキーワード（関数本体ではありません）はPostgreSQLではRETURNSになります。同様にISはASになります。PL/pgSQL以外の言語でも関数を記述できるため、LANGUAGE句が必要となります。
- PostgreSQLは関数本体を文字列リテラルと考えます。したがって、それを囲むドル引用符または他の引用符が必要です。これは/で終了するOracleの方法の代替です。
- PostgreSQLにはshow errorsコマンドはありません。また、エラーが自動的に表示されるため、必要ありません。

それではPostgreSQLに移植されると、この関数がどのようになるか見てみましょう。

```
CREATE OR REPLACE FUNCTION cs_fmt_browser_version(v_name varchar,
                                                  v_version varchar)
RETURNS varchar AS $$
BEGIN
    IF v_version IS NULL THEN
        RETURN v_name;
    END IF;
    RETURN v_name || '/' || v_version;
END;
$$ LANGUAGE plpgsql;
```

例 42.10は、他の関数を生成する関数を移植する方法、ならびに、その結果発生する引用符問題を扱う方法を示します。

例42.10 他の関数を生成するPL/SQLをPL/pgSQLに移植

以下の手続きは、SELECT文からの行を取って、効率のためにIF文で結果を巨大な関数に埋め込んでいます。

以下はOracle版です。

```
CREATE OR REPLACE PROCEDURE cs_update_referrer_type_proc IS
    CURSOR referrer_keys IS
        SELECT * FROM cs_referrer_keys
        ORDER BY try_order;
    func_cmd VARCHAR(4000);
BEGIN
    func_cmd := 'CREATE OR REPLACE FUNCTION cs_find_referrer_type(v_host IN VARCHAR2,
                 v_domain IN VARCHAR2, v_url IN VARCHAR2) RETURN VARCHAR2 IS BEGIN';

    FOR referrer_key IN referrer_keys LOOP
        func_cmd := func_cmd ||
```

```
          ' IF v_' || referrer_key.kind
          || ' LIKE ''' || referrer_key.key_string
          || ''' THEN RETURN ''' || referrer_key.referrer_type
          || '''; END IF;';
    END LOOP;

    func_cmd := func_cmd || ' RETURN NULL; END;';

    EXECUTE IMMEDIATE func_cmd;
END;
/
show errors;
```

この関数をPostgreSQLで記述するとこうなるでしょう。

```
CREATE OR REPLACE PROCEDURE cs_update_referrer_type_proc() AS $func$
DECLARE
    referrer_keys CURSOR IS
        SELECT * FROM cs_referrer_keys
        ORDER BY try_order;
    func_body text;
    func_cmd text;
BEGIN
    func_body := 'BEGIN';

    FOR referrer_key IN referrer_keys LOOP
        func_body := func_body ||
          ' IF v_' || referrer_key.kind
          || ' LIKE ' || quote_literal(referrer_key.key_string)
          || ' THEN RETURN ' || quote_literal(referrer_key.referrer_type)
          || '; END IF;' ;
    END LOOP;

    func_body := func_body || ' RETURN NULL; END;';

    func_cmd :=
      'CREATE OR REPLACE FUNCTION cs_find_referrer_type(v_host varchar,
                                                        v_domain varchar,
                                                        v_url varchar)
        RETURNS varchar AS '
      || quote_literal(func_body)
      || ' LANGUAGE plpgsql;' ;

    EXECUTE func_cmd;
END;
$func$ LANGUAGE plpgsql;
```

関数本体を別途作成し、それをquote_literalに渡して本体内の引用符を二重化する方法に注目してください。新規の関数を定義する時ドル引用符の使用が安全とは限らないため、この方法が必要となります。それはreferrer_key.key_stringの領域に、どのような文字列が書き込まれているか不明だからです。(referrer_key.kindは常に信用できるhostかdomainかurlであると仮定しますが、どんなものでもreferrer_key.key_stringになり得るので、ドル記号を含む可能性があります。) この関数はOracle版より実際に改善されています。それはreferrer_key.key_stringまたはreferrer_key.referrer_typeが引用符を含む時、おかしなコードを生成しないからです。

例 42.11は、OUTパラメータを持ち、文字列操作を行う関数の移植方法を示します。PostgreSQLには組み込みのinstr関数はありませんが、他の関数を組み合わせることで作成できます。42.13.3に、移植を簡略化できるようにinstrのPL/pgSQLによる実装を示します。

例42.11 文字列操作を行い、OUTパラメータを持つPL/SQLプロシージャのPL/pgSQLへの移植

以下のOracle PL/SQLプロシージャは、URLを解析していくつかの要素(ホスト、パス、問い合わせ)を返します。

以下はOracle版です。

```
CREATE OR REPLACE PROCEDURE cs_parse_url(
    v_url IN VARCHAR2,

    v_host OUT VARCHAR2,  -- この値は戻されます
    v_path OUT VARCHAR2,  -- この値も戻されます
    v_query OUT VARCHAR2) -- この値も戻されます
IS
    a_pos1 INTEGER;
    a_pos2 INTEGER;
BEGIN
    v_host := NULL;
    v_path := NULL;
    v_query := NULL;
    a_pos1 := instr(v_url, '//');

    IF a_pos1 = 0 THEN
        RETURN;
    END IF;
    a_pos2 := instr(v_url, '/', a_pos1 + 2);
    IF a_pos2 = 0 THEN
        v_host := substr(v_url, a_pos1 + 2);
        v_path := '/';
        RETURN;
    END IF;

    v_host := substr(v_url, a_pos1 + 2, a_pos2 - a_pos1 - 2);
    a_pos1 := instr(v_url, '?', a_pos2 + 1);
```

```
    IF a_pos1 = 0 THEN
        v_path := substr(v_url, a_pos2);
        RETURN;
    END IF;

    v_path := substr(v_url, a_pos2, a_pos1 - a_pos2);
    v_query := substr(v_url, a_pos1 + 1);
END;
/
show errors;
```

PL/pgSQLへの可能な変換は以下のようになります。

```
CREATE OR REPLACE FUNCTION cs_parse_url(
    v_url IN VARCHAR,

    v_host OUT VARCHAR,  -- この値は戻されます
    v_path OUT VARCHAR,  -- この値も戻されます
    v_query OUT VARCHAR) -- この値も戻されます
AS $$
DECLARE
    a_pos1 INTEGER;
    a_pos2 INTEGER;
BEGIN
    v_host := NULL;
    v_path := NULL;
    v_query := NULL;
    a_pos1 := instr(v_url, '//');

    IF a_pos1 = 0 THEN
        RETURN;
    END IF;
    a_pos2 := instr(v_url, '/', a_pos1 + 2);
    IF a_pos2 = 0 THEN
        v_host := substr(v_url, a_pos1 + 2);
        v_path := '/';
        RETURN;
    END IF;

    v_host := substr(v_url, a_pos1 + 2, a_pos2 - a_pos1 - 2);
    a_pos1 := instr(v_url, '?', a_pos2 + 1);

    IF a_pos1 = 0 THEN
        v_path := substr(v_url, a_pos2);
        RETURN;
    END IF;
```

```
    v_path := substr(v_url, a_pos2, a_pos1 - a_pos2);
    v_query := substr(v_url, a_pos1 + 1);
END;
$$ LANGUAGE plpgsql;
```

この関数は以下のように使用できます。

```
SELECT * FROM cs_parse_url('http://foobar.com/query.cgi?baz');
```

例 42.12は、Oracleに特化した多くの機能を使用したプロシージャの移植方法を示します。

例42.12 PL/SQLプロシージャのPL/pgSQLへの移植

以下はOracle版です。

```
CREATE OR REPLACE PROCEDURE cs_create_job(v_job_id IN INTEGER) IS
    a_running_job_count INTEGER;
BEGIN
    LOCK TABLE cs_jobs IN EXCLUSIVE MODE;

    SELECT count(*) INTO a_running_job_count FROM cs_jobs WHERE end_stamp IS NULL;

    IF a_running_job_count > 0 THEN

        COMMIT; -- ロックを解放
        raise_application_error(-20000,
                'Unable to create a new job: a job is currently running.');
    END IF;

    DELETE FROM cs_active_job;
    INSERT INTO cs_active_job(job_id) VALUES (v_job_id);

    BEGIN
        INSERT INTO cs_jobs (job_id, start_stamp) VALUES (v_job_id, now());
    EXCEPTION

        WHEN dup_val_on_index THEN NULL; -- 既存であっても問題なし
    END;
    COMMIT;
END;
/
show errors
```

それでは、このプロシージャをPL/pgSQLに移植することができた方法を見てみましょう。

```
CREATE OR REPLACE PROCEDURE cs_create_job(v_job_id integer) AS $$
```

```
DECLARE
    a_running_job_count integer;
BEGIN
    LOCK TABLE cs_jobs IN EXCLUSIVE MODE;

    SELECT count(*) INTO a_running_job_count FROM cs_jobs WHERE end_stamp IS NULL;

    IF a_running_job_count > 0 THEN

        COMMIT; -- ロックを解放
        RAISE EXCEPTION 'Unable to create a new job: a job is currently running'; -- 1
    END IF;

    DELETE FROM cs_active_job;
    INSERT INTO cs_active_job(job_id) VALUES (v_job_id);

    BEGIN
        INSERT INTO cs_jobs (job_id, start_stamp) VALUES (v_job_id, now());
    EXCEPTION
        WHEN unique_violation THEN -- 2

            -- 既存であっても問題なし
    END;
    COMMIT;
END;
$$ LANGUAGE plpgsql;
```

1 基本のRAISE exception_nameである場合は同様に操作できますが、RAISE構文はOracleにおける文とかなり異なります。

2 PL/pgSQLがサポートする例外の名称は、Oracleと異なります。提供する例外の名称は、はるかに広範囲です（付録Aを参照してください）。今のところ、ユーザ定義の例外名称を宣言できません。しかし代わりにユーザが選択したSQLSTATE値を返すことができます。

42.13.2. その他の注意事項

本節では、Oracle PL/SQL関数のPostgreSQLへの移植における、その他の注意事項を説明します。

42.13.2.1. 例外後の暗黙的ロールバック

PL/pgSQLにおいてEXCEPTION句が例外を捕捉すると、BEGIN以降のそのブロックにおけるデータベースの変更が自動的にロールバックされます。すなわち、Oracleで以下のプログラムと同等の処理が実行されます。

```
BEGIN
    SAVEPOINT s1;
    ... code here ...
EXCEPTION
```

```
    WHEN ... THEN
        ROLLBACK TO s1;
        ... code here ...
    WHEN ... THEN
        ROLLBACK TO s1;
        ... code here ...
END;
```

このような方式でSAVEPOINTとROLLBACK TOを使用したOracleのプロシージャの書き換えは簡単です。単にSAVEPOINTとROLLBACK TOの処理を削除すればよいだけです。これと異なった方式でSAVEPOINTとROLLBACK TOを使用したプロシージャの時は、それに応じた工夫が必要になると思われます。

42.13.2.2. EXECUTE

PL/pgSQLのEXECUTEはPL/SQL版とよく似ています。しかし42.5.4で説明されているquote_literalとquote_identを使うことを覚えておかなければいけません。これらの関数を使用しない限りEXECUTE ''SELECT * from $1'';という構文の動作には、信頼性がありません。

42.13.2.3. PL/pgSQL関数の最適化

PostgreSQLには実行を最適化するために2つの関数生成修飾子があります。変動性（同じ引数が与えられた場合常に同じ結果を返します）と「厳密性」（引数のいずれかにNULLが含まれる場合NULLを返します）です。詳細はCREATE FUNCTIONを参照してください。

これらの最適化属性を利用するためには、CREATE FUNCTION文を以下のようにします。

```
CREATE FUNCTION foo(...) RETURNS integer AS $$
...
$$ LANGUAGE plpgsql STRICT IMMUTABLE;
```

42.13.3. 付録

本節には、移植作業を簡略化するために使用できる、Oracle互換のinstr関数のコードがあります。

```
--
-- Oracleのものと同じ動作をするinstr関数
-- 構文: instr(string1, string2 [, n [, m]]) ただし、
      []は省略可能なパラメータ
--
-- string1内のn番目の文字から始めて、m番目のstring2を探します。
-- nが負の場合、string1の終わりからabs(n)番目の文字から始めて、逆方向に検索します。
-- nが渡されなかった場合は、1とみなします（最初の文字から探し始めます）。
-- mが渡されなかった場合は、1とみなします（最初に一致するものを見つけます）。
-- string1内のstring2の開始位置を、string2が見つからなければ0を返します。
--
```

```
CREATE FUNCTION instr(varchar, varchar) RETURNS integer AS $$
BEGIN
    RETURN instr($1, $2, 1);
END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;


CREATE FUNCTION instr(string varchar, string_to_search_for varchar,
                      beg_index integer)
RETURNS integer AS $$
DECLARE
    pos integer NOT NULL DEFAULT 0;
    temp_str varchar;
    beg integer;
    length integer;
    ss_length integer;
BEGIN
    IF beg_index > 0 THEN
        temp_str := substring(string FROM beg_index);
        pos := position(string_to_search_for IN temp_str);

        IF pos = 0 THEN
            RETURN 0;
        ELSE
            RETURN pos + beg_index - 1;
        END IF;
    ELSIF beg_index < 0 THEN
        ss_length := char_length(string_to_search_for);
        length := char_length(string);
        beg := length + 1 + beg_index;

        WHILE beg > 0 LOOP
            temp_str := substring(string FROM beg FOR ss_length);
            IF string_to_search_for = temp_str THEN
                RETURN beg;
            END IF;

            beg := beg - 1;
        END LOOP;

        RETURN 0;
    ELSE
        RETURN 0;
    END IF;
END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;
```

```
CREATE FUNCTION instr(string varchar, string_to_search_for varchar,
                      beg_index integer, occur_index integer)
RETURNS integer AS $$
DECLARE
    pos integer NOT NULL DEFAULT 0;
    occur_number integer NOT NULL DEFAULT 0;
    temp_str varchar;
    beg integer;
    i integer;
    length integer;
    ss_length integer;
BEGIN
    IF occur_index <= 0 THEN
        RAISE 'argument ''%'' is out of range', occur_index
          USING ERRCODE = '22003';
    END IF;

    IF beg_index > 0 THEN
        beg := beg_index - 1;
        FOR i IN 1..occur_index LOOP
            temp_str := substring(string FROM beg + 1);
            pos := position(string_to_search_for IN temp_str);
            IF pos = 0 THEN
                RETURN 0;
            END IF;
            beg := beg + pos;
        END LOOP;

        RETURN beg;
    ELSIF beg_index < 0 THEN
        ss_length := char_length(string_to_search_for);
        length := char_length(string);
        beg := length + 1 + beg_index;

        WHILE beg > 0 LOOP
            temp_str := substring(string FROM beg FOR ss_length);
            IF string_to_search_for = temp_str THEN
                occur_number := occur_number + 1;
                IF occur_number = occur_index THEN
                    RETURN beg;
                END IF;
            END IF;

            beg := beg - 1;
```

```
        END LOOP;

        RETURN 0;
    ELSE
        RETURN 0;
    END IF;
END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;
```

第43章 PL/Tcl — Tcl手続き言語

PL/Tclとは、PostgreSQLデータベースシステムにロード可能な手続き言語で、Tcl言語[1]を使ったPostgreSQL関数を作成できます。

43.1. 概要

PL/Tclは、いくつか制限がありますが、C言語で書かれた関数と同じような能力を提供します。さらに、Tclで利用できる、強力な文字列処理ライブラリを持っています。

すべてがTclインタプリタの安全なコンテキスト内で実行されるという制約はやむを得ないものですが、逆に良い制約でもあります。安全なTclの制約付きのコマンドセットに、SPIを使ってデータベースにアクセスするコマンドと、elog()を使ってメッセージを処理するためのコマンドなどの、わずかなコマンドが追加されています。C関数では可能ですが、PL/Tclにはデータベースサーバ内部にアクセスする方法や、PostgreSQLサーバプロセスの権限によるOSレベルのアクセスを行う方法はありません。この結果、非特権データベースユーザがこの言語を信頼して使用することができます。つまり、無制限の権限は与えられません。

その他の注意すべき実装上の制約として、Tcl関数を使用して新しいデータ型用の入出力関数を作成することはできません。

例えば、メールを送るTcl関数が必要な場合など、安全なTclに制約されないTcl関数を書くことが望ましい場合があります。このような場合、PL/TclU（信頼されないTcl）というPL/Tclの亜種を使用します。これは、完全なTclインタプリタが使用されているという点以外の違いはありません。*PL/TclU*を使用する場合は、信頼されていない手続き言語としてインストールする必要があります。そうすることによって、データベースのスーパーユーザのみが関数を作成することができるようになります。PL/TclU関数ではデータベース管理者としてログインしたユーザができるあらゆることの実行が可能となるので、作成する際に、この関数が意図された以外のことを行わないように細心の注意を払う必要があります。

インストール時にTclサポートの設定が指定されていれば、PL/TclとPL/TclU呼び出しハンドラの共有オブジェクトコードは自動的に作成され、PostgreSQLのライブラリディレクトリにインストールされます。PL/TclまたはPL/TclUの一方あるいは両方を特定のデータベースにインストールしたい場合は、CREATE EXTENSIONコマンドを使用してください。例えば、CREATE EXTENSION pltclあるいはCREATE EXTENSION pltcluです。

43.2. PL/Tcl関数と引数

PL/Tcl言語で関数を作成するには、以下の標準的なCREATE FUNCTION構文を使用してください。

```
CREATE FUNCTION funcname (argument-types) RETURNS return-type AS $$

    # PL/Tcl関数本体
$$ LANGUAGE pltcl;
```

PL/TclUでも、言語にpltcluを指定しなければならない点以外は同様です。

[1] https://www.tcl.tk/

関数本体は、単なる小さなTclスクリプトです。関数が呼び出された時、引数の値はTclスクリプトに1 ... nという名前の変数として渡されます。結果は通常通りreturn文を使用してTclのコードから返されます。プロシージャでは、Tclコードからの戻り値は無視されます。

例えば、2つの整数のうち大きな方を返す関数は以下のように定義できます。

```
CREATE FUNCTION tcl_max(integer, integer) RETURNS integer AS $$
    if {$1 > $2} {return $1}
    return $2
$$ LANGUAGE pltcl STRICT;
```

STRICT句に注意してください。これによりプログラマは、入力にNULL値が与えられた場合を検討する手間を省くことができます。NULLが渡された場合、関数はまったく呼び出されず、単にNULLという結果が自動的に返されます。

厳密（strict）でない関数では、引数の実際の値がNULLである場合、対応する$n変数は空文字列に設定されます。ある引数がNULLかどうかを検出するためには、argisnull関数を使用してください。例えば、引数の片方がNULL、もう片方が非NULLであって、NULLではなく、非NULLの引数の方を返すtcl_maxを考えると、以下のようになります。

```
CREATE FUNCTION tcl_max(integer, integer) RETURNS integer AS $$
    if {[argisnull 1]} {
        if {[argisnull 2]} { return_null }
        return $2
    }
    if {[argisnull 2]} { return $1 }
    if {$1 > $2} {return $1}
    return $2
$$ LANGUAGE pltcl;
```

上で示した通り、NULL値をPL/Tcl関数から返すためには、return_nullを実行してください。これは、関数が厳密かどうかに関係なく、実行することができます。

複合型の引数は、Tcl配列として関数に渡されます。配列の要素名は複合型の属性名です。渡された行の属性がNULL値の場合、その属性は配列内には現れません。以下に例を示します。

```
CREATE TABLE employee (
    name text,
    salary integer,
    age integer
);

CREATE FUNCTION overpaid(employee) RETURNS boolean AS $$
    if {200000.0 < $1(salary)} {
        return "t"
    }
    if {$1(age) < 30 && 100000.0 < $1(salary)} {
```

```
        return "t"
    }
    return "f"
$$ LANGUAGE pltcl;
```

PL/Tcl関数は複合型の結果を返すこともできます。このためには、Tclコードは期待する結果型と一致する列の名前/値のペアのリストを返さなければなりません。そのリストで省略された列名は結果がNULLになり、期待されない列名があるとエラーが生じます。例を示します。

```
CREATE FUNCTION square_cube(in int, out squared int, out cubed int) AS $$
    return [list squared [expr {$1 * $1}] cubed [expr {$1 * $1 * $1}]]
$$ LANGUAGE pltcl;
```

プロシージャの出力引数は同様に返されます。以下に例を示します。

```
CREATE PROCEDURE tcl_triple(INOUT a integer, INOUT b integer) AS $$
    return [list a [expr {$1 * 3}] b [expr {$2 * 3}]]
$$ LANGUAGE pltcl;

CALL tcl_triple(5, 10);
```

ヒント

array get Tclコマンドを使って、希望するタプルの配列表現から結果リストを作成することができます。

```
CREATE FUNCTION raise_pay(employee, delta int) RETURNS employee AS $$
    set 1(salary) [expr {$1(salary) + $2}]
    return [array get 1]
$$ LANGUAGE pltcl;
```

PL/Tcl関数は集合を返すことができます。このためにはTclコードで、return_nextを返却する行ごとに呼び出します。スカラー値を返却する場合は適切な値を、複合型を返す場合は列の名前/値ペアのリストを渡します。スカラー型を返す例を示します。

```
CREATE FUNCTION sequence(int, int) RETURNS SETOF int AS $$
    for {set i $1} {$i < $2} {incr i} {
        return_next $i
    }
$$ LANGUAGE pltcl;
```

複合型を返す例を示します。

```
CREATE FUNCTION table_of_squares(int, int) RETURNS TABLE (x int, x2 int) AS $$
    for {set i $1} {$i < $2} {incr i} {
```

```
        return_next [list x $i x2 [expr {$i * $i}]]
    }
$$ LANGUAGE pltcl;
```

43.3. PL/Tclにおけるデータの値

PL/Tcl関数コードに与えられる引数の値は、単に、テキスト形式（SELECT文によりそれを表示した場合と同じ形式）に変換された入力引数です。逆に、returnコマンドとreturn_nextコマンドは、その関数宣言における戻り値の型、あるいは複合型の戻り値型の入力書式として受け付けることができる、任意の文字列を受け付けます。

43.4. PL/Tclにおけるグローバルデータ

ある関数の複数の呼び出し間で保持される、もしくは、異なる関数間で共有されるような、いくつかのグローバルデータを持つことが有意な場合があります。これはPL/Tclで簡単に実現できますが、理解する必要がある制限がいくつかあります。

セキュリティ上の理由のため、PL/Tclは、任意のSQLロールによって呼び出された関数をそのロール用の別のTclインタプリタで実行します。これにより、他のユーザのPL/Tcl関数の処理によってあるユーザへの事故または悪意の干渉を防止します。こうしたインタプリタはそれぞれ任意の「グローバル」なTcl変数を持ちます。したがって、同じSQLロールにより実行されていれば、2つのPL/Tcl関数は同じグローバル変数を共有します。単一セッション内で（SECURITY DEFINER関数またはSET ROLEなどを通して）複数のSQLロールでコードを実行するアプリケーションでは、PL/Tcl関数が確実にデータを共有できるように明示的な処理を行う必要があるかもしれません。このためには、通信しなければならない関数が同一ユーザで所有されていること、 および、それがSECURITY DEFINERとして印がついていることを確実にしてください。当然ながら、こうした関数が意図しない動作を行うために使われることのないよう注意しなければなりません。

セッション内で使用されるすべてのPL/TclU関数は、当然ながらPL/Tcl関数とは別のインタプリタですが、 同一のTclインタプリタ内で実行されます。このためPL/TclU関数間ではグローバルデータは自動的に共有されます。すべてのPL/TclU関数は同じ信頼レベル、すなわちデータベーススーパーユーザで実行されますので、これはセキュリティ上危険とはみなされません。

PL/Tcl関数が予期しない相互作用に巻き込まれないようにするために、upvarコマンドを使用することによって、各関数でアクセスできるグローバルな配列を作成することができます。この変数のグローバル名は関数の内部名で、ローカル名はGDとなります。関数の永続局所データではGDを使用することを推奨します。複数の関数で共用させる予定の値に対してのみ、通常のTclのグローバル変数を使用してください。（GD配列が特定のインタプリタ内のみでグローバルであることに注意してください。このため、これらは上記のセキュリティ制限を迂回することはありません。）

後述のspi_execpの例の中にGDの使用例があります。

43.5. PL/Tclからのデータベースアクセス

下記のコマンドは、PL/Tcl関数内からデータベースアクセスを行う時に使用できるコマンドです。

spi_exec ?-count n? ?-array name? command ?loop-body?

文字列として与えられたSQL問い合わせを実行します。コマンド内のエラーは、エラーの発生となります。さもなければ、このspi_execの戻り値はコマンドによって処理（選択、挿入、更新、削除）された行数、または、コマンドがユーティリティ文の場合はゼロとなります。さらに、コマンドがSELECT文の場合、選択された列の値は以下のようにTclの変数に格納されます。

-countオプションの値は、spi_execに対し、そのコマンドで処理する最大行数を指示します。これにより、問い合わせをカーソルとして設定し、FETCH nを実行することと同じことができます。

コマンドがSELECT文の場合、その結果得られた列の値は、列名にちなんだ名前のTcl変数に格納されます。-arrayオプションが付与された場合は、列の値は指定された名前の連想配列の要素に格納され、 その配列のインデックスとして列名が使用されます。加えて、結果内での現在の行番号（ゼロから数えます）が「.tupno」という名前の配列要素に格納されます。ただし、その名前が結果内の列名として使われていない場合に限られます。

問い合わせ文がSELECT文、かつ、loop-bodyスクリプトが付与されなかった場合、結果のうち最初の行だけがTclの変数または配列要素に格納されます。他にも行があったとしても、それらは無視されます。問い合わせが行を返さなかった場合は、変数への格納は発生しません（spi_execの戻り値を検査することで、これを検出することができます）。以下に例を示します。

```
spi_exec "SELECT count(*) AS cnt FROM pg_proc"
```

これは、$cnt Tcl変数を、pg_procシステムカタログの行数に設定します。

loop-bodyオプション引数が付与された場合、それは、問い合わせの結果内の行それぞれに対して一度だけ実行される小さなTclスクリプトです（loop-bodyはSELECT以外の問い合わせで付与された場合は無視されます）。処理中の行の列値は、各繰り返しの前にTclの変数または配列要素に格納されます。以下に例を示します。

```
spi_exec -array C "SELECT * FROM pg_class" {
    elog DEBUG "have table $C(relname)"
}
```

これは、pg_classの各行に対してログメッセージを出力します。この機能は他のTclの繰り返し構文でも同様に動作します。特にループ本体内のcontinueとbreakは通常通り動作します。

問い合わせの結果、列がNULLであった場合、対象となる変数は代入されずに、「未設定状態」になります。

spi_prepare query typelist

後の実行のために問い合わせ計画の準備、保存を行います。保存された計画は現在のセッションが終了するまで保持されます。

問い合わせはパラメータ、つまり、計画が実際に実行される時に常に与えられる値用のプレースホルダを持つことができます。問い合わせ文字列の中では、$1 ... $nというシンボルを使用して引数を参照してください。問い合わせがパラメータを使用する場合、Tclのリストとしてパラメータの型名を指定する必要があります。（パラメータを使用しない場合はtypelistには空のリストを指定してください。）

spi_prepareの戻り値は問い合わせIDです。このIDは後にspi_execpを呼び出す時に使用されます。使用例についてはspi_execpを参照してください。

spi_execp ?-count n? ?-array name? ?-nulls string? queryid ?value-list? ?loop-body?

spi_prepareにより事前に準備された問い合わせを実行します。queryidはspi_prepareにより返されたIDです。その問い合わせがパラメータを参照する場合、value-listを与える必要があります。これは、そのパラメータの実際の値を持つTclのリストです。このリストの長さは、事前にspi_prepareで指定した引数型のリストの長さと同じでなければなりません。問い合わせにパラメータがない場合は、value-listを省略してください。

-nullsオプションの値は、空白文字と'n'という文字からなる文字列で、spi_execpに対し、どの引数がNULL値かを示します。指定された場合、その文字列の長さはvalue-listの長さと正確に一致していなければなりません。指定されない場合は、すべてのパラメータの値は非NULLです。

問い合わせとそのパラメータをどこで指定するのかという点を除き、spi_execpはspi_execと同様に動作します。-count、-array、loop-bodyオプションも、そして、結果の値も同じです。

ここで、プリペアド計画を使用した、PL/Tcl関数の例を示します。

```
CREATE FUNCTION t1_count(integer, integer) RETURNS integer AS $$
    if {![ info exists GD(plan) ]} {

        # 最初の呼び出しでは保存する計画を準備します。
        set GD(plan) [ spi_prepare \
                "SELECT count(*) AS cnt FROM t1 WHERE num >= \$1 AND num <= \$2" \
                [ list int4 int4 ] ]
    }
    spi_execp -count 1 $GD(plan) [ list $1 $2 ]
    return $cnt
$$ LANGUAGE pltcl;
```

spi_prepareに与える問い合わせ文字列の内側では、$n記号が確実にそのままspi_prepareに渡され、Tcl変数の代入による置き換えが起こらないようにバックスラッシュが必要です。

subtransaction command

commandに含まれるTclスクリプトが、SQLサブトランザクション中で実行されます。スクリプトがエラーを返すと、上位のTclコードにエラーを返す前に、そのサブトランザクションをロールバックします。更なる詳細と使用例については43.9を参照してください。

quote string

指定された文字列内のすべての単一引用符とバックスラッシュ文字を二重化します。spi_execやspi_prepareで与えられたSQL問い合わせに挿入される予定の文字列を安全に引用符付けするために、これを使用することができます。例えば、以下のような問い合わせ文字列を考えます。

```
"SELECT '$val' AS ret"
```

ここで、val Tcl変数にdoesn't が実際に含まれているものとします。これは最終的に以下の問い合わせ文字列になってしまいます。

```
SELECT 'doesn't' AS ret
```

これでは、spi_execまたはspi_prepareの実行中に解析エラーが発生してしまいます。正しく稼動させるには、実行したい問い合わせは以下のようにしなければなりません。

```
SELECT 'doesn''t' AS ret
```

これは、PL/Tclでは以下により形成することができます。

```
"SELECT '[ quote $val ]' AS ret"
```

spi_execpの持つ1つの利点は、パラメータはSQL問い合わせ文字列の一部として解析されることがありませんので、このようにパラメータの値を引用符付けする必要がないことです。

elog level msg

ログまたはエラーメッセージを発行します。使用できるレベルは、DEBUG、LOG、INFO、NOTICE、WARNING、ERROR、およびFATALです。ERRORはエラー状態を発生します。その上位レベルのTclコードで例外が捕捉されなければ、このエラーは問い合わせ呼び出し処理の外部へ伝播され、その結果、現在のトランザクションもしくはサブトランザクションはアボートされます。これは実質的にTclのerrorコマンドと同一です。FATALはトランザクションをアボートし、現在のセッションを停止させます。(PL/Tcl関数においてこのエラーレベルを使用すべき理由はおそらく存在しませんが、完全性のために用意されています。) 他のレベルは、異なる重要度のメッセージを生成するだけです。log_min_messagesとclient_min_messages設定パラメータは、特定の重要度のメッセージをクライアントに報告するか、サーバのログに書き出すか、あるいはその両方かを制御します。詳細については第19章および43.8を参照してください。

43.6. PL/Tclのトリガ関数

トリガ関数をPL/Tclで作成することができます。PostgreSQLでは、トリガとして呼び出される関数は、trigger型の戻り値を返す引数のない関数として宣言する必要があります。

トリガマネージャからの情報は、以下の変数内に格納されて関数本体に渡されます。

$TG_name

CREATE TRIGGER文によるトリガ名。

$TG_relid

そのトリガ関数の呼び出しを発生させたテーブルのオブジェクトID。

$TG_table_name

そのトリガ関数の呼び出しを発生させたテーブルの名前。

$TG_table_schema

そのトリガプロシージャ呼び出しが発生したテーブルのスキーマ。

$TG_relatts

先頭に空のリスト要素を持つ、テーブルの列名のTclリスト。Tclのlsearchコマンドを使用して、そのリストから列名を検索することで、最初の列を1とした要素番号が返されます。これは、PostgreSQLでの通常の列の番号付けと同じです。(また空のリスト要素は、右側の列の属性番号を正しくするために、削除された列の位置に現れます。)

$TG_when

トリガイベントの種類に応じた、BEFORE、AFTERまたはINSTEAD OFという文字列。

$TG_level

トリガイベントの種類に応じた、ROWまたはSTATEMENTという文字列。

$TG_op

トリガイベントの種類に応じた、INSERT、UPDATE、DELETE、またはTRUNCATEの文字列。

$NEW

INSERT/UPDATE動作の場合は新しいテーブル行の値を、DELETE動作の場合は空を持つ連想配列。配列のインデックスは列名です。NULLの列はこの配列内には現れません。文レベルのトリガに対しては設定されません。

$OLD

UPDATE/DELETE動作の場合は古いテーブル行の値を、INSERT動作の場合は空を持つ連想配列。配列のインデックスは列名です。NULLの列はこの配列内には現れません。文レベルのトリガに対しては設定されません。

$args

CREATE TRIGGER文で指定された、関数への引数のTclリスト。この引数は、関数本体から$1 ... $nとしてもアクセスすることができます。

トリガ関数からの戻り値は、OKという文字列、SKIPという文字列、列名/値の組のリスト、の内の1つを取ることができます。戻り値がOKの場合、トリガを発行した操作(INSERT/UPDATE/DELETE)は正常に処理されます。SKIPはトリガマネージャにこの行に対する操作を何も出力せずに中止するように通知します。リストが返された場合は、PL/Tclに対し、変更した行をトリガマネージャに返すことを通知します。変更行の内容はリスト内の列名と値により指定されます。リストで言及されなかった列は全てNULLが置かれます。変更された行を返すことは、$NEW内で与えられる行ではなく変更された行が挿入される、行レベルのBEFORE INSERTまたはUPDATEトリガ、または、返される行がINSERT RETURNINGおよびUPDATE RETURNING句の元データとして使われる、行レベルのINSTEAD OF INSERTまたはUPDATEトリガでのみ有意です。行レベルのBEFORE DELETEまたはINSTEAD OF DELETEトリガでは、変更された行が返されることがOKが返されるのと同じ効果を持ち、その操作は処理されます。この他の種類のトリガでは戻り値は無視されます。

ヒント

結果リストはarray get Tclコマンドによる変更されたタプルの配列表現から作ることができます。

ここで、テーブル内の整数値としてその行に対する更新数を記録させる、小さめのトリガプロシージャの例を示します。新規の行が挿入された場合は、その値はゼロに初期化され、その後の各更新操作時に1が加算されます。

```
CREATE FUNCTION trigfunc_modcount() RETURNS trigger AS $$
    switch $TG_op {
        INSERT {
            set NEW($1) 0
        }
        UPDATE {
            set NEW($1) $OLD($1)
            incr NEW($1)
        }
        default {
            return OK
        }
    }
    return [array get NEW]
$$ LANGUAGE pltcl;

CREATE TABLE mytab (num integer, description text, modcnt integer);

CREATE TRIGGER trig_mytab_modcount BEFORE INSERT OR UPDATE ON mytab
    FOR EACH ROW EXECUTE FUNCTION trigfunc_modcount('modcnt');
```

トリガ関数自身は列名を認識していない点に注目してください。これはトリガの引数として与えられます。これにより、このトリガ関数を別のテーブルで再利用することができます。

43.7. PL/Tclにおけるイベントトリガ関数

イベントトリガ関数をPL/Tclで作成することができます。PostgreSQLでは、イベントトリガとして呼び出される関数は、event_trigger型の戻り値を返す引数のない関数として宣言する必要があります。

トリガマネージャからの情報は、以下の変数内に格納されて関数本体に渡されます。

$TG_event

トリガが発行されたイベント名

$TG_tag

トリガが発行されたコマンドタグ

トリガ関数の戻り値は無視されます。

サポートするコマンドが実行される度に、単にNOTICEメッセージを発行するイベントトリガ関数の例を、以下に示します。

```
CREATE OR REPLACE FUNCTION tclsnitch() RETURNS event_trigger AS $$
  elog NOTICE "tclsnitch: $TG_event $TG_tag"
$$ LANGUAGE pltcl;

CREATE EVENT TRIGGER tcl_a_snitch ON ddl_command_start EXECUTE FUNCTION tclsnitch();
```

43.8. PL/Tclのエラー処理

PL/Tcl関数中の、あるいはPL/Tcl関数から呼ばれるTclコードは、無効な演算の実行により、あるいはTclのerrorコマンドやPL/Tclのelogコマンドを使ってエラーを生成することにより、エラーとなることがありえます。これらエラーはTclのcatchコマンドを使ってTcl内で捕捉することができます。あるエラーが捕捉されず、PL/Tcl関数実行のトップレベルに伝播することが許容されているなら、関数が呼び出している問合せにおけるSQエラーとして報告されます。

逆に、PL/Tclのspi_exec、spi_prepare、spi_execpコマンドの中で起きるSQLエラーは、Tclのエラーとして報告され、したがって、これらはTclのcatchコマンドにより捕捉可能です。(各々のPL/Tclコマンドは、エラー時にロールバックするSQL操作をサブトランザクション中で実行するので、部分的に完了した操作は自動的に後始末されます。) ここでも同様に、捕捉されることなくトップレベルに伝播するならSQLエラーに戻ります。

Tclは、Tclプログラムで解釈しやすい形式でエラーに関する追加情報を表現できるerrorCode変数を提供します。変数の中身はTclリスト形式で、1番目の語でエラーを報告したサブシステムまたはライブラリを識別します。それ以降の内容は個々のサブシステムやライブラリに任されています。PL/Tclコマンドで報告されるデータベースエラーむけには、1番目の語がPOSTGRES、2番目の語がPostgreSQLのバージョン番号で、それ続く語はエラーの詳細情報を提供するフィールド名と値の組です。フィールドSQLSTATE、condition、およびmessageは常に与えられます(最初の2つは付録Aにあるエラーコードと状態名です)。出現しうるフィールドとしては、detail、hint、context、schema、table、column、datatype、constraint、statement、cursor_position、filename、linenoおよびfuncnameがあります。

PL/TclのerrorCode情報を処理する便利な方法は、それを配列に読み込むことです。これによりフィールド名は配列の添え字になります。これを行うコードは以下のようになります。

```
if {[catch { spi_exec $sql_command }]} {
    if {[lindex $::errorCode 0] == "POSTGRES"} {
        array set errorArray $::errorCode
        if {$errorArray(condition) == "undefined_table"} {
            # deal with missing table
        } else {
            # deal with some other type of SQL error
        }
    }
}
```

（二重コロンはerrorCodeがグローバル変数であることを明示的に指定します。）

43.9. PL/Tclにおける明示的サブトランザクション

43.8で説明されているように、データベースアクセスによって生じたエラーからの回復により、操作のうちいくつかが失敗する前に他の操作が成功し、エラーからの回復後、データの一貫性が失われた望ましくない状態になってしまう可能性があります。PL/Tclは明示的なトランザクションの手法でこの問題を解決する手段を提供しています。

2つのアカウントの間の送金を実装する関数を考えます。

```
CREATE FUNCTION transfer_funds() RETURNS void AS $$
    if [catch {
        spi_exec "UPDATE accounts SET balance = balance - 100 WHERE account_name = 'joe'"
        spi_exec "UPDATE accounts SET balance = balance + 100 WHERE account_name = 'mary'"
    } errormsg] {
        set result [format "error transferring funds: %s" $errormsg]
    } else {
        set result "funds transferred successfully"
    }
    spi_exec "INSERT INTO operations (result) VALUES ('[quote $result]')"
$$ LANGUAGE pltcl;
```

ふたつ目のUPDATE文で例外が発生する結果になると、この関数は失敗を記録しますが、それにもかかわらず、最初のUPDATEはコミットされます。言い換えると、Joeのアカウントから資金が引き出されたのに、Maryのアカウントには転送されません。これは、それぞれのspi_execが別々のサブトランザクションになっていて、そのうち一つのサブトランザクションだけがロールバックされるからです。

このような状況に対応するには、複数のデータベース操作を、全体が成功するか、あるいは失敗する明示的なサブトランザクションで包みます。PL/Tclは、これを管理するためのsubtransactionコマンドを提供しています。

```
CREATE FUNCTION transfer_funds2() RETURNS void AS $$
    if [catch {
        subtransaction {
            spi_exec "UPDATE accounts SET balance = balance - 100 WHERE account_name = 'joe'"
            spi_exec "UPDATE accounts SET balance = balance + 100 WHERE account_name = 'mary'"
        }
    } errormsg] {
        set result [format "error transferring funds: %s" $errormsg]
    } else {
        set result "funds transferred successfully"
    }
    spi_exec "INSERT INTO operations (result) VALUES ('[quote $result]')"
$$ LANGUAGE pltcl;
```

この目的のために、catchが必要であることに注意してください。そうでないと、エラーが関数のトップレベルまで伝搬し、期待したようなoperationsテーブルへの挿入が阻害されてしまいます。subtransactionコマンドはエラーを補足しません。エラーが報告された際に、スコープの内側で実行されたすべてのデータベース操作がロールバックされることを保証するだけです。

明示的なサブトランザクションのロールバックは、Tclのコードの中でエラーが報告された際だけでなく、データベースアクセスに起因するエラーの際にも起こります。ですから、subtransactionコマンド内の内側で起こった通常のTcl例外は、サブトランザクションのロールバックも引き起こします。しかし、Tclコードからのエラーによらない脱出（たとえばreturnによるもの）は、ロールバックをもたらしません。

43.10. トランザクション制御

トップレベル、あるいは、トップレベルから呼ばれた無名コードブロック（DOコマンド）から呼ばれたプロシージャでは、トランザクション制御が可能です。現在のトランザクションをコミットするには、commitコマンドを呼びます。現在のトランザクションをロールバックするには、rollbackコマンドを呼びます。（SQLコマンドのCOMMITやROLLBACKをspi_execなどを通して実行することはできない点に注意してください。前述の関数を使って行う必要があります。）トランザクションが終了した後、新たなトランザクションが自動的に開始されますので、開始するための別途のコマンドはありません。

以下に例を示します。

```
CREATE PROCEDURE transaction_test1()
LANGUAGE pltcl
AS $$
for {set i 0} {$i < 10} {incr i} {
    spi_exec "INSERT INTO test1 (a) VALUES ($i)"
    if {$i % 2 == 0} {
        commit
    } else {
        rollback
    }
}
$$;

CALL transaction_test1();
```

明示的なサブトランザクションの中ではトランザクションを終了することはできません。

43.11. PL/Tclの設定

この節では、PL/Tclに影響がある設定パラメータを列挙します。

pltcl.start_proc (string)

このパラメータが空文字以外に設定された場合、PL/Tclのための新しいTclインタプリタが作成された際に実行すべきパラメータなしのPL/Tcl関数の名前（スキーマ修飾される場合もあります）を指定しま

す。そうした関数は、追加のTclコードをロードするような、Tclセッションごとの初期化を実施できます。データベースセッションの中で新しいPL/Tcl関数が最初に実行された際、あるいはPL/Tcl関数が新しいロールから呼び出されたためにインタプリタを追加で作成しなければならない際に、新しいTclインタプリタが作られます。

参照されている関数はpltcl言語で記述しなければならず、またSECURITY DEFINERとしてマークされていてはいけません。(この制約により、その関数が初期化すると想定しているインタプリタ内で実行されることが保証されます。) また、現在のユーザはその関数を呼び出すことが許可されていなければなりません。

関数がエラーで失敗すると、関数呼び出しをアボートし、その結果新しいインタプリタが作成され、エラーは呼び出し元のクエリに伝搬し、現在のトランザクションあるいはサブトランザクションがアボートします。Tcl内でそれまでに行われた操作は取り消されません。しかし、インタプリタは再使用できません。言語が再び使用されると、新しいTclインタプリタ内で初期化が再び試みられます。

スーパーユーザだけがこの設定を変更できます。この設定はセッション内で変更できますが、すでに作成されたTclインタプリタには影響しません。

pltclu.start_proc (string)

このパラメータはPL/TclUに適用される点を除けば、pltcl.start_procと完全に類似しています。参照される関数はpltclu言語で書かれていなければなりません。

43.12. Tclプロシージャ名

PostgreSQLでは、その関数の引数の数または引数の型が異なっていれば、同じ関数名を異なる関数定義に使用することが可能です。しかし、Tclではプロシージャ名の重複は許されません。PL/Tclでは、プロシージャ名の一部にpg_procシステムテーブルにあるその関数のオブジェクトIDを持たせた内部的なTclプロシージャ名を作成することでこれに対応しています。したがって、こういった異なる引数の型を持つ同じ名前のPostgreSQL関数は、異なるTclプロシージャになります。PL/Tclプログラマから見ますと、通常は問題にはなりませんが、デバッグの際に表面に現れます。

第44章 PL/Perl — Perl手続き言語

PL/PerlはPerlプログラミング言語[1]を使用してPostgreSQL関数を作成することができる、ロード可能な手続き言語です。

PL/Perlを使用する主たる利点は、ストアドプロシージャの中で、さまざまな「文字列操作」やPerlで使用可能な関数を使用できるという点です。複雑な文字列解析は、PL/pgSQLで提供される文字列関数や制御構造体を使用するよりPerlを使用する方が簡単に行うことができます。

PL/Perlを特定のデータベースにインストールするには、`CREATE EXTENSION plperl`を使用してください。

> **ヒント**
>
> 言語を`template1`にインストールすると、その後に作成されるデータベース全てにその言語は自動的にインストールされます。

> **注記**
>
> ソースパッケージを使用するユーザは、インストール作業時にPL/Perlを特別に使用可能にする必要があります。(詳細については、第16章を参照してください。) バイナリパッケージを使用する場合は、別個のサブパッケージにPL/Perlが入っている可能性があります。

44.1. PL/Perl関数と引数

PL/Perl言語で関数を作成するには、以下の標準的なCREATE FUNCTION構文を使用してください。

```
CREATE FUNCTION funcname (argument-types)
RETURNS return-type

-- 関数の属性はここに来る
AS $$

    # PL/Perl関数本体はここに来る
$$ LANGUAGE plperl;
```

関数本体は通常のPerlのコードです。実際、PL/Perlの糊付けコードは、これをPerlのサブルーチンの内部に格納します。PL/Perl関数はスカラコンテキストとして呼び出されます。このためリストを返すことはできません。後述の通り、参照を返すことによりスカラ以外の値(配列、レコード、集合)を返すことができます。

PL/Perlプロシージャでは、Perlコードからのあらゆる戻り値は無視されます。

またPL/PerlはDO文で呼び出される匿名コードブロックをサポートします。

[1] https://www.perl.org

```
DO $$
    # PL/Perl code
$$ LANGUAGE plperl;
```

匿名コードブロックは引数を取りません。また何らかの値を返したとしても破棄されます。その他は関数と同様に動作します。

> **注記**
>
> Perl、特にその閉ざされたスコープで局所変数を参照するような場合では、名前付きの入れ子状サブルーチンの使用は危険です。PL/Perl関数はサブルーチン内に格納されますので、内部に記述した名前付きのサブルーチンはすべて入れ子にされます。一般的に、コード参照を介して呼び出す匿名サブルーチンを作成する方がかなり安全です。詳細はperldiagマニュアルページ内のVariable "%s" will not stay sharedおよびVariable "%s" is not availableを参照してください。またはインターネットで「perl nested named subroutine」を検索してください。

CREATE FUNCTIONコマンドの構文では、関数本体は文字列定数として記述されることを必須としています。通常、文字列定数にはドル引用符付け(4.1.2.4を参照)を使用することが最も便利です。エスケープ文字列構文E''を使用することを選択した場合、関数本体で使用される単一引用符(')とバックスラッシュ(\)をすべて二重にしなければなりません(4.1.2.1を参照)。

引数と結果は他のPerlサブルーチンと同様に扱われます。引数は@_の中に渡され、結果値はreturn、または、その関数で最後に評価された式として返されます。

例えば、2つの整数のうち大きな方を返す関数は以下のように定義できます。

```
CREATE FUNCTION perl_max (integer, integer) RETURNS integer AS $$
    if ($_[0] > $_[1]) { return $_[0]; }
    return $_[1];
$$ LANGUAGE plperl;
```

> **注記**
>
> PL/Perl内部での使用のため、引数はデータベースの符号化方式からUTF-8に変換され、返されるときにUTF-8からデータベースの符号化方式に戻されます。

SQLのNULL値が関数に渡された場合、その引数値はPerlにおける「未定義」として現れます。上の関数定義では、NULL値が入力された場合うまく動作しないでしょう(実際はそれがゼロであるかのように動作するでしょう)。STRICTを関数定義に加えることで、PostgreSQLの動作をより合理的にすることができます。NULL値が渡された場合、関数はまったく呼び出されず、単にNULLという結果が自動的に返されます。他の方法として、関数本体で未定義な入力を検査することもできます。例えば、perl_maxの引数の片方がNULL、もう片方が非NULLの場合に、NULL値ではなく非NULLの引数を返すようにするとします。

```
CREATE FUNCTION perl_max (integer, integer) RETURNS integer AS $$
    my ($x, $y) = @_;
    if (not defined $x) {
```

```
        return undef if not defined $y;
        return $y;
    }
    return $x if not defined $y;
    return $x if $x > $y;
    return $y;
$$ LANGUAGE plperl;
```

上で示した通り、PL/Perl関数からSQLのNULL値を返すためには、未定義値を返すようにしてください。これは、関数が厳密かどうかに関係なく、実行することができます。

関数引数の内で参照ではないものは、対応するデータ型向けのPostgreSQLの標準的な外部テキスト表現で表された文字列です。通常の数値やテキスト型では、Perlは正確に処理を行いますので、通常プログラマは心配することはありません。しかし、この他の場合では、引数をPerlでより使用しやすいように変換する必要があります。例えば、decode_bytea関数はbytea型の引数をエスケープしないバイナリに変換するために使用することができます。

同様に、PostgreSQLに戻される値を外部テキスト表現書式で表さなければなりません。例えば、bytea型の戻り値をバイナリデータにエスケープするためにencode_byteaを使用することができます。

特に重要な場合の1つは真偽値です。つい先ほど述べたように、bool値のデフォルトの振舞いはPerlにテキストとして、すなわち't'または'f'で渡されるというものです。Perlは'f'を偽とは扱いませんので、これは問題をはらんでいます。「変換」(CREATE TRANSFORMを参照してください)を使って問題を改善することができます。適切な変換がbool_plperl拡張で提供されています。使うには、拡張をインストールします。

```

CREATE EXTENSION bool_plperl;  -- PL/PerlUに対してはbool_plperlu
```

次にboolを受け取ったり返したりするPL/Perl関数に対してTRANSFORM関数属性を使います。例えば以下の通りです。

```
CREATE FUNCTION perl_and(bool, bool) RETURNS bool
TRANSFORM FOR TYPE bool
AS $$
  my ($a, $b) = @_;
  return $a && $b;
$$ LANGUAGE plperl;
```

この変換が適用されると、bool引数はPerlからは1もしくは空、すなわち正しく真または偽と見えます。関数の結果が型boolなら、Perlが戻り値を真と評価したかどうかに従って真または偽となります。同様の変換は、関数の内部で行われる真偽値の問い合わせ引数やSPI問い合わせの結果(44.3.1)でも実行されます。

Perlは、PostgreSQLの配列をPerl配列への参照として返すことができます。以下に例を示します。

```
CREATE OR REPLACE function returns_array()
RETURNS text[][] AS $$
    return [['a"b','c,d'],['e\\f','g']];
$$ LANGUAGE plperl;
```

```
select returns_array();
```

PerlはPostgreSQLの配列をblessされたPostgreSQL::InServer::ARRAYオブジェクトとして渡します。9.1より過去のPostgreSQLで作成されたPerlコードを実行させるための後方互換性のため、このオブジェクトは配列への参照または文字列として扱うことができます。以下に例を示します。

```
CREATE OR REPLACE FUNCTION concat_array_elements(text[]) RETURNS TEXT AS $$
    my $arg = shift;
    my $result = "";
    return undef if (!defined $arg);


    # 配列への参照として
    for (@$arg) {
        $result .= $_;
    }


    # 文字列としても働く
    $result .= $arg;

    return $result;
$$ LANGUAGE plperl;

SELECT concat_array_elements(ARRAY['PL','/','Perl']);
```

注記

Perlプログラマの常識のように、多次元配列は低次元配列の参照への参照として表現されます。

複合型の引数はハッシュへの参照として関数に渡されます。ハッシュのキーは複合型の属性名です。以下に例を示します。

```
CREATE TABLE employee (
    name text,
    basesalary integer,
    bonus integer
);

CREATE FUNCTION empcomp(employee) RETURNS integer AS $$
    my ($emp) = @_;
    return $emp->{basesalary} + $emp->{bonus};
$$ LANGUAGE plperl;
```

```
SELECT name, empcomp(employee.*) FROM employee;
```

必要な属性を持つハッシュの参照を返すという同じ方法で、PL/Perl関数は複合型の結果を返すことができます。以下に例を示します。

```
CREATE TYPE testrowperl AS (f1 integer, f2 text, f3 text);

CREATE OR REPLACE FUNCTION perl_row() RETURNS testrowperl AS $$
    return {f2 => 'hello', f1 => 1, f3 => 'world'};
$$ LANGUAGE plperl;

SELECT * FROM perl_row();
```

宣言された結果データ型の任意の列の内、ハッシュ内に存在しないものはNULL値として返されます。

同様に、プロシージャの出力引数はハッシュ参照で返すことができます。

```
CREATE PROCEDURE perl_triple(INOUT a integer, INOUT b integer) AS $$
    my ($a, $b) = @_;
    return {a => $a * 3, b => $b * 3};
$$ LANGUAGE plperl;

CALL perl_triple(5, 10);
```

また、PL/Perl関数はスカラ型の配列や複合型の配列を返すこともできます。通常ならば、起動処理の高速化とメモリ内の結果セット全体を待ち行列に保持できることから、1度に1行を返す方がよいでしょう。以下に示すreturn_nextを使用して、これを行うことができます。最後のreturn_nextの後で、returnまたはreturn undef（推奨）を記述しなければならないことに注意してください。

```
CREATE OR REPLACE FUNCTION perl_set_int(int)
RETURNS SETOF INTEGER AS $$
    foreach (0..$_[0]) {
        return_next($_);
    }
    return undef;
$$ LANGUAGE plperl;

SELECT * FROM perl_set_int(5);

CREATE OR REPLACE FUNCTION perl_set()
RETURNS SETOF testrowperl AS $$
    return_next({ f1 => 1, f2 => 'Hello', f3 => 'World' });
    return_next({ f1 => 2, f2 => 'Hello', f3 => 'PostgreSQL' });
    return_next({ f1 => 3, f2 => 'Hello', f3 => 'PL/Perl' });
    return undef;
$$ LANGUAGE plperl;
```

小規模な結果セットでは、それぞれ単純な型、配列型、複合型に対応する、スカラ、配列への参照、ハッシュへの参照を含む配列への参照を返すことができます。以下に、配列への参照として結果セット全体を返す単純な例をいくつか示します。

```
CREATE OR REPLACE FUNCTION perl_set_int(int) RETURNS SETOF INTEGER AS $$
    return [0..$_[0]];
$$ LANGUAGE plperl;

SELECT * FROM perl_set_int(5);

CREATE OR REPLACE FUNCTION perl_set() RETURNS SETOF testrowperl AS $$
    return [
        { f1 => 1, f2 => 'Hello', f3 => 'World' },
        { f1 => 2, f2 => 'Hello', f3 => 'PostgreSQL' },
        { f1 => 3, f2 => 'Hello', f3 => 'PL/Perl' }
    ];
$$ LANGUAGE plperl;

SELECT * FROM perl_set();
```

コード内でstrictプラグマを使用したいのであればいくつか選択肢があります。一時的に大域的に使用するために、SET plperl.use_strictを真にすることができます。このパラメータは、その後のPL/Perl関数のコンパイルに影響しますが、現在のセッションでコンパイル済みの関数には影響しません。永続的に大域的に使用するためには、postgresql.confファイル内でplperl.use_strictを真に設定します。

特定の関数で永続的に使用するためには単純に以下を関数本体の先頭に記載してください。

```
use strict;
```

また、Perlのバージョンが5.10.0以上であればuseでfeatureプラグマが利用可能です。

44.2. PL/Perlにおけるデータ値

PL/Perl関数のコードに渡される引数値は、単に(SELECT文で表示される場合と同様の)テキスト形式に変換された入力引数です。反対にreturnおよびreturn_nextコマンドは、関数の宣言された戻り値の型で受け付け可能な入力書式で表された任意の文字列を受け付けます。

この動作が特定の場合には不都合であるなら、前にbool値の例で説明したように、変換を使って改善できます。変換モジュールの例がいくつかPostgreSQLの配布物に含まれています。

44.3. 組み込み関数

44.3.1. PL/Perlからのデータベースアクセス

Perl関数からデータベースそのものにアクセスするには以下の関数で行います。

spi_exec_query(query [, max-rows])

spi_exec_queryはSQLコマンドを実行し、行セット全体をハッシュへの参照を要素とする配列への参照として返します。結果が相対的に小規模であることが分かっている場合にのみこのコマンドを使用してください。以下に最大行数オプションを持った問い合わせ（SELECTコマンド）の例を示します。

```
$rv = spi_exec_query('SELECT * FROM my_table', 5);
```

これはmy_tableテーブルから5行までを返します。my_tableにmy_column列がある場合、結果の第$i行の列値を以下のように取り出すことができます。

```
$foo = $rv->{rows}[$i]->{my_column};
```

SELECT問い合わせから返される行の総数は以下のようにアクセスできます。

```
$nrows = $rv->{processed}
```

以下は他の種類のコマンドを使用する例です。

```
$query = "INSERT INTO my_table VALUES (1, 'test')";
$rv = spi_exec_query($query);
```

この後、以下のようにコマンドステータス（例えばSPI_OK_INSERT）にアクセスすることができます。

```
$res = $rv->{status};
```

影響を受けた行数を取り出すには以下を行います。

```
$nrows = $rv->{processed};
```

以下に複雑な例を示します。

```
CREATE TABLE test (
    i int,
    v varchar
);

INSERT INTO test (i, v) VALUES (1, 'first line');
INSERT INTO test (i, v) VALUES (2, 'second line');
INSERT INTO test (i, v) VALUES (3, 'third line');
INSERT INTO test (i, v) VALUES (4, 'immortal');

CREATE OR REPLACE FUNCTION test_munge() RETURNS SETOF test AS $$
    my $rv = spi_exec_query('select i, v from test;');
    my $status = $rv->{status};
    my $nrows = $rv->{processed};
    foreach my $rn (0 .. $nrows - 1) {
```

```
        my $row = $rv->{rows}[$rn];
        $row->{i} += 200 if defined($row->{i});
        $row->{v} =~ tr/A-Za-z/a-zA-Z/ if (defined($row->{v}));
        return_next($row);
    }
    return undef;
$$ LANGUAGE plperl;

SELECT * FROM test_munge();
```

spi_query(command)
spi_fetchrow(cursor)
spi_cursor_close(cursor)

spi_queryおよびspi_fetchrowは、大規模になる可能性がある行セット用、または、行を順番通りに返したい場合向けに組み合わせて動作します。spi_fetchrowはspi_queryと一緒でなければ動作しません。組み合わせて使用する方法について、以下の例で示します。

```
CREATE TYPE foo_type AS (the_num INTEGER, the_text TEXT);

CREATE OR REPLACE FUNCTION lotsa_md5 (INTEGER) RETURNS SETOF foo_type AS $$
    use Digest::MD5 qw(md5_hex);
    my $file = '/usr/share/dict/words';
    my $t = localtime;
    elog(NOTICE, "opening file $file at $t" );
    open my $fh, '<', $file # ooh, it's a file access!
        or elog(ERROR, "cannot open $file for reading: $!");
    my @words = <$fh>;
    close $fh;
    $t = localtime;
    elog(NOTICE, "closed file $file at $t");
    chomp(@words);
    my $row;
    my $sth = spi_query("SELECT * FROM generate_series(1,$_[0]) AS b(a)");
    while (defined ($row = spi_fetchrow($sth))) {
        return_next({
            the_num => $row->{a},
            the_text => md5_hex($words[rand @words])
        });
    }
    return;
$$ LANGUAGE plperlu;

SELECT * from lotsa_md5(500);
```

通常spi_fetchrowは、読み取る行がなくなったことを示すundefが返されるまで繰り返されるはずです。spi_fetchrowがundefを返すとspi_queryで返されるカーソルは自動的に解放されます。すべての行を

読み取りたくない場合は代わりにspi_cursor_closeを呼び出してカーソルを解放してください。これに失敗するとメモリリークという結果になります。

```
spi_prepare(command, argument types)
spi_query_prepared(plan, arguments)
spi_exec_prepared(plan [, attributes], arguments)
spi_freeplan(plan)
```

spi_prepare、spi_query_prepared、spi_exec_prepared、spi_freeplanは、プリペアド問い合わせ用に同様の機能を実装します。spi_prepareは番号付き引数プレースホルダ($1、$2など)を持つ問い合わせ文字列と引数の型を表す文字列リストを受け付けます。

```
$plan = spi_prepare('SELECT * FROM test WHERE id > $1 AND name = $2',
                                                    'INTEGER', 'TEXT');
```

spi_prepareを呼び出すことで問い合わせ計画が準備されると、spi_exec_queryにより返されるものと同様の結果となるspi_exec_preparedやspi_queryとまったく同じカーソルが返されるspi_query_prepared(このカーソルは後でspi_fetchrowに渡すことができます)の中で、その計画を問い合わせ文字列の代わりに使用することができます。spi_exec_preparedの省略可能な第二パラメータは属性のハッシュ参照です。現在サポートされる唯一の属性は、問い合わせで返される最大行数を設定するlimitです。

プリペアド問い合わせの利点は、1つの準備された計画を複数回使用して問い合わせを実行することができるという点です。計画が不要になった後、spi_freeplanを使用して、計画を解放することができます。

```
CREATE OR REPLACE FUNCTION init() RETURNS VOID AS $$
        $_SHARED{my_plan} = spi_prepare('SELECT (now() + $1)::date AS now',
                                        'INTERVAL');
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION add_time( INTERVAL ) RETURNS TEXT AS $$
        return spi_exec_prepared(
                $_SHARED{my_plan},
                $_[0]
        )->{rows}->[0]->{now};
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION done() RETURNS VOID AS $$
        spi_freeplan( $_SHARED{my_plan});
        undef $_SHARED{my_plan};
$$ LANGUAGE plperl;

SELECT init();
SELECT add_time('1 day'), add_time('2 days'), add_time('3 days');
SELECT done();
```

```
  add_time  |  add_time  |  add_time
------------+------------+------------
 2005-12-10 | 2005-12-11 | 2005-12-12
```

spi_prepare内のパラメータ添字が$1、$2、$3などを介して定義されることに注意してください。そのため、検出困難な不具合が簡単に発生することになる二重引用符内での問い合わせ文字列宣言はやめてください。

他の例は、spi_exec_preparedにおける省略可能なパラメータの使用について示しています。

```
CREATE TABLE hosts AS SELECT id, ('192.168.1.'||id)::inet AS address
                      FROM generate_series(1,3) AS id;

CREATE OR REPLACE FUNCTION init_hosts_query() RETURNS VOID AS $$
        $_SHARED{plan} = spi_prepare('SELECT * FROM hosts
                                      WHERE address << $1', 'inet');
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION query_hosts(inet) RETURNS SETOF hosts AS $$
        return spi_exec_prepared(
                $_SHARED{plan},
                {limit => 2},
                $_[0]
        )->{rows};
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION release_hosts_query() RETURNS VOID AS $$
        spi_freeplan($_SHARED{plan});
        undef $_SHARED{plan};
$$ LANGUAGE plperl;

SELECT init_hosts_query();
SELECT query_hosts('192.168.1.0/30');
SELECT release_hosts_query();

    query_hosts
-----------------
 (1,192.168.1.1)
 (2,192.168.1.2)
(2 rows)
```

spi_commit()
spi_rollback()

現在のトランザクションをコミットあるいはロールバックします。これはプロシージャ、あるいはトップレベルから呼ばれた無名コードブロック（DOコマンド）の中からのみ呼び出すことができます。（SQLコマンドのCOMMITやROLLBACKをspi_exec_queryなどを通して実行することはできない点に注意してください。前

述の関数を使って行う必要があります。) トランザクションが終了した後、新たなトランザクションが自動的に開始されますので、開始するための別途の関数はありません。

以下に例を示します。

```
CREATE PROCEDURE transaction_test1()
LANGUAGE plperl
AS $$
foreach my $i (0..9) {
    spi_exec_query("INSERT INTO test1 (a) VALUES ($i)");
    if ($i % 2 == 0) {
        spi_commit();
    } else {
        spi_rollback();
    }
}
$$;

CALL transaction_test1();
```

44.3.2. PL/Perlのユーティリティ関数

elog(level, msg)

ログまたはエラーメッセージを発行します。使用できるレベルは、DEBUG、LOG、INFO、NOTICE、WARNING、およびERRORです。ERRORはエラー状態を発生します。その上位のPerlコードでこのエラーを捕捉しない場合、エラーは問い合わせの呼び出し元まで伝播し、その結果、現在のトランザクションもしくはサブトランザクションはアボートします。これは実質Perlのdieコマンドと同じです。他のレベルは、異なる重要度のメッセージを生成するだけです。log_min_messagesとclient_min_messages設定パラメータは、特定の重要度のメッセージをクライアントに報告するか、サーバのログに書き出すか、あるいはその両方かを制御します。詳細は第19章を参照してください。

quote_literal(string)

与えられた文字列を、SQL文の文字列内で文字列リテラルとして使用するために適切に引用符付けして返します。埋め込まれた単一引用符およびバックスラッシュは適切に二重化されます。quote_literalは入力がundefならばundefを返すことに注意してください。引数がundefの可能性があるのであれば、quote_nullableの方が適しています。

quote_nullable(string)

与えられた文字列を、SQL文の文字列内で文字列リテラルとして使用するために適切に引用符付けして返します。引数がundefの場合引用符付けされない文字列"NULL"を返します。埋め込まれた単一引用符およびバックスラッシュは適切に二重化されます。

quote_ident(string)

与えられた文字列を、SQL文の文字列内で識別子として使用するために適切に引用符付けして返します。必要な場合(つまり文字列に識別子用ではない文字列が含まれる、または、大文字小文字を保持する場合)のみ引用符が付けられます。埋め込まれた引用符は適切に二重化されます。

decode_bytea(string)

与えられた文字列の内容を表す、エスケープのないバイナリデータを返します。これはbytea符号化でなければなりません。

encode_bytea(string)

与えられた文字列の内容をバイナリデータ形式で符号化したbyteaを返します。

encode_array_literal(array)
encode_array_literal(array, delimiter)

参照先の配列の内容を、配列リテラル書式で表した文字列として返します(8.15.2参照)。配列への参照でない場合は引数の値は変更されません。配列リテラルの要素間の区切り文字は指定がない、または、undefの場合、デフォルトで", "です。

encode_typed_literal(value, typename)

Perl変数を2番目の引数として渡されたデータ型の値に変換し、その値の文字列表現を返します。入れ子状の配列や複合型の値を正しく扱います。

encode_array_constructor(array)

参照先の配列の内容を配列生成書式で表した文字列として返します(4.2.12参照)。個々の値はquote_nullableを使用して引用符付けされます。配列への参照でない場合は、quote_nullableを使用して引用符付けされた引数の値が返されます。

looks_like_number(string)

与えられた文字列の内容がPerlの流儀で数値でありそうな場合に真値を、さもなくば偽を返します。引数がundefならばundefを返します。先頭の空白、末尾の空白は無視されます。InfおよびInfinityは数値とみなします。

is_array_ref(argument)

指定された引数が配列参照として扱うことができる場合、つまり、引数のrefがARRAYまたはPostgreSQL::InServer::ARRAYの場合、真を返します。さもなくば偽を返します。

44.4. PL/Perlにおけるグローバルな値

現在のセッションの有効期間中の関数呼び出し間でデータ(コード参照を含む)を受け渡しするためにグローバルな%_SHAREDハッシュを使用することができます。

データの共有について簡単な例を以下に示します。

```
CREATE OR REPLACE FUNCTION set_var(name text, val text) RETURNS text AS $$
    if ($_SHARED{$_[0]} = $_[1]) {
        return 'ok';
```

```
    } else {
        return "cannot set shared variable $_[0] to $_[1]";
    }
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION get_var(name text) RETURNS text AS $$
    return $_SHARED{$_[0]};
$$ LANGUAGE plperl;

SELECT set_var('sample', 'Hello, PL/Perl!  How''s tricks?');
SELECT get_var('sample');
```

以下は、コード参照を使用した、多少複雑な例です。

```
CREATE OR REPLACE FUNCTION myfuncs() RETURNS void AS $$
    $_SHARED{myquote} = sub {
        my $arg = shift;
        $arg =~ s/(['\\])/\\$1/g;
        return "'$arg'";
    };
$$ LANGUAGE plperl;


SELECT myfuncs(); /* 関数の初期化 */


/* 引用符関数を使用する関数を作成 */

CREATE OR REPLACE FUNCTION use_quote(TEXT) RETURNS text AS $$
    my $text_to_quote = shift;
    my $qfunc = $_SHARED{myquote};
    return &$qfunc($text_to_quote);
$$ LANGUAGE plperl;
```

（可読性を犠牲にすると、上は`return $_SHARED{myquote}->($_[0]);`という1行のみで置き換えることができます。）

セキュリティ上の理由により、PL/Perlは、あるロールで呼び出された関数をそのロール用に独立したPerlインタプリタ内で実行します。これにより、あるユーザの事故または悪意によって他のユーザのPL/Perl関数の動作が干渉されてしまうことを防ぎます。こうしたインタプリタはそれぞれ独自の%_SHAREDなどのグローバル状態を持ちます。したがって、同一のSQLロールによって実行された場合のみ、2つのPL/Perl関数は同じ%_SHARED値を共有します。1つのセッション内で複数のSQLロールの元でコードを（SECURITY DEFINER経由、SET ROLEの使用など）実行するアプリケーションでは、確実にPL/Perl関数が%_SHAREDを介してデータを共有することができるように、明示的な処理を行う必要があります。このためには、通信しなければならない関数が同じユーザによって所有されること、およびSECURITY DEFINERと印付けられていることを確実にしなければなりません。当然ながらこうした関数が意図していないことを行うために使用することができないように注意しなければなりません。

44.5. 信頼されたPL/Perlおよび信頼されないPL/Perl

通常、PL/Perlはplperlという名前で「信頼された」プログラミング言語としてインストールされます。この設定では、セキュリティを確保するためにPerlの特定の操作は無効にされます。一般的には、制限される操作は環境に作用するものです。これには、ファイルハンドル操作やrequire、use（外部モジュール用）が含まれます。C関数では可能ですが、Perlでは、データベースサーバ内部にアクセスする方法や、サーバプロセスの権限によるOSレベルのアクセスを行う方法はありません。この結果、データベースの全ての非特権ユーザはこの言語を使用することができます。

セキュリティ上の理由により許されていないファイルシステム操作を行うため、うまく動作しない関数の例を以下に示します。

```
CREATE FUNCTION badfunc() RETURNS integer AS $$
    my $tmpfile = "/tmp/badfile";
    open my $fh, '>', $tmpfile
        or elog(ERROR, qq{could not open the file "$tmpfile": $!});
    print $fh "Testing writing to a file\n";
    close $fh or elog(ERROR, qq{could not close the file "$tmpfile": $!});
    return 1;
$$ LANGUAGE plperl;
```

許されていない操作の使用が検証機能によって検出されますので、この関数の作成は失敗します。

制限のないPerl関数の作成が望ましい場合があります。例えば、Perl 関数を使用してメールを送信するような場合です。このような場合を扱うために、PL/Perlを「信頼されない」言語（通常PL/PerlUと呼ばれます）としてインストールすることもできます。この場合は完全なPerl言語を使用することができます。言語がインストールされた場合、plperluという言語名によって、信頼されないPL/Perlの亜種が選択されます。

PL/PerlU関数の作成者は、その関数を不必要なことに使用できないように注意する必要があります。この関数は、データベース管理者としてログインしたユーザが実行できることを全て実行できるからです。データベースシステムはデータベースのスーパーユーザにのみ信頼されない言語による関数作成を許可していることに注意してください。

上記の関数が、スーパーユーザによってplperlu言語を使用して作成された場合、実行は可能となります。

同じ方法で、言語をplperlではなくplperluと指定することで、Perl内に作成された匿名コードブロックは制限された操作を使用することができます。ただし呼び出し元はスーパーユーザでなければなりません。

> **注記**
>
> PL/Perl関数はSQLロール毎に別々のPerlインタプリタ内で実行されますが、あるセッションで実行されるPL/PerlU関数はすべて、単一のPerlインタプリタ（PL/Perl関数用に使用されるインタプリタのいずれかではありません）内で実行されます。これによりPL/PerlU関数はデータを自由に共有することができます。しかしPL/Perl関数とPL/PerlU関数の間で通信することはできません。

注記

Perlは適切なフラグ、すなわちusemultiplicityまたはuseithreadsを付けて構築していない限り、1つのプロセス内で複数のインタプリタをサポートすることはできません。(実際にスレッドの使用が必要でなければusemultiplicityを勧めます。詳細はperlembedマニュアルページを参照してください。) PL/Perlがこの方法で構築されていないPerlのコピーを使用する場合、1つのセッション内で1つのPerlインタプリタしか持つことができません。このため、1つのセッションでは、PL/PerlU関数、もしくは、すべて同一のSQLロールで呼び出されるPL/Perl関数のいずれかのみを実行することができます。

44.6. PL/Perlトリガ

PL/Perlを使用してトリガ関数を作成することができます。トリガ関数では、$_TDというハッシュへの参照に、現在のトリガイベントに関する情報が含まれています。$_TDは大域変数であり、各トリガ呼び出しに対して局所的な値を別々に取り出します。以下に$_TDというハッシュへの参照のフィールドを示します。

$_TD->{new}{foo}

NEWのfoo列値。

$_TD->{old}{foo}

OLDのfoo列値。

$_TD->{name}

呼び出されたトリガの名前。

$_TD->{event}

トリガイベント。INSERT、UPDATE、DELETE、TRUNCATE、もしくはUNKNOWN。

$_TD->{when}

トリガがいつ呼び出されたか。BEFORE、AFTER、INSTEAD OFもしくはUNKNOWN。

$_TD->{level}

トリガレベル。ROW、STATEMENT、もしくはUNKNOWN。

$_TD->{relid}

トリガの発行元テーブルのOID。

$_TD->{table_name}

トリガの発行元テーブルの名前。

$_TD->{relname}

トリガの発行元テーブルの名前。これは廃止予定で、将来のリリースで削除される可能性があります。代わりに$_TD->{table_name}を使用してください。

$_TD->{table_schema}

トリガの発行元テーブルが存在するスキーマの名前。

$_TD->{argc}

トリガ関数の引数の数。

@{$_TD->{args}}

トリガ関数の引数。$_TD->{argc}が0の場合は存在しません。

行レベルトリガは以下のいずれかを返すことができます。

return;

操作を実行します。

"SKIP"

操作を実行しません。

"MODIFY"

トリガ関数によってNEW行が変更されたことを示します。

以下はトリガ関数の例で、ここまでの説明の一部を例証するものです。

```
CREATE TABLE test (
    i int,
    v varchar
);

CREATE OR REPLACE FUNCTION valid_id() RETURNS trigger AS $$
    if (($_TD->{new}{i} >= 100) || ($_TD->{new}{i} <= 0)) {

        return "SKIP";    # INSERT/UPDATEコマンドを取消します。
    } elsif ($_TD->{new}{v} ne "immortal") {
        $_TD->{new}{v} .= "(modified by trigger)";

        return "MODIFY";  # 行を変更し、
      INSERT/UPDATEコマンドを実行します。
    } else {

        return;           # INSERT/UPDATEコマンドを実行します。
    }
$$ LANGUAGE plperl;

CREATE TRIGGER test_valid_id_trig
    BEFORE INSERT OR UPDATE ON test
    FOR EACH ROW EXECUTE FUNCTION valid_id();
```

44.7. PL/Perlイベントトリガ

PL/Perlを使用してイベントトリガ関数を作成することができます。イベントトリガ関数では、$_TDというハッシュへの参照に、現在のトリガイベントに関する情報が含まれています。$_TDはグローバル変数であり、各トリガ呼び出しに対してローカルな値を別々に取り出します。以下に$_TDというハッシュへの参照のフィールドを示します。

$_TD->{event}

イベントトリガ名が発行された

$_TD->{tag}

トリガの発行元コマンドタグ

トリガ関数の戻り値は無視されます

以下はトリガ関数の例で、ここまでの説明の一部を例証するものです。

```
CREATE OR REPLACE FUNCTION perlsnitch() RETURNS event_trigger AS $$
  elog(NOTICE, "perlsnitch: " . $_TD->{event} . " " . $_TD->{tag} . " ");
$$ LANGUAGE plperl;

CREATE EVENT TRIGGER perl_a_snitch
    ON ddl_command_start
    EXECUTE FUNCTION perlsnitch();
```

44.8. PL/Perlの内部

44.8.1. 設定

本節ではPL/Perlに影響する設定パラメータを列挙します。

plperl.on_init (string)

Perlインタプリタが最初に初期化され、plperlまたはplperluでの使用のための準備がなされる前に実行されるperlコードを指定します。このコードが実行される時にはSPI関数を利用できません。このコードがエラーで失敗した場合、インタプリタの初期化は中断され、呼び出し元の問い合わせに伝わり、現在のトランザクションまたはサブトランザクションがアボートすることになります。

このPerlコードは単一文字列に制限されます。長いコードをモジュール化し、on_init文字列でロードすることができます。以下に例を示します。

```
plperl.on_init = 'require "plperlinit.pl"'
plperl.on_init = 'use lib "/my/app"; use MyApp::PgInit;'
```

plperl.on_initにより直接または間接的に読み込まれるモジュールはすべて、plperlにより使用可能になります。これはセキュリティの危険性が発生する可能性があります。どんなモジュールが読み込まれたかを確認するためには以下を使用します。

```
DO 'elog(WARNING, join ", ", sort keys %INC)' LANGUAGE plperl;
```

plperlライブラリがshared_preload_librariesに含まれている場合、初期化はpostmaster内部で起こります。この場合、postmasterが不安定になる危険が出てくるため、一層の考慮が必要です。この機能を使用できるようにした大きな理由は、plperl.on_initでロードされるPerlモジュールはpostmaster起動時点のみでロードされなければならないためです。このため個々のデータベースセッション内にロードというオーバーヘッドをもたらすことなく即座に利用できるようになります。しかし、データベースセッションで最初に使用されるPerlインタプリタ（PL/PerlUまたはPL/Perl関数を呼び出す最初のSQLロール用のPL/Perl）に対してのみ、このオーバーヘッドを防ぐことができる点に注意してください。データベースセッション内でその後に作成されるPerlインタプリタはすべて、新たにplperl.on_initを実行する必要があります。また、postmasterプロセス内で作成されるPerlインタプリタは子プロセスに伝播されませんので、Windowsにおける事前ロードには何かを節約することはまったくありません。

このパラメータはpostgresql.confファイルまたはサーバのコマンドラインでのみ設定可能です。

plperl.on_plperl_init (string)
plperl.on_plperlu_init (string)

これらのパラメータはそれぞれ、plperlまたはplperlu用にPerlインタプリタを特化する時に実行されるPerlコードを指定します。これは、データベースセッション内でPL/PerlまたはPL/PerlU関数が最初に実行される時、または、他の言語が呼び出されたため、あるいは新しいSQLロールでPL/Perl関数が呼び出されたために追加のインタプリタを呼び出す必要があった時に起こります。この後にplperl.on_initによる初期化が行われます。このコードを実行する時にはSPI関数は利用できません。plperl.on_plperl_init内のPerlコードはインタプリタを「権限で制限した」後に実行されます。このためPerlコードは信頼できる操作のみを行うことができます。

コードがエラーで失敗した場合、初期化は中断され、呼び出し元にエラーが伝わります。その結果現在のトランザクションまたはサブトランザクションはアボートします。Perl内ですでに行われた処理は取り消されません。言語が再度使用される時、初期化は新しいインタプリタの中で再度試行されます。

スーパーユーザのみがこれらの設定を変更することができます。これらの設定はセッション内で変更することができますが、このような変更は関数を実行するためにすでに使用されたPerlインタプリタには影響を与えません。

plperl.use_strict (boolean)

真の場合、その後のPL/Perl関数のコンパイルはstrictプラグマが有効になります。このパラメータは現在のセッションでコンパイル済みの関数には影響しません。

44.8.2. 制限および存在しない機能

現時点では、以下の機能はPL/Perlにありません。各機能の寄稿を歓迎します。

- PL/Perl関数は互いに直接呼び出すことができません。

- SPIはまだ完全に実装されていません。
- spi_exec_queryを使用して、非常に大規模なデータセットを取り出そうとする場合、これらがすべてメモリ内に保存されることに注意しなければなりません。上で示した通り、spi_query/spi_fetchrowを使用することで、これを避けることができます。

  集合を返す関数が大規模な行セットをreturnを介してPostgreSQLに返す場合、同様の問題が起こります。前述の通り、この問題もreturn_nextを使用して行毎に返すことで避けることができます。
- セッションが正常に終了した時、致命的なエラーによるものでなければ、定義された任意のENDブロックが実行されます。現在、その他の動作は行われません。特にファイルハンドルは自動的に吐き出されません。またオブジェクトも自動的に破棄されません。

第45章 PL/Python — Python手続き言語

PL/Python手続き言語を使用してPostgreSQLの関数をPython言語[1]で作成できます。

PL/Pythonを特定のデータベースにインストールするには、`CREATE EXTENSION plpythonu`を使用してください (ただし45.1も参照してください)。

> **ヒント**
>
> 言語を`template1`にインストールすると、その後に作成されるデータベース全てにその言語は自動的にインストールされます。

PL/Pythonは「信頼されない」、つまり、ユーザが実行可能なことを制限する方法を提供しない言語としてのみ利用可能です。したがって、`plpythonu`という名前に変更されました。Pythonで新しい安全な実行手法が開発されたら、将来信頼できる`plpython`の亜種は利用可能になるかもしれません。データベース管理者としてログインしたユーザにより行えることをすべて行うことができますので、信頼されないPL/Pythonによる関数開発者は、その関数は不必要なものを行うために使用できないことに注意しなければなりません。スーパーユーザのみが`plpythonu`などの信頼されない言語で関数を作成することができます。

> **注記**
>
> ソースパッケージを使用するユーザは、インストール処理の過程でPL/Pythonの構築が有効になるように指定する必要があります。(詳細については、インストール手順を参照してください。) バイナリパッケージを使用する場合は、別のサブパッケージにPL/Pythonが入っている可能性があります。

45.1. Python 2対Python 3

PL/PythonはPython 2およびPython 3言語の両方をサポートします。(PostgreSQLのインストール手順では、サポートするPythonの細かなマイナーバージョンに関して、より正確な情報が記載されています。) Python 2及びPython 3言語には重要な側面でいくつか互換性がありませんので、混在を防ぐためにPL/Pythonでは以下の命名ならびに移行計画が使用されています。

- `plpython2u`という名前のPostgreSQL言語はPython 2言語に基づいたPL/Pythonを実装します。
- `plpython3u`という名前のPostgreSQL言語はPython 3言語に基づいたPL/Pythonを実装します。
- `plpythonu`という名前の言語はデフォルトのPython言語(現時点ではPython 2)に基づいたPL/Pythonを実装します。(このデフォルトはどのローカルにインストールされたPythonがその「デフォルト」とみなされるか、例えば`/usr/bin/python`が何を示すか、とは独立しています。) Pythonコミュニティにおける Python 3への移行の進行状況に依存しますが、おそらく近い将来のPostgreSQLのデフォルトはPython 3に変わります。

[1] https://www.python.org

この計画は、PEP 394[2]内の、pythonコマンドの命名と移行に関する推奨に類似しています。

Python 2用のPL/PythonかPython 3用のPL/Python、またはその両方が利用できるかどうかは、構築時の設定またはインストールしたパッケージに依存します。

> **ヒント**
>
> どのバージョンのPythonがインストール時に検出されるか、または、PYTHON環境変数を使用してバージョンを明示的に設定することで、構築される亜種が決まります。16.4を参照してください。1つのインストレーションで両方のPL/Pythonを利用可能にするためには、ソースツリーでconfigureと構築を2回行う必要があります。

この結果以下のような使用方法と移行戦略となります。

- 既存のユーザおよび現時点でPython 3に興味を持たないユーザはplpythonuという名前の言語を使用し、当分の間何も変更する必要はありません。最終的なPython 3への移行を簡単にするために、Python 2.6/2.7への移行を介してコードを少しずつ「将来に備える」ことを勧めます。

  実際には、多くのPL/Python関数はそのまま、またはわずかな変更を行うことでPython 3に移行されます。

- Python 2に大きく依存するコードがあることが分かっていて、変更する予定がないユーザはplpython2u言語名を使用することができます。これは、PostgreSQLでPython 2のサポートが完全になくなる、非常に先の将来まで動作し続けます。

- Python 3に挑戦したいユーザは、現在の標準では永久に動作し続けるplpython3u言語名を使用できます。遠い将来、Python 3がデフォルトになった時、美的な理由から「3」という文字はなくなることになるでしょう。

- Python 3のみのオペレーティングシステム環境を構築したい、恐れを知らぬユーザは、plpythonuの拡張制御ファイルとスクリプトファイルの内容を変更して、plpythonuがplpython3uを示すようにすることができます。ただし、世間一般と互換性がないインストレーションを作成していることを覚えておいてください。

また、Python 3への移植に関する情報についてはPython 3.0における新機能[3]文書を参照してください。

Python 2を基にしたPL/PythonとPython 3を基にしたPL/Pythonを同じセッションで使用することはできません。動的モジュール内のシンボルが相反するため、こうするとPostgreSQLサーバプロセスがクラッシュしてしまうためです。あるセッション内でPythonのメジャーバージョンが混在させないための検査があり、不一致が見つかるとセッションは中断されます。しかし別々のセッションからであれば、同じデータベースにおいて異なるPL/Pythonの両方を使用することができます。

45.2. PL/Python関数

PL/Pythonで作成された関数は標準的なCREATE FUNCTION構文で宣言されます。

```
CREATE FUNCTION funcname (argument-list)
```

[2] https://www.python.org/dev/peps/pep-0394/

[3] https://docs.python.org/3/whatsnew/3.0.html

```
  RETURNS return-type
AS $$
  # PL/Python function body
$$ LANGUAGE plpythonu;
```

関数本体は単なるPythonスクリプトです。関数が呼び出されると、引数はargs[]リストの要素として渡されます。名前付きの引数も通常の変数としてPythonスクリプトに渡されます。通常、名前付き引数の方が可読性が高くなります。結果は、Pythonコードから通常の方法、returnまたはyield（結果セット文の場合）で返されるものです。戻り値を提供しない場合、PythonはデフォルトのNoneを返します。PL/PythonはPythonのNoneをSQLのNULL値に変換します。プロシージャでは、Pythonコードからの結果はNoneでなければなりません（典型的にはreturn文を使わずプロシージャを終了したり、return文を引数無しで使うことで達成されます）。さもないとエラーが起きます。

たとえば、2つの整数の内大きな数を返す関数は以下のように定義することができます。

```
CREATE FUNCTION pymax (a integer, b integer)
  RETURNS integer
AS $$
  if a > b:
    return a
  return b
$$ LANGUAGE plpythonu;
```

関数定義の本体として提供されたPythonのコードはPythonの関数に変換されます。例えば上の例は以下のようになります。

```
def __plpython_procedure_pymax_23456():
  if a > b:
    return a
  return b
```

ここで、23456はPostgreSQLにより割り当てられたこの関数のOIDです。

引数はグローバル変数として設定されます。Pythonのスコープ規則のため、これは、ブロック内でグローバルとして再宣言されていない限り、関数内で引数変数に変数名自身を含む式の値として再代入できないという難解な結果をもたらします。例えば以下は動作しません。

```
CREATE FUNCTION pystrip(x text)
  RETURNS text
AS $$
  x = x.strip()  # error
  return x
$$ LANGUAGE plpythonu;
```

xへの代入は、xをブロック全体に対するローカル変数にしようとし、そして、代入の右辺のxがPL/Pythonの関数パラメータではなく、まだ割り当てられていないローカル変数xを参照するためです。global文を使用することで、動作するようになります。

```
CREATE FUNCTION pystrip(x text)
  RETURNS text
AS $$
  global x
  x = x.strip()  # ok now
  return x
$$ LANGUAGE plpythonu;
```

しかし、PL/Pythonのこうした詳細な実装に依存しないようにすることを勧めます。関数パラメータは読み取りのみとして扱うことを勧めます。

45.3. データ値

一般的にいって、PL/Pythonの目標はPostgreSQLとPythonの世界の間で「自然な」対応付けを提供することです。これは以下のようなデータの対応付けを形成します。

45.3.1. データ型の対応付け

PL/Python関数が呼ばれると、その引数は、以下のようにPostgreSQLの型から対応するPython型に変換されます。

- PostgreSQLのbooleanはPythonのboolに変換されます。
- PostgreSQLのsmallintおよびintはPythonのintに変換されます。PostgreSQLのbigintおよびoidは、Python 2ではlongに、Python 3ではintに変換されます。
- PostgreSQLのrealおよびdoubleはPythonのfloatに変換されます。
- PostgreSQLのnumericはPythonのDecimalに変換されます。この型は可能ならばcdecimalパッケージからインポートできます。可能でなければ、標準ライブラリのdecimal.Decimalが使用できます。cdecimalはdecimalより高速です。しかしPython 3.3から、cdecimalはdecimalという名前で標準ライブラリに統合されたので、もはや差異はありません。
- PostgreSQLのbyteaは、Python 2ではstrに、Python 3ではbytesに変換されます。Python 2では文字列は文字符号化方式を持たないバイト列として扱われるはずです。
- PostgreSQLの文字列型を含む、上記以外のデータ型はすべてPythonのstrに変換されます。Python 2ではこの文字列はPostgreSQLのサーバ符号化方式で表されます。Python 3ではすべての文字列と同様にUnicode文字列となります。
- スカラ型以外については後述します。

PL/Python関数が戻る時には、その戻り値は、以下のようにPostgreSQLの宣言された戻り値データ型に変換されます。

- PostgreSQLの戻り値の型がbooleanの場合、戻り値は*Python*の規約に従った真に対して評価されます。つまり、0や空文字列は偽です。'f'が真となることには注意してください。
- PostgreSQLの戻り値の型がbyteaの場合、戻り値は文字列(Python 2)またはbytes(Python 3)に、それぞれ対応するPythonのビルトインを使用して変換され、その結果がbyteaに変換されます。

- この他のPostgreSQLの戻り値型では、返される値はPythonのビルトインstrを使用して文字列に変換され、その結果がPostgreSQLデータ型の入力関数に渡されます。(Pythonの値がfloatであれば、精度が失われるのを避けるため、strの代わりにreprビルトインを使って変換されます。)

  Python 2における文字列はPostgreSQLに渡される時にPostgreSQLサーバの符号化方式でなければなりません。現在のサーバ符号化方式で無効な文字列はエラーとなりますが、符号化方式の不一致がすべて検知されることはありません。このため正確に行われなかった場合にはゴミデータという結果になります。Unicode文字列は自動的に正しい符号化方式に変換されます。このためこれを使用することがより安全でより便利です。Python 3における文字列はすべてUnicode文字列です。

- スカラデータ型以外については後述します。

宣言されたPostgreSQLの戻り値型と実際に返されるオブジェクトのPythonデータ型との間の論理的な不整合が伝わらないことに注意してください。値はいかなる場合でも変換されます。

45.3.2. NullとNone

SQLのNULL値が関数に渡されると、その引数値はPythonではNoneとなります。例えば、45.2に示されたpymax関数の定義では、NULL入力に対して間違った結果が返されます。関数定義にSTRICTを付与してPostgreSQLを、NULL値が渡された場合にその関数を呼び出さず、自動的に単にNULL結果を返すという、より理想的に動作させることができます。他に、関数本体でNULL入力を検査することもできます。

```
CREATE FUNCTION pymax (a integer, b integer)
  RETURNS integer
AS $$
  if (a is None) or (b is None):
    return None
  if a > b:
    return a
  return b
$$ LANGUAGE plpythonu;
```

上で示したように、PL/Python関数からSQL NULL値を返すには、Noneという値を返してください。関数を厳密とした場合でも厳密としない場合でも、これを行うことができます。

45.3.3. 配列、リスト

SQL配列値はPythonのリストとしてPL/Pythonに渡されます。PL/Python関数の外部にSQL配列値を返すためには、Pythonのリストを返します。

```
CREATE FUNCTION return_arr()
  RETURNS int[]
AS $$
return [1, 2, 3, 4, 5]
$$ LANGUAGE plpythonu;

SELECT return_arr();
 return_arr
```

```
-------------
 {1,2,3,4,5}
(1 row)
```

多次元配列はPL/Pythonに入れ子のPythonのリストとして渡されます。例えば、2次元配列はリストのリストです。PL/Pythonから多次元のSQLの配列を返す場合には、各レベルの内側のリストはすべて同じ大きさでなければなりません。例えば、

```
CREATE FUNCTION test_type_conversion_array_int4(x int4[]) RETURNS int4[] AS $$
plpy.info(x, type(x))
return x
$$ LANGUAGE plpythonu;

SELECT * FROM test_type_conversion_array_int4(ARRAY[[1,2,3],[4,5,6]]);
INFO:  ([[1, 2, 3], [4, 5, 6]], <type 'list'>)
 test_type_conversion_array_int4
---------------------------------
 {{1,2,3},{4,5,6}}
(1 row)
```

タプル等のその他のPythonのシーケンスも、PostgreSQLバージョン9.6以下との後方互換性のために受け入れられます。当時は、多次元配列はサポートされていませんでした。しかしながら、複合型と区別できないため、常に1次元配列として扱われます。同じ理由で、複合型を多次元配列内で使う場合、リストではなくタプルとして表現しなければなりません。

Pythonでは、文字列はシーケンスであることに注意してください。これは予想できない影響を与えることがありますが、Pythonプログラマには慣れたものでしょう。

```
CREATE FUNCTION return_str_arr()
  RETURNS varchar[]
AS $$
return "hello"
$$ LANGUAGE plpythonu;

SELECT return_str_arr();
 return_str_arr
----------------
 {h,e,l,l,o}
(1 row)
```

45.3.4. 複合型

複合型の引数はPythonのマップとして渡されます。マップの要素名は複合型の属性名です。渡された行の属性値がNULLの場合、マップ上ではNoneという値となります。以下に例を示します。

```
CREATE TABLE employee (
  name text,
```

```
  salary integer,
  age integer
);

CREATE FUNCTION overpaid (e employee)
  RETURNS boolean
AS $$
  if e["salary"] > 200000:
    return True
  if (e["age"] < 30) and (e["salary"] > 100000):
    return True
  return False
$$ LANGUAGE plpythonu;
```

Python関数から行または複合型を返す方法は複数存在します。以下の例では

```
CREATE TYPE named_value AS (
  name   text,
  value  integer
);
```

を前提とします。複合型の結果は以下のように返されます。

シーケンス型（タプルまたはリスト。ただしインデックス付けができないためset は不可）

返されるシーケンスオブジェクトは、結果の複合型が持つフィールドと同じ項目数をもたなければなりません。0というインデックスの項目が複合型の最初のフィールド、1が次のフィールド、などとなります。以下に例を示します。

```
CREATE FUNCTION make_pair (name text, value integer)
  RETURNS named_value
AS $$
  return ( name, value )

  # もしくは、タプルとして返すなら: return [ name, value ]
$$ LANGUAGE plpythonu;
```

任意の列でSQL NULL値を返すには、対応する位置にNoneを挿入します。

複合型の配列を返す場合、Pythonのリストが複合型を表しているのか、また別の配列の次元を表しているのかあいまいですので、リストとして返すことはできません。

マップ（辞書）

結果型の列の値は、列名をキーとして持つマップから取り出されます。以下に例を示します。

```
CREATE FUNCTION make_pair (name text, value integer)
  RETURNS named_value
AS $$
```

```
  return { "name": name, "value": value }
$$ LANGUAGE plpythonu;
```

余計な辞書のキーと値の組み合わせは無視されます。存在しないキーはエラーとして扱われます。任意の列でSQL NULLを返すためには、対応する列名をキーとしてNoneを挿入してください。

オブジェクト（__getattr__メソッドを提供する任意のオブジェクト）

これはマップと同じように動作します。以下に例を示します。

```
CREATE FUNCTION make_pair (name text, value integer)
  RETURNS named_value
AS $$
  class named_value:
    def __init__ (self, n, v):
      self.name = n
      self.value = v
  return named_value(name, value)

  # or simply
  class nv: pass
  nv.name = name
  nv.value = value
  return nv
$$ LANGUAGE plpythonu;
```

OUTパラメータを用いる関数もサポートされています。以下に例を示します。

```
CREATE FUNCTION multiout_simple(OUT i integer, OUT j integer) AS $$
return (1, 2)
$$ LANGUAGE plpythonu;

SELECT * FROM multiout_simple();
```

プロシージャの出力パラメータは同様に戻されます。以下に例を示します。

```
CREATE PROCEDURE python_triple(INOUT a integer, INOUT b integer) AS $$
return (a * 3, b * 3)
$$ LANGUAGE plpythonu;

CALL python_triple(5, 10);
```

45.3.5. 集合を返す関数

また、PL/Python関数はスカラまたは複合型の集合を返すこともできます。返されるオブジェクトは内部的にイテレータに変換されるため、複数の実現方法があります。以下の例では、以下の複合型が存在することを仮定します。

```
CREATE TYPE greeting AS (
  how text,
  who text
);
```

集合という結果は以下から返されます。

シーケンス型（タプル、リスト、セット）

```
CREATE FUNCTION greet (how text)
  RETURNS SETOF greeting
AS $$
  # return tuple containing lists as composite types
  # all other combinations work also
  return ( [ how, "World" ], [ how, "PostgreSQL" ], [ how, "PL/Python" ] )
$$ LANGUAGE plpythonu;
```

イテレータ（__iter__メソッドとnextメソッドを提供する任意のオブジェクト）

```
CREATE FUNCTION greet (how text)
  RETURNS SETOF greeting
AS $$
  class producer:
    def __init__ (self, how, who):
      self.how = how
      self.who = who
      self.ndx = -1

    def __iter__ (self):
      return self

    def next (self):
      self.ndx += 1
      if self.ndx == len(self.who):
        raise StopIteration
      return ( self.how, self.who[self.ndx] )

  return producer(how, [ "World", "PostgreSQL", "PL/Python" ])
$$ LANGUAGE plpythonu;
```

ジェネレータ(yield)

```
CREATE FUNCTION greet (how text)
  RETURNS SETOF greeting
AS $$
  for who in [ "World", "PostgreSQL", "PL/Python" ]:
```

```
    yield ( how, who )
$$ LANGUAGE plpythonu;
```

(RETURNS SETOF recordを使用して)OUTパラメータを持つ集合を返す関数もサポートされます。以下に例を示します。

```
CREATE FUNCTION multiout_simple_setof(n integer, OUT integer, OUT integer) RETURNS SETOF record AS
 $$
return [(1, 2)] * n
$$ LANGUAGE plpythonu;

SELECT * FROM multiout_simple_setof(3);
```

45.4. データの共有

グローバルなSD辞書は、同じ関数に対する繰り返しの呼び出しの間でのプライベートなデータ保存のために使用することができます。グローバルなGD辞書は、共有データであり、セッション内の全てのPython関数で使用することができます。注意して使用してください。

各関数は、Pythonインタプリタ内で自身の実行環境を入手します。そのため、myfuncによるグローバルデータと関数の引数はmyfunc2から使用することはできません。上記で説明した通り、GD辞書内のデータは例外です。

45.5. 匿名コードブロック

PL/PythonはDO文で呼び出される匿名コードブロックもサポートします。

```
DO $$
    # PL/Python code
$$ LANGUAGE plpythonu;
```

匿名コードブロックは引数を持たず、また、何か値を返したとしても破棄されます。その他は関数とまったく同様に動作します。

45.6. トリガ関数

トリガとして関数を使用した場合、TD辞書にトリガに関連した値が格納されます。

TD["event"]

次のイベントが文字列として格納されます。INSERT、UPDATE、DELETE、TRUNCATE

TD["when"]

BEFORE、AFTER、またはINSTEAD OFのいずれかが格納されます。

TD["level"]

ROWまたはSTATEMENTが格納されます。

TD["new"]
TD["old"]

行レベルトリガにおいてトリガイベントに依存して、これらのフィールドの片方または両方に対応するトリガ行が格納されます。

TD["name"]

トリガ名が格納されます。

TD["table_name"]

トリガの発生元のテーブルの名前が格納されます。

TD["table_schema"]

トリガの発生元のテーブルのスキーマが格納されます。

TD["relid"]

トリガの発生元テーブルのOIDが格納されます。

TD["args"]

CREATE TRIGGERに引数が含まれていた場合、その引数はTD["args"][0]からTD["args"][n-1]までの範囲で使用することができます。

TD["when"]がBEFOREまたはINSTEAD OFで、かつ、TD["level"]がROWの場合、Pythonの関数から、行が変更されないことを示すNoneまたは"OK"、イベントを中断したことを示す"SKIP"を返すことができます。また、TD["event"]がINSERTまたはUPDATEの場合、行を変更したことを示す"MODIFY"を返すことができます。さもなければ、戻り値は無視されます。

45.7. データベースアクセス

PL/Python言語モジュールは自動的にplpyというPythonモジュールをインポートします。このモジュールの関数と定数は、plpy.fooのように作成したPythonコードから使用することができます。

45.7.1. データベースアクセス関数

plpyモジュールはデータベースコマンドを実行するために数個の関数を用意しています。

plpy.execute(query [, max-rows])

plpy.executeを、問い合わせ文字列および省略可能な行数制限引数を付けて呼び出すと、問い合わせが実行され、結果オブジェクトとして問い合わせ結果が返ります。

結果オブジェクトはリストもしくは辞書オブジェクトをエミュレートします。結果オブジェクトは、行番号や列名によってアクセスすることができます。例を示します。

```
rv = plpy.execute("SELECT * FROM my_table", 5)
```

これは、my_tableから5行までを返します。my_tableにmy_column列が存在する場合、その列には以下のようにアクセスできます。

```
foo = rv[i]["my_column"]
```

戻った行数はビルトインlen関数を使用して取得できます。

結果オブジェクトには以下のメソッドが追加されています。

nrows()

コマンドによる処理の行数を返します。戻った行数と同じとは限らないことに注意してください。例えば、UPDATEコマンドではゼロでない値を返しますが、行を戻すことはありません（RETURNINGを使用したときは別です）。

status()

SPI_execute()関数の戻り値を返します。

colnames()
coltypes()
coltypmods()

各々、列名のリスト、列の型OIDのリスト、列に関する型独自の型修飾子のリストを返します。

RETURNINGを持たないUPDATEやDROP TABLEなど、結果セットを生成しないコマンドによる結果オブジェクトに対して呼び出された場合、これらのメソッドは例外を発生します。しかし、ゼロ行の結果セットに対してこれらのメソッドを使用することには問題ありません。

__str__()

標準の__str__メソッドが定義されていますので、例えば問い合わせの実行結果をplpy.debug(rv)を使ってデバッグできます。

結果オブジェクトは変更できます。

plpy.executeを呼び出すことにより、結果セット全体がメモリ内に読み込まれることに注意してください。結果セットが比較的小さいことが確実な場合だけ、この関数を使用してください。大規模な結果を取り込む場合の過度のメモリ使用に関する危険を回避したい場合は、plpy.executeではなくplpy.cursorを使用してください。

plpy.prepare(query [, argtypes])
plpy.execute(plan [, arguments [, max-rows]])

plpy.prepareは問い合わせの実行計画を準備します。問い合わせ内にパラメータ参照がある場合、問い合わせ文字列および引数型のリストとともに呼び出されます。例を示します。

```
plan = plpy.prepare("SELECT last_name FROM my_users WHERE first_name = $1", ["text"])
```

textは$1として渡される変数の型です。問い合わせにパラメータを渡さない場合、2番目の引数は省略可能です。

文を準備した後、それを実行するために関数plpy.executeの亜種を使用します。

```
rv = plpy.execute(plan, ["name"], 5)
```

実行計画を（問い合わせ文字列ではなく）最初の引数として渡してください。問い合わせに代入する値のリストを、2番目の引数として渡してください。問い合わせにパラメータがない場合、2番目の引数は省略可能です。3番目の引数は、前に述べた省略可能な行数制限引数です。

代わりに、計画オブジェクトのexecuteメソッドを呼び出すことができます。

```
rv = plan.execute(["name"], 5)
```

問い合わせパラメータおよび結果行のフィールドは45.3で示した通り、PostgreSQLとPythonのデータ型の間で変換されます。

PL/Pythonモジュールを使用して準備した計画は自動的に保存されます。これが何を意味するのかについてはSPIの文書（第46章）を参照してください。これを複数呼び出しにおいて効果的に使用するためには、永続的な格納用辞書であるSDまたはGD（45.4を参照）のいずれかを使用する必要があります。例を示します。

```
CREATE FUNCTION usesavedplan() RETURNS trigger AS $$
    if "plan" in SD:
        plan = SD["plan"]
    else:
        plan = plpy.prepare("SELECT 1")
        SD["plan"] = plan
    # rest of function
$$ LANGUAGE plpythonu;
```

`plpy.cursor(query)`
`plpy.cursor(plan [, arguments])`

plpy.cursor関数はplpy.executeと同じ引数を受け取り（行数制限引数を除いた）カーソルオブジェクトとして返します。これにより大規模な結果セットをより小さな塊の中で処理することができます。plpy.executeの場合と同様、問い合わせ文字列または引数リスト付きの計画オブジェクトを使用できますし、計画オブジェクトのメソッドとしてcursor関数を呼ぶことができます。

カーソルオブジェクトは、整数パラメータを受付け、結果オブジェクトを返すfetchメソッドを提供します。fetchを呼び出す度に、返されるオブジェクトには次の一群の行が含まれます。この行数はパラメータ値より多くなることはありません。全ての行が出し尽くされると、fetchは空の結果オブジェクトを返すようになります。カーソルオブジェクトはまた、すべての行を出し尽くすまで一度に1行を生成するイテレータインタフェース[4]を提供します。この方法で取り出されたデータは結果オブジェクトとしては返されず、1つの辞書が単一の結果行に対応する辞書群として返されます。

大きなテーブルのデータを処理する、2つの方法の例を示します。

```
CREATE FUNCTION count_odd_iterator() RETURNS integer AS $$
```

[4] https://docs.python.org/library/stdtypes.html#iterator-types

```
odd = 0
for row in plpy.cursor("select num from largetable"):
    if row['num'] % 2:
         odd += 1
return odd
$$ LANGUAGE plpythonu;

CREATE FUNCTION count_odd_fetch(batch_size integer) RETURNS integer AS $$
odd = 0
cursor = plpy.cursor("select num from largetable")
while True:
    rows = cursor.fetch(batch_size)
    if not rows:
        break
    for row in rows:
        if row['num'] % 2:
            odd += 1
return odd
$$ LANGUAGE plpythonu;

CREATE FUNCTION count_odd_prepared() RETURNS integer AS $$
odd = 0
plan = plpy.prepare("select num from largetable where num % $1 <> 0", ["integer"])

rows = list(plpy.cursor(plan, [2]))  # または = list(plan.cursor([2]))

return len(rows)
$$ LANGUAGE plpythonu;
```

カーソルは自動的に処分されます。しかし、カーソルが保有していた資源を明示的に解放したい場合は、closeメソッドを使用してください。閉じた後、カーソルからこれ以上取り込むことはできません。

> **ヒント**
>
> plpy.cursorによって作成されたオブジェクトと、Pythonデータベース API仕様[5]において定義されたDB-APIカーソルとを混同しないでください。名称以外の共通点はありません。

45.7.2. エラーの捕捉

データベースにアクセスする関数はエラーに遭遇し、エラーが関数をアボートして例外を発生させる原因となります。plpy.executeおよびplpy.prepareは、デフォルトでは関数を終了させるplpy.SPIErrorのサブクラスのインスタンスを発生させることができます。このエラーは、try/except構文を使用して、Pythonの他の例外と同様に処理できます。例を示します。

[5] https://www.python.org/dev/peps/pep-0249/

```
CREATE FUNCTION try_adding_joe() RETURNS text AS $$
    try:
        plpy.execute("INSERT INTO users(username) VALUES ('joe')")
    except plpy.SPIError:

        "うまくいかなかった" を返す
    else:

        "Joeが追加された" を返す
$$ LANGUAGE plpythonu;
```

発生される例外の実クラスはエラーを引き起こした特定の条件と対応します。表 A.1にあり得る条件のリストがありますので参照してください。plpy.spiexceptionsモジュールはPostgreSQLの条件それぞれに対して、その条件名に因んだ名前の例外クラスを定義しています。例えばdivision_by_zeroはDivisionByZero、unique_violationはUniqueViolationに、fdw_errorはFdwErrorなどのようになります。これらの例外クラスはそれぞれSPIErrorを継承したものです。このように分離することで特定のエラーをより簡単に扱うことができるようになります。以下に例を示します。

```
CREATE FUNCTION insert_fraction(numerator int, denominator int) RETURNS text AS $$
from plpy import spiexceptions
try:
    plan = plpy.prepare("INSERT INTO fractions (frac) VALUES ($1 / $2)", ["int", "int"])
    plpy.execute(plan, [numerator, denominator])
except spiexceptions.DivisionByZero:
    return "denominator cannot equal zero"
except spiexceptions.UniqueViolation:
    return "already have that fraction"
except plpy.SPIError as e:
    return "other error, SQLSTATE %s" % e.sqlstate
else:
    return "fraction inserted"
$$ LANGUAGE plpythonu;
```

plpy.spiexceptionsモジュールからの全ての例外はSPIErrorを継承するため、例外を処理するexcept句は全てのデータベースアクセスエラーを捕捉することに注意してください。

異なったエラー条件を処理する代りの方法として、SPIError例外を捕捉して、例外オブジェクトのsqlstate属性を調べることにより、exceptブロック内部の明細なエラー条件を決定できます。この属性は「SQLSTATE」エラーコードを含む文字列値です。この方法は、ほぼ同じ機能を提供します。

45.8. 明示的サブトランザクション

45.7.2で説明したデータベースアクセスによって引き起こるエラーからの復旧は、操作の中の1つが失敗する前に、一部の操作が成功し、エラーからの復旧の後一貫性のないデータが残ってしまうという望ましくない

状態を導く可能性があります。PL/Pythonは明示的サブトランザクションにより、この問題の解法を提供します。

45.8.1. サブトランザクションのコンテキスト管理

2つの口座の間の振替えを実装する関数を考えてみます。

```
CREATE FUNCTION transfer_funds() RETURNS void AS $$
try:
    plpy.execute("UPDATE accounts SET balance = balance - 100 WHERE account_name = 'joe'")
    plpy.execute("UPDATE accounts SET balance = balance + 100 WHERE account_name = 'mary'")
except plpy.SPIError as e:
    result = "error transferring funds: %s" % e.args
else:
    result = "funds transferred correctly"
plan = plpy.prepare("INSERT INTO operations (result) VALUES ($1)", ["text"])
plpy.execute(plan, [result])
$$ LANGUAGE plpythonu;
```

2番目のUPDATE文が例外を発生させる結果となった場合、この関数はエラーを記録しますが、それにもかかわらず最初のUPDATEはコミットされます。言い換えると、資金はジョーの口座から引き落とされますが、メアリーの口座には移転しません。

こうした問題を防ぐために、plpy.execute呼び出しを明示的なサブトランザクションで囲むことができます。plpyモジュールは、plpy.subtransaction()関数で作成される明示的なサブトランザクションを管理するための補助オブジェクトを提供します。この関数によって作成されるオブジェクトはコンテキストマネージャインタフェース[6]を実装します 明示的なサブトランザクションを使用して、上の関数を以下のように書き換えることができます。

```
CREATE FUNCTION transfer_funds2() RETURNS void AS $$
try:
    with plpy.subtransaction():
        plpy.execute("UPDATE accounts SET balance = balance - 100 WHERE account_name = 'joe'")
        plpy.execute("UPDATE accounts SET balance = balance + 100 WHERE account_name = 'mary'")
except plpy.SPIError as e:
    result = "error transferring funds: %s" % e.args
else:
    result = "funds transferred correctly"
plan = plpy.prepare("INSERT INTO operations (result) VALUES ($1)", ["text"])
plpy.execute(plan, [result])
$$ LANGUAGE plpythonu;
```

try/catchの使用がまだ必要なことに注意してください。さもないと例外がPythonスタックの最上位まで伝播され、関数全体がPostgreSQLエラーにより中断され、この結果、operationsテーブルには挿入されるは

[6] https://docs.python.org/library/stdtypes.html#context-manager-types

ずの行が存在しないことになります。サブトランザクションのコンテキストマネージャはエラーを捕捉しません。これはそのスコープの内側で実行されるデータベース操作すべてが、原子的にコミットされるかロールバックされるかだけを保証します。サブトランザクションブロックのロールバックは、データベースアクセスを元にしたエラーによって引き起こる例外だけではなく、何らかの種類の例外終了でも起こります。明示的なサブトランザクションブロックの内側で発生した通常のPython例外も同様にサブトランザクションをロールバックさせます。

45.8.2. より古いPythonのバージョン

デフォルトでは、withキーワードを使用したコンテキストマネージャ構文はPython 2.6で利用可能です。これより古いバージョンのPythonとの互換性のために、サブトランザクションマネージャの__enter__および__exit__関数を、enterおよびexitという便利な別名を使用して、呼び出すことができます。資金の振替えを行う関数の例は以下のように記述できます。

```
CREATE FUNCTION transfer_funds_old() RETURNS void AS $$
try:
    subxact = plpy.subtransaction()
    subxact.enter()
    try:
        plpy.execute("UPDATE accounts SET balance = balance - 100 WHERE account_name = 'joe'")
        plpy.execute("UPDATE accounts SET balance = balance + 100 WHERE account_name = 'mary'")
    except:
        import sys
        subxact.exit(*sys.exc_info())
        raise
    else:
        subxact.exit(None, None, None)
except plpy.SPIError as e:
    result = "error transferring funds: %s" % e.args
else:
    result = "funds transferred correctly"

plan = plpy.prepare("INSERT INTO operations (result) VALUES ($1)", ["text"])
plpy.execute(plan, [result])
$$ LANGUAGE plpythonu;
```

45.9. トランザクション制御

トップレベル、またはトップレベルから呼ばれた無名コードブロック（DOコマンド）から呼ばれたプロシージャでは、トランザクションの制御が可能です。現在のトランザクションをコミットするには、plpy.commit()を呼びます。現在のロールバックするには、plpy.rollback()を呼びます。（SQLコマンドのCOMMITやROLLBACKをplpy.executeなどを通して実行することはできない点に注意してください。前述の関数を使って行う必要があります。）トランザクションが終了した後は新たなトランザクションが自動的に開始されますので、開始のための別の関数はありません。

以下に例を示します。

```
CREATE PROCEDURE transaction_test1()
LANGUAGE plpythonu
AS $$
for i in range(0, 10):
    plpy.execute("INSERT INTO test1 (a) VALUES (%d)" % i)
    if i % 2 == 0:
        plpy.commit()
    else:
        plpy.rollback()
$$;

CALL transaction_test1();
```

トランザクションは明示的なサブトランザクションの中では終了できません。

45.10. ユーティリティ関数

plpyモジュールでは以下の関数も提供しています。

```
plpy.debug(msg, **kwargs)
plpy.log(msg, **kwargs)
plpy.info(msg, **kwargs)
plpy.notice(msg, **kwargs)
plpy.warning(msg, **kwargs)
plpy.error(msg, **kwargs)
plpy.fatal(msg, **kwargs)
```

plpy.errorおよびplpy.fatalは、実際にPythonの例外を発生させます。これが捕捉されない場合、呼び出し中の問い合わせに伝わり、その結果、現在のトランザクションもしくはサブトランザクションがアボートします。raise plpy.Error(msg)およびraise plpy.Fatal(msg)は、それぞれplpy.errorおよびplpy.fatalの呼び出しと同じですが、raise形式ではキーワード引数を渡すことができません。他の関数は異なる重要度のメッセージを生成するだけです。log_min_messagesとclient_min_messages設定変数は、特定の重要度のメッセージをクライアントに報告するか、サーバのログに書き出すか、あるいはその両方かを制御します。詳細は第19章を参照してください。

msg引数は位置引数として与えられます。後方互換性のために、2つ以上の位置引数を与えることができます。その場合、位置引数のタプルの文字列表現がクライアントに報告されるメッセージになります。

以下のキーワードのみの引数を受け付けます。

```
detail
hint
sqlstate
schema_name
table_name
```

column_name
datatype_name
constraint_name

キーワードのみの引数として渡されたオブジェクトの文字列表現は、クライアントへ報告されるメッセージを豊富にするのに使われます。例えば、

```
CREATE FUNCTION raise_custom_exception() RETURNS void AS $$
plpy.error("custom exception message",
           detail="some info about exception",
           hint="hint for users")
$$ LANGUAGE plpythonu;

=# SELECT raise_custom_exception();
ERROR:  plpy.Error: custom exception message
DETAIL:  some info about exception
HINT:  hint for users
CONTEXT:  Traceback (most recent call last):
  PL/Python function "raise_custom_exception", line 4, in <module>
    hint="hint for users")
PL/Python function "raise_custom_exception"
```

この他のユーティリティ関数群にはplpy.quote_literal(string)、plpy.quote_nullable(string)およびplpy.quote_ident(string)があります。これらは9.4で説明する組み込みの引用符付け関数と同等です。これらはその場限りの問い合わせを構築する時に有用です。例 42.1の動的SQLと同等なPL/Pythonを以下に示します。

```
plpy.execute("UPDATE tbl SET %s = %s WHERE key = %s" % (
    plpy.quote_ident(colname),
    plpy.quote_nullable(newvalue),
    plpy.quote_literal(keyvalue)))
```

45.11. 環境変数

Pythonインタプリタにより受け付けられる環境変数の一部はまた、PL/Pythonの動作を変更するために使用することができます。これらは例えば起動スクリプト内など主PostgreSQLサーバプロセスの環境で設定される必要があります。利用可能な環境変数はPythonのバージョンに依存します。詳細に付いてはPythonの文書を参照してください。適切なバージョンのPythonであることが前提ですが、本章の執筆時点では以下の環境変数がPL/Pythonに影響を与えます。

- PYTHONHOME
- PYTHONPATH
- PYTHONY2K
- PYTHONOPTIMIZE

- PYTHONDEBUG
- PYTHONVERBOSE
- PYTHONCASEOK
- PYTHONDONTWRITEBYTECODE
- PYTHONIOENCODING
- PYTHONUSERBASE
- PYTHONHASHSEED

（pythonマニュアルページに列挙された環境変数の一部はコマンドラインインタプリタでのみ影響を与え埋め込みPythonインタプリタには影響しないというPL/Pythonの制御を超えたPythonの詳細実装があるようです。）

第46章 サーバプログラミングインタフェース

サーバプログラミングインタフェース(SPI)は、ユーザ定義のC関数からSQL問い合わせを実行する機能をユーザに提供します。SPIはパーサ、プランナ、エクゼキュータへのアクセスを単純化したインタフェース関数の集合です。また、SPIは多少のメモリ管理を行います。

> **注記**
>
> 利用可能な手続き言語は、関数からSQLコマンドを実行するための各種手段を提供します。これらのほとんどは、SPIを基にしていますので、この文書はこれらの言語のユーザにとっても有用な場合があります。

コマンドがSPIの失敗を起こした場合、その制御はC関数には戻らないことに注意してください。それどころか、プロシージャを実行していたトランザクションもしくは副トランザクションはロールバックされます (これはSPI関数のほとんどでエラーを返す規約があることから奇妙に思われるかもしれません。しかし、こうした規約はSPI関数自身でエラーを検知した時にのみ適用されるものです)。失敗する可能性があるSPI呼び出しを囲む副トランザクションを独自に用意することで、エラーの後の制御を戻すことができます。

SPI関数は成功時に非負の結果を(戻り値、もしくは後述のSPI_resultグローバル変数の中に)返します。エラー時、負の結果もしくはNULLを返します。

SPIを使用するソースコードファイルではexecutor/spi.hヘッダファイルをincludeしなければなりません。

46.1. インタフェース関数

SPI_connect

SPI_connect — SPIマネージャにC関数を接続する

概要

```
int SPI_connect(void)
```

```
int SPI_connect_ext(int options)
```

説明

SPI_connectはC関数の呼び出しからSPIマネージャへの接続を開きます。SPIを経由してコマンドを実行させる場合、この関数を呼び出さなければなりません。SPIユーティリティ関数の中には、未接続のC関数から呼び出し可能なものがあります。

SPI_connect_extは同様に動作しますが、オプションフラグを渡せる引数を一つもちます。今のところ以下のオプション値が使えます。

SPI_OPT_NONATOMIC

SPI接続を非原子的になるように設定します。これはトランザクション制御呼び出しのSPI_commit、SPI_rollback、および、SPI_start_transactionが可能であることを意味します。このフラグなしで、これら関数を呼び出すと即座にエラーになります。

SPI_connect()はSPI_connect_ext(0)と同義です。

戻り値

SPI_OK_CONNECT

成功した場合。

SPI_ERROR_CONNECT

エラーが発生した場合。

SPI_finish

SPI_finish — C関数をSPIマネージャから切断する

概要

```
int SPI_finish(void)
```

説明

SPI_finishは既存のSPIマネージャへの接続を切断します。C関数の現在の呼び出し期間内で必要なSPI操作が完了した後この関数を呼び出さなければなりません。しかし、elog(ERROR)経由でトランザクションを中断させる場合は、この関数が何を行うかを気にする必要はありません。その場合、SPIは自動的に自身を整理します。

戻り値

SPI_OK_FINISH

適切に切断された場合。

SPI_ERROR_UNCONNECTED

未接続のC関数から呼び出された場合。

SPI_execute

SPI_execute — コマンドを実行する

概要

```
int SPI_execute(const char * command, bool read_only, long count)
```

説明

SPI_executeは指定したSQLコマンドを、count行分実行します。read_onlyがtrueの場合、そのコマンドは読み取りのみでなければなりませんが、多少のオーバーヘッドが削減されます。

この関数は接続済みのC関数からのみ呼び出し可能です。

countが0の場合、そのコマンドを、適用される全ての行に対して実行します。countが0より多ければ、countを超えない数の行が取り出されます。問い合わせにLIMIT句と追加するの同じように、countに達すれば、実行は止まります。例えば、

```
SPI_execute("SELECT * FROM foo", true, 5);
```

は、テーブルから多くても5行しか取り出しません。この制限はコマンドが実際に行を返した場合にのみ有効なことに注意して下さい。例えば

```
SPI_execute("INSERT INTO foo SELECT * FROM bar", false, 5);
```

は、countパラメータを無視して、barからすべての行を挿入します。しかし、

```
SPI_execute("INSERT INTO foo SELECT * FROM bar RETURNING *", false, 5);
```

は、5番目のRETURNINGの結果行を取り出した後に実行が止まりますので、多くても5行を挿入するだけです。

複数のコマンドを1つの文字列として渡すことができます。SPI_executeは最後に実行したコマンドの結果を返します。count制限は（最後の結果が返されただけだとしても）それぞれのコマンドに独立に適用されます。この制限はルールによって生成される隠れたコマンドには適用されません。

read_onlyがfalseの場合、文字列内の各コマンドを実行する前にSPI_executeはコマンドカウンタを増分し、新しいスナップショットを作成します。このスナップショットは、現在のトランザクション分離レベルがSERIALIZABLEまたはREPEATABLE READの場合は変更されません。しかし、READ COMMITTEDモードでは、このスナップショットは更新され、他のセッションで新しくコミットされたトランザクションの結果を各コマンドから参照できます。これは、そのコマンドがデータベースを変更する場合、一貫性の維持に重要です。

read_onlyがtrueの場合は、SPI_executeはスナップショットもコマンドカウンタも更新しません。さらに、普通のSELECTコマンドのみをコマンド文字列内に記述することができます。このコマンドは、その前後の問い合わせによって事前に確立済みのスナップショットを使用して実行されます。この実行モードは読み書きモード

よりもコマンドごとのオーバーヘッドが省略される分多少高速です。また、これにより本当に安定(stable)な関数を構築することができます。つまり、連続した実行は全て同じスナップショットを使用しますので、結果は変わることがないということです。

一般的に、SPIを使用する1つの関数内で読み取りのみコマンドと読み書きコマンドを混在させることは勧めません。読み取りのみの問い合わせでは、読み書き問い合わせでなされたデータベースの更新結果を参照しないため、非常に混乱した動作に陥ることがあります。

(最後の)コマンドが実行した実際の行数は、SPI_processedグローバル変数に返されます。関数の戻り値がSPI_OK_SELECT、SPI_OK_INSERT_RETURNING、SPI_OK_DELETE_RETURNING、またはSPI_OK_UPDATE_RETURNINGの場合、SPITupleTable *SPI_tuptableグローバルポインタを使用して、結果の行にアクセスすることができます。また、一部のユーティリティコマンド(EXPLAINなど)は行セットを返しますが、この場合もSPI_tuptableにはその結果が含まれます。一部のユーティリティコマンド(COPY, CREATE TABLE AS)は行セットを返しません。このためSPI_tuptableはNULLですが、SPI_processedの中で処理行数を返します。

SPITupleTable構造体は以下のように定義されています。

```
typedef struct SPITupleTable
{

    /* 公開メンバ */
    TupleDesc   tupdesc;        /* タプル記述子 */
    HeapTuple  *vals;           /* タプルの配列 */
    uint64      numvals;        /* 有効なタプルの数 */


    /* 非公開メンバ、外部呼び出し側のためのものではない */
    uint64      alloced;        /* vals配列に割り当てられた長さ */
    MemoryContext tuptabcxt;    /* 結果テーブルのメモリコンテキスト */
    slist_node  next;           /* 内部情報のためのリンク */
    SubTransactionId subid;     /* SPITupleTableが生成されたサブトランザクション */

} SPITupleTable;
```

フィールドtupdesc、vals、numvalsはSPIの呼び出し側で使えます。残りのフィールドは内部のものです。valsは行へのポインタの配列です。行数はnumvalsで与えられます(ちょっとした歴史的理由により、この数はSPI_processedでも返されます)。tupdescは、行を扱うSPI関数に渡すことのできる行記述子です。

SPI_finishは、現在のC関数で割り当てられたSPITupleTableをすべて解放します。SPI_freetuptableを呼び出して解放する場合、特定の結果テーブルを早めに解放することができます。

引数

const char * command

実行するコマンドを含む文字列。

bool read_only

読み取りのみの実行の場合true。

long count

返される行の最大数。無制限なら0。

戻り値

コマンドの実行に成功した場合、以下のいずれかの(非負の)値が返されます。

SPI_OK_SELECT

SELECT(SELECT INTOを除く)が実行された場合。

SPI_OK_SELINTO

SELECT INTOが実行された場合。

SPI_OK_INSERT

INSERTが実行された場合。

SPI_OK_DELETE

DELETEが実行された場合。

SPI_OK_UPDATE

UPDATEが実行された場合。

SPI_OK_INSERT_RETURNING

INSERT RETURNINGが実行された場合。

SPI_OK_DELETE_RETURNING

DELETE RETURNINGが実行された場合。

SPI_OK_UPDATE_RETURNING

UPDATE RETURNINGが実行された場合。

SPI_OK_UTILITY

ユーティリティコマンド(CREATE TABLEなど)が実行された場合。

SPI_OK_REWRITTEN

ルールによって(例えば、UPDATEがINSERTになったような)あるコマンドが他の種類のコマンドに書き換えられた場合です。

エラーの場合、以下のいずれかの負の値が返されます。

SPI_ERROR_ARGUMENT

commandがNULL、あるいはcountが0未満の場合。

SPI_ERROR_COPY

COPY TO stdoutあるいはCOPY FROM stdinが試行された場合。

SPI_ERROR_TRANSACTION

トランザクション操作を行うコマンド（BEGIN、COMMIT、ROLLBACK、SAVEPOINT、PREPARE TRANSACTION、COMMIT PREPARED、ROLLBACK PREPARED、およびこれらの亜種）が試行された場合。

SPI_ERROR_OPUNKNOWN

コマンド種類が不明な場合（起きてはなりません）。

SPI_ERROR_UNCONNECTED

未接続なC関数から呼び出された場合。

注意

SPI問い合わせ実行関数はすべてSPI_processedとSPI_tuptableの両方を変更します（ポインタのみで、構造体の内容は変更しません）。SPI_execや他の問い合わせ実行関数の結果テーブルを後の呼び出しでまたがってアクセスしたいのであれば、これら2つのグローバル変数を局所的なプロシージャ変数に保存してください。

SPI_exec

SPI_exec — 読み書きコマンドを実行する

概要

```
int SPI_exec(const char * command, long count)
```

説明

SPI_execは、常にread_onlyパラメータをfalseとしたSPI_executeと同じです。

引数

const char * command

実行するコマンドを含む文字列。

long count

返される行の最大数。無制限なら0。

戻り値

SPI_executeを参照してください。

SPI_execute_with_args

SPI_execute_with_args — 行外のパラメータを持つコマンドを実行する

概要

```
int SPI_execute_with_args(const char *command,
                          int nargs, Oid *argtypes,
                          Datum *values, const char *nulls,
                          bool read_only, long count)
```

説明

SPI_execute_with_argsは外部から供給されるパラメータへの参照を含むコマンドを実行します。コマンドテキストはパラメータを$nとして参照し、呼び出しはこうしたシンボル毎にデータ型と値を指定します。read_onlyとcountはSPI_executeと同じ解釈をします。

SPI_executeと比較して、このルーチンの主たる利点は、データ値を面倒な引用やエスケープを要せずコマンドに埋め込むことができることで、従ってSQLインジェクション攻撃の危険性を軽減します。

後にSPI_execute_planが続いたSPI_prepareでも同様の結果が得られますが、この関数を使用するときには、提供された特定のパラメータ値に対して問い合わせ計画が必ずカスタマイズされます。1回限りの問い合わせ実行に対しては、この関数を選ぶべきです。多くの異なったパラメータを持つ同一のコマンドを実行する場合、再計画のコストと独自計画による利益に依存して、どちらか一方の方法がより早くなります。

引数

const char * command

コマンド文字列

int nargs

入力パラメータ（$1、$2など）の数

Oid * argtypes

パラメータのデータ型のOIDを含む、nargs長の配列

Datum * values

実パラメータ値を含む、nargs長の配列

const char * nulls

どのパラメータがnullかを記述する、nargs長の配列

nullsがNULLであれば、SPI_execute_with_argsはどのパラメータもnullでないと看做します。さもなければ、nulls配列の各項目は、対応するパラメータが非NULLならば' '、対応するパラメータがNULLならば'n'です。(後者の場合、values内の対応する値は注意されません。) nullsはテキスト文字列ではなく単なる配列であることに注意してください。'\0'終端は必要ありません。

bool read_only

読み取りのみの実行の場合true

long count

返される行の最大数。無制限なら0。

戻り値

戻り値はSPI_executeと同じです。

成功した場合SPI_processedとSPI_tuptableはSPI_executeと同様に設定されます。

SPI_prepare

SPI_prepare — 文を準備する。文の実行はまだ行わない

概要

```
SPIPlanPtr SPI_prepare(const char * command, int nargs, Oid * argtypes)
```

説明

SPI_prepareは指定したコマンド用の準備済み文を作成し、それを返します。しかし、そのコマンドは実行しません。その準備済み文はSPI_execute_planを使って後で繰り返し実行できます。

同じ、あるいは類似のコマンドが繰り返し実行される場合、一度だけ解析を計画作成を行うことには一般に利点があります。また、コマンドの実行計画を再利用することにはさらに利点があるかも知れません。SPI_prepareはコマンド文字列を、解析結果をカプセル化した準備済み文に変換します。実行の度に独自計画を生成するのが役に立たないと分かった場合には、準備済み文は実行計画をキャッシュする場所も提供します。

プリペアドコマンドは、通常のコマンド内の定数となる場所を（$1、$2などの）パラメータで記述することで一般化することができます。そしてパラメータの実際の値は、SPI_execute_plan が呼び出される時に指定されます。これにより、プリペアドコマンドは、パラメータがない場合に比べ、より広範な状況で使用できるようになります。

SPI_finishが文のために割り当てられたメモリを解放しますので、SPI_prepareで返される文は、そのC関数の現在の呼び出し内でのみ使用することができます。しかし、関数SPI_keepplanやSPI_saveplanを使用して長期間文を保存することもできます。

引数

`const char * command`

コマンド文字列

`int nargs`

入力パラメータ（$1、$2など）の数

`Oid * argtypes`

パラメータのデータ型のOIDを持つ配列へのポインタ

戻り値

SPI_prepareはSPIPlanへの非NULLのポインタを返します。ここでSPIPlanは準備済み文を表すopaque構造体です エラーの場合、NULLが返され、SPI_executeで使用されるエラーコードと同じコードの1つ

がSPI_resultに設定されます。しかし、commandがNULLの場合や、nargsが0未満の場合、nargsが0より大きくかつargtypesがNULLの場合は、SPI_ERROR_ARGUMENTに設定されます。

注意

パラメータが定義されていなければ、SPI_execute_planが最初に使用された時に一般的な計画が作成され、以降の実行すべてでも利用されます。パラメータがあれば、始めの何回かのSPI_execute_planの使用で、与えられたパラメータの値に固有の独自計画が作成されます。同じ準備済み文が十分に使用された後、SPI_execute_planは一般的な計画を作成し、独自計画よりもそれほど高価でなければ、毎回再計画する代わりに一般的な計画を使い始めるようになります。このデフォルトの動作が不適切であれば、SPI_prepare_cursorにCURSOR_OPT_GENERIC_PLANまたはCURSOR_OPT_CUSTOM_PLANフラグを設定することで、それぞれ一般的な計画か独自計画を強制的に利用するよう変更できます。

プリペアド文の主要な利点は、文の解析処理と計画作成処理の繰り返しを防止することですが、PostgreSQLでは、以前にそのプリペアド文を使用してから、文の中で使用されているデータベースオブジェクトが定義（DDL）の変更を受けた時は常に再解析処理と計画再作成処理を強制します。また、一度使用してからsearch_pathの値が変わった場合も、文は新しいsearch_pathを使用して再解析されます。（後者の振る舞いはPostgreSQL 9.3の時に追加されました。）プリペアド文の動作についてはPREPAREを参照してください。

この関数は接続済みのC関数からのみ呼び出してください。

SPIPlanPtrはspi.h内でopaque構造体型へのポインタとして宣言されています。たいていの場合将来のバージョンのPostgreSQLでそのコードが壊れてしまうため、この内容に直接アクセスすることは避けてください。

そのデータ構造はもはや実行計画を含むとは限りませんので、SPIPlanPtrという名前はいくらか歴史的なものです。

SPI_prepare_cursor

SPI_prepare_cursor — 文を準備する。まだ実行は行わない

概要

```
SPIPlanPtr SPI_prepare_cursor(const char * command, int nargs,
                              Oid * argtypes, int cursorOptions)
```

説明

SPI_prepare_cursorは、プランナの「カーソルオプション」パラメータを指定できる点を除き、SPI_prepareと同じです。これはDeclareCursorStmtのoptionsフィールド用にnodes/parsenodes.hで示された値を持つビットマスクです。SPI_prepareでは常にカーソルオプションをゼロとして扱います。

引数

`const char * command`

コマンド文字列

`int nargs`

入力パラメータ(`$1`、`$2`など)の数

`Oid * argtypes`

パラメータのデータ型のOIDを持つ配列へのポインタ

`int cursorOptions`

カーソルオプションの整数ビットマスク。ゼロはデフォルトの動作を引き起こします

戻り値

SPI_prepare_cursorはSPI_prepareと同じ戻り値の規則を持ちます。

注釈

cursorOptionsに指定できるビットには、CURSOR_OPT_SCROLL、CURSOR_OPT_NO_SCROLL、CURSOR_OPT_FAST_PLAN、CURSOR_OPT_GENERIC_PLAN、CURSOR_OPT_CUSTOM_PLANがあります。特にCURSOR_OPT_HOLDは無視される点に注意してください。

SPI_prepare_params

SPI_prepare_params — 文を準備する。まだ実行は行わない

概要

```
SPIPlanPtr SPI_prepare_params(const char * command,
                              ParserSetupHook parserSetup,
                              void * parserSetupArg,
                              int cursorOptions)
```

説明

SPI_prepare_paramsは指定したコマンドの準備済み文を作成し返します。しかしそのコマンドを実行しません。この関数はSPI_prepare_cursorと同じですが、呼び出し元が外部パラメータ参照の解析を制御するパーサフック関数を指定できる点が追加されています。

引数

const char * command

コマンド文字列

ParserSetupHook parserSetup

パーサフック設定関数

void * parserSetupArg

parserSetupに渡される引数

int cursorOptions

カーソルオプションの整数ビットマスク。ゼロはデフォルトの動作を引き起こします

戻り値

SPI_prepare_paramsはSPI_prepareと同じ戻り値の規則を持ちます。

SPI_getargcount

SPI_getargcount — SPI_prepareにより準備した文に必要とされる引数の数を返す

概要

```
int SPI_getargcount(SPIPlanPtr plan)
```

説明

SPI_getargcountは、SPI_prepareにより準備された文を実行する時に必要とされる引数の数を返します。

引数

SPIPlanPtr plan

(SPI_prepareで返される)準備済み文です。

戻り値

planで想定される引数の数です。planがNULLまたは無効な場合はSPI_resultにSPI_ERROR_ARGUMENTが設定され、-1が返されます。

SPI_getargtypeid

SPI_getargtypeid — SPI_prepareで準備された文で指定される引数のデータ型のOIDを返す

概要

```
Oid SPI_getargtypeid(SPIPlanPtr plan, int argIndex)
```

説明

SPI_getargtypeidは、SPI_prepareで準備された文におけるargIndex番目の引数の型を表すOIDを返します。インデックス0は最初の引数を示します。

引数

SPIPlanPtr plan

(SPI_prepareで返される)準備済み文

int argIndex

0から始まる引数のインデックス

戻り値

指定したインデックスにおける引数の型OIDです。planがNULLまたは無効、あるいはargIndexが0未満、planで宣言された引数の数以上の場合、SPI_resultにSPI_ERROR_ARGUMENTが設定され、InvalidOidが返されます。

SPI_is_cursor_plan

SPI_is_cursor_plan — SPI_prepareで準備された文がSPI_cursor_openで使用できる場合にtrueを返す

概要

```
bool SPI_is_cursor_plan(SPIPlanPtr plan)
```

説明

SPI_prepareで準備済み文がSPI_cursor_openへの引数として渡すことができる場合、
SPI_is_cursor_planはtrueを返します。渡すことができない場合はfalseを返します。この基準は、planが単一のコマンドであり、かつ、そのコマンドが呼び出し元にタプルを返すことです。例えば、INTO句を含んでいないSELECTは可能です。そして、RETURNING句を含む場合のみUPDATEも可能です。

引数

SPIPlanPtr plan

(SPI_prepareで返される)準備済み文

戻り値

planがカーソルを生成することができるかどうかを示すtrueもしくはfalseです。そしてSPI_resultをゼロに設定します。解答を決定することができない場合(例えばplanがNULL、または無効な場合、もしくはSPI未接続時に呼び出された場合)はSPI_resultに適切なエラーコードが設定され、falseが返されます。

SPI_execute_plan

SPI_execute_plan — SPI_prepareで準備された文を実行する

概要

```
int SPI_execute_plan(SPIPlanPtr plan, Datum * values, const char * nulls,
                     bool read_only, long count)
```

説明

SPI_execute_planは、SPI_prepareもしくは類似の関数で準備された文を実行します。
read_onlyとcountはSPI_executeと同様の解釈がなされます。

引数

SPIPlanPtr plan

(SPI_prepareで返される)準備済み文

Datum * values

実パラメータ値の配列。文の引数の数と同じ長さでなければなりません

const char * nulls

どのパラメータがNULLであるかを示す配列。文の引数の数と同じ長さでなければなりません。

nullsがNULLの場合、SPI_execute_planはすべてのパラメータがNULLではないとみなします。さもなければ、nulls配列の各項目は、対応するパラメータが非NULLならば' '、対応するパラメータがNULLならば'n'です。(後者の場合、values内の対応する値は注意されません。) nullsはテキスト文字列ではなく単なる配列であることに注意してください。'\0'終端は必要ありません。

bool read_only

読み取りのみの実行の場合true

long count

返される行の最大数。無制限なら0。

戻り値

戻り値は、SPI_execute同様のものに加え、以下のエラー(負)の結果を取ることがあります。

SPI_ERROR_ARGUMENT

planがNULLまたは無効、あるいは、countが0未満の場合

SPI_ERROR_PARAM

valuesがNULL、かつ、planがパラメータ付きで準備された場合

成功時、SPI_processedとSPI_tuptableがSPI_execute同様に設定されます。

SPI_execute_plan_with_paramlist

SPI_execute_plan_with_paramlist — SPI_prepareで準備された文を実行する

概要

```
int SPI_execute_plan_with_paramlist(SPIPlanPtr plan,
                                    ParamListInfo params,
                                    bool read_only,
                                    long count)
```

説明

SPI_execute_plan_with_paramlistはSPI_prepareで準備された文を実行します。この関数はSPI_execute_planと同じですが、問い合わせに渡されるパラメータ値に関する情報が別途存在する点が異なります。ParamListInfo表現は、すでに利用可能な形式で値を渡すために便利です。またParamListInfoで指定されたフック関数経由での動的なパラメータ群の使用をサポートします。

引数

SPIPlanPtr plan

(SPI_prepareで返される)準備済み文

ParamListInfo params

パラメータの型と値からなるデータ構造。なければヌル。

bool read_only

読み取りのみの実行の場合true

long count

返される行の最大数。無制限なら0。

戻り値

戻り値はSPI_execute_planと同じです。

成功時、SPI_processedとSPI_tuptableがSPI_execute_plan同様に設定されます。

SPI_execp

SPI_execp — 読み書きモードで文を実行する

概要

```
int SPI_execp(SPIPlanPtr plan, Datum * values, const char * nulls, long count)
```

説明

SPI_execpは、常にread_onlyパラメータをfalseとしたSPI_execute_planと同じです。

引数

SPIPlanPtr plan

(SPI_prepareで返される)準備済み文

Datum * values

実パラメータ値の配列。文の引数の数と同じ長さでなければなりません。

const char * nulls

どのパラメータがNULLであるかを示す配列。文の引数の数と同じ長さでなければなりません。

nullsがNULLの場合、SPI_execpはすべてのパラメータがNULLではないとみなします。さもなければ、nulls配列の各項目は、対応するパラメータが非NULLならば' '、対応するパラメータがNULLならば'n'です。(後者の場合、values内の対応する値は注意されません。) nullsはテキスト文字列ではなく単なる配列であることに注意してください。'\0'終端は必要ありません。

long count

返される行の最大数。無制限なら0。

戻り値

SPI_execute_planを参照してください。

成功時、SPI_execute同様にSPI_processedとSPI_tuptableが設定されます。

SPI_cursor_open

SPI_cursor_open — SPI_prepareで作成された文を使用したカーソルを設定する

概要

```
Portal SPI_cursor_open(const char * name, SPIPlanPtr plan,
                       Datum * values, const char * nulls,
                       bool read_only)
```

説明

SPI_cursor_openは、SPI_prepareによって準備された文を実行するカーソル（内部的にはポータル）を設定します。このパラメータはSPI_execute_planの対応するパラメータと同じ意味を持ちます。

文を直接実行するのではなくカーソルを使用することには2つの利点があります。1つ目は、結果行を一度に少なく取り出し、多くの行を返す問い合わせでのメモリの過使用を防ぐことができる点です。2つ目は、ポータルは現在のC関数の外部でも有効である点です（実際、現在のトランザクションの終端まで有効とすることができます）。C関数の呼び出し元にポータルの名前を返すことで、結果として行セットを返す手段を提供します。

渡されるパラメータデータはカーソルのポータルにコピーされます。そのため、カーソルが存在している間にそのデータを解放することができます。

引数

const char * name

ポータルの名前、あるいはシステムに名前を決定させる場合はNULL

SPIPlanPtr plan

（SPI_prepareで返される）準備済み文

Datum * values

実パラメータ値の配列。文の引数の数と同じ長さでなければなりません。

const char * nulls

どのパラメータがNULLであるかを示す配列。文の引数の数と同じ長さでなければなりません。

nullsがNULLの場合、SPI_cursor_openは全てのパラメータがNULLではないとみなします。さもなければ、nulls配列の各項目は、対応するパラメータが非NULLならば' '、対応するパラメータがNULLならば'n'です。（後者の場合、values内の対応する値は注意されません。）nullsはテキスト文字列ではなく単なる配列であることに注意してください。'\0'終端は必要ありません。

`bool read_only`

読み取りのみの実行の場合true

戻り値

カーソルを含むポータルへのポインタ。戻り値の規約にはエラーを表すものがないことに注意してください。エラーはすべてelog経由で報告されます。

SPI_cursor_open_with_args

SPI_cursor_open_with_args — 問い合わせとパラメータを使ってカーソルを設定する

概要

```
Portal SPI_cursor_open_with_args(const char *name,
                                 const char *command,
                                 int nargs, Oid *argtypes,
                                 Datum *values, const char *nulls,
                                 bool read_only, int cursorOptions)
```

説明

SPI_cursor_open_with_argsは特定の問い合わせを実行するカーソル（内部的にはポータル）を設定します。ほとんどのパラメータはSPI_prepare_cursorとSPI_cursor_openに対応するパラメータと同じ意味を持っています。

1回限りの問い合わせ実行に対しては、後にSPI_cursor_openが続いたSPI_prepare_cursorよりも、この関数を選ぶべきです。多くの異なったパラメータを持つ同一のコマンドを実行する場合、再計画のコストと独自計画による利益に依存して、どちらか一方の方法がより早くなります。

渡されたパラメータデータはカーソルのポータルにコピーされますので、カーソルが存在している間は解放することができます。

引数

`const char * name`

ポータルの名前、またはシステムに名前を選択させるNULL

`const char * command`

コマンド文字列

`int nargs`

入力パラメータ（$1、$2など）の数

`Oid * argtypes`

パラメータのデータ型のOIDを含む、nargs長の配列

`Datum * values`

実パラメータ値を含む、nargs長の配列

`const char * nulls`

どのパラメータがnullかを記述する、nargs長の配列

nullsがNULLであれば、SPI_cursor_open_with_argsはどのパラメータもnullでないとみなします。さもなければ、nulls配列の各項目は、対応するパラメータが非NULLならば' '、対応するパラメータがNULLならば'n'です。(後者の場合、values内の対応する値は注意されません。) nullsはテキスト文字列ではなく単なる配列であることに注意してください。'\0'終端は必要ありません。

`bool read_only`

読み取りのみの実行の場合true

`int cursorOptions`

カーソルオプションの整数ビットマスク。ゼロの場合はデフォルトの動作

戻り値

カーソルを含んだポータルへのポインタ。エラーを返す規約がないことに注意してください。すべてのエラーはelogで報告されます。

SPI_cursor_open_with_paramlist

SPI_cursor_open_with_paramlist — パラメータを使ってカーソルを設定する

概要

```
Portal SPI_cursor_open_with_paramlist(const char *name,
                                      SPIPlanPtr plan,
                                      ParamListInfo params,
                                      bool read_only)
```

説明

SPI_cursor_open_with_paramlistはSPI_prepareで準備された文を実行するカーソル(内部的にはポータル)を設定します。この関数はSPI_cursor_openと同じですが、問い合わせに渡されるパラメータ値に関する情報が別途存在することが異なります。ParamListInfo表現は、すでに利用可能な形式で値を渡すために便利です。またParamListInfoで指定されたフック関数経由での動的なパラメータ群の使用をサポートします。

渡されるパラメータデータはカーソルのポータルにコピーされます。そのため、カーソルが存在している間にそのデータを解放することができます。

引数

`const char * name`

ポータルの名前、あるいはシステムに名前を決定させる場合はNULL

`SPIPlanPtr plan`

(SPI_prepareで返される)準備済み文

`ParamListInfo params`

パラメータの型と値からなるデータ構造。なければヌル。

`bool read_only`

読み取りのみの実行の場合true

戻り値

カーソルを含むポータルへのポインタ。戻り値の規約にはエラーを表すものがないことに注意してください。エラーはすべてelog経由で報告されます。

SPI_cursor_find

SPI_cursor_find — 既存のカーソルを名前で検索する

概要

```
Portal SPI_cursor_find(const char * name)
```

説明

SPI_cursor_findは既存のカーソルを名前で検索します。これは主に、他の何らかの関数でテキストとして返されたカーソル名の名前解決の際に使用されます。

引数

const char * name

ポータルの名前

戻り値

指定された名前のポータルへのポインタ。見つからない場合はNULLです。

SPI_cursor_fetch

SPI_cursor_fetch — カーソルから数行を取り出す

概要

```
void SPI_cursor_fetch(Portal portal, bool forward, long count)
```

説明

SPI_cursor_fetchはカーソルから数行を取り出します。これは、FETCH SQLコマンドと部分的に等価です。（詳細機能についてはSPI_scroll_cursor_fetchを参照してください。）

引数

`Portal portal`

カーソルを持つポータル

`bool forward`

前方方向の取り出しの場合、真。後方方向の場合は偽。

`long count`

取り出す最大行数。

戻り値

成功時、SPI_processedとSPI_tuptableがSPI_execute同様に設定されます。

注釈

カーソルの計画が CURSOR_OPT_SCROLLオプションを付けて作成されなかった場合、後方方向の取り出しは失敗する可能性があります。

SPI_cursor_move

SPI_cursor_move — カーソルを移動する

概要

```
void SPI_cursor_move(Portal portal, bool forward, long count)
```

説明

SPI_cursor_moveはカーソル内で、数行を飛ばします。これはMOVE SQLコマンドと部分的に等価です。(詳細機能についてはSPI_scroll_cursor_moveを参照してください。)

引数

`Portal portal`

カーソルを持つポータル

`bool forward`

前方方向の移動の場合、真。後方方向の場合は偽。

`long count`

移動する最大行数。

注釈

カーソルの計画がCURSOR_OPT_SCROLLオプション付きで作成されなかった場合、後方方向への移動は失敗する可能性があります。

SPI_scroll_cursor_fetch

SPI_scroll_cursor_fetch — カーソルから一部の行を取り出す

概要

```
void SPI_scroll_cursor_fetch(Portal portal, FetchDirection direction,
                             long count)
```

説明

SPI_scroll_cursor_fetchはカーソルから行の一部を取り出します。これはSQLコマンドFETCHと等価です。

引数

`Portal portal`

カーソルを含むポータル

`FetchDirection direction`

FETCH_FORWARD、FETCH_BACKWARD、FETCH_ABSOLUTE、FETCH_RELATIVEのいずれか

`long count`

FETCH_FORWARDまたはFETCH_BACKWARDでは取り出す行数。FETCH_ABSOLUTEでは取り出す行の絶対番号。FETCH_RELATIVEでは取り出す行の相対的番号。

戻り値

成功時、SPI_execute同様にSPI_processedとSPI_tuptableが設定されます。

注釈

directionパラメータおよびcountパラメータの解釈の詳細についてはSQL FETCHコマンドを参照してください。

カーソルの計画がCURSOR_OPT_SCROLLオプション付きで作成されていない場合、FETCH_FORWARD以外の方向値は失敗する可能性があります。

SPI_scroll_cursor_move

SPI_scroll_cursor_move — カーソルを移動する

概要

```
void SPI_scroll_cursor_move(Portal portal, FetchDirection direction,
                            long count)
```

説明

SPI_scroll_cursor_moveはカーソル内の行の一部を飛ばします。これはSQLコマンドMOVEと等価です。

引数

`Portal portal`

カーソルを含むポータル

`FetchDirection direction`

FETCH_FORWARD、FETCH_BACKWARD、FETCH_ABSOLUTE、FETCH_RELATIVEのいずれか

`long count`

FETCH_FORWARDまたはFETCH_BACKWARDでは移動する行数。FETCH_ABSOLUTEでは移動する行の絶対番号。FETCH_RELATIVEでは移動する行の相対的番号。

戻り値

成功時、SPI_execute同様にSPI_processedが設定されます。この関数は行を返しませんので、SPI_tuptableはNULLに設定されます。

注釈

directionパラメータおよびcountパラメータの解釈の詳細についてはSQL FETCHコマンドを参照してください。

カーソルの計画がCURSOR_OPT_SCROLLオプション付きで作成されていない場合、FETCH_FORWARD以外の方向値は失敗する可能性があります。

SPI_cursor_close

SPI_cursor_close — カーソルを閉じる

概要

```
void SPI_cursor_close(Portal portal)
```

説明

SPI_cursor_closeは事前に作成されたカーソルを閉じ、そのポータル用の領域を解放します。

トランザクションの終了時に全ての開いているカーソルが自動的に閉ざされます。SPI_cursor_closeは、リソースの解放をより早めに行いたい場合にのみ呼び出す必要があります。

引数

Portal portal

カーソルを持つポータル

SPI_keepplan

SPI_keepplan — 準備済み文を保持する

概要

```
int SPI_keepplan(SPIPlanPtr plan)
```

説明

SPI_keepplanは渡された(SPI_prepareで準備された)文をSPI_finishとトランザクションマネージャで解放されないメモリ内に保存します。これは、現在のセッションにおける、その後のC関数の呼び出しで準備済み文を再利用できる機能を提供します。

引数

SPIPlanPtr plan

保存する準備済み文

戻り値

成功時は0。planがNULLまたは無効な場合はSPI_ERROR_ARGUMENT

注意

渡された文はポインタの調整により永続的記憶領域に再配置されます(データコピーは不要です)。後ほど削除したければ、SPI_freeplanを実行してください。

SPI_saveplan

SPI_saveplan — 準備済み文を保存する

概要

```
SPIPlanPtr SPI_saveplan(SPIPlanPtr plan)
```

説明

SPI_saveplanは渡された(SPI_prepareで準備された)文をSPI_finishとトランザクションマネージャで解放されないメモリ内にコピーします。そして、コピーした文のポインタを返します。これは、現在のセッションにおける、その後のC関数の呼び出しで準備済み文を再利用できる機能を提供します。

引数

SPIPlanPtr plan

保存する準備済み文

戻り値

コピーした文へのポインタ。失敗した場合はNULLです。エラー時、SPI_resultは以下のように設定されます。

SPI_ERROR_ARGUMENT

planがNULL、または無効な場合

SPI_ERROR_UNCONNECTED

未接続のC関数から呼び出された場合

注意

渡された元の文は解放されません。ですので、SPI_finishを行うまでのメモリリークを防ぎたければSPI_freeplanを実行してください。

準備済み文のデータ構造を物理的にコピーする必要なく、ほとんど同じ結果をもたらしますので、たいていの場合、この関数よりもSPI_keepplanの方が好ましいです。

SPI_register_relation

SPI_register_relation — 短命の名前付きリレーションをSPIの問い合わせから名前で参照可能にする

概要

```
int SPI_register_relation(EphemeralNamedRelation enr)
```

説明

SPI_register_relationは短命の名前付きリレーションを、現在のSPI接続を通して計画され、実行される問い合わせに対して、関連情報と一緒に参照できるようにします。

引数

EphemeralNamedRelation enr

短命の名前付きリレーションの登録エントリ

戻り値

コマンドの実行に成功したときは、次の（負でない）値が返されます。

SPI_OK_REL_REGISTER

リレーションが名前で登録できた場合

エラーが発生したときは、以下の負の値の一つが返されます。

SPI_ERROR_ARGUMENT

enrがNULLか、そのnameフィールドがNULLの場合

SPI_ERROR_UNCONNECTED

接続されていないC関数から呼び出された場合

SPI_ERROR_REL_DUPLICATE

enrのnameフィールドで指定された名前が、現在の接続で既に登録済みの場合

SPI_unregister_relation

SPI_unregister_relation — 短命の名前付きリレーションをSPIのレジストリから削除する

概要

```
int SPI_unregister_relation(const char * name)
```

説明

SPI_unregister_relationは短命の名前付きリレーションを現在の接続のレジストリから削除します。

引数

const char * name

リレーションのレジストリエントリの名前

戻り値

コマンドの実行に成功したときは、次の（負でない）値が返されます。

SPI_OK_REL_UNREGISTER

タプルストアがレジストリから削除された場合

エラーが発生したときは、以下の負の値の一つが返されます。

SPI_ERROR_ARGUMENT

nameがNULLの場合

SPI_ERROR_UNCONNECTED

接続されていないC関数から呼び出された場合

SPI_ERROR_REL_NOT_FOUND

nameが現在の接続のレジストリに見つからない場合

SPI_register_trigger_data

SPI_register_trigger_data — 短命のトリガーデータをSPIの問い合わせから利用可能にする

概要

```
int SPI_register_trigger_data(TriggerData *tdata)
```

説明

SPI_register_trigger_dataはトリガによって捕捉される任意の短命のリレーションを、現在のSPI接続を通して計画され、実行される問い合わせで利用可能にします。現在のところ、これはREFERENCING OLD/NEW TABLE ASの句で定義されるAFTERトリガによって捕捉される遷移テーブルを意味します。この関数は接続後にPLのトリガハンドラ関数から呼び出されるようにします。

引数

TriggerData *tdata

トリガハンドラ関数にfcinfo->contextとして渡されるTriggerDataオブジェクト

戻り値

コマンドの実行に成功したときは、次の（負でない）値が返されます。

SPI_OK_TD_REGISTER

捕捉されたトリガデータ（あれば）が登録された場合

エラーが発生したときは、以下の負の値の一つが返されます。

SPI_ERROR_ARGUMENT

tdataがNULLの場合

SPI_ERROR_UNCONNECTED

接続されていないC関数から呼び出された場合

SPI_ERROR_REL_DUPLICATE

トリガデータの遷移リレーションのどれかの名前が、この接続で既に登録されている場合

46.2. インタフェースサポート関数

以下で説明する関数は、SPI_executeや他のSPI関数で返される結果セットから情報を取り出すためのインタフェースを提供します。

本節で説明する関数は全て、接続、未接続のC関数のどちらからでも使用することができます。

SPI_fname

SPI_fname — 指定した列番号に対する列名を決定する

概要

```
char * SPI_fname(TupleDesc rowdesc, int colnumber)
```

説明

SPI_fname は指定した列の列名のコピーを返します（名前のコピーが不要になった場合pfreeを使用してその領域を解放することができます）。

引数

`TupleDesc rowdesc`

入力行の記述

`int colnumber`

（1から始まる）列番号

戻り値

列の名前です。colnumberが範囲外の場合はNULLです。エラー時、
SPI_resultはSPI_ERROR_NOATTRIBUTEに設定されます。

SPI_fnumber

SPI_fnumber — 指定した列名から列番号を決定する

概要

```
int SPI_fnumber(TupleDesc rowdesc, const char * colname)
```

説明

SPI_fnumberは指定した名前の列の列番号を返します。

colnameが（ctidなどの）システム列を参照する場合、適切な負の列番号が返されます。呼び出し元は、エラーを検知するために戻り値がSPI_ERROR_NOATTRIBUTEと正確に同一であるかどうかを注意して検査しなければなりません。システム列を拒絶させたくなければ、結果が0あるいは0未満かを検査するという方法は、正しくありません。

引数

TupleDesc rowdesc

入力行の記述

const char * colname

列名

戻り値

（ユーザ定義の列について1から始まる）列番号。指定された名前の列が見つからなければ、SPI_ERROR_NOATTRIBUTEです。

SPI_getvalue

SPI_getvalue — 指定された列の文字列値を返す

概要

```
char * SPI_getvalue(HeapTuple row, TupleDesc rowdesc, int colnumber)
```

説明

SPI_getvalueは指定された列の値の文字列表現を返します。

結果は、pallocを使用して割り当てられたメモリ内に返されます（不要になった段階で、pfreeを使用してメモリを解放することができます）。

引数

`HeapTuple row`

検査対象の入力行

`TupleDesc rowdesc`

入力行の記述

`int colnumber`

（1から始まる）列番号

戻り値

列の値。列がNULLの場合、あるいはcolnumberが範囲外の場合はNULLです（SPI_resultがSPI_ERROR_NOATTRIBUTEに設定されます）。利用できる出力関数が存在しない場合は、NULLです（SPI_resultがSPI_ERROR_NOOUTFUNCに設定されます）。

SPI_getbinval

SPI_getbinval — 指定した列のバイナリ値を返す

概要

```
Datum SPI_getbinval(HeapTuple row, TupleDesc rowdesc, int colnumber,
                    bool * isnull)
```

説明

SPI_getbinval は指定された列の値を内部形式で（Datumとして）返します。

この関数はデータ用に新しい領域を確保しません。参照渡しのデータ型の場合、戻り値は渡された行の内部を示すポインタとなります。

引数

HeapTuple row

対象とする入力行

TupleDesc rowdesc

入力行の記述

int colnumber

（1から始まる）列番号

bool * isnull

列のNULL値についてのフラグ

戻り値

列のバイナリ値が返されます。isnullで指し示される変数は、列がNULLならば真に、さもなくば、偽に設定されます。

エラー時、SPI_resultはSPI_ERROR_NOATTRIBUTEに設定されます。

SPI_gettype

SPI_gettype — 指定された列のデータ型名を返す

概要

```
char * SPI_gettype(TupleDesc rowdesc, int colnumber)
```

説明

SPI_gettypeは指定された列のデータ型名のコピーを返します（不要になった段階で、pfreeを使用して名前のコピーを解放することができます）。

引数

`TupleDesc rowdesc`

入力行の記述

`int colnumber`

（1から始まる）列番号

戻り値

指定された列のデータ型名。エラー時はNULLです。エラー時、SPI_resultはSPI_ERROR_NOATTRIBUTEに設定されます。

SPI_gettypeid

SPI_gettypeid — 指定された列のデータ型のOIDを返す

概要

```
Oid SPI_gettypeid(TupleDesc rowdesc, int colnumber)
```

説明

SPI_gettypeidは指定された列のデータ型のOIDを返します。

引数

`TupleDesc rowdesc`

入力行の記述

`int colnumber`

(1から始まる)列番号

戻り値

指定された列のデータ型のOIDです。エラー時はInvalidOidです。エラー時、
SPI_resultはSPI_ERROR_NOATTRIBUTEに設定されます。

SPI_getrelname

SPI_getrelname — 指定されたリレーションの名前を返す

概要

```
char * SPI_getrelname(Relation rel)
```

説明

SPI_getrelnameは指定リレーションの名前のコピーを返します（不要になった段階で、pfreeを使用して名前のコピーを解放することができます）。

引数

`Relation rel`

入力リレーション

戻り値

指定されたリレーションの名前です。

SPI_getnspname

SPI_getnspname — 指定されたリレーションの名前空間を返す

概要

```
char * SPI_getnspname(Relation rel)
```

説明

SPI_getnspnameは、指定したRelationが属する名前空間名のコピーを返します。これはリレーションのスキーマと同じです。作業終了時に、この関数の戻り値に対してpfreeを行わなければなりません。

引数

Relation rel

入力リレーション

戻り値

指定したリレーションの名前空間の名称です。

SPI_result_code_string

SPI_result_code_string — 文字列でエラーコードを返します

概要

```
const char * SPI_result_code_string(int code);
```

Description

SPI_result_code_stringは、様々なSPI関数から返されたか、SPI_resultに格納された結果コードの文字列表現を返します。

Arguments

int code

結果コード

結果値

結果コードの文字列表現

46.3. メモリ管理

PostgreSQLは、メモリコンテキスト内にメモリを確保します。これは、様々な場所で、必要な有効期間がそれぞれ異なるような割り当てを管理する便利な方法を提供します。コンテキストを破壊することで、そこで割り当てられた全てのメモリを解放します。したがって、メモリリークを防ぐための個々のオブジェクトの追跡を維持することは不要です。その代わり、相対的に少量のコンテキストを管理する必要があります。pallocと関連する関数は「現在の」コンテキストからメモリを確保します。

SPI_connectは新しくメモリコンテキストを作成し、それを現在のコンテキストとします。SPI_finishは直前の現在のメモリコンテキストを戻し、SPI_connectで作成されたコンテキストを破壊します。これらの動作により、C関数内で割り当てが行われる一時的なメモリがC関数の終了時に回収され、メモリリークが防止されることが保証されます。

しかし、(参照渡しのデータ型の値といった)C関数が割り当てられたメモリ内のオブジェクトを返す必要がある場合、少なくともSPIに接続していない期間は、pallocを使用してメモリを確保することができません。これを試行すると、そのオブジェクトはSPI_finishで解放されてしまい、C関数は正しく動作しないでしょう。この問題を解決するには、SPI_pallocを使用して、戻り値となるオブジェクト用のメモリを確保してください。SPI_pallocは「上位エクゼキュータコンテキスト」内にメモリを割り当てます。このメモリコンテキストは、SPI_connectが呼び出された時点において現在のコンテキストだったものであり、C関数の戻り値用のコンテ

キストとしてまさに正しいものです。この章で説明されているほかのユーティリティ関数のいくつかも、上位エクゼキュータコンテキスト内で作成されたオブジェクトを返します。

SPI_connectが呼び出されると、SPI_connectによって作成されるC関数固有のコンテキストが現在のコンテキストに作成されます。palloc、repalloc、SPIユーティリティ関数（この章で説明されているものは除きます）によって作成される割り当ては全て、このコンテキスト内に作成されます。C関数がSPIマネージャから（SPI_finish経由で）切断した時、現在のコンテキストは上位エクゼキュータコンテキストに戻され、C関数のメモリコンテキスト内で割り当てられたメモリは全て解放され、二度と使用することができません。

SPI_palloc

SPI_palloc — 上位エクゼキュータコンテキスト内にメモリを割り当てる

概要

```
void * SPI_palloc(Size size)
```

説明

SPI_pallocは上位エクゼキュータコンテキスト内にメモリを割り当てます。

この関数はSPIに接続されている間にのみ使うことができます。それ以外の場合はエラーを発生させます。

引数

Size size

割り当てる領域のバイト数

戻り値

指定サイズの新しい格納領域へのポインタ

SPI_repalloc

SPI_repalloc — 上位エクゼキュータコンテキスト内にメモリを再割り当てる

概要

```
void * SPI_repalloc(void * pointer, Size size)
```

説明

SPI_repallocは、以前にSPI_pallocを使用して割り当てられたメモリセグメントのサイズを変更します。

この関数はもはや通常のrepallocとは異なるものではありません。単に既存コードの後方互換性のために保持されています。

引数

`void * pointer`

変更する既存の領域へのポインタ

`Size size`

割り当てる領域のバイト数

戻り値

指定サイズに新規に割り当てられ、既存領域の内容をコピーした領域へのポインタ

SPI_pfree

SPI_pfree — 上位エクゼキュータコンテキスト内のメモリを解放する

概要

```
void SPI_pfree(void * pointer)
```

説明

SPI_pfreeは、以前にSPI_pallocやSPI_repallocを使用して割り当てられたメモリを解放します。

この関数はもはや通常のpfreeとは異なるものではありません。単に既存コードの後方互換性のために保持されています。

引数

void * pointer

解放する既存の領域へのポインタ

SPI_copytuple

SPI_copytuple — 上位エクゼキュータ内に行のコピーを作成する

概要

```
HeapTuple SPI_copytuple(HeapTuple row)
```

説明

SPI_copytupleは上位エクゼキュータコンテキスト内に行のコピーを作成します。これは通常、トリガから変更した行を返す時に使用されます。複合型を返すものと宣言された関数では、代わりにSPI_returntupleを使用してください。

この関数はSPIに接続されている間にのみ使うことができます。それ以外の場合はNULLを返し、SPI_resultをSPI_ERROR_UNCONNECTEDにセットします。

引数

HeapTuple row

コピーされる行

戻り値

コピーされた行、あるいはエラー時はNULL（エラーの表示についてはSPI_resultを参照してください）

SPI_returntuple

SPI_returntuple — Datumとしてタプルを返す準備をする

概要

```
HeapTupleHeader SPI_returntuple(HeapTuple row, TupleDesc rowdesc)
```

説明

SPI_returntupleは上位エクゼキュータコンテキスト内に行の複製を作成し、それを行型のDatum形式で返します。返されるポインタは、返す前にPointerGetDatumを使用してDatumに変換することのみが必要です。

この関数はSPIに接続されている間にのみ使うことができます。それ以外の場合はNULLを返し、SPI_resultをSPI_ERROR_UNCONNECTEDにセットします。

この関数は複合型を返すものと宣言された関数に対して使用しなければなりません。トリガでは使用されません。トリガで変更された行を返すにはSPI_copytupleを使用してください。

引数

HeapTuple row

コピーされる行

TupleDesc rowdesc

行の記述子（最も効率的にキャッシュを行えるように毎回同一の記述子を渡してください）

戻り値

コピーされた行を指し示すHeapTupleHeader、あるいはエラー時はNULLです。（エラーの表示についてはSPI_resultを参照してください）

SPI_modifytuple

SPI_modifytuple — 与えられた行の選択フィールドを置き換えた行を作成する

概要

```
HeapTuple SPI_modifytuple(Relation rel, HeapTuple row, int ncols,
                          int * colnum, Datum * values, const char * nulls)
```

説明

SPI_modifytupleは、選択された列は新しい値で置き換え、その他の位置は元の行の列をコピーした、新しい行を作成します。入力行は変更されません。新しい行は上位エクゼキュータコンテキスト内に返されます。

この関数はSPIに接続されている間にのみ使うことができます。それ以外の場合はNULLを返し、SPI_resultをSPI_ERROR_UNCONNECTEDにセットします。

引数

`Relation rel`

行の行記述子のソースとしてのみ使用されます。(行記述子ではなくリレーションを渡すことは設計ミスです。)

`HeapTuple row`

変更される行

`int ncols`

変更された列数

`int * colnum`

変更される列番号を含む、ncols長の配列(列番号は1から始まります)

`Datum * values`

指定された列の新しい値を含む、ncols長の配列

`const char * nulls`

新しい値のどれがNULLかを記述する、ncols長の配列

nullsがNULLであれば、SPI_modifytupleはどの新しい値もnullでないとみなします。さもなければ、nulls配列の各項目は、対応するパラメータが非NULLならば' '、対応するパラメータがNULLならば'n'です。(後者の場合、values内の対応する値は注意されません。) nullsはテキスト文字列ではなく単なる配列であることに注意してください。'\0'終端は必要ありません。

戻り値

変更された新しい行。上位エクゼキュータコンテキストに割り当てられます。エラー時はNULLです。（エラーの表示についてはSPI_resultを参照してください）

エラー時、SPI_resultが以下のように設定されます。

SPI_ERROR_ARGUMENT

relがNULLの場合、rowがNULLの場合、ncolsが0以下の場合、colnumがNULLの場合、valuesがNULLの場合。

SPI_ERROR_NOATTRIBUTE

colnumが無効な列番号を持つ場合（0以下、rowの列数以上）。

SPI_ERROR_UNCONNECTED

SPIが動作していない場合

SPI_freetuple

SPI_freetuple — 上位エクゼキュータコンテキスト内に割り当てられた行を解放する

概要

```
void SPI_freetuple(HeapTuple row)
```

説明

SPI_freetupleは以前に上位エクゼキュータコンテキスト内に割り当てられた行を解放します。

この関数はもはや通常のheap_freetupleとは異なるものではありません。単に既存コードの後方互換性のために保持されています。

引数

HeapTuple row

解放する行

SPI_freetuptable

SPI_freetuptable — SPI_executeや類似の関数によって生成された行セットを解放する

概要

```
void SPI_freetuptable(SPITupleTable * tuptable)
```

説明

SPI_freetuptableは、以前にSPI_executeなどのSPIコマンド実行関数によって作成された行セットを解放します。そのため、この関数はよくSPI_tuptableグローバル変数を引数として呼び出されます。

この関数はSPIプロシージャが複数のコマンドを実行する必要があり、かつ、初期のコマンドの結果を終わりまで保持したくない場合に有用です。解放されない行セットは、SPI_finish時に全て解放されることに注意してください。また副トランザクションが始まった後SPIプロシージャの実行中にアボートした場合、SPIは自動的に副トランザクションが実行中に作成された行セットすべてを解放します。

PostgreSQL 9.3からSPI_freetuptableには同一行セットに対して重複する削除要求から保護する保護ロジックが含まれます。過去のリリースでは重複する削除がクラッシュをもたらすかもしれませんでした。

引数

SPITupleTable * tuptable

解放する行セットへのポインタ。または何も行わないことを示すNULL。

SPI_freeplan

SPI_freeplan — 以前に保存した準備済み文を解放する

概要

```
int SPI_freeplan(SPIPlanPtr plan)
```

説明

SPI_freeplanは、以前にSPI_prepareから返された、あるいはSPI_keepplanやSPI_saveplanで保存された準備済み文を解放します。

引数

SPIPlanPtr plan

解放する文へのポインタ

戻り値

成功時は0。planがNULLまたは無効な場合、SPI_ERROR_ARGUMENTです。

46.4. トランザクション管理

COMMITやROLLBACKといったトランザクション制御コマンドをSPI_executeなどのSPI関数を通して実行することはできません。しかしながら、SPIを通してトランザクション制御ができる別のインタフェース関数があります。

どこで呼び出されるかという文脈を考慮することなく、ユーザ定義された任意のSQL呼び出し可能な関数でトランザクションを開始・終了することは、一般的に安全でも思慮のあることではありません。例えば、SQLコマンドの一部の複雑なSQL式の一部である関数中のトランザクションブロックは、おそらく不明瞭な内部エラーやクラッシュになります。ここに示されるインタフェース関数は、CALL起動の文脈を考慮しており、主としてCALLコマンドから起動される手続き言語から使われることを意図しています。SPIを使ったCで実装されたプロシージャは同じロジックを実装できますが、その詳細は本文書の範囲を超えます。

SPI_commit

SPI_commit, SPI_commit_and_chain — 現在のトランザクションをコミットします。

概要

```
void SPI_commit(void)
```

```
void SPI_commit_and_chain(void)
```

説明

SPI_commitは現在のトランザクションをコミットします。これはSQLコマンドのCOMMITを実行することと概ね同等です。トランザクションがコミットされた後には、続くデータベース操作を実行する前にSPI_start_transactionを使って新たなトランザクションを開始しなければなりません。

SPI_commit_and_chainは同じですが、新しいトランザクションは、SQLコマンドCOMMIT AND CHAINと同じように、直前に完了したものと同じトランザクションの特性で即時に開始されます。

これらの関数はSPI接続がSPI_connect_extの呼び出しで非原子的と設定されている場合のみ、実行できます。

SPI_rollback

SPI_rollback, SPI_rollback_and_chain — 現在のトランザクションを中断します。

概要

```
void SPI_rollback(void)
```

```
void SPI_rollback_and_chain(void)
```

説明

SPI_rollbackは現在のトランザクションをロールバックします。これはSQLコマンドのROLLBACKを実行することと概ね同等です。トランザクションをロールバックした後、続くデータベース操作を実行する前にSPI_start_transactionを使用して新たなトランザクションを開始しなければなりません。

SPI_rollback_and_chainは同じですが、新しいトランザクションは、SQLコマンドROLLBACK AND CHAINと同じように、直前に完了したものと同じトランザクションの特性で即時に開始されます。

これらの関数はSPI接続がSPI_connect_extの呼び出しで非原子的と設定されている場合のみ、実行できます。

SPI_start_transaction

SPI_start_transaction — 新たなトランザクションを開始します

概要

```
void SPI_start_transaction(void)
```

説明

SPI_start_transactionは新たなトランザクションを開始します。そこがトランザクション内では無いように、SPI_commitまたはSPI_rollbackの後でのみ呼び出しできます。通常は、何らかSPIを使ったプロシージャが呼ばれたとき、トランザクションは既に開始済みです。そのため、現在のトランザクションを閉じる前に別のトランザクションを開始しようとするとエラーになります。

この関数はSPI接続がSPI_connect_extの呼び出しで非原子的と設定されている場合のみ、実行できます。

46.5. データ変更の可視性

SPI（や他の任意のC関数）を使用する関数内のデータの可視性は、以下の規則に従います。

- SQLコマンドの実行中、そのコマンドで行われたデータ変更はそのコマンドからは不可視です。例えば、

  ```
  INSERT INTO a SELECT * FROM a;
  ```

  では、挿入された行はSELECT部からは不可視です。

- コマンドCで行われた変更は、Cの後に開始された全てのコマンドからは可視です。Cの内側（処理中）に開始したかCの処理後に開始したかは関係ありません。

- SQLコマンドによって呼び出される関数（普通の関数やトリガ関数）の内側で、SPIを使用して実行されるコマンドは、SPIに渡される読み書きフラグに応じて上の規則のいくつかに従います。読み取りのみモードで実行されるコマンドは、呼び出し中のコマンドの変更は不可視であるという最初の規則に従います。読み書きモードで実行されるコマンドは、今までに行われた変更はすべて可視であるという2番目の規則に従います。

- 標準の手続き言語は全て、関数の変動属性に応じてSPI読み書きモードを設定します。STABLEおよびIMMUTABLE関数のコマンドは、読み取りのみモードで行われ、VOLATILE関数のコマンドは読み書きモードで行われます。C言語関数の作者はこの規約を無視することができますが、それはほとんどの場合勧められません。

次節には、これら規則の適用についてを示す例があります。

46.6. 例

本節には、SPIを使用する非常に簡単な例があります。C関数execqは1つ目の引数としてSQLコマンドを、2つ目の引数として行数を取り、SPI_execコマンドを実行し、そのコマンドで処理された行数を返します。SPIのより複雑な例はソースツリー内のsrc/test/regress/regress.cとspiモジュールにあります。

```
#include "postgres.h"

#include "executor/spi.h"
#include "utils/builtins.h"

PG_MODULE_MAGIC;

PG_FUNCTION_INFO_V1(execq);

Datum
execq(PG_FUNCTION_ARGS)
{
    char *command;
    int cnt;
    int ret;
    uint64 proc;


    /* 与えられたテキストオブジェクトをC文字列に変換 */
    command = text_to_cstring(PG_GETARG_TEXT_PP(0));
    cnt = PG_GETARG_INT32(1);

    SPI_connect();

    ret = SPI_exec(command, cnt);

    proc = SPI_processed;

    /*

     * 何らかの行が取り出された場合は、行をelog(INFO)を使用して表示
     */
    if (ret > 0 && SPI_tuptable != NULL)
    {
        SPITupleTable *tuptable = SPI_tuptable;
        TupleDesc tupdesc = tuptable->tupdesc;
        char buf[8192];
        uint64 j;

        for (j = 0; j < tuptable->numvals; j++)
```

```
        {
            HeapTuple tuple = tuptable->vals[j];
            int i;

            for (i = 1, buf[0] = 0; i <= tupdesc->natts; i++)
                snprintf(buf + strlen(buf), sizeof(buf) - strlen(buf), " %s%s",
                        SPI_getvalue(tuple, tupdesc, i),
                        (i == tupdesc->natts) ? " " : " |");
            elog(INFO, "EXECQ: %s", buf);
        }
    }

    SPI_finish();
    pfree(command);

    PG_RETURN_INT64(proc);
}
```

以下は、コンパイルし共有ライブラリ(37.10.5を参照)を作成した後で、関数を宣言する方法です。

```
CREATE FUNCTION execq(text, integer) RETURNS int8
    AS 'filename'
    LANGUAGE C STRICT;
```

以下はセッションの例です。

```
=> SELECT execq('CREATE TABLE a (x integer)', 0);
 execq
-------
     0
(1 row)

=> INSERT INTO a VALUES (execq('INSERT INTO a VALUES (0)', 0));
INSERT 0 1
=> SELECT execq('SELECT * FROM a', 0);

INFO:  EXECQ:  0    -- execqによって挿入された
INFO:  EXECQ:  1    -- execqによって返され、
      上位のINSERTによって挿入された

 execq
-------
     2
(1 row)

=> SELECT execq('INSERT INTO a SELECT x + 2 FROM a', 1);
```

```
 execq
-------
     1
(1 row)

=> SELECT execq('SELECT * FROM a', 10);
INFO:  EXECQ:  0
INFO:  EXECQ:  1

INFO:  EXECQ:  2    -- 指定された、
      0 + 2という1つの行のみが挿入された

 execq
-------

     3              -- 10は最大値を示すのみで、
      3が実際の行数です。
(1 row)

=> DELETE FROM a;
DELETE 3
=> INSERT INTO a VALUES (execq('SELECT * FROM a', 0) + 1);
INSERT 0 1
=> SELECT * FROM a;
 x
---

 1                  -- aテーブルに行がない(0) + 1
(1 row)

=> INSERT INTO a VALUES (execq('SELECT * FROM a', 0) + 1);
INFO:  EXECQ:  1
INSERT 0 1
=> SELECT * FROM a;
 x
---
 1

 2                  -- aテーブルに1行あり + 1
(2 rows)


--    これはデータ変更に関する可視性規則を説明します。

=> INSERT INTO a SELECT execq('SELECT * FROM a', 0) * x FROM a;
INFO:  EXECQ:  1
```

```
INFO:  EXECQ:  2
INFO:  EXECQ:  1
INFO:  EXECQ:  2
INFO:  EXECQ:  2
INSERT 0 2
=> SELECT * FROM a;
 x
---
 1
 2

 2                  -- 2 行 * 1 （xは最初の行）
 6                  -- 3 行 （2 + 挿入された 1） * 2 （2行目のx）
(4 rows)               ^^^^^^
                       別の呼び出しで execq()で可視な行
```

第47章 バックグラウンドワーカプロセス

PostgreSQLはユーザ提供のコードを別のプロセスとして実行できるように拡張することができます。このプロセスはpostgresによって起動、終了、監視され、サーバの状態に密接にリンクした寿命を持つことができます。これらのプロセスはPostgreSQLの共有メモリ領域にアタッチしたり、データベースの内部に接続するオプションを持ちます。これらはまた、通常のクライアントに接続された実際のサーバプロセスのように複数のトランザクションを連続して実行することができます。また、アプリケーションはlibpqとリンクすることにより通常のクライアントアプリケーションのようにサーバに接続して動作することができます。

> **警告**
>
> バックグラウンドワーカを使うにあたっては、堅牢性とセキュリティリスクを考慮しなくてはなりません。なぜならば、C言語で書かれており、データへのアクセスが制限されていないためです。バックグラウンドワーカプロセスを含むモジュールを有効にしたいと思っている管理者は、細心の注意を払って実践してください。バックグラウンドワーカプロセスの実行は、注意深く検査されたモジュールだけを許可する必要があります。

バックグラウンドワーカは、モジュールをshared_preload_librariesに記すことによって、PostgreSQLスタート時に初期化できます。バックグラウンドワーカとして実行したいモジュールは、_PG_init()関数からRegisterBackgroundWorker(BackgroundWorker *worker)を呼び出すことで登録できます。バックグラウンドワーカはシステム起動後もRegisterDynamicBackgroundWorker(BackgroundWorker *worker, BackgroundWorkerHandle **handle)を呼び出すことによって開始することができます。postmasterプロセスからのみ呼び出すことができるRegisterBackgroundWorkerとは異なり、RegisterDynamicBackgroundWorkerは通常のバックエンドまたは他のバックグラウンドワーカから呼び出す必要があります。

BackgroundWorkerの構造体は以下のように定義されます。

```
typedef void (*bgworker_main_type)(Datum main_arg);
typedef struct BackgroundWorker
{
    char        bgw_name[BGW_MAXLEN];
    char        bgw_type[BGW_MAXLEN];
    int         bgw_flags;
    BgWorkerStartTime bgw_start_time;
    int         bgw_restart_time;       /* in seconds, or BGW_NEVER_RESTART */
    char        bgw_library_name[BGW_MAXLEN];
    char        bgw_function_name[BGW_MAXLEN];
    Datum       bgw_main_arg;
    char        bgw_extra[BGW_EXTRALEN];
    int         bgw_notify_pid;
} BackgroundWorker;
```

bgw_nameやbgw_typeは、ログメッセージ、プロセス一覧、および同様の場面で使用される文字列です。bgw_typeは、同じ種類のバックグラウンドワーカで全て同じになるため、例えば同じ種類のワーカをプロセス

一覧でグループ化することができます。一方でbgw_nameは、特定のプロセスに関する追加情報を含むことができます。(通常、bgw_nameの文字列は何らかの形で種類に関する情報を含んでいますが、必須であるというわけではありません。)

bgw_flagsは、モジュールが要求する機能をOR演算したビットマスクです。可能な値は以下の通りです。

BGWORKER_SHMEM_ACCESS

共有メモリへのアクセスを要求します。共有メモリアクセスがないワーカは、重量または軽量のロック、共有バッファ、ワーカが作成して利用したいカスタムデータ構造等、PostgreSQLの共有データ構造にアクセスできません。

BGWORKER_BACKEND_DATABASE_CONNECTION

トランザクションやクエリの実行が出来るデータベース接続を要求します。BGWORKER_BACKEND_DATABASE_CONNECTIONを使用してデータベースに接続するバックグラウンドワーカはBGWORKER_SHMEM_ACCESSを使用して共有メモリにアタッチしなければなりません。さもなければ起動時に失敗します。

bgw_start_timeは、postgresがプロセスを起動するべきタイミングを指定します。そのタイミングは、以下のうちの1つです。BgWorkerStart_PostmasterStart(postgres自身が初期化を終えるとすぐに起動します。これを要求するプロセスはデータベース接続に望ましいものではありません)、BgWorkerStart_ConsistentState(ホットスタンバイで一貫性のある状態に到達し、データベースに接続して参照のみのクエリが実行できるようになると起動します)、BgWorkerStart_RecoveryFinished(システムが通常の参照/更新クエリを実行できるようになると起動します)。最後の2つの値は、ホットスタンバイでないサーバでは同等であることに注意してください。この設定はいつプロセスが起動されるかを示すだけであることに注意してください。これらのプロセスは、違う状態になったときに停止するわけではありません。

bgw_restart_timeは、プロセスがクラッシュした場合にpostgresがそのプロセスを再起動するために待つ必要のある間隔を秒単位で指定します。これは任意の正の値、またはクラッシュしても再起動させない場合にBGW_NEVER_RESTARTを指定します。

bgw_library_nameはバックグラウンドワーカの初期エントリーポイントのためのライブラリ名です。その指定されたライブラリがワーカプロセスによって動的にロードされます。呼び出すべき関数を特定するためにbgw_function_nameが使用されます。コアコードから関数をロードする場合、"postgres"を設定する必要があります。

bgw_function_nameは新しいバックグラウンドワーカから動的にロードされるときに初期エントリポイントの関数名です。

bgw_main_argは、バックグラウンドワーカのメイン関数のDatum引数です。メイン関数は単一のDatum引数を取り、voidを返します。bgw_main_argは引数として渡されます。加えて、グローバル変数MyBgworkerEntryは、登録時に渡されたBackgroundWorker構造体のコピーを指しています。ワーカはこの構造を調べることがあり、役に立ちます。

Windowsの(どこか他の場所でEXEC_BACKENDが定義されている)場合、または動的バックグラウンドワーカは、Datumを参照で渡すのは安全ではありません。値のみで渡してください。引数が必要な場合は、int32型または他の小さな値を渡し、共有メモリに割り当てられた配列へのインデックスとしてそれを使用するのが最も安全です。cstringやtextのようなポインタを渡された場合は、新しいバックグラウンドワーカプロセスから有効になりません。

bgw_extraはバックグラウンドワーカに渡す追加データを含めることが出来ます。bgw_main_argとは異なり、このデータはワーカのメイン関数の引数として渡されていませんが、上述したようにMyBgworkerEntryを介してアクセスすることが出来ます。

bgw_notify_pidはプロセスの開始時と終了時にpostmasterがSIGUSR1を送信するPostgreSQLバックエンドプロセスのPIDです。それはpostmasterの起動時に登録されたワーカの場合、またはワーカを登録しているバックエンドがワーカーの起動を待ちたくない場合は0にする必要があります。それ以外の場合は、MyProcPidで初期化する必要があります。

ひとたび実行すると、このプロセスはBackgroundWorkerInitializeConnection(char *dbname, char *username, uint32 flags)またはBackgroundWorkerInitializeConnectionByOid(Oid dboid, Oid useroid, uint32 flags)を呼び出すことによって、データベースに接続できます。これはプロセスにSPIインタフェースを使用してのトランザクションとクエリの実行を許します。もし、dbnameがNULLであった場合、またはdboidがInvalidOidであった場合には、そのセッションは特定のデータベースに接続しません。しかし、共有カタログにはアクセス出来ます。もし、usernameがNULLの場合、またはuseroidがInvalidOidの場合には、そのプロセスはinitdb時に作成されたスーパーユーザとして実行されます。flagsとしてBGWORKER_BYPASS_ALLOWCONNが設定されている場合、データベースへ接続する際のユーザ接続を許可しない制約を回避することが出来ます。バックグラウンドワーカはこれら2つの関数をどちらかを一度だけ呼ぶことが出来ます。データベースを切り替えることができません。

バックグラウンドワーカのメイン関数に制御が達したとき、シグナルは最初にブロックされています。このブロックは解除されなければなりません。これは、必要に応じてプロセスがシグナルハンドラをカスタマイズできるようにするためです。シグナルは、新しいプロセスでBackgroundWorkerUnblockSignalsを呼び出すことにより解除でき、BackgroundWorkerBlockSignalsを呼び出すことでブロックできます。

バックグラウンドワーカは、bgw_restart_timeがBGW_NEVER_RESTARTに設定されている場合、または終了コード0で終了した場合、またはTerminateBackgroundWorkerによって終了した場合、postmasterに自動的に登録が解除されて終了します。それ以外の場合、bgw_restart_timeで設定された時間の後に再起動します。または、バックエンドの障害のためにpostmasterがクラスタを再初期化した場合は、すぐに再起動します。一時的に実行を中断するだけでよいバックエンドは、終了するのではなく、割り込み可能なスリープを使用する必要があります。これはWaitLatch()を呼び出すことによって可能になります。この関数を呼び出すときにはWL_POSTMASTER_DEATHフラグが設定されているか確認し、postgres自身が終了する緊急事態には、リターンコードを確認するようにしてください。

バックグラウンドワーカをRegisterDynamicBackgroundWorker関数により登録している場合、登録を実行するバックエンドはワーカの状態に関する情報を取得することが可能です。取得したい場合はRegisterDynamicBackgroundWorkerに2番目の引数としてBackgroundWorkerHandle *のアドレスを渡す必要があります。もし登録に成功した場合、このポインタは後でGetBackgroundWorkerPid(BackgroundWorkerHandle *,pid_t *)またはTerminateBackgroundWorker(BackgroundWorkerHandle *)に渡すことができるopaque(不透明)ハンドルで、初期化されます。GetBackgroundWorkerPidはワーカの状態を監視することができます。以下の返り値が得られます。BGWH_NOT_YET_STARTEDワーカはまだpostmasterにより開始されていない。BGWH_STOPPED開始されたが、もはや実行されていない。BGWH_STARTED実行中です。この最後のケースでは、PIDは、2番目の引数を介して返されます。TerminateBackgroundWorkerはワーカが実行していた場合postmasterがワーカにSIGTERMを送信し、実行が終了次第すぐに登録を解除します。

場合によっては、バックグラウンドワーカが起動するのを待ってから、ワーカを登録したい場合もあるでしょう。これは bgw_notify_pidをMyProcPidで初期化し、登録時に得られたBackgroundWorkerHandle *を使用してWaitForBackgroundWorkerStartup(BackgroundWorkerHandle *handle,pid_t *)関

数を呼び出すことで実現します。postmasterがバックグラウンドワーカを開始しようと試みたか、 postmasterが死ぬまで、この関数はブロックします。バックグラウンドワーカが実行されている場合、 戻り値はBGWH_STARTEDとなり、指定されたアドレスにPIDが書き込まれます。そうでない場合、戻り値 はBGWH_STOPPEDまたはBGWH_POSTMASTER_DIEDになります。

登録時に得られたBackgroundWorkerHandle *を使用し てWaitForBackgroundWorkerShutdown(BackgroundWorkerHandle *handle)関数を呼び出すことで、バックグラウンドワーカがシャットダウンするのを待つこともできます。バックグラウンドワーカが終了するか、 postmasterが死ぬまで、この関数はブロックします。バックグラウンドワーカが終了した場合の戻り値 はBGWH_STOPPED、postmasterが死んだ場合の戻り値はBGWH_POSTMASTER_DIEDになります。

バックグラウンドワーカは、サーバプログラミングインタフェース(SPI)経由でNOTIFYコマンドにより非同期に通知を送る場合、囲んでいるトランザクションをコミットした後、通知を配信することができるように明示的にProcessCompletedNotifiesを呼ぶ必要があります。バックグラウンドワーカは、SPIを通じてLISTENによる非同期通知の受信を登録した場合、ワーカがこれらの通知をログに記録しますが、ワーカが傍受し、それらの通知に応答するためのプログラム的な方法はありません。

バックグラウンドワーカの実例として、src/test/modules/worker_spiというモジュールがあります。これはいくつかの有用な技術を示しています。

登録できるバックグラウンドワーカの最大数はmax_worker_processesによって制限されています。

第48章 ロジカルデコーディング

PostgreSQLは、SQLによって実行された更新結果を外部のコンシューマにストリーミングする基盤を提供しています。

更新結果は、ロジカルレプリケーションスロット(logical replication slots)で識別されるストリームに送出されます。

ストリームに送出される更新データのフォーマットは、使用するプラグインで決まります。サンプルプラグインがPostgreSQLの配布物に含まれています。追加のプラグインを書くことにより、PostgreSQLのコア部分のコードを一切変更することなく、利用可能なフォーマットの選択肢を増やすことができます。すべてのプラグインから、INSERTによって作成された個々の新しい行と、UPDATEによって作成された新しい個々の行のバージョンにアクセスできます。UPDATEとDELETEによって生じた古いバージョンの行へのアクセスが可能かどうかは、レプリカアイデンティティ(replica identity)の設定によって決まります(REPLICA IDENTITY参照)。

変更データの消費は、ストリーミングレプリケーションのプロトコル(52.4と 48.3を参照)を使うか、SQL関数(48.4)を使って行います。また、コア部分に手を入れなくても、レプリケーションスロットの出力を消費する別の方法を実装することもできます (48.7参照)。

48.1. ロジカルデコーディングの例

以下はロジカルデコーディングをSQLを使って制御する例です。

ロジカルデコーディングを使う前に、wal_level を logical に、そしてmax_replication_slotsを少なくとも1に設定しなければなりません。次に、使用するデータベースにスーパーユーザ(以下の例ではpostgres)として接続します。

```
postgres=# -- Create a slot named 'regression_slot' using the output plugin 'test_decoding'
postgres=# SELECT * FROM pg_create_logical_replication_slot('regression_slot', 'test_decoding');
    slot_name    |    lsn
-----------------+-----------
 regression_slot | 0/16B1970
(1 row)

postgres=# SELECT slot_name, plugin, slot_type, database, active, restart_lsn,
 confirmed_flush_lsn FROM pg_replication_slots;
    slot_name    |    plugin     | slot_type | database | active | restart_lsn |
 confirmed_flush_lsn
-----------------+---------------+-----------+----------+--------+-------------+-----------------
 regression_slot | test_decoding | logical   | postgres | f      | 0/16A4408   | 0/16A4440
(1 row)

postgres=# -- There are no changes to see yet
postgres=# SELECT * FROM pg_logical_slot_get_changes('regression_slot', NULL, NULL);
 lsn | xid | data
-----+-----+------
(0 rows)
```

```
postgres=# CREATE TABLE data(id serial primary key, data text);
CREATE TABLE

postgres=# -- DDL isn't replicated, so all you'll see is the transaction
postgres=# SELECT * FROM pg_logical_slot_get_changes('regression_slot', NULL, NULL);
    lsn    |  xid  |     data
-----------+-------+--------------
 0/BA2DA58 | 10297 | BEGIN 10297
 0/BA5A5A0 | 10297 | COMMIT 10297
(2 rows)

postgres=# -- Once changes are read, they're consumed and not emitted
postgres=# -- in a subsequent call:
postgres=# SELECT * FROM pg_logical_slot_get_changes('regression_slot', NULL, NULL);
 lsn | xid | data
-----+-----+------
(0 rows)

postgres=# BEGIN;
postgres=*# INSERT INTO data(data) VALUES('1');
postgres=*# INSERT INTO data(data) VALUES('2');
postgres=*# COMMIT;

postgres=# SELECT * FROM pg_logical_slot_get_changes('regression_slot', NULL, NULL);
    lsn    |  xid  |                          data
-----------+-------+--------------------------------------------------------
 0/BA5A688 | 10298 | BEGIN 10298
 0/BA5A6F0 | 10298 | table public.data: INSERT: id[integer]:1 data[text]:'1'
 0/BA5A7F8 | 10298 | table public.data: INSERT: id[integer]:2 data[text]:'2'
 0/BA5A8A8 | 10298 | COMMIT 10298
(4 rows)

postgres=# INSERT INTO data(data) VALUES('3');

postgres=# -- You can also peek ahead in the change stream without consuming changes
postgres=# SELECT * FROM pg_logical_slot_peek_changes('regression_slot', NULL, NULL);
    lsn    |  xid  |                          data
-----------+-------+--------------------------------------------------------
 0/BA5A8E0 | 10299 | BEGIN 10299
 0/BA5A8E0 | 10299 | table public.data: INSERT: id[integer]:3 data[text]:'3'
 0/BA5A990 | 10299 | COMMIT 10299
(3 rows)

postgres=# -- The next call to pg_logical_slot_peek_changes() returns the same changes again
postgres=# SELECT * FROM pg_logical_slot_peek_changes('regression_slot', NULL, NULL);
```

```
    lsn    |  xid  |                          data
-----------+-------+-------------------------------------------------------
 0/BA5A8E0 | 10299 | BEGIN 10299
 0/BA5A8E0 | 10299 | table public.data: INSERT: id[integer]:3 data[text]:'3'
 0/BA5A990 | 10299 | COMMIT 10299
(3 rows)

postgres=# -- options can be passed to output plugin, to influence the formatting
postgres=# SELECT * FROM pg_logical_slot_peek_changes('regression_slot', NULL, NULL, 'include-
timestamp', 'on');
    lsn    |  xid  |                          data
-----------+-------+-------------------------------------------------------
 0/BA5A8E0 | 10299 | BEGIN 10299
 0/BA5A8E0 | 10299 | table public.data: INSERT: id[integer]:3 data[text]:'3'
 0/BA5A990 | 10299 | COMMIT 10299 (at 2017-05-10 12:07:21.272494-04)
(3 rows)

postgres=# -- Remember to destroy a slot you no longer need to stop it consuming
postgres=# -- server resources:
postgres=# SELECT pg_drop_replication_slot('regression_slot');
 pg_drop_replication_slot
----------------------

(1 row)
```

以下はPostgreSQLに付属するプログラムpg_recvlogicalを用いてロジカルデコーディングを ストリーミングレプリケーションのプロトコルによって制御する例です。この方法を使うには、レプリケーション接続を許すようにクライアント認証を設定し(26.2.5.1参照)、max_wal_sendersを十分に大きくして追加の接続ができるようにしておかなければなりません。

```
$ pg_recvlogical -d postgres --slot=test --create-slot
$ pg_recvlogical -d postgres --slot=test --start -f -
Control+Z
$ psql -d postgres -c "INSERT INTO data(data) VALUES('4');"
$ fg
BEGIN 693
table public.data: INSERT: id[integer]:4 data[text]:'4'
COMMIT 693
Control+C
$ pg_recvlogical -d postgres --slot=test --drop-slot
```

48.2. ロジカルデコーディングのコンセプト

48.2.1. ロジカルデコーディング

ロジカルデコーディングは、データベースのテーブルへの恒久的な更新を、一貫性があって、データベース内部の状態に関する詳細な知識がなくても容易に理解できる形式として取得するプロセスです。

PostgreSQLにおいてロジカルデコーディングは、記憶装置のレベルで更新を記述する書き込み先行ログの内容を、タプルやSQL文のストリームといったアプリケーション固有の形式にデコードすることによって実装されています。

48.2.2. レプリケーションスロット

ロジカルレプリケーションの文脈ではスロットは、元のサーバで行われた変更と同じ順序でクライアント上でリプレイできるようなストリームを表します。それぞれのスロットは、単一のデータベース上の変更操作の連鎖をストリームとして流します。

> **注記**
>
> またPostgreSQLには、ストリーミングレプリケーションスロットがあります (26.2.5参照)。しかし、ここでの説明とは少し違う使い方がされています。

それぞれのレプリケーションスロットはPostgreSQLクラスタの中で一意な識別子を持っています。スロットは、そのために使用される接続とは独立しており、クラッシュセーフです。

ロジカルスロットは、通常の操作においては、各々の変更操作を一度だけ送出します。それぞれのスロットにおける現在位置は、チェックポイントのときにだけ永続的になります。ですからクラッシュすると、スロットは以前のLSNに戻ってしまうかもしれませんし、サーバの再起動時には最近の変更が再送されることになります。ロジカルデコーディングのクライアントは、同じメッセージを複数回扱うことによる好ましくない結果を避けることに対して責任を追っています。クライアントはデコーディングの際に最後に確認したLSNを記録し、繰り返されるデータをスキップしたり、(レプリケーションプロトコルを使う場合に)サーバに開始時点を決めさせるのではなく、記録しておいたLSNからデコーディングを始めるように要求するかもしれません。レプリケーション進捗追跡機能はこの目的のために設計されています。replication originsを参照してください。

単一のデータベース中に、お互いに独立した複数のスロットが存在しても構いません。それぞれのスロットは自分自身の状態を持っており、データベース更新のストリーム上の別の場所から変更データを受信する異なる消費者があり得ます。多くのアプリケーションにとっては、各消費者に対して個別のスロットが必要となるでしょう。

ロジカルレプリケーションスロットは、受信者の状態については関知しません。同時にでなければ、同じスロットを使う複数の異なる受信者を持つことさえできます。その場合は、直近の受信者がストリームの消費を終了した時点から更新データを受信するだけです。

> **注意**
>
> レプリケーションスロットは、クラッシュをまたがって永続し、消費者の状態については関知しません。スロットを使う接続がない場合でも、消費者が必要としているリソースが削除されることを防ぎます。これによりストレージが消費されます。何故ならば、関連するWALもシステムカタログの行も、レプリケーションスロットが必要とする限りVACUUMによって削除されないからです。極端な場合、トランザクションIDの周回(24.1.5を参照)を防ぐためのデータベース停止をもたらす可能性があります。したがって、必要でなくなったスロットは削除すべきです。

48.2.3. 出力プラグイン

出力プラグインは、書き込み先行ログの内部データ表現を、レプリケーションスロットの消費者が必要とする形式に変換します。

48.2.4. スナップショットのエクスポート

ストリーミングレプリケーションのインタフェースを使って新しいスロットを作ると(CREATE_REPLICATION_SLOT参照)、スナップショットがエクスポートされます(9.27.5参照)。このスナップショットはまさにその時点でのデータベースの状態を示しており、スナップショット以後のすべての変更は更新ストリームに含まれるようになります。このことを利用して、スロットが作られた際のデータベースの状態をSET TRANSACTION SNAPSHOTを使って読み込むことにより、新しいレプリカを作ることができます。このトランザクションは、その時点のデータベースの状態をダンプするために使用することができます。また、スロットに含まれるデータを使って、ダンプした後で行われた更新を失うことなくデータベースを更新できます。

スナップショットの作成はいつでも可能なわけではありません。とりわけ、ホットスタンバイに接続するときは失敗します。スナップショットのエクスポートが必要ないアプリケーションは、NOEXPORT_SNAPSHOTオプションを使ってスナップショットのエクスポートを抑止できます。

48.3. ストリームレプリケーションプロトコルインタフェース

コマンド

- `CREATE_REPLICATION_SLOT slot_name LOGICAL output_plugin`
- `DROP_REPLICATION_SLOT slot_name [WAIT]`
- `START_REPLICATION SLOT slot_name LOGICAL ...`

は、それぞれレプリケーションスロットに対して更新データを生成、削除、ストリームします。これらのコマンドは、レプリケーション接続でのみ使用できます。SQLでは使用できません。これらのコマンドの詳細については52.4を参照してください。

コマンドpg_recvlogicalを使ってストリーミングコネクション上のロジカルデコーディングを制御できます(このコマンドは内部で上記のコマンドを使っています)。

48.4. ロジカルデコーディングSQLインタフェース

ロジカルデコーディングのSQLレベルのAPIの詳細については、9.27.6を参照してください。

同期レプリケーション(26.2.8参照)は、ストリーミングレプリケーションによるレプリケーションスロット上でのみサポートされます。関数インタフェースおよびコアに対する追加のインタフェースでは同期レプリケーションをサポートしていません。

48.5. ロジカルデコーディング関連のシステムカタログ

pg_replication_slotsビューとpg_stat_replicationビューは、レプリケーションスロット、ストリーミングレプリケーションのコネクションのそれぞれの現在の状態に関する情報を提供します。

48.6. ロジカルデコーディングの出力プラグイン

PostgreSQLのソースコードのサブディレクトリ contrib/test_decoding にサンプル出力プラグインがあります。

48.6.1. 初期化関数

出力プラグインは、出力プラグインの名前をライブラリのベース名として持つ共有ライブラリを動的にロードすることによってロードされます。必要な出力プラグインコールバックを提供し、そのライブラリが実際に出力プラグインであることを示すために、_PG_output_plugin_initという名前の関数を作成しなければなりません。この関数には、各々のアクションに対応するコールバック関数へのポインタを持つ構造体が渡されます。

```
typedef struct OutputPluginCallbacks
{
    LogicalDecodeStartupCB startup_cb;
    LogicalDecodeBeginCB begin_cb;
    LogicalDecodeChangeCB change_cb;
    LogicalDecodeTruncateCB truncate_cb;
    LogicalDecodeCommitCB commit_cb;
    LogicalDecodeMessageCB message_cb;
    LogicalDecodeFilterByOriginCB filter_by_origin_cb;
    LogicalDecodeShutdownCB shutdown_cb;
} OutputPluginCallbacks;

typedef void (*LogicalOutputPluginInit) (struct OutputPluginCallbacks *cb);
```

コールバック関数のbegin_cb、change_cb、および、commit_cbは必須ですが、startup_cb、filter_by_origin_cb、truncate_cb、および、shutdown_cbは必須ではありません。truncate_cbが設定されていないけれども、TRUNCATEがデコードされることになった場合、この動作は無視されます。

48.6.2. 機能

更新データをデコード、整形、出力するために、出力関数を呼び出すことを含め、出力プラグインはバックエンドの通常のインフラストラクチャのほとんどを利用できます。テーブルは、initdbで作られ、pg_catalogス

キーマに含まれているか、以下のコマンドでユーザ定義のカタログテーブルであると印が付けられている限り、読み込み専用のアクセスが許可されます。

```
ALTER TABLE user_catalog_table SET (user_catalog_table = true);
CREATE TABLE another_catalog_table(data text) WITH (user_catalog_table = true);
```

トランザクションIDの割り当てが発生するような動作は許可されていません。そのような動作としては、テーブルへの書き込み、DDLの変更操作、pg_current_xact_id()の呼び出しなどがあります。

48.6.3. 出力モード

出力プラグインコールバックは、かなり自由な形式で消費者にデータを渡すことができます。SQLで変更データを見るような場合、任意のかたちでデータを返すことのできるデータ型(たとえばbytea)は扱いにくいです。出力プラグインがサーバエンコーディングのテキストデータのみを含むことにするには、起動コールバックで、OutputPluginOptions.output_typeにOUTPUT_PLUGIN_BINARY_OUTPUTではなく、OUTPUT_PLUGIN_TEXTUAL_OUTPUTを設定することによって宣言できます。この場合、textdatumが格納することができるように、すべてのデータはサーバエンコーディングでエンコードされていなければなりません。

48.6.4. 出力プラグインコールバック

出力プラグインには、必要に応じて発生した更新に関する通知が様々なコールバックを通じて送られます。

同時に実行されたトランザクションは、コミットした順番にデコードされます。指定したトランザクションに含まれる更新だけがbeginとcommitの間のコールバックによってデコードされます。明示的あるいは暗黙的にロールバックされたトランザクションは、決してデコードされません。成功したセーブポイントは、実行された順番にセーブポイントが実行されたトランザクションの中に折り込まれます。

> **注記**
>
> ディスクに安全に書きだされたトランザクションだけがデコードされます。そのため、synchronous_commitがoffの場合には、直後に呼び出されたpg_logical_slot_get_changes()がそのCOMMITをデコードしないことがあります。

48.6.4.1. 開始コールバック

ストリームに投入可能な更新の数に関係なく、レプリケーションスロットが作られるか、ストリームの変更がリクエストされた場合にオプションのstartup_cbコールバック呼び出されます。

```
typedef void (*LogicalDecodeStartupCB) (struct LogicalDecodingContext *ctx,
                                        OutputPluginOptions *options,
                                        bool is_init);
```

is_init パラメータは、レプリケーションスロットが作られる際にはtrue、それ以外ではfalseになります。optionsは、出力プラグインが書き込む以下の構造体を指します。

```
typedef struct OutputPluginOptions
{
    OutputPluginOutputType output_type;
    bool        receive_rewrites;
} OutputPluginOptions;
```

output_typeはOUTPUT_PLUGIN_TEXTUAL_OUTPUTかOUTPUT_PLUGIN_BINARY_OUTPUTのどちらかです。48.6.3も参照してください。receive_rewritesが真なら、何らかDDL操作時のヒープ書き換えで生じた変更に対して、出力プラグインも呼ばれます。これはDDLレプリケーションを処理するプラグインを対象としていますが、これらは特別な処理を必要とします。

開始コールバックでは、ctx->output_plugin_optionsで指定されるオプションを検証しましょう。出力プラグインが状態を持つ必要がある場合には、ctx->output_plugin_privateを利用できます。

48.6.4.2. 終了コールバック

以前アクティブだったレプリケーションスロットが使われなくなったら、いつでもshutdown_cbコールバックが呼び出され、出力プラグインのプライベートリソースが解放されます。スロットは削除される必要はありません。単にストリームが停止します。

```
typedef void (*LogicalDecodeShutdownCB) (struct LogicalDecodingContext *ctx);
```

48.6.4.3. トランザクション開始コールバック

必須であるbegin_cbコールバックは、コミットしたトランザクションの開始がデコードされる際に必ず呼び出されます。アボートしたトランザクションとその内容は決してデコードされません。

```
typedef void (*LogicalDecodeBeginCB) (struct LogicalDecodingContext *ctx,
                                      ReorderBufferTXN *txn);
```

txn引数は、コミット時のタイムスタンプやトランザクションIDなどのトランザクションに関するメタ情報を含みます。

48.6.4.4. トランザクション終了コールバック

必須であるcommit_cbコールバックは、トランザクションのコミットがデコードされる際に必ず呼び出されます。行が更新された場合は、それぞれの行に対してchange_cbコールバックが、commit_cbの前に呼び出されます。

```
typedef void (*LogicalDecodeCommitCB) (struct LogicalDecodingContext *ctx,
                                       ReorderBufferTXN *txn,
```

```
                                        XLogRecPtr commit_lsn);
```

48.6.4.5. 更新コールバック

トランザクション内のINSERT、UPDATE、DELETEの更新に対して、必須コールバックであるchange_cbが呼び出されます。元の更新コマンドが複数の行を一度に更新する場合は、それぞれの行に対してこのコールバックが呼び出されます。

```
typedef void (*LogicalDecodeChangeCB) (struct LogicalDecodingContext *ctx,
                                       ReorderBufferTXN *txn,
                                       Relation relation,
                                       ReorderBufferChange *change);
```

ctxとtxnは、begin_cb、commit_cbコールバックでは同じ内容になります。これに加えてrelationは行が属するリレーションを指定し、行の変更を記述するchangeパラメータが渡されます。

> **注記**
>
> unloggedテーブル(UNLOGGED参照)と(TEMPORARYまたはTEMP参照)以外のユーザ定義テーブルだけが、ロジカルデコーディングを使って更新データを取得できます。

48.6.4.6. TRUNCATEコールバック

truncate_cbコールバックは、TRUNCATEコマンドに対して呼ばれます。

```
typedef void (*LogicalDecodeTruncateCB) (struct LogicalDecodingContext *ctx,
                                         ReorderBufferTXN *txn,
                                         int nrelations,
                                         Relation relations[],
                                         ReorderBufferChange *change);
```

パラメータはchange_cbコールバックと似ています。しかしながら、外部キーで結びついたテーブル群のTRUNCATE動作は一緒に実行される必要があるため、このコールバックは単一リレーションではなく、リレーションの配列を受け取ります。詳しくはTRUNCATE文の説明を参照してください。

48.6.4.7. オリジンフィルターコールバック

オプションのfilter_by_origin_cbコールバックは、origin_idからリプレイされたデータがアウトプットプラグインの対象となるかどうかを判定するために呼び出されます。

```
typedef bool (*LogicalDecodeFilterByOriginCB) (struct LogicalDecodingContext *ctx,
                                               RepOriginId origin_id);
```

ctxパラメータは、他のコールバックと同じ内容を持ちます。オリジンの情報だけが得られます。渡されたノードで発生した変更が無関係であることを伝えるには、trueを返します。これにより、その変更は無視されることになります。無視されたトランザクション変更に関わる他のコールバックは呼び出されません。

これは、カスケード、あるいは双方向レプリケーションソリューションを実装する際に有用です。オリジンでフィルターすることにより、そのような構成で、同じ変更のレプリケーションが往復するのを防ぐことができます。トランザクションや変更もオリジンに関する情報を持っていますが、このコールバックでフィルターするほうがずっと効率的です。

48.6.4.8. 汎用メッセージコールバック

オプションのmessage_cbコールバックは、ロジカルデコーディングメッセージがデコードされる度に呼び出されます。

```
typedef void (*LogicalDecodeMessageCB) (struct LogicalDecodingContext *ctx,
                                        ReorderBufferTXN *txn,
                                        XLogRecPtr message_lsn,
                                        bool transactional,
                                        const char *prefix,
                                        Size message_size,
                                        const char *message);
```

txnパラメータは、コミット時のタイムスタンプとXIDのような、トランザクションに関するメタ情報を含んでいます。ただし、そのメッセージがトランザクション扱いではなく、メッセージをログしたトランザクションにXIDが割り当てられてない場合はNULLになることに注意してください。lsnは、メッセージに対応するWALの位置です。transactionalは、メッセージがトランザクションとして送られたものかどうかを表しています。prefixはnull終端された任意の接頭辞で、現在のプラグインが興味のあるメッセージを特定するために利用できます。最後に、messageパラメータは、大きさがmessage_sizeの、実際のメッセージを保持します。

出力プラグインが利用を考慮している接頭辞が一意になるように、特に注意を払ってください。拡張の名前か、出力プラグインの名前を使うのが良い場合が多いです。

48.6.5. 出力生成関数

begin_cb、commit_cb、change_cbコールバックにおいて、出力プラグインは実際にデータ出力するためにctx->outのStringInfo出力バッファに書き込みます。出力バッファに書き込む前に、OutputPluginPrepareWrite(ctx, last_write)を呼び出します。また、書き込みバッファにデータを書き終えたら、OutputPluginWrite(ctx, last_write)を呼び出してデータの書き込みを実施します。last_write引数により、その書き込みがコールバックの最終的な書き込みであるかどうかを指定します。

以下の例では、出力プラグインにおいて消費者に向けてデータを出力する方法を示します。

```
OutputPluginPrepareWrite(ctx, true);
appendStringInfo(ctx->out, "BEGIN %u", txn->xid);
OutputPluginWrite(ctx, true);
```

48.7. ロジカルデコーディング出力ライタ

ロジカルデコーディングに、別な出力方法を追加することもできます。src/backend/replication/logical/logicalfuncs.cを参照してください。基本的に、3つの関数を用意する必要があります。WALを読む関数、出力データの書き込みを準備する関数、それに出力データを書き込む関数です。(48.6.5参照)。

48.8. ロジカルデコーディングにおける同期レプリケーションのサポート

ストリーミングレプリケーションにおける同期レプリケーションと同じユーザインタフェースで、ロジカルデコーディングを使って同期レプリケーションソリューションを構築することができます。そのためには、ストリーミングレプリケーションインタフェース(48.3参照)を使ってデータをストリーム出力します。ストリーミングレプリケーションクライアントが行っているのと同じように、状態の更新(52.4参照)メッセージを送信しなければなりません。

> **注記**
>
> synchronous_standby_namesがサーバ全体に適用されるのに対し、ロジカルデコーディングを通じて変更データを受け取る同期レプリカは、単一のデータベースのスコープの範囲で動作します。このことにより、複数のデータベースが同時に使用される環境では、ロジカルデコーディングを使った同期レプリケーションはうまく動きません。

第49章 レプリケーション進捗の追跡

レプリケーション起点(replication origins)は、ロジカルデコーディングの上に、ロジカルレプリケーションソリューションを実装しやすくすることを意図しています。以下の2つの良くある問題に対して解決の方策を提供します。

- レプリケーション進捗をどうやって安全に追跡するか
- たとえば双方向レプリケーションにおけるループを回避するために、起点の行ごとに、いかにしてレプリケーションの挙動を変えるか

レプリケーション起点は、名前とOIDから構成されます。システム中で起点を参照する際に使われる名前は、任意のtextです。その名前は、たとえばレプリケーションソリューションの名前を接頭辞にすることにより、別々のレプリケーションソリューションによって作成されたレプリケーションオリジンが衝突することがないように使われるべきです。oidは、空間効率が重要な場合に、長い名前を格納することを避けたいときにのみ使用します。oidはシステム間で共有すべきものではありません。

レプリケーション起点はpg_replication_origin_create()で作成し、pg_replication_origin_drop()で削除し、pg_replication_originシステムカタログを使って参照します。

レプリケーションソリューションを構築する際に無視できない問題は、どうやってリプレイの進捗を安全に追跡するか、ということです。(訳注: ログを)適用するプロセス、あるいはシステム全体が死んだ時に、どこまでデータのレプリケーションが成功したかを見つけることができなければなりません。トランザクションのリプレイの度にテーブルの行を更新するような素朴なソリューションは、実行時のオーバヘッドとデータベースの肥大化問題を起こします。

レプリケーション起点のインフラを使用することにより、あるセッションに対してリモートノードからリプレイしていることの目印を付けることができます。(pg_replication_origin_session_setup()を使います) また、pg_replication_origin_xact_setup()を使ってすべてのソーストランザクションに対してトランザクション単位でLSNとタイムスタンプを記録するように設定することができます。終了後は、クラッシュに対して安全な方法で、レプリケーションの進捗は永続的に記録されます。すべてのレプリケーション起点のリプレイの進捗は、pg_replication_origin_statusビューで参照できます。たとえばレプリケーションの再開の際などには、個々の起点の進捗を、pg_replication_origin_progress()で参照できます。現在のセッションに起点が設定されている場合は、pg_replication_origin_session_progress()を使用します。

厳密に一つのシステムから別の一つのシステムにレプリケーションする以上のより複雑なレプリケーションのトポロジでは、リプレイされた行を再びレプリケーションするのを避けるのが難しいという別な問題が発生するかもしれません。これにより、レプリケーションの巡回と、非効率性の両方が発生するかもしれません。レプリケーション起点には、この問題を認識し、避けるためのオプションの機構があります。前段で言及した関数を使うと、出力プラグインコールバック(48.6参照)に渡されるすべての更新とトランザクションに、セッションを生成しているレプリケーション起点がタグ付けされます。これにより、出力プラグインの中でそれらの扱いを分けることができます。たとえばローカルに起因する行以外はすべて無視するような場合です。また追加で、ソースに基づくロジカルデコーディングの変更ストリームをフィルターするためにfilter_by_origin_cbコールバックを使うことができます。これは柔軟ではありませんが、アウトプットプラグインを通してフィルタリングするのはずっと効率的です。

パート VI. リファレンス

このリファレンス内の項目では、対応する主題に関する信頼できる、完全な、形式の整った要約を、適切な長さで提供することが意図されています。PostgreSQLの使用に関する詳細は、物語、チュートリアル、例形式として、本書の他の部分にて説明されています。各リファレンスページで挙げたクロスリファレンスを参照してください。

このリファレンスは伝統的な「man」ページとしても入手できます。

目次

SQLコマンド

ここにはPostgreSQLでサポートされるSQLコマンドのリファレンス情報があります。「SQL」とは一般に言語を意味し、各コマンドの標準への準拠や互換性に関する情報がそれぞれの対応するリファレンスページから分かります。

目次

ABORT

ABORT — 現在のトランザクションをアボートする

概要

```
ABORT [ WORK | TRANSACTION ] [ AND [ NO ] CHAIN ]
```

説明

ABORTは現在のトランザクションをロールバックし、そのトランザクションで行われた全ての更新を廃棄します。このコマンドの動作は標準SQLのROLLBACKコマンドと同一であり、歴史的な理由のためだけに存在しています。

パラメータ

WORK
TRANSACTION

省略可能なキーワードです。何も効果がありません。

AND CHAIN

AND CHAINが指定されていれば、新しいトランザクションは、直前に終了したものと同じトランザクションの特性(SET TRANSACTIONを参照してください)で即時に開始されます。そうでなければ、新しいトランザクションは開始されません。

注釈

トランザクションを正常に終了させる場合はCOMMITを使用してください。

トランザクションブロックの外部でABORTを発行すると警告が発生しますが、それ以外は何の効果もありません。

例

全ての変更をアボートします。

```
ABORT;
```

互換性

このコマンドはPostgreSQLの拡張で、歴史的な理由で存在します。ROLLBACKは、このコマンドと等価な標準SQLコマンドです。

関連項目

BEGIN, COMMIT, ROLLBACK

ALTER AGGREGATE

ALTER AGGREGATE — 集約関数定義を変更する

概要

```
ALTER AGGREGATE name ( aggregate_signature ) RENAME TO new_name
ALTER AGGREGATE name ( aggregate_signature )
                OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER AGGREGATE name ( aggregate_signature ) SET SCHEMA new_schema


ここでaggregate_signatureは以下の通りです。

* |
[ argmode ] [ argname ] argtype [ , ... ] |
[ [ argmode ] [ argname ] argtype [ , ... ] ] ORDER BY [ argmode ] [ argname ] argtype [ , ... ]
```

説明

ALTER AGGREGATEは集約関数の定義を変更します。

ALTER AGGREGATEを使用するには集約関数の所有者でなければなりません。集約関数のスキーマを変更するには、新しいスキーマにおけるCREATE権限も必要です。所有者を変更するには、直接または間接的に新しい所有者ロールのメンバでなければなりません。また、そのロールは集約関数のスキーマにおいてCREATE権限を持たなければなりません。(この制限により、集約関数の削除と再作成を行ってもできないことが、所有者の変更によってもできないようにしています。しかし、スーパーユーザはすべての集約関数の所有者を変更することができます。)

パラメータ

name

既存の集約関数の名前です(スキーマ修飾名も可)。

argmode

引数のモードで、INあるいはVARIADICです。省略された時のデフォルトはINです。

argname

引数の名前です。ALTER AGGREGATEは実際には引数の名前を無視することに注意してください。これは、集約関数の本体を特定するのに必要になるのは、引数のデータ型だけだからです。

argtype

集約関数が演算する入力データ型です。引数を持たない集約関数を参照するには、引数指定のリストに*と記載してください。順序集約関数を参照するには、直接引数の指定と集約引数の指定の間にORDER BYと書いてください。

new_name

新しい集約関数の名前です。

new_owner

新しい集約関数の所有者です。

new_schema

集約関数の新しいスキーマです。

注釈

順序集約関数を参照するときの推奨される構文は、CREATE AGGREGATEと同じ形式で、直接引数の指定と集約引数の指定の間にORDER BYと書くことです。しかし、ORDER BYを省略して、単に直接引数と集約引数を1つのリストにまとめても動作します。VARIADIC "any"が直接引数のリストと集約引数のリストの両方に対して使われていた場合、この省略形式ではVARIADIC "any"を1度だけ書いてください。

例

integer型用のmyavg集約関数の名前をmy_averageに変更します。

```
ALTER AGGREGATE myavg(integer) RENAME TO my_average;
```

integer型用のmyavg集約関数の所有者をjoeに変更します。

```
ALTER AGGREGATE myavg(integer) OWNER TO joe;
```

直接引数がfloat8型、集約引数がinteger型の順序集約関数mypercentileをmyschemaスキーマに移動します。

```
ALTER AGGREGATE mypercentile(float8 ORDER BY integer) SET SCHEMA myschema;
```

以下も動作します。

```
ALTER AGGREGATE mypercentile(float8, integer) SET SCHEMA myschema;
```

互換性

標準SQLにはALTER AGGREGATE文はありません。

関連項目

CREATE AGGREGATE, DROP AGGREGATE

ALTER COLLATION

ALTER COLLATION — 照合順序の定義を変更する

概要

```
ALTER COLLATION name REFRESH VERSION

ALTER COLLATION name RENAME TO new_name
ALTER COLLATION name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER COLLATION name SET SCHEMA new_schema
```

説明

ALTER COLLATIONは照合順序の定義を変更します。

ALTER COLLATIONを使用するためには照合順序を所有していなければなりません。所有者を変更するためには、新しい所有ロールの直接あるいは間接的なメンバでなければならず、かつ、そのロールが照合順序のスキーマにおけるCREATE権限を持たなければなりません（この制限により、照合順序の削除と再作成を行ってもできないことが所有者の変更で行えないようにします。ただし、スーパーユーザはすべての照合順序の所有者を変更することができます）。

パラメータ

name

既存の照合順序の名前（スキーマ修飾可）です。

new_name

照合順序の新しい名前です。

new_owner

照合順序の新しい所有者です。

new_schema

照合順序の新しいスキーマです。

REFRESH VERSION

照合順序のバージョンを更新します。以下のNotesを参照してください。

注釈

ICUライブラリが提供する照合順序を使う場合、照合順序オブジェクトを作成する時に、照合器のICU固有のバージョンがシステムカタログに記録されます。照合順序が使われるとき、現在のバージョンと記録されているバージョンが比較され、不適合の場合は以下の例のように警告が発行されます。

```
WARNING:  collation "xx-x-icu" has version mismatch
DETAIL:  The collation in the database was created using version 1.2.3.4, but the operating
 system provides version 2.3.4.5.
HINT:  Rebuild all objects affected by this collation and run ALTER COLLATION pg_catalog."xx-x-
icu" REFRESH VERSION, or build PostgreSQL with the right library version.
```

データベースシステムは、保存されたオブジェクトが一定のソート順になっていることに依存しているため、照合順序を変更するとインデックスが破損するなどといった問題につながります。一般的にこれは避けるべきですが、pg_upgradeを使って新しいバージョンのICUライブラリとリンクされたサーバのバイナリへとアップグレードする場合など、仕方のない場合もあります。これが発生する場合は、照合順序に依存するすべてのオブジェクトを、例えばREINDEXを使って再構築します。これがされれば、照合順序のバージョンをコマンドALTER COLLATION ... REFRESH VERSIONを使って更新できます。これにより、システムカタログが更新されて照合器の現在のバージョンが記録され、警告が出なくなります。これは、影響を受けるすべてのオブジェクトが正しく再構築されたかどうかを実際に確認するわけではないことに注意してください。

libcが提供する照合順序を使い、PostgreSQLがGNU Cライブラリで構築されている場合、照合順序としてはCライブラリのものが使われます。照合順序の定義は、典型的にはGNU Cライブラリのリリースで変わるだけですので、これは破損に対していくらかの防御を提供しますが、完全には信頼できるものではありません。

今のところ、データベースのデフォルトの照合順序に対するバージョン追跡はありません。

以下の問い合わせを使って、現在のデータベース内の更新が必要なすべての照合順序と、それに依存するオブジェクトを特定することができます。

```
SELECT pg_describe_object(refclassid, refobjid, refobjsubid) AS "Collation",
       pg_describe_object(classid, objid, objsubid) AS "Object"
  FROM pg_depend d JOIN pg_collation c
       ON refclassid = 'pg_collation'::regclass AND refobjid = c.oid
  WHERE c.collversion <> pg_collation_actual_version(c.oid)
  ORDER BY 1, 2;
```

例

照合順序de_DEの名前をgermanに変更します。

```
ALTER COLLATION "de_DE" RENAME TO german;
```

照合順序en_USの所有者をjoeに変更します。

```
ALTER COLLATION "en_US" OWNER TO joe;
```

互換性

標準SQLにはALTER COLLATION文はありません。

関連項目

CREATE COLLATION, DROP COLLATION

ALTER CONVERSION

ALTER CONVERSION — 変換の定義を変更する

概要

```
ALTER CONVERSION name RENAME TO new_name
ALTER CONVERSION name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER CONVERSION name SET SCHEMA new_schema
```

説明

ALTER CONVERSIONは変換の定義を変更します。

ALTER CONVERSIONを使用するには変換の所有者でなければなりません。所有者を変更するには、直接または間接的に新しいロールのメンバでなければなりません。また、そのロールは変換のスキーマにおいてCREATE権限を持たなければなりません。(この制限により、所有者の変更に伴い変換の削除や再作成ができなくなるといった問題が起こらないようになります。しかし、スーパーユーザはすべての変換の所有者を変更することができます。)

パラメータ

name

既存の変換の名前です(スキーマ修飾名も可)。

new_name

変換の新しい名前です。

new_owner

変換の新しい所有者です。

new_schema

変換の新しいスキーマです。

例

変換iso_8859_1_to_utf8の名前をlatin1_to_unicodeへ変更します。

```
ALTER CONVERSION iso_8859_1_to_utf8 RENAME TO latin1_to_unicode;
```

変換iso_8859_1_to_utf8の所有者をjoeに変更します。

```
ALTER CONVERSION iso_8859_1_to_utf8 OWNER TO joe;
```

互換性

標準SQLにはALTER CONVERSION文はありません。

関連項目

CREATE CONVERSION, DROP CONVERSION

ALTER DATABASE

ALTER DATABASE — データベースを変更する

概要

```
ALTER DATABASE name [ [ WITH ] option [ ... ] ]


ここでoptionは以下の通りです。

    ALLOW_CONNECTIONS allowconn
    CONNECTION LIMIT connlimit
    IS_TEMPLATE istemplate

ALTER DATABASE name RENAME TO new_name

ALTER DATABASE name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }

ALTER DATABASE name SET TABLESPACE new_tablespace

ALTER DATABASE name SET configuration_parameter { TO | = } { value | DEFAULT }
ALTER DATABASE name SET configuration_parameter FROM CURRENT
ALTER DATABASE name RESET configuration_parameter
ALTER DATABASE name RESET ALL
```

説明

ALTER DATABASEはデータベースの属性を変更します。

最初の構文はデータベース毎の設定を変更します。(詳細は後述します。) データベース所有者とスーパーユーザのみがこの設定を変更することができます。

2番目の構文は、データベースの名前を変更します。データベースの名前を変更できるのは、データベースの所有者またはスーパーユーザのみです。ただし、スーパーユーザではない所有者はCREATEDB権限を所有していなければなりません。現在のデータベースの名前を変更することはできません(必要ならば、別のデータベースに接続してください)。

3番目の構文は、データベースの所有者を変更します。所有者を変更するにはデータベースを所有し、かつ、新しい所有者ロールの間接的あるいは直接的なメンバでなければなりません。さらに、CREATEDB権限も持たなければなりません。(スーパーユーザはこれらの権限を自動的に持っていることに注意してください。)

4番目の構文は、データベースのデフォルトのテーブル空間を変更します。データベースの所有者またはスーパーユーザのみがこれを行うことができます。また、新しいテーブル空間における作成権限を持つ必要

があります。このコマンドはデータベースの古いデフォルトのテーブル空間にあるテーブルまたはインデックスを新しいテーブル空間に物理的にすべて移動します。新しいデフォルトのテーブル空間は、このデータベースについては空でなければならず、誰もデータベースに接続されていてはなりません。デフォルト以外のテーブル空間にあるテーブルまたはインデックスには影響がありません。

残りの構文は、PostgreSQLデータベースにおける実行時設定変数のセッションのデフォルト値を変更します。コマンド実行後にデータベースで開始される新規セッションにおいて、指定された値がデフォルト値になります。データベース固有のデフォルト値は、postgresql.confファイルに記述されている設定やpostgresコマンドラインから受け取った設定よりも優先します。データベースにおけるセッションのデフォルト値を変更できるのは、データベースの所有者またはスーパーユーザのみです。この方法では設定できない変数や、スーパーユーザのみが設定できる変数も存在します。

パラメータ

name

属性変更の対象となるデータベースの名前です。

allowconn

falseの場合、このデータベースには誰も接続できません。

connlimit

データベースへの最大同時接続数です。-1は無制限を意味します。

istemplate

trueの場合、CREATEDB権限のあるユーザは誰でも、このデータベースを複製できます。falseの場合、スーパーユーザー、あるいはデータベースの所有者のみが、このデータベースを複製できます。

new_name

新しいデータベース名です。

new_owner

新しいデータベースの所有者です。

new_tablespace

新しいデータベースのデフォルトのテーブル空間です。

この構文のコマンドはトランザクションブロックの内側で実行することはできません。

configuration_parameter
value

指定した設定パラメータについて、データベースのセッションのデフォルト値を指定した値に設定します。valueがDEFAULTの場合、あるいは等価なRESETが使用されている場合、データベース固有の設定は無効になり、新しいセッションではシステム全体のデフォルト設定が継承されます。全てのデータベース

固有の設定をクリアするには、RESET ALLを使用してください。SET FROM CURRENTは、データベース固有の値としてセッションにおけるパラメータの現在値を保管します。

設定可能なパラメータ名とその値に関する詳細についてはSETおよび第19章を参照してください。

注釈

データベースではなく特定のロールにセッションのデフォルト値を関連付けることもできます。ALTER ROLEを参照してください。設定が競合する場合には、ロール固有の設定が、データベース固有の設定を上書きします。

例

データベースtest内のインデックススキャンをデフォルトで無効にします。

```
ALTER DATABASE test SET enable_indexscan TO off;
```

互換性

ALTER DATABASE文はPostgreSQLの拡張です。

関連項目

CREATE DATABASE, DROP DATABASE, SET, CREATE TABLESPACE

ALTER DEFAULT PRIVILEGES

ALTER DEFAULT PRIVILEGES — デフォルトのアクセス権限を定義する

概要

```
ALTER DEFAULT PRIVILEGES
    [ FOR { ROLE | USER } target_role [, ...] ]
    [ IN SCHEMA schema_name [, ...] ]
    abbreviated_grant_or_revoke


ここでabbreviated_grant_or_revokeは以下のいずれかです。

GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }
    [, ...] | ALL [ PRIVILEGES ] }
    ON TABLES
    TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { { USAGE | SELECT | UPDATE }
    [, ...] | ALL [ PRIVILEGES ] }
    ON SEQUENCES
    TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { EXECUTE | ALL [ PRIVILEGES ] }
    ON { FUNCTIONS | ROUTINES }
    TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON TYPES
    TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | CREATE | ALL [ PRIVILEGES ] }
    ON SCHEMAS
    TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

REVOKE [ GRANT OPTION FOR ]
    { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }
    [, ...] | ALL [ PRIVILEGES ] }
    ON TABLES
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
```

```
    { { USAGE | SELECT | UPDATE }
    [, ...] | ALL [ PRIVILEGES ] }
    ON SEQUENCES
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { EXECUTE | ALL [ PRIVILEGES ] }
    ON { FUNCTIONS | ROUTINES }
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON TYPES
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | CREATE | ALL [ PRIVILEGES ] }
    ON SCHEMAS
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]
```

説明

ALTER DEFAULT PRIVILEGESにより今後作成されるオブジェクトに適用される権限を設定することができます。(既存のオブジェクトに割り当てられている権限には影響しません。) 現時点ではスキーマ、テーブル(ビュー、外部テーブルを含む)、シーケンス、関数、型(ドメインを含む)用の権限のみを変更可能です。このコマンドでは、関数は集約とプロシージャを含みます。FUNCTIONSとROUTINESは、このコマンドでは同じです。(ROUTINESが、関数とプロシージャを合わせた標準の用語ですので、今後は好まれます。PostgreSQLの以前のリリースではFUNCTIONSだけが許されていました。関数とプロシージャに対して別々にデフォルト権限を設定することはできません。)

ユーザ自身あるいは、ユーザがメンバとして属するロールにより作成されるオブジェクトについてのみ、デフォルト権限を変更することができます。権限は大域的に(つまり現在のデータベース内に作成されるすべてのオブジェクトに対して)設定することも、指定したスキーマ内に作成されるオブジェクトのみに対して設定することもできます。

5.7の説明にある通り、どの種類のオブジェクトについてもデフォルト権限は通常、オブジェクト所有者に対して付与可能な権限すべてを付与します。さらに、PUBLICに対して一部の権限を付与することがあります。しかしALTER DEFAULT PRIVILEGESを用いて大域デフォルト権限を変更することで、この動作を変更できます。

スキーマ単位で指定されるデフォルト権限は、大域的な個々の種類のオブジェクト用のデフォルト権限に追加されます。これは、スキーマ単位の権限が(デフォルトであれ、以前のスキーマを指定しないALTER DEFAULT PRIVILEGESコマンドよってであれ)大域的に付与されているのなら、それを取り消せないことを意味します。スキーマ単位のREVOKEは、以前のスキーマ単位のGRANTの効果を取り消すのにのみ有用です。

パラメータ

target_role

現在のロールがメンバとして属する、既存のロールの名前です。FOR ROLEを省略した場合、現在のロールとみなされます。

schema_name

既存のスキーマの名前です。指定された場合、そのスキーマ内で後に作成されるオブジェクトに対するデフォルト権限が変更されます。IN SCHEMAを省略した場合、大域的なデフォルト権限が変更されます。スキーマはネストできないため、スキーマに対する権限を設定する場合にはIN SCHEMAを指定できません。

role_name

権限を付与または取り消す、既存のロールの名前です。このパラメータ、およびabbreviated_grant_or_revoke内の他のパラメータは、GRANTやREVOKEの説明通りに動作します。ただし、指定したオブジェクトではなくオブジェクトクラス全体に対して権限を設定する点が異なります。

注釈

デフォルト権限としてすでに割り当てられている情報を入手するためにはpsqlの\ddpコマンドを使用してください。権限の表示の意味は、5.7の\dpの説明と同じです。

デフォルト権限を変更したロールを削除したい場合、デフォルト権限の項目を取り除くために、そのデフォルト権限の変更を元に戻すかDROP OWNED BYを使用する必要があります。

例

スキーマmyschema内に今後作成されるすべてのテーブル（およびビュー）に対して、全員にSELECT権限を付与します。また、ロールwebuserにはそれらに挿入できるようにします。

```
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema GRANT SELECT ON TABLES TO PUBLIC;
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema GRANT INSERT ON TABLES TO webuser;
```

今後作成されるテーブルが通常以外の権限を持たないように、上を元に戻します。

```
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema REVOKE SELECT ON TABLES FROM PUBLIC;
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema REVOKE INSERT ON TABLES FROM webuser;
```

今後adminロールにより作成されるすべての関数について、通常関数に付与される、全員に対するEXECUTE権限を取り除きます。

```
ALTER DEFAULT PRIVILEGES FOR ROLE admin REVOKE EXECUTE ON FUNCTIONS FROM PUBLIC;
```

しかしながら、スキーマ1つに限定されたコマンドではそのような効果は達成できないということに注意してください。対応するGRANTを取り消さない限り、以下のコマンドは効果がありません。

```
ALTER DEFAULT PRIVILEGES IN SCHEMA public REVOKE EXECUTE ON FUNCTIONS FROM PUBLIC;
```

スキーマ単位のデフォルト権限は、大域的な設定に権限を追加できるだけで、付与された権限を削除することはできないからです。

互換性

標準SQLにはALTER DEFAULT PRIVILEGES文はありません。

関連項目

GRANT, REVOKE

ALTER DOMAIN

ALTER DOMAIN —ドメイン定義を変更する

概要

```
ALTER DOMAIN name
    { SET DEFAULT expression | DROP DEFAULT }
ALTER DOMAIN name
    { SET | DROP } NOT NULL
ALTER DOMAIN name
    ADD domain_constraint [ NOT VALID ]
ALTER DOMAIN name
    DROP CONSTRAINT [ IF EXISTS ] constraint_name [ RESTRICT | CASCADE ]
ALTER DOMAIN name
     RENAME CONSTRAINT constraint_name TO new_constraint_name
ALTER DOMAIN name
    VALIDATE CONSTRAINT constraint_name
ALTER DOMAIN name
    OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER DOMAIN name
    RENAME TO new_name
ALTER DOMAIN name
    SET SCHEMA new_schema
```

説明

ALTER DOMAINは既存ドメインの定義を変更します。以下に示す副構文があります。

SET/DROP DEFAULT

この構文はドメインのデフォルト値の設定または削除を行います。指定したデフォルト値は、その後のINSERTコマンドのみに適用されることに注意してください。そのドメインを使用したテーブルの既存の行には影響を与えません。

SET/DROP NOT NULL

この構文はドメインがNULL値を持つことができるかどうかを変更します。SET NOT NULLを実行できるのは、ドメインを使用する列にNULL値が含まれていない場合のみです。

ADD domain_constraint [NOT VALID]

この構文はCREATE DOMAINと同一の構文を使って、ドメインに新しい制約を付与します。新しい制約がドメインに追加された時、そのドメインを使用するすべての列が新しく追加された制約に対して検査されます。NOT VALIDオプションを使用して新しい制約を追加することでこれらの検査を抑制することがで

きます。ALTER DOMAIN ... VALIDATE CONSTRAINTを使用することで、後で制約を有効にすることができます。新しく挿入または更新される行については、NOT VALIDと印が付いていたとしても、常にすべての制約が検査されます。NOT VALIDはCHECK制約でのみ受け付けられます。

DROP CONSTRAINT [IF EXISTS]

この構文はドメイン上の制約を削除します。IF EXISTSが指定された場合、制約が存在しなくてもエラーになりません。この場合は代わりに注意メッセージが発生します。

RENAME CONSTRAINT

この構文はドメイン上の制約名を変更します。

VALIDATE CONSTRAINT

この構文は、以前にNOT VALIDとして追加された制約を検証します。つまり、そのドメイン型のテーブル列の値すべてが指定された制約を満たすかどうかを検証します。

OWNER

この構文はドメインの所有者を指定したユーザに変更します。

RENAME

この構文はドメインの名前を変更します。

SET SCHEMA

この構文はドメインのスキーマを変更します。ドメインに関連する制約もすべて新しいスキーマに移動します。

ALTER DOMAINを使用するにはドメインを所有していなければなりません。ドメインのスキーマを変更するには、新しいスキーマにおけるCREATE権限を持たなければなりません。所有者を変更するには、直接または間接的に新しいロールのメンバでなければなりません。また、そのロールはドメインのスキーマにおいてCREATEを持たなければなりません。(この制限により、所有者の変更により、ドメインの削除と再作成でできないことは何もできないようにしています。しかし、スーパーユーザはすべてのドメインの所有者を変更することができます。)

パラメータ

name

変更対象となる既存のドメインの名前です(スキーマ修飾名も可)。

domain_constraint

ドメイン用の新しいドメイン制約です。

constraint_name

削除または名前を変更する既存の制約名です。

NOT VALID

既存の格納されたデータについて制約の妥当性を検証しません。

CASCADE

その制約に依存するオブジェクトを自動的に削除し、さらにそれらのオブジェクトに依存するすべてのオブジェクトも削除します（5.14参照）。

RESTRICT

依存するオブジェクトがある場合、制約の削除要求を拒否します。これがデフォルトの動作です。

new_name

ドメインの新しい名前です。

new_constraint_name

制約の新しい名前です。

new_owner

ドメインの新しい所有者となるユーザの名前です。

new_schema

ドメインの新しいスキーマです。

注釈

ALTER DOMAIN ADD CONSTRAINTは、既存の格納されたデータが新しい制約を満たすか検証しようとしますが、この確認は万全なものではありません。このコマンドが新しく挿入または更新されてまだコミットされていないテーブル行を「見る」ことはできないからです。同時に実行される操作が不正なデータを挿入する危険があり、処理方法がNOT VALIDオプションを使った制約を追加することであるなら、そのコマンドをコミットして、そのコミットよりも前に開始したトランザクションがすべて完了するのを待ってから、制約に違反するデータを探すためにALTER DOMAIN VALIDATE CONSTRAINTを発行してください。制約が一度コミットされれば、新しいトランザクションはすべてドメイン型の新しい値に対してその制約を強制していることが保証されますので、この方法は信頼できます。

今のところ、ALTER DOMAIN ADD CONSTRAINT、ALTER DOMAIN VALIDATE CONSTRAINTおよびALTER DOMAIN SET NOT NULLは、指定ドメインまたは任意の派生ドメインがデータベース内のいずれかのテーブルのコンテナ型の列（複合型、配列型もしくは範囲型の列）で使用されていた場合、失敗します。これは将来的には、こうした入れ子になった値に対する新しい制約を検証できるように改良されるべきです。

例

ドメインにNOT NULL制約を付与します。

```
ALTER DOMAIN zipcode SET NOT NULL;
```

ドメインからNOT NULL制約を削除します。

```
ALTER DOMAIN zipcode DROP NOT NULL;
```

ドメインにCHECK制約を付与します。

```
ALTER DOMAIN zipcode ADD CONSTRAINT zipchk CHECK (char_length(VALUE) = 5);
```

ドメインからCHECK制約を削除します。

```
ALTER DOMAIN zipcode DROP CONSTRAINT zipchk;
```

ドメイン上の検査制約の名前を変更します。

```
ALTER DOMAIN zipcode RENAME CONSTRAINT zipchk TO zip_check;
```

ドメインを新しいスキーマに移動します。

```
ALTER DOMAIN zipcode SET SCHEMA customers;
```

互換性

ALTER DOMAIN文は標準SQLに準拠しています。ただし、OWNER、RENAME、SET SCHEMA、VALIDATE CONSTRAINT構文は例外で、PostgreSQLの拡張です。ADD CONSTRAINT構文のNOT VALID句もPostgreSQLの拡張です。

関連項目

CREATE DOMAIN, DROP DOMAIN

ALTER EVENT TRIGGER

ALTER EVENT TRIGGER — イベントトリガの定義を変更する

概要

```
ALTER EVENT TRIGGER name DISABLE
ALTER EVENT TRIGGER name ENABLE [ REPLICA | ALWAYS ]
ALTER EVENT TRIGGER name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER EVENT TRIGGER name RENAME TO new_name
```

説明

ALTER EVENT TRIGGERは既存のイベントトリガの属性を変更します。

イベントトリガを変更するためにはスーパーユーザでなければなりません。

パラメータ

name

変更する既存のトリガの名前です。

new_owner

イベントトリガの新しい所有者となるユーザの名前です。

new_name

イベントトリガの新しい名前です。

DISABLE/ENABLE [REPLICA | ALWAYS] TRIGGER

この構文はイベントトリガの発行処理を設定します。無効化されたトリガはまだシステムで認識されていますが、きっかけとなるイベントが起きたとしても実行されません。session_replication_roleも参照してください。

互換性

標準SQLにはALTER EVENT TRIGGER文はありません。

関連項目

CREATE EVENT TRIGGER, DROP EVENT TRIGGER

ALTER EXTENSION

ALTER EXTENSION — 拡張の定義を変更する

概要

```
ALTER EXTENSION name UPDATE [ TO new_version ]
ALTER EXTENSION name SET SCHEMA new_schema
ALTER EXTENSION name ADD member_object
ALTER EXTENSION name DROP member_object


ここでmember_objectは以下の通りです。

  ACCESS METHOD object_name |
  AGGREGATE aggregate_name ( aggregate_signature ) |
  CAST (source_type AS target_type) |
  COLLATION object_name |
  CONVERSION object_name |
  DOMAIN object_name |
  EVENT TRIGGER object_name |
  FOREIGN DATA WRAPPER object_name |
  FOREIGN TABLE object_name |
  FUNCTION function_name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ] |
  MATERIALIZED VIEW object_name |
  OPERATOR operator_name (left_type, right_type) |
  OPERATOR CLASS object_name USING index_method |
  OPERATOR FAMILY object_name USING index_method |
  [ PROCEDURAL ] LANGUAGE object_name |
  PROCEDURE procedure_name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ] |
  ROUTINE routine_name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ] |
  SCHEMA object_name |
  SEQUENCE object_name |
  SERVER object_name |
  TABLE object_name |
  TEXT SEARCH CONFIGURATION object_name |
  TEXT SEARCH DICTIONARY object_name |
  TEXT SEARCH PARSER object_name |
  TEXT SEARCH TEMPLATE object_name |
  TRANSFORM FOR type_name LANGUAGE lang_name |
  TYPE object_name |
  VIEW object_name
```

またaggregate_signatureは以下の通りです。

```
* |
[ argmode ] [ argname ] argtype [ , ... ] |
[ [ argmode ] [ argname ] argtype [ , ... ] ] ORDER BY [ argmode ] [ argname ] argtype [ , ... ]
```

説明

ALTER EXTENSIONはインストールされた拡張の定義を変更します。複数の副構文があります。

UPDATE

この構文は拡張を新しいバージョンに更新します。拡張は、現在インストールされているバージョンから要求するバージョンに変更することができる、適切な更新スクリプト（またはスクリプト群）を提供しなければなりません。

SET SCHEMA

この構文は拡張のオブジェクトを別のスキーマに移動します。このコマンドを成功させるためには、拡張は再配置可能でなければなりません。

ADD member_object

この構文は既存のオブジェクトを拡張に追加します。これは主に拡張の更新スクリプトで有用です。オブジェクトはその後拡張のメンバとして扱われます。特に、オブジェクトの削除は拡張の削除によってのみ可能です。

DROP member_object

この構文は拡張からメンバオブジェクトを削除します。これは主に拡張の更新スクリプトで有用です。オブジェクトは削除されません。拡張との関連がなくなるだけです。

これらの操作の詳細については37.17を参照してください。

ALTER EXTENSIONを使用するためには拡張の所有者でなければなりません。ADD/DROP構文では追加されるオブジェクトまたは削除されるオブジェクトの所有者でもなければなりません。

パラメータ

name

インストールされた拡張の名前です。

new_version

更新したい新しい拡張のバージョンです。これは識別子または文字列リテラルのいずれかで記述することができます。指定がない場合、ALTER EXTENSION UPDATEは拡張の制御ファイル内でデフォルトバージョンとして示されるものへの更新を試行します。

new_schema

拡張の新しいスキーマです。

object_name
aggregate_name
function_name
operator_name
procedure_name
routine_name

拡張に追加する、または、拡張から削除するオブジェクトの名前です。テーブル、集約、ドメイン、外部テーブル、関数、演算子、演算子クラス、演算子族、プロシージャ、ルーチン、シーケンス、全文検索オブジェクト、型、ビューの名前はスキーマ修飾可能です。

source_type

キャストの変換元データ型の名前です。

target_type

キャストの変換先データ型の名前です。

argmode

関数、プロシージャ、または集約の引数のモードでIN、OUT、INOUT、VARIADICのいずれかです。省略時のデフォルトはINです。関数を識別するためには入力引数だけが必要ですので、実際のところALTER EXTENSIONはOUT引数を考慮しないことに注意してください。このためIN、INOUTおよびVARIADIC引数を列挙するだけで十分です。

argname

関数、プロシージャ、または集約の引数の名前です。関数を識別するためには入力引数だけが必要ですので、実際のところALTER EXTENSIONは引数名を考慮しないことに注意してください。

argtype

関数、プロシージャ、または集約の引数のデータ型です。

left_type
right_type

演算子の引数のデータ型（スキーマ修飾可）です 前置または後置演算子における存在しない引数にはNONEと記述してください。

PROCEDURAL

これは無意味な単語です。

type_name

変換のデータ型の名前です。

lang_name

変換の言語の名前です。

例

hstore拡張をバージョン2.0に更新します。

```
ALTER EXTENSION hstore UPDATE TO '2.0';
```

hstore拡張のスキーマをutilsに変更します。

```
ALTER EXTENSION hstore SET SCHEMA utils;
```

hstore拡張に既存の関数を追加します。

```
ALTER EXTENSION hstore ADD FUNCTION populate_record(anyelement, hstore);
```

互換性

ALTER EXTENSIONはPostgreSQLの拡張です。

関連項目

CREATE EXTENSION, DROP EXTENSION

ALTER FOREIGN DATA WRAPPER

ALTER FOREIGN DATA WRAPPER — 外部データラッパの定義を変更する

概要

```
ALTER FOREIGN DATA WRAPPER name
    [ HANDLER handler_function | NO HANDLER ]
    [ VALIDATOR validator_function | NO VALIDATOR ]
    [ OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ... ]) ]
ALTER FOREIGN DATA WRAPPER name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER FOREIGN DATA WRAPPER name RENAME TO new_name
```

説明

ALTER FOREIGN DATA WRAPPERは外部データラッパの定義を変更します。このコマンドの第1の構文はサポート関数または外部データラッパの一般的なオプションを変更します。（少なくとも1つの句が必要です。）第2の構文は外部データラッパの所有者を変更します。

スーパーユーザのみが外部データラッパを変更することができます。さらにスーパーユーザのみが外部データラッパを所有することができます。

パラメータ

name

既存の外部データラッパの名前です。

HANDLER handler_function

外部データラッパ用の新しいハンドラ関数を指定します。

NO HANDLER

これを使用して外部データラッパがハンドラ関数を持たないことを指定します。

ハンドラを持たない外部データラッパを使用する外部テーブルにはアクセスできないことに注意してください。

VALIDATOR validator_function

外部データラッパ用の新しい検証関数を指定します。

外部データラッパや依存するサーバ、ユーザマップ、外部テーブルの既存のオプションが新しい検証関数に対して無効となる可能性があることに注意してください。PostgreSQLはこの検査を行いません。変更された外部データラッパを使用する前にこれらのオプションが正しいことを確実にすることはユーザの

責任です。しかしこのALTER FOREIGN DATA WRAPPERコマンドで指定されたオプションはすべて新しい検証関数で検査されます。

NO VALIDATOR

これは、外部データラッパが検証関数を持たないことを指定するために使用されます。

OPTIONS ([ADD | SET | DROP] option ['value'] [, ...])

外部データラッパ用のオプションを変更します。ADD、SET、DROPは実行する動作を指定します。明示的な動作の指定がなければADDとみなされます。オプション名は一意でなければなりません。また名前と値は外部データラッパの検証関数を使用して検証されます。

new_owner

外部データラッパの新しい所有者のユーザ名です。

new_name

外部データラッパの新しい名前です。

例

外部データラッパdbiを変更し、fooオプションを追加し、barオプションを削除します。

```
ALTER FOREIGN DATA WRAPPER dbi OPTIONS (ADD foo '1', DROP 'bar');
```

外部データラッパdbiの検証関数をbob.myvalidatorに変更します。

```
ALTER FOREIGN DATA WRAPPER dbi VALIDATOR bob.myvalidator;
```

互換性

ALTER FOREIGN DATA WRAPPERはISO/IEC 9075-9 (SQL/MED)に従います。ただし、HANDLER、VALIDATOR、OWNER TO、RENAME句は拡張です。

関連項目

CREATE FOREIGN DATA WRAPPER, DROP FOREIGN DATA WRAPPER

ALTER FOREIGN TABLE

ALTER FOREIGN TABLE — 外部テーブルの定義を変更する

概要

```
ALTER FOREIGN TABLE [ IF EXISTS ] [ ONLY ] name [ * ]
    action [, ... ]
ALTER FOREIGN TABLE [ IF EXISTS ] [ ONLY ] name [ * ]
    RENAME [ COLUMN ] column_name TO new_column_name
ALTER FOREIGN TABLE [ IF EXISTS ] name
    RENAME TO new_name
ALTER FOREIGN TABLE [ IF EXISTS ] name
    SET SCHEMA new_schema


ここでactionは以下のいずれかです。

    ADD [ COLUMN ] column_name data_type [ COLLATE collation ] [ column_constraint [ ... ] ]
    DROP [ COLUMN ] [ IF EXISTS ] column_name [ RESTRICT | CASCADE ]
    ALTER [ COLUMN ] column_name [ SET DATA ] TYPE data_type [ COLLATE collation ]
    ALTER [ COLUMN ] column_name SET DEFAULT expression
    ALTER [ COLUMN ] column_name DROP DEFAULT
    ALTER [ COLUMN ] column_name { SET | DROP } NOT NULL
    ALTER [ COLUMN ] column_name SET STATISTICS integer
    ALTER [ COLUMN ] column_name SET ( attribute_option = value [, ... ] )
    ALTER [ COLUMN ] column_name RESET ( attribute_option [, ... ] )
    ALTER [ COLUMN ] column_name SET STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN }
    ALTER [ COLUMN ] column_name OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ... ])
    ADD table_constraint [ NOT VALID ]
    VALIDATE CONSTRAINT constraint_name
    DROP CONSTRAINT [ IF EXISTS ]  constraint_name [ RESTRICT | CASCADE ]
    DISABLE TRIGGER [ trigger_name | ALL | USER ]
    ENABLE TRIGGER [ trigger_name | ALL | USER ]
    ENABLE REPLICA TRIGGER trigger_name
    ENABLE ALWAYS TRIGGER trigger_name
    SET WITHOUT OIDS
    INHERIT parent_table
    NO INHERIT parent_table
    OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
    OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ... ])
```

説明

ALTER FOREIGN TABLEは既存の外部テーブルの定義を変更します。以下のように複数の副構文があります。

ADD COLUMN

この構文は、CREATE FOREIGN TABLEと同じ文法を使用して、外部テーブルに新しい列を追加します。通常のテーブルに列を追加する場合と異なり、背後のストレージには何も起こりません。この操作は単に、外部テーブルを通して何らかの新しい列がアクセスできるようになったことを宣言します。

DROP COLUMN [IF EXISTS]

この構文は外部テーブルから列を削除します。ビューなど何らかのその他のテーブルがこの列に依存する場合、CASCADEを付けなければなりません。IF EXISTSが指定された場合、列が存在しなくてもエラーになりません。この場合、注意メッセージが代わりに発生します。

SET DATA TYPE

この構文は外部テーブルの列の型を変更します。この場合も、背後のストレージには何の影響もありません。この動作は単に、PostgreSQLが想定しているその列の型を変更するだけです。

SET/DROP DEFAULT

この構文は列に対するデフォルト値の設定または削除を行います。デフォルト値はその後に行われるINSERTまたはUPDATEコマンドにのみ適用されます。すでにテーブル内に存在する行の変更は行われません。

SET/DROP NOT NULL

列にNULL値を許すか許さないかどうか印を付けます。

SET STATISTICS

この構文は、この後のANALYZE操作における列単位の統計情報収集目標を設定します。詳細についてはALTER TABLEの類似の構文を参照してください。

SET (attribute_option = value [, ...])
RESET (attribute_option [, ...])

この構文は属性単位のオプションを設定またはリセットします。詳細についてはALTER TABLEにおける類似の構文を参照してください。

SET STORAGE

この構文は、列のストレージモードを設定します。詳しくはALTER TABLEの類似の構文を参照して下さい。ストレージモードは、テーブルの外部データラッパがそれに注意するようになっていなければ、何の効果もないことに注意して下さい。

ADD table_constraint [NOT VALID]

この構文は、CREATE FOREIGN TABLEと同じ構文を使って、外部テーブルに新しい制約を追加します。現在のところ、CHECK制約のみがサポートされています。

通常のテーブルに制約を追加する場合とは異なり、制約が正しいことを検証するために、何も実行されません。そうではなく、この動作は単に、ある新しい条件が、外部テーブルのすべての行に対して成り立つことを仮定すべきだと宣言するものです。(CREATE FOREIGN TABLEの記述を参照して下さい。) 制

約がNOT VALIDであるとされている場合、それが成り立つことは仮定されず、将来利用される場合に備えて記録されているだけになります。

VALIDATE CONSTRAINT

この構文は、それまでNOT VALIDであるとされていた制約をvalidに変更します。制約を検証するために何の動作も実行されませんが、以後の問い合わせではそれが成り立つと仮定されます。

DROP CONSTRAINT [IF EXISTS]

この構文は、外部テーブル上の指定された制約を削除します。IF EXISTSが指定され、その制約が存在しない場合は、エラーにはなりません。その場合、代わりに注意が発行されます。

DISABLE/ENABLE [REPLICA | ALWAYS] TRIGGER

これらの構文は外部テーブルに属するトリガーの発行について設定します。詳細についてはALTER TABLEにおける類似の構文を参照してください。

SET WITHOUT OIDS

システム列oidを削除する、後方互換のための構文です。システム列oidは今では追加できませんので、これは効果がありません。

INHERIT parent_table

この構文は対象の外部テーブルを指定した親テーブルの新しい子テーブルとして追加します。詳しくはALTER TABLEの類似の構文を参照して下さい。

NO INHERIT parent_table

この構文は、対象の外部テーブルを指定した親テーブルの子テーブルのリストから削除します。

OWNER

この構文は外部テーブルの所有者を指定ユーザに変更します。

OPTIONS ([ADD | SET | DROP] option ['value'] [, ...])

外部テーブルもしくはその列の1つについてのオプションを変更します。ADD、SET、DROPは実行する操作を指定します。明示的な操作指定がない場合ADDとみなされます。重複したオプション名は許されません。(しかしテーブルオプションと列オプションとで同じ名前を持たせることは問題ありません。) またオプションの名前と値は外部データラッパのライブラリを使用して検証されます。

RENAME

RENAME構文は外部テーブルの名前または外部テーブル内の個々の列の名前を変更します。

SET SCHEMA

この構文は外部テーブルを別のスキーマに移動します。

RENAMEおよびSET SCHEMA以外の操作はすべて、複数変更項目リストにまとめて並行に適用することができます。例えば、複数の列の追加、複数の列の型変更、またはその両方を単一のコマンドで行うことができます。

コマンドがALTER FOREIGN TABLE IF EXISTS ...と記述されていて外部テーブルが存在しない場合、エラーにはなりません。この場合、注意が発行されます。

ALTER FOREIGN TABLEを使用するためにはテーブルの所有者でなければなりません。また外部テーブルのスキーマを変更するためには、新しいスキーマに対してCREATE権限を持っていなければなりません。所有者を変更するためには、新しく所有者となるロールの直接的または間接的なメンバでなければなりません。また新しく所有者となるロールはテーブルのスキーマに対してCREATE権限を持っていなければなりません。(これらの制限により、テーブルの削除と再作成を行ってもできないことを所有者の変更で行えないようにします。しかし、スーパーユーザはどのテーブルの所有者も変更できます。) 列の追加または列の型の変更を行うためには、そのデータ型に対するUSAGE権限も必要です。

パラメータ

name

変更対象の既存外部テーブルの名前(スキーマ修飾可)です。テーブル名の前にONLYが指定されていた場合、そのテーブルのみが変更されます。ONLYが指定されていない場合、そのテーブルと、そのすべての子孫のテーブル(あれば)が変更されます。子孫のテーブルが含まれることを明示的に示すために、テーブル名の後に*を指定することができますが、これは省略可能です。

column_name

新しい列または既存の列の名前です。

new_column_name

既存の列に対する新しい名前です。

new_name

テーブルの新しい名前です。

data_type

新しい列のデータ型、または既存の列に対する新しいデータ型です。

table_constraint

外部テーブルの新しいテーブル制約です。

constraint_name

削除する既存の制約の名前です。

CASCADE

削除される列または制約に依存するオブジェクト(その列を参照するビューなど)を自動的に削除し、さらにそれらのオブジェクトに依存するすべてのオブジェクトも削除します(5.14参照)。

RESTRICT

依存するオブジェクトが存在する場合、列または制約の削除を拒否します。これがデフォルトの動作です。

trigger_name

無効にする、あるいは有効にするトリガーの名前です。

ALL

外部テーブルに属するすべてのトリガーを無効、あるいは有効にします。(内部的に生成されたトリガーが含まれる場合、スーパーユーザー権限が必要です。コアシステムは外部テーブルにそのようなトリガーを追加することはありませんが、アドオンが追加することはあり得ます。)

USER

内部的に生成されたトリガーを除き、外部テーブルに属するすべてのトリガーを無効、あるいは有効にします。

parent_table

外部テーブルと関連付ける、あるいは関連を取り消す親テーブルです。

new_owner

テーブルの新しい所有者のユーザ名です。

new_schema

テーブルの移動先となるスキーマの名前です。

注釈

COLUMNキーワードには意味がなく、省略可能です。

ADD COLUMNまたはDROP COLUMNにより列が追加、削除される時、NOT NULLまたはCHECK制約が追加される時、SET DATA TYPEにより列の型が変更される時、外部サーバとの一貫性は検査されません。確実にテーブル定義をリモート側に合わせることはユーザの責任です。

有効なパラメータに関する詳しい説明についてはCREATE FOREIGN TABLEを参照してください。

例

列を非NULLと印付けします。

```
ALTER FOREIGN TABLE distributors ALTER COLUMN street SET NOT NULL;
```

外部テーブルのオプションを変更します。

```
ALTER FOREIGN TABLE myschema.distributors OPTIONS (ADD opt1 'value', SET opt2 'value2', DROP opt3
 'value3');
```

互換性

ADD、DROP、SET DATA TYPE構文は標準SQLに準拠します。他の構文は標準SQLに対するPostgreSQLの拡張です。単一のALTER FOREIGN TABLEコマンドに複数の操作を指定する機能も拡張です。

ALTER FOREIGN TABLE DROP COLUMNを用いて外部テーブルのたった1つの列を削除してゼロ列のテーブルとすることができます。これは拡張であり、SQLではゼロ列の外部テーブルを許しません。

関連項目

CREATE FOREIGN TABLE, DROP FOREIGN TABLE

ALTER FUNCTION

ALTER FUNCTION — 関数定義を変更する

概要

```
ALTER FUNCTION name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    action [ ... ] [ RESTRICT ]
ALTER FUNCTION name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    RENAME TO new_name
ALTER FUNCTION name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER FUNCTION name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    SET SCHEMA new_schema
ALTER FUNCTION name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    [ NO ] DEPENDS ON EXTENSION extension_name


ここで、
      actionは以下のいずれかです。

    CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
    IMMUTABLE | STABLE | VOLATILE | [ NOT ] LEAKPROOF
    [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
    PARALLEL { UNSAFE | RESTRICTED | SAFE }
    COST execution_cost
    ROWS result_rows
    SUPPORT support_function
    SET configuration_parameter { TO | = } { value | DEFAULT }
    SET configuration_parameter FROM CURRENT
    RESET configuration_parameter
    RESET ALL
```

説明

ALTER FUNCTIONは関数定義を変更します。

ALTER FUNCTIONを使用するには関数の所有者でなければなりません。関数のスキーマを変更するには、新しいスキーマにおけるCREATE権限も必要です。所有者を変更するには、直接または間接的に新しいロールのメンバでなければなりません。また、そのロールは関数のスキーマにおいてCREATE権限を持たなければなりません。(この制限により、関数の削除と再作成で行うことができない処理を所有者の変更で行えないようになります。しかし、スーパーユーザはすべての関数の所有者を変更することができます。)

パラメータ

name

既存の関数名です（スキーマ修飾名も可）。引数リストを指定しない場合、名前はスキーマ内で一意でなければなりません。

argmode

引数のモードで、IN、OUT、INOUT、VARIADICのいずれかです。省略された場合のデフォルトはINです。関数の識別を行うには入力引数のみが必要ですので、実際にはALTER FUNCTIONがOUT引数を無視することに注意してください。ですので、IN、INOUTおよびVARIADIC引数を列挙することで十分です。

argname

引数の名前です。関数の識別を行うには引数のデータ型のみが必要ですので、実際にはALTER FUNCTIONは引数の名前を無視することに注意してください。

argtype

もしあれば、その関数の引数のデータ型（スキーマ修飾可能）です。

new_name

新しい関数名です。

new_owner

新しい関数の所有者です。関数にSECURITY DEFINERが指定されている場合、その後は新しい所有者の権限で関数が実行されることに注意してください。

new_schema

関数の新しいスキーマです。

DEPENDS ON EXTENSION extension_name
NO DEPENDS ON EXTENSION extension_name

この構文は、関数が拡張に依存している、もしくはNOが指定された場合には拡張にもはや依存していないと印を付けます。拡張に依存していると印をつけられた関数は、拡張が削除されると自動的に削除されます。

CALLED ON NULL INPUT
RETURNS NULL ON NULL INPUT
STRICT

CALLED ON NULL INPUTは、引数の一部またはすべてがNULLの場合に関数が呼び出されるように変更します。RETURNS NULL ON NULL INPUTもしくはSTRICTは、引数の一部がNULLの場合に関数が呼び出されないように変更します。代わりに自動的にNULLという結果とされます。詳細はCREATE FUNCTIONを参照してください。

IMMUTABLE
STABLE
VOLATILE

関数の揮発性を指定した設定に変更します。詳細についてはCREATE FUNCTIONを参照してください。

[EXTERNAL] SECURITY INVOKER
[EXTERNAL] SECURITY DEFINER

関数のセキュリティを定義者にするか否かを変更します。EXTERNALキーワードはSQLとの互換性のためのものであり、無視されます。この機能の詳細についてはCREATE FUNCTIONを参照してください。

PARALLEL

関数が並列処理に対して安全であると見なされるかどうかを変更します。詳しくはCREATE FUNCTIONを参照してください。

LEAKPROOF

関数を漏洩防止関数とみなすか否かを変更します。この機能に関する詳細についてはCREATE FUNCTIONを参照してください。

COST execution_cost

関数の推定実行コストを変更します。詳細についてはCREATE FUNCTIONを参照してください。

ROWS result_rows

集合を返す関数で返される推定行数を変更します。詳細についてはCREATE FUNCTIONを参照してください。

SUPPORT support_function

この関数のために使うプランナサポート関数を設定もしくは変更します。詳細は37.11を参照してください。このオプションを使うにはスーパーユーザでなければなりません。

新しいサポート関数の名前でなければならないため、このオプションはサポート関数を同時に削除するのに使うことはできません。そうする必要があるなら、CREATE OR REPLACE FUNCTIONを使ってください。

configuration_parameter
value

関数呼び出し時に設定パラメータに対して行われる設定を追加または変更します。valueがDEFAULT、またはそれと等価なRESETが使用された場合、関数の局所的な設定は削除されます。このため、関数はその環境内に存在する値で実行されます。すべての関数の局所的な設定を消去したければRESET ALLを使用してください。SET FROM CURRENTは、ALTER FUNCTIONが実行された時点でのパラメータの現在値を、関数起動時に適用される値として保管します。

設定可能なパラメータとその値に関する詳細については、SETおよび第19章を参照してください。

RESTRICT

標準SQLとの互換性のためのものであり、無視されます。

例

integer型用のsqrt関数の名前をsquare_rootに変更します。

```
ALTER FUNCTION sqrt(integer) RENAME TO square_root;
```

integer型用のsqrt関数の所有者をjoeに変更します。

```
ALTER FUNCTION sqrt(integer) OWNER TO joe;
```

integer型用のsqrt関数のスキーマをmathsに変更します。

```
ALTER FUNCTION sqrt(integer) SET SCHEMA maths;
```

integer型に対する関数sqrtが、拡張mathlibに依存するとして印をつけるには、次のようにします。

```
ALTER FUNCTION sqrt(integer) DEPENDS ON EXTENSION mathlib;
```

関数用に検索パスを自動的に設定するように調整します。

```
ALTER FUNCTION check_password(text) SET search_path = admin, pg_temp;
```

関数用のsearch_pathの自動設定を無効にします。

```
ALTER FUNCTION check_password(text) RESET search_path;
```

呼び出し元で使用される検索パスでこの関数が実行されるようになります。

互換性

この文は標準SQLのALTER FUNCTION文に部分的に従っています。標準ではより多くの関数の属性を変更できますが、関数名の変更、関数を定義者の権限で実行するかどうかの変更、関数と設定パラメータ値の関連付け、関数の所有者やスキーマ、揮発性の変更を行う機能はありません。また、標準ではRESTRICTキーワードを必須としていますが、PostgreSQLでは省略可能です。

関連項目

CREATE FUNCTION, DROP FUNCTION, ALTER PROCEDURE, ALTER ROUTINE

ALTER GROUP

ALTER GROUP — ロールの名前またはメンバ資格を変更する

概要

```
ALTER GROUP role_specification ADD USER user_name [, ... ]
ALTER GROUP role_specification DROP USER user_name [, ... ]


ここでrole_specificationは以下の通りです。

    role_name
  | CURRENT_USER
  | SESSION_USER

ALTER GROUP group_name RENAME TO new_name
```

説明

ALTER GROUPはユーザグループの属性を変更します。後方互換性のために受け付けられていますが、このコマンドは廃止予定です。グループ(ユーザも同様)は、より一般化されたロールという概念に置き換えられたからです。

最初の2つの構文は、ユーザをグループに追加もしくはグループから削除します (この場合、任意のロールを「ユーザ」部分として、または「グループ」 部分として使用することができます)。この種の構文は、実のところ、「グループ」として指名されたロール内のメンバ資格の付与、取消と同じです。ですので、GRANTやREVOKEを使用する方法を薦めます。

3番目の構文はグループの名前を変更します。これは、ALTER ROLEを使用したロール名の変更とまったく同じです。

パラメータ

group_name

変更するグループ(ロール)の名前です。

user_name

グループに追加または削除されるユーザ(ロール)です。指定するユーザは存在するものでなければいけません。ALTER GROUPは、ユーザの作成も削除も行いません。

new_name

新しいグループ名です。

例

ユーザをグループに追加します。

```
ALTER GROUP staff ADD USER karl, john;
```

ユーザをグループから削除します。

```
ALTER GROUP workers DROP USER beth;
```

互換性

標準SQLにはALTER GROUP文はありません。

関連項目

GRANT, REVOKE, ALTER ROLE

ALTER INDEX

ALTER INDEX — インデックス定義を変更する

概要

```
ALTER INDEX [ IF EXISTS ] name RENAME TO new_name
ALTER INDEX [ IF EXISTS ] name SET TABLESPACE tablespace_name
ALTER INDEX name ATTACH PARTITION index_name
ALTER INDEX name DEPENDS ON EXTENSION extension_name
ALTER INDEX [ IF EXISTS ] name SET ( storage_parameter [= value] [, ... ] )
ALTER INDEX [ IF EXISTS ] name RESET ( storage_parameter [, ... ] )
ALTER INDEX [ IF EXISTS ] name ALTER [ COLUMN ] column_number
    SET STATISTICS integer
ALTER INDEX ALL IN TABLESPACE name [ OWNED BY role_name [, ... ] ]
    SET TABLESPACE new_tablespace [ NOWAIT ]
```

説明

ALTER INDEXは既存のインデックスの定義を変更します。以下のような副構文が存在します。要求されるロックレベルはそれぞれの副構文によって異なることに注意してください。特に記述がなければACCESS EXCLUSIVEロックを取得します。複数のサブコマンドが使われるときは、それらのサブコマンドが要求するうち、もっとも高いレベルのロックを取得します。

RENAME

このRENAME構文は、インデックスの名前を変更します。インデックスがテーブル制約(UNIQUE、PRIMARY KEY、EXCLUDEのいずれか)と関連付けられていた場合、制約名も変更されます。格納されたデータには影響しません。

インデックスの名前の変更にはSHARE UPDATE EXCLUSIVEロックが必要です。

SET TABLESPACE

この構文は、インデックスのテーブル空間を指定したテーブル空間に変更し、インデックスに関連するデータファイルを移動します。インデックスのテーブル空間を変更するには、インデックスの所有者であり、かつ新しいテーブル空間のCREATE権限を有している必要があります。ALL IN TABLESPACE構文を使うことで、テーブル空間内の現在のデータベースのすべてのインデックスを移動することができます。この場合、移動されるすべてのインデックスがロックされ、それから1つずつ移動されます。この構文はOWNED BYもサポートしており、これを使うと、指定のロールが所有しているインデックスだけを移動します。NOWAITオプションを指定した場合、必要とするすべてのロックを即座に獲得できなければ、このコマンドは失敗します。このコマンドではシステムカタログは移動されないことに注意してください。必要であれば、ALTER DATABASEを使うか、あるいはALTER INDEXで明示的に指定してください。CREATE TABLESPACEも参照してください。

ATTACH PARTITION

指定されたインデックスを変更するインデックスに付加します。指定されたインデックスは、変更するインデックスを持つテーブルのパーティションに対するもので、かつ、同じ定義を持たなければなりません。付加されたインデックスは、それ自身として削除できず、親インデックスが削除された場合に自動的に削除されます。

DEPENDS ON EXTENSION extension_name
NO DEPENDS ON EXTENSION extension_name

この構文は、インデックスが拡張に依存している、もしくはNOが指定された場合には拡張にもはや依存していないと印を付けます。拡張に依存していると印をつけられたインデックスは、拡張が削除されると自動的に削除されます。

SET (storage_parameter [= value] [, ...])

この構文は、インデックスに対し、インデックスメソッド固有の1つ以上の格納パラメータを変更します。設定可能なパラメータについてはCREATE INDEXを参照してください。このコマンドにより、インデックスの内容がすぐに変更されるわけではないことに注意してください。パラメータによりますが、期待する効果を得るためにREINDEXを使用してインデックスを再構築しなければならない場合もあります。

RESET (storage_parameter [, ...])

この構文は、1つ以上のインデックスメソッド固有の格納パラメータをデフォルト値に再設定します。SET同様、インデックスを完全に更新するためにREINDEXが必要になる場合があります。

ALTER [COLUMN] column_number SET STATISTICS integer

この構文は、以後のANALYZE操作にむけた、列ごとの統計収集対象を設定します。ただし、式として定義されたインデックス列のみに使えます。式には一意な名前が無いため、これらはインデックス列の序数を使って参照します。対象は0から10000の範囲で設定できます。代わりに-1と設定すると、システムのデフォルト統計対象（default_statistics_target）に戻します。PostgreSQLの問い合わせプランナによる統計の利用についての詳細は14.2を参照してください。

パラメータ

IF EXISTS

インデックスが存在しない場合にエラーとしません。この場合注意メッセージが発生します。

column_number

インデックス列の順序位置（左から右）を参照する序数。

name

変更対象の既存のインデックスの名前です（スキーマ修飾名も可）。

new_name

インデックスの新しい名前です。

tablespace_name

インデックスの移動先のテーブル空間です。

extension_name

インデックスが依存することになる拡張の名前です。

storage_parameter

インデックスメソッド固有の格納パラメータの名前です。

value

インデックスメソッド固有の格納パラメータの新しい値です。パラメータに応じてこれが数値になることも文字列になることもあります。

注釈

これらの操作はALTER TABLEを使用して行うこともできます。実際には、ALTER INDEXは、ALTER TABLEのインデックス用構文の別名に過ぎません。

以前はALTER INDEX OWNERという種類の構文がありましたが、(警告の上)無視されるようになりました。インデックスの所有者は基のテーブルの所有者と異なるものにすることができません。テーブルの所有者を変更すると自動的にインデックスの所有者も変わります。

システムカタログ用インデックスに対する変更は許されていません。

例

既存のインデックスの名前を変更します。

```
ALTER INDEX distributors RENAME TO suppliers;
```

インデックスを別のテーブル空間に移動します。

```
ALTER INDEX distributors SET TABLESPACE fasttablespace;
```

インデックスのフィルファクタを変更します(インデックスメソッドがフィルファクタをサポートしていることを前提とします)。

```
ALTER INDEX distributors SET (fillfactor = 75);
REINDEX INDEX distributors;
```

式インデックスに対して統計収集対象を設定します。

```
CREATE INDEX coord_idx ON measured (x, y, (z + t));
ALTER INDEX coord_idx ALTER COLUMN 3 SET STATISTICS 1000;
```

互換性

`ALTER INDEX`はPostgreSQLの拡張です。

関連項目

CREATE INDEX, REINDEX

ALTER LANGUAGE

ALTER LANGUAGE — 手続き言語の定義を変更する

概要

```
ALTER [ PROCEDURAL ] LANGUAGE name RENAME TO new_name
ALTER [ PROCEDURAL ] LANGUAGE name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
```

説明

ALTER LANGUAGEは、手続き言語の定義を変更します。言語名の変更および言語の所有者の変更のみが可能です。ALTER LANGUAGEを使用するためにはスーパーユーザまたは言語の所有者でなければなりません。

パラメータ

name

言語の名前です。

new_name

新しい言語の名前です。

new_owner

新しい言語の所有者です。

互換性

標準SQLにはALTER LANGUAGE文はありません。

関連項目

CREATE LANGUAGE, DROP LANGUAGE

ALTER LARGE OBJECT

ALTER LARGE OBJECT — ラージオブジェクトの定義を変更する

概要

```
ALTER LARGE OBJECT large_object_oid OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
```

説明

ALTER LARGE OBJECTはラージオブジェクトの定義を変更します。

ALTER LARGE OBJECTを使用するためにはラージオブジェクトを所有していなければなりません。所有者を変更するためには、新しく所有するロールの直接または間接のメンバでもなければなりません。(しかし、スーパーユーザはラージオブジェクトを変更できます。) 今のところ、唯一の機能は新しい所有者の割り当てですので、両方の制限が常に適用されます。

パラメータ

large_object_oid

変更対象のラージオブジェクトのOIDです。

new_owner

ラージオブジェクトの新しい所有者です。

互換性

標準SQLにはALTER LARGE OBJECT文はありません。

関連項目

第34章

ALTER MATERIALIZED VIEW

ALTER MATERIALIZED VIEW — マテリアライズドビューの定義を変更する

概要

```
ALTER MATERIALIZED VIEW [ IF EXISTS ] name
    action [, ... ]
ALTER MATERIALIZED VIEW name
    DEPENDS ON EXTENSION extension_name
ALTER MATERIALIZED VIEW [ IF EXISTS ] name
    RENAME [ COLUMN ] column_name TO new_column_name
ALTER MATERIALIZED VIEW [ IF EXISTS ] name
    RENAME TO new_name
ALTER MATERIALIZED VIEW [ IF EXISTS ] name
    SET SCHEMA new_schema
ALTER MATERIALIZED VIEW ALL IN TABLESPACE name [ OWNED BY role_name [, ... ] ]
    SET TABLESPACE new_tablespace [ NOWAIT ]


ここでactionは以下のいずれかです。

    ALTER [ COLUMN ] column_name SET STATISTICS integer
    ALTER [ COLUMN ] column_name SET ( attribute_option = value [, ... ] )
    ALTER [ COLUMN ] column_name RESET ( attribute_option [, ... ] )
    ALTER [ COLUMN ] column_name SET STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN }
    CLUSTER ON index_name
    SET WITHOUT CLUSTER
    SET ( storage_parameter [= value] [, ... ] )
    RESET ( storage_parameter [, ... ] )
    OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
```

説明

ALTER MATERIALIZED VIEWは既存のマテリアライズドビューの各種補助属性を変更します。

ALTER MATERIALIZED VIEWを使用するためにはそのマテリアライズドビューを所有していなければなりません。マテリアライズドビューのスキーマを変更するためには、新しいスキーマに対するCREATE権限を持たなければなりません。所有者を変更するためには、新しく所有者となるロールの直接または間接的なメンバでなければなりません。またそのロールはマテリアライズドビューのスキーマに対してCREATE権限を持たなければなりません。(これらの制限により、マテリアライズドビューを削除し再作成することによってできる以上のことを所有者の変更で行えないようにします。しかしスーパーユーザはいずれにせよ任意のビューの所有権を変更することができます。)

ALTER MATERIALIZED VIEW文で利用可能な副構文と操作は、ALTER TABLEで利用できるものの部分集合であり、マテリアライズドビューに対して使用した場合も同じ意味を持ちます。詳細についてはALTER TABLEの説明を参照してください。

パラメータ

name

既存のマテリアライズドビューの名前（スキーマ修飾可）です。

column_name

新しいまたは既存の列の名前です。

extension_name

マテリアライズドビューが依存する(もしくはNOが指定された場合にはもはや依存していない)拡張の名前です。拡張に依存していると印をつけられたマテリアライズドビューは、拡張が削除されると自動的に削除されます。

new_column_name

既存の列に対する新しい名前です。

new_owner

マテリアライズドビューの新しい所有者となるユーザの名前です。

new_name

マテリアライズドビューの新しい名前です。

new_schema

マテリアライズドビューの新しいスキーマです。

例

マテリアライズドビューfooの名前をbarに変更します。

```
ALTER MATERIALIZED VIEW foo RENAME TO bar;
```

互換性

ALTER MATERIALIZED VIEWはPostgreSQLの拡張です。

関連項目

CREATE MATERIALIZED VIEW, DROP MATERIALIZED VIEW, REFRESH MATERIALIZED VIEW

ALTER OPERATOR

ALTER OPERATOR — 演算子の定義を変更する

概要

```
ALTER OPERATOR name ( { left_type | NONE } , { right_type | NONE } )
    OWNER TO { new_owner | CURRENT_USER | SESSION_USER }

ALTER OPERATOR name ( { left_type | NONE } , { right_type | NONE } )
    SET SCHEMA new_schema

ALTER OPERATOR name ( { left_type | NONE } , { right_type | NONE } )
    SET ( {  RESTRICT = { res_proc | NONE }
           | JOIN = { join_proc | NONE }
         } [, ... ] )
```

説明

ALTER OPERATORは演算子の定義を変更します。

ALTER OPERATORを使用するには演算子の所有者でなければなりません。所有者を変更するには、直接または間接的に新しい所有者ロールのメンバでなければなりません。また、そのロールが演算子のスキーマにおいてCREATEを持たなければなりません。(この制限により、演算子の削除と再作成で行うことができない処理を所有者の変更で行えないようになります。しかし、スーパーユーザはすべての演算子の所有者を変更することができます。)

パラメータ

name

既存の演算子の名前です(スキーマ修飾名も可)。

left_type

演算子の左オペランドのデータ型です。左オペランドがない演算子の場合はNONEを指定します。

right_type

演算子の右オペランドのデータ型です。右オペランドがない演算子の場合はNONEを指定します。

new_owner

演算子の新しい所有者です。

new_schema

演算子の新しいスキーマです。

res_proc

この演算子の制約選択評価関数です。既存の制約選択評価関数を削除するにはNONEを指定します。

join_proc

この演算子の結合選択評価関数です。既存の結合選択評価関数を削除するにはNONEを指定します。

例

text型用の独自の演算子a @@ bの所有者を変更します。

```
ALTER OPERATOR @@ (text, text) OWNER TO joe;
```

int[]型用の独自の演算子a && bの制約および結合選択評価関数を変更します。

```
ALTER OPERATOR && (_int4, _int4) SET (RESTRICT = _int_contsel, JOIN = _int_contjoinsel);
```

互換性

標準SQLにはALTER OPERATOR文はありません。

関連項目

CREATE OPERATOR, DROP OPERATOR

ALTER OPERATOR CLASS

ALTER OPERATOR CLASS — 演算子クラスの定義を変更する

概要

```
ALTER OPERATOR CLASS name USING index_method
    RENAME TO new_name

ALTER OPERATOR CLASS name USING index_method
    OWNER TO { new_owner | CURRENT_USER | SESSION_USER }

ALTER OPERATOR CLASS name USING index_method
    SET SCHEMA new_schema
```

説明

ALTER OPERATOR CLASSは演算子クラスの定義を変更します。

ALTER OPERATOR CLASSを使用するには演算子クラスの所有者でなければなりません。所有者を変更するには、直接または間接的に新しい所有者ロールのメンバでなければなりません。また、そのロールが演算子クラスのスキーマにおいてCREATEを持たなければなりません。(この制限により、演算子クラスの削除と再作成で行うことができない処理を所有者の変更で行えないようになります。しかし、スーパーユーザはすべての演算子クラスの所有者を変更することができます。)

パラメータ

name

既存の演算子クラスの名前です(スキーマ修飾名も可)。

index_method

演算子クラス用のインデックスメソッドの名前です。

new_name

新しい演算子クラス名です。

new_owner

演算子クラスの新しい所有者です。

new_schema

演算子クラスの新しいスキーマです。

互換性

標準SQLにはALTER OPERATOR CLASS文はありません。

関連項目

CREATE OPERATOR CLASS, DROP OPERATOR CLASS, ALTER OPERATOR FAMILY

ALTER OPERATOR FAMILY

ALTER OPERATOR FAMILY — 演算子族の定義を変更する

概要

```
ALTER OPERATOR FAMILY name USING index_method ADD
  {  OPERATOR strategy_number operator_name ( op_type, op_type )
             [ FOR SEARCH | FOR ORDER BY sort_family_name ]
   | FUNCTION support_number [ ( op_type [ , op_type ] ) ]
             function_name [ ( argument_type [, ...] ) ]
  } [, ... ]

ALTER OPERATOR FAMILY name USING index_method DROP
  {  OPERATOR strategy_number ( op_type [ , op_type ] )
   | FUNCTION support_number ( op_type [ , op_type ] )
  } [, ... ]

ALTER OPERATOR FAMILY name USING index_method
    RENAME TO new_name

ALTER OPERATOR FAMILY name USING index_method
    OWNER TO { new_owner | CURRENT_USER | SESSION_USER }

ALTER OPERATOR FAMILY name USING index_method
    SET SCHEMA new_schema
```

説明

ALTER OPERATOR FAMILYは演算子族の定義を変更します。演算子やサポート関数を演算子族に追加することやそれらを演算子族から削除すること、演算子族の名前や所有者を変更することが可能です。

ALTER OPERATOR FAMILYを使用して演算子とサポート関数が演算子族に追加される時、これらは演算子族内の特定の演算子クラスの一部とはならず、単に演算子族内で「自由」なものになります。これは、これらの演算子と関数が演算子族と意味的な互換性を持つが、特定のインデックスの正しい動作には必要とされないことを意味します。（必要な演算子と関数は演算子クラスの一部として宣言しなければなりません。CREATE OPERATOR CLASSを参照してください。）PostgreSQLでは演算子族の自由なメンバをいつでも演算子族から削除することができます。しかし演算子クラス内のメンバは、クラス全体と依存するインデックスすべてを削除しなければ削除することはできません。通常、単一データ型の演算子と関数は、特定のデータ型に対するインデックスをサポートするために必要ですので、演算子クラスの一部となります。一方、データ型を跨る演算子と関数は、演算子族内の自由なメンバとなります。

ALTER OPERATOR FAMILYを使用するには、スーパーユーザでなければなりません（誤った演算子族定義はサーバを混乱させクラッシュさせることさえありますので、この制限がなされています）。

現時点ではALTER OPERATOR FAMILYは、インデックスメソッドで必要とされる演算子族がすべての演算子と関数を含んでいるかどうかを検査しません。また、演算子と関数が自身で整合性のある集合を形成しているかどうかも検査しません。有効な演算子族を定義することはユーザの責任です。

詳細は37.16を参照してください。

パラメータ

name

既存の演算子族の名前です(スキーマ修飾可)。

index_method

演算子族が対象とするインデックスメソッドの名前です。

strategy_number

演算子族と関連した演算子に対するインデックスメソッドの戦略番号です。

operator_name

演算子族と関連した演算子の名前です(スキーマ修飾可)。

op_type

OPERATOR句では演算子の入力データ型、または左単項演算子、右単項演算子を表すNONEです。CREATE OPERATOR CLASSと類似の構文と異なり、入力データ型を常に指定しなければなりません。

ADD FUNCTION句では、関数がサポートする予定の入力データ型です(関数の入力データ型と異なる場合)。B-tree比較関数およびHash関数では、関数の入力データ型は常に正しく使用するデータ型であるため、op_typeを指定する必要がありません。B-treeソートサポート関数、B-tree等価イメージ関数とGiST、SP-GiST、GIN演算子クラスのすべての関数では、関数が使用する入力データ型を指定する必要があります。

DROP FUNCTION句では、関数がサポートする予定の入力データ型を指定しなければなりません。

sort_family_name

順序付け演算子に関連するソート順序を記述する、既存のbtree演算子族の名前(スキーマ修飾も可)です。

FOR SEARCHもFOR ORDER BYも指定されない場合、FOR SEARCHがデフォルトです。

support_number

演算子族に関連する関数用のインデックスメソッドのサポート関数の番号です。

function_name

演算子族用のインデックスメソッドのサポート関数となる関数の名前です(スキーマ修飾名でも可)。引数リストを指定しない場合、名前はスキーマ内で一意でなければなりません。

argument_type

関数のパラメータのデータ型です。

new_name

演算子族の新しい名前です。

new_owner

演算子族の新しい所有者です。

new_schema

演算子族の新しいスキーマです。

OPERATORとFUNCTION句は任意の順番で記述できます。

注釈

DROP構文が、戦略番号またはサポート番号と入力データ型という、演算子族の「スロット」のみを指定していることに注意してください。そのスロットに存在する演算子または関数の名前については言及されません。また、DROP FUNCTIONでは、指定する型は関数がサポートする予定の入力データ型です。GiST、SP-GiSTおよびGINインデックスでは、関数の実際の入力引数の型と関連しない可能性があります。

インデックス機構は使用する前に関数のアクセス権限を検査しません。演算子族内の関数や演算子を含めることは、公的な実行権限を与えることと同じです。これは通常、演算子族内で使用される関数では問題になりません。

演算子をSQL関数で定義してはいけません。SQL関数はよく、呼び出し元の問い合わせ内でインライン展開されます。すると、オプティマイザが問い合わせがインデックスに一致するかどうか認識できなくなります。

PostgreSQL 8.4より前までは、OPERATOR句にRECHECKオプションを含めることができました。インデックス演算子に「損失がある」かどうかは実行時にその場で決定されるようになりましたので、これはサポートされなくなりました。これにより、演算子に損失があるかもしれないしないかもしれないような場合を効率的に扱うことができるようになりました。

例

以下のコマンド例は、データ型を跨る演算子とサポート関数をint4とint2データ型用のB-Tree演算子クラスをすでに含む演算子族に追加します。

```
ALTER OPERATOR FAMILY integer_ops USING btree ADD

  -- int4 vs int2
  OPERATOR 1 < (int4, int2) ,
  OPERATOR 2 <= (int4, int2) ,
  OPERATOR 3 = (int4, int2) ,
  OPERATOR 4 >= (int4, int2) ,
```

```
  OPERATOR 5 > (int4, int2) ,
  FUNCTION 1 btint42cmp(int4, int2) ,

  -- int2 vs int4
  OPERATOR 1 < (int2, int4) ,
  OPERATOR 2 <= (int2, int4) ,
  OPERATOR 3 = (int2, int4) ,
  OPERATOR 4 >= (int2, int4) ,
  OPERATOR 5 > (int2, int4) ,
  FUNCTION 1 btint24cmp(int2, int4) ;
```

これらの項目を再度削除します。

```
ALTER OPERATOR FAMILY integer_ops USING btree DROP

  -- int4 vs int2
  OPERATOR 1 (int4, int2) ,
  OPERATOR 2 (int4, int2) ,
  OPERATOR 3 (int4, int2) ,
  OPERATOR 4 (int4, int2) ,
  OPERATOR 5 (int4, int2) ,
  FUNCTION 1 (int4, int2) ,

  -- int2 vs int4
  OPERATOR 1 (int2, int4) ,
  OPERATOR 2 (int2, int4) ,
  OPERATOR 3 (int2, int4) ,
  OPERATOR 4 (int2, int4) ,
  OPERATOR 5 (int2, int4) ,
  FUNCTION 1 (int2, int4) ;
```

互換性

標準SQLにはALTER OPERATOR FAMILY文はありません。

関連項目

CREATE OPERATOR FAMILY, DROP OPERATOR FAMILY, CREATE OPERATOR CLASS, ALTER OPERATOR CLASS, DROP OPERATOR CLASS

ALTER POLICY

ALTER POLICY — 行単位のセキュリティポリシーの定義を変更する

概要

```
ALTER POLICY name ON table_name RENAME TO new_name

ALTER POLICY name ON table_name
    [ TO { role_name | PUBLIC | CURRENT_USER | SESSION_USER } [, ...] ]
    [ USING ( using_expression ) ]
    [ WITH CHECK ( check_expression ) ]
```

説明

ALTER POLICYは既存の行単位のセキュリティポリシーの定義を変更します。ALTER POLICYはポリシーが適用されるロールの集合、およびUSINGとWITH CHECKの式を変更できるだけであることに注意してください。適用されるコマンドや、許容と制限の別といったその他のポリシーの属性を変更するには、ポリシーを削除して再作成しなければなりません。

ALTER POLICYを使うには、ポリシーの適用対象のテーブルの所有者でなければなりません。

ALTER POLICYの2番目の構文で、ロールのリスト、using_expression、check_expressionが指定された時は、それぞれ独立して置換されます。それらの1つが省略された場合、ポリシーのその部分については変更されません。

パラメータ

name

変更対象の既存のポリシーの名前です。

table_name

ポリシーが適用されているテーブルの名前（スキーマ修飾可）です。

new_name

ポリシーの新しい名前です。

role_name

ポリシーの適用対象のロールです。複数のロールを一度に指定することができます。ポリシーをすべてのロールに適用するには、PUBLICを指定します。

using_expression

ポリシーのUSING式です。詳しくはCREATE POLICYを参照して下さい。

check_expression

ポリシーのWITH CHECK式です。詳しくはCREATE POLICYを参照して下さい。

互換性

ALTER POLICYはPostgreSQLの拡張です。

関連項目

CREATE POLICY, DROP POLICY

ALTER PROCEDURE

ALTER PROCEDURE — プロシージャの定義を変更する

概要

```
ALTER PROCEDURE name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    action [ ... ] [ RESTRICT ]
ALTER PROCEDURE name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    RENAME TO new_name
ALTER PROCEDURE name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER PROCEDURE name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    SET SCHEMA new_schema
ALTER PROCEDURE name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    DEPENDS ON EXTENSION extension_name


ここでactionは以下のいずれかです。

    [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
    SET configuration_parameter { TO | = } { value | DEFAULT }
    SET configuration_parameter FROM CURRENT
    RESET configuration_parameter
    RESET ALL
```

説明

ALTER PROCEDUREはプロシージャ定義を変更します。

ALTER PROCEDUREを使用するにはプロシージャの所有者でなければなりません。プロシージャのスキーマを変更するには、新しいスキーマにおけるCREATE権限も必要です。所有者を変更するには、直接または間接的に新しいロールのメンバでなければなりません。また、そのロールはプロシージャのスキーマにおいてCREATE権限を持たなければなりません。(この制限により、プロシージャの削除と再作成で行うことができない処理を所有者の変更で行えないようになります。しかし、スーパーユーザはすべての関数の所有者を変更することができます。)

パラメータ

name

既存のプロシージャ名です(スキーマ修飾も可)。引数リストを指定しない場合、名前はスキーマ内で一意でなければなりません。

argmode

引数モードで、INかVARIADICのいずれかです。省略した場合のデフォルトはINです。

argname

引数の名前です。プロシージャの識別を行うには引数のデータ型のみが必要ですので、実際にはALTER PROCEDUREは引数の名前を無視することに注意してください。

argtype

もしあれば、そのプロシージャの引数のデータ型（スキーマ修飾も可）です。

new_name

新たなプロシージャ名。

new_owner

新しいプロシージャの所有者です。プロシージャにSECURITY DEFINERが指定されている場合、その後は新しい所有者の権限でプロシージャが実行されることに注意してください。

new_schema

プロシージャの新しいスキーマ。

extension_name

プロシージャが依存することになる拡張の名前。

[EXTERNAL] SECURITY INVOKER
[EXTERNAL] SECURITY DEFINER

プロシージャを定義者セキュリティにするか否かを変更します。EXTERNALキーワードはSQLとの互換性のためのものであり、無視されます。この機能の詳細についてはCREATE PROCEDUREを参照してください。

configuration_parameter
value

プロシージャ呼び出し時に設定パラメータに対して行われる設定を追加または変更します。valueがDEFAULT、またはそれと等価なRESETが使用された場合、プロシージャの局所的な設定は削除されます。このため、プロシージャはその環境内に存在する値で実行されます。すべてのプロシージャの局所的な設定を消去したければRESET ALLを使用してください。SET FROM CURRENTは、ALTER PROCEDUREが実行された時点でのパラメータの現在値を、プロシージャ起動時に適用される値として保管します。

使用できるパラメータ名と値についての更なる詳細はSETと第19章を参照してください。

RESTRICT

標準SQLに準拠するためのものであり、無視されます。

例

integer型の二つの引数を持つプロシージャinsert_dataをinsert_recordに名前変更します。

```
ALTER PROCEDURE insert_data(integer, integer) RENAME TO insert_record;
```

integer型の二つの引数を持つプロシージャinsert_dataの所有者をjoeに変更します。

```
ALTER PROCEDURE insert_data(integer, integer) OWNER TO joe;
```

integer型の二つの引数を持つプロシージャinsert_dataのスキーマをaccountingに変更します。

```
ALTER PROCEDURE insert_data(integer, integer) SET SCHEMA accounting;
```

プロシージャinsert_data(integer, integer)を拡張myextに依存するものと印付けします。

```
ALTER PROCEDURE insert_data(integer, integer) DEPENDS ON EXTENSION myext;
```

プロシージャに対して自動的に設定されるようにサーチパスを調整します。

```
ALTER PROCEDURE check_password(text) SET search_path = admin, pg_temp;
```

プロシージャに対するsearch_pathの自動的な設定を無効化します。

```
ALTER PROCEDURE check_password(text) RESET search_path;
```

このプロシージャは何であれ呼び出し側で使われるサーチパスで実行されるようになります。

互換性

この文はSQL標準のALTER PROCEDUREと部分的に互換性があります。標準ではより多くのプロシージャの属性を変更できますが、プロシージャの名前変更、定義者の権限で実行するかどうかの変更、設定パラメータ値の付与、および、プロシージャの所有者、スキーマ、変動性の変更は提供されません。また、標準ではRESTRICTキーワードが必要ですが、PostgreSQLでは省略可能です。

See Also

CREATE PROCEDURE, DROP PROCEDURE, ALTER FUNCTION, ALTER ROUTINE

ALTER PUBLICATION

ALTER PUBLICATION — パブリケーションの定義を変更する

概要

```
ALTER PUBLICATION name ADD TABLE [ ONLY ] table_name [ * ] [, ...]
ALTER PUBLICATION name SET TABLE [ ONLY ] table_name [ * ] [, ...]
ALTER PUBLICATION name DROP TABLE [ ONLY ] table_name [ * ] [, ...]
ALTER PUBLICATION name SET ( publication_parameter [= value] [, ... ] )
ALTER PUBLICATION name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER PUBLICATION name RENAME TO new_name
```

説明

コマンドALTER PUBLICATIONはパブリケーションの属性を変更できます。

最初の3つの構文では、パブリケーションにどのテーブルが含まれるかを変更します。SET TABLE句は、パブリケーションのテーブルのリストを指定したもので置き換えます。ADD TABLE句とDROP TABLE句はパブリケーションに1つ以上のテーブルを追加または削除します。既にサブスクライブされているパブリケーションにテーブルを追加した場合、それを有効にするにはサブスクライブしている側でALTER SUBSCRIPTION ... REFRESH PUBLICATIONの操作を行う必要があることに注意してください。

このコマンドの概要に挙げられている4番目の構文では、CREATE PUBLICATIONで指定されたすべてのパブリケーションの属性を変更できます。このコマンドで属性を指定しなかったものについては、以前の設定が保持されます。

残りの構文では、パブリケーションの所有者および名前を変更します。

ALTER PUBLICATIONを使用するには、そのパブリケーションを所有していなければなりません。所有者を変更するには、新しい所有ロールの直接的あるいは間接的なメンバーでもなければなりません。新しい所有者は、データベースにCREATE権限を持っていなければなりません。また、FOR ALL TABLESのパブリケーションの新しい所有者はスーパーユーザでなければなりません。しかし、スーパーユーザはこれらの制限を回避してパブリケーションの所有者を変更することができます。

パラメータ

name

定義の変更の対象となる既存のパブリケーションの名前です。

table_name

既存のテーブルの名前です。テーブル名の前にONLYが指定されたときは、そのテーブルだけが影響を受けます。テーブル名の前にONLYが指定されていないときは、そのテーブルとそのすべての子テーブル

（あれば）が影響を受けます。オプションでテーブル名の後に*を指定して、子テーブルが含まれることを明示的に示すことができます。

SET (publication_parameter [= value] [, ...])

この句では、元はCREATE PUBLICATIONにより設定されたパブリケーションのパラメータを変更します。詳細な情報はそちらを参照してください。

new_owner

パブリケーションの新しい所有者のユーザ名です。

new_name

パブリケーションの新しい名前です。

例

deleteとupdateのみをパブリッシュするようにパブリケーションを変更します。

```
ALTER PUBLICATION noinsert SET (publish = 'update, delete');
```

パブリケーションにいくつかのテーブルを追加します。

```
ALTER PUBLICATION mypublication ADD TABLE users, departments;
```

互換性

ALTER PUBLICATIONはPostgreSQLの拡張です。

関連項目

CREATE PUBLICATION, DROP PUBLICATION, CREATE SUBSCRIPTION, ALTER SUBSCRIPTION

ALTER ROLE

ALTER ROLE — データベースロールを変更する

概要

```
ALTER ROLE role_specification [ WITH ] option [ ... ]


ここでoptionは以下の通りです。

      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | REPLICATION | NOREPLICATION
    | BYPASSRLS | NOBYPASSRLS
    | CONNECTION LIMIT connlimit
    | [ ENCRYPTED ] PASSWORD 'password' | PASSWORD NULL
    | VALID UNTIL 'timestamp'

ALTER ROLE name RENAME TO new_name

ALTER ROLE { role_specification | ALL } [ IN DATABASE database_name ] SET configuration_parameter
 { TO | = } { value | DEFAULT }
ALTER ROLE { role_specification | ALL } [ IN DATABASE database_name ] SET configuration_parameter
 FROM CURRENT
ALTER ROLE { role_specification | ALL } [ IN DATABASE database_name ]
 RESET configuration_parameter
ALTER ROLE { role_specification | ALL } [ IN DATABASE database_name ] RESET ALL


ここでrole_specificationは以下の通りです。

    role_name
  | CURRENT_USER
  | SESSION_USER
```

説明

ALTER ROLEはPostgreSQLのロールの属性を変更します。

このコマンドの最初の構文では、CREATE ROLEで指定可能な多くのロール属性を変更できます。(指定し得るすべての属性に対応していますが、メンバ資格の追加および削除用のオプションはありません。追加および削除にはGRANTとREVOKEを使用してください。) このコマンドで指定しなかった属性は以前の設定のまま残ります。データベーススーパーユーザはすべてのロールに対して設定すべてを変更できます。CREATEROLE権限を持つロールは、SUPERUSER、REPLICATION、BYPASSRLSを除く設定すべてを変更できます。ただし、スーパーユーザおよびレプリケーション以外のロールに対してのみです。通常のロールは自身のパスワードのみを変更できます。

2番目の構文ではロールの名前を変更することができます。データベーススーパーユーザはすべてのロールの名前を変更できます。CREATEROLE権限を持つロールはスーパーユーザ以外のロールに対してその名前を変更できます。現在のセッションユーザの名前を変更することはできません。(必要ならば別のユーザで接続してください)。MD5暗号化パスワードではロール名を暗号用のソルトとして使用しますので、パスワードがMD5で暗号化されている場合、ロール名を変更するとパスワードは空になります。

残りの構文では、全データベース用、またはIN DATABASE句が指定された場合はそのデータベース用のセッションに対するロールの設定変数についてのセッションデフォルトを変更します。ロール名の代わりにALLを指定すると、すべてのロール用の設定を変更します。ALLとIN DATABASEを一緒に使用することは実質ALTER DATABASE ... SET ...コマンドの使用と同じです。

その後、ロールが新しいセッションを始めると常に、postgresql.conf内の設定やpostgresコマンドラインから受け取った設定よりも優先されて、指定された値がセッションのデフォルトとなります。これはログイン時のみに発生します。SET ROLEまたはSET SESSION AUTHORIZATIONを実行しても新しい設定値は設定されません。全データベースに対する設定よりも、ロールに割り当てたデータベース固有の設定が優先します。特定のデータベースまたは特定のロールに対する設定は、すべてのロールに対する設定よりも優先します。

スーパーユーザはすべてのユーザのセッションのデフォルトを変更することができます。CREATEROLE権限を持つロールはスーパーユーザ以外のロールのデフォルトを変更することができます。通常のロールは自身のデフォルトのみを設定することができます。設定変数の中にはこの方法で変更できないものがあります。また、スーパーユーザがこのコマンドを発行した時にのみ変更できるものもあります。スーパーユーザのみがすべてのデータベースにおけるすべてのロール用の設定を変更することができます。

パラメータ

name

属性を変更するロールの名前です。

CURRENT_USER

明示的にロールを指定する代わりに現在のユーザを変更します。

SESSION_USER

明示的にロールを指定する代わりに現在のセッションユーザを変更します。

SUPERUSER
NOSUPERUSER
CREATEDB
NOCREATEDB
CREATEROLE
NOCREATEROLE
INHERIT
NOINHERIT
LOGIN
NOLOGIN
REPLICATION
NOREPLICATION
BYPASSRLS
NOBYPASSRLS
CONNECTION LIMIT connlimit
[ENCRYPTED] PASSWORD 'password'
PASSWORD NULL
VALID UNTIL 'timestamp'

これらの句は、元々CREATE ROLEで設定された属性を変更します。詳細はCREATE ROLEのマニュアルページを参照してください。

new_name

ロールの新しい名前です。

database_name

設定変数を設定する対象のデータベースの名前です。

configuration_parameter
value

指定した設定パラメータに対して、ロールのセッションデフォルトを指定した値に設定します。valueがDEFAULT、またはRESETが使用されていた場合、ロール固有の変数設定は削除され、新しいセッションではロールはシステム全体のデフォルト設定を継承します。すべてのロール固有の設定を削除するにはRESET ALLを使用してください。SET FROM CURRENTはセッションのパラメータ値をロール固有の値として保管します。IN DATABASEが指定された場合、設定パラメータは指定されたロールとデータベースのみで設定または削除されます。

ロール固有の変数設定はログイン時のみに影響を与えます。SET ROLEおよびSET SESSION AUTHORIZATIONはロール固有の変数設定を処理しません。

取り得るパラメータ名とその値に関する詳細はSETおよび第19章を参照してください。

注釈

新規にロールを追加するにはCREATE ROLEを使用してください。また、ロールを削除するにはDROP ROLEを使用してください。

ALTER ROLEではロールのメンバ資格を変更できません。メンバ資格の変更にはGRANTおよびREVOKEを使用してください。

このコマンドで暗号化しないパスワードを指定するときには注意しなければなりません。パスワードはサーバに平文で送信されます。クライアントのコマンド履歴やサーバのログにこれが残ってしまうかもしれません。psqlには\passwordコマンドがあります。これを使用してロールのパスワードを平文のパスワードをさらすことなく変更することができます。

ロールではなくデータベースにセッションのデフォルトを結びつけることもできます。ALTER DATABASEを参照してください。競合する場合、データベースとロールの組み合わせに固有な設定はロール固有の設定よりも優先し、ロール固有の設定はデータベース固有の設定よりも優先します。

例

ロールのパスワードを変更します。

```
ALTER ROLE davide WITH PASSWORD 'hu8jmn3';
```

ロールのパスワードを削除します。

```
ALTER ROLE davide WITH PASSWORD NULL;
```

パスワードの有効期限を変更し、UTCの1時間進んだタイムゾーンを使用して、2015年5月4日正午にパスワードが無効となるように指定します。

```
ALTER ROLE chris VALID UNTIL 'May 4 12:00:00 2015 +1';
```

パスワードの有効期限を無効にします。

```
ALTER ROLE fred VALID UNTIL 'infinity';
```

ロールに他のロールの作成権限と新しいデータベースの作成権限を与えます。

```
ALTER ROLE miriam CREATEROLE CREATEDB;
```

ロールにmaintenance_work_memパラメータ用のデフォルトとは異なる設定を与えます。

```
ALTER ROLE worker_bee SET maintenance_work_mem = 100000;
```

ロールにデータベース固有のclient_min_messagesパラメータ用のデフォルトとは異なる設定を与えます。

```
ALTER ROLE fred IN DATABASE devel SET client_min_messages = DEBUG;
```

互換性

ALTER ROLE文はPostgreSQLの拡張です。

関連項目

CREATE ROLE, DROP ROLE, ALTER DATABASE, SET

ALTER ROUTINE

ALTER ROUTINE — ルーチンの定義を変更する

概要

```
ALTER ROUTINE name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    action [ ... ] [ RESTRICT ]
ALTER ROUTINE name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    RENAME TO new_name
ALTER ROUTINE name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER ROUTINE name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    SET SCHEMA new_schema
ALTER ROUTINE name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    DEPENDS ON EXTENSION extension_name


ここでactionは以下のいずれかです。

    IMMUTABLE | STABLE | VOLATILE | [ NOT ] LEAKPROOF
    [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
    PARALLEL { UNSAFE | RESTRICTED | SAFE }
    COST execution_cost
    ROWS result_rows
    SET configuration_parameter { TO | = } { value | DEFAULT }
    SET configuration_parameter FROM CURRENT
    RESET configuration_parameter
    RESET ALL
```

説明

ALTER ROUTINEはルーチン、すなわち、集約関数や通常の関数、プロシージャの定義を変更します。
パラメータ説明や更なる例、より詳細については、ALTER AGGREGATE、ALTER FUNCTION、ALTER PROCEDUREを参照してください。

例

integer型に対するルーチンfooをfoobarに名前変更します。

```
ALTER ROUTINE foo(integer) RENAME TO foobar;
```

このコマンドはfooが集約、関数、プロシージャの何れであるかによらず動作します。

互換性

この文はSQL標準のALTER ROUTINE文と部分的に互換性があります。より詳しくはALTER FUNCTIONとALTER PROCEDUREを参照してください。ルーチン名が集約関数を参照できるのはPostgreSQLの拡張です。

関連項目

ALTER AGGREGATE, ALTER FUNCTION, ALTER PROCEDURE, DROP ROUTINE

CREATE ROUTINEコマンドは無いことに注意してください。

ALTER RULE

ALTER RULE — ルールの定義を変更する

概要

```
ALTER RULE name ON table_name RENAME TO new_name
```

説明

ALTER RULEは既存のルールの属性を変更します。現時点で利用可能な操作はルールの名称変更のみです。

ALTER RULEを使用するためには、ルールを適用するテーブルまたはビューの所有者でなければなりません。

パラメータ

name

変更対象の既存のルールの名前です。

table_name

ルールを適用するテーブルまたはビューの名前（スキーマ修飾可）です。

new_name

ルールの新しい名前です。

例

既存のルールの名前を変更します。

```
ALTER RULE notify_all ON emp RENAME TO notify_me;
```

互換性

ALTER RULEはPostgreSQLの言語拡張で、問い合わせ書き換えシステム全体も言語拡張です。

関連項目

CREATE RULE, DROP RULE

ALTER SCHEMA

ALTER SCHEMA — スキーマ定義を変更する

概要

```
ALTER SCHEMA name RENAME TO new_name
ALTER SCHEMA name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
```

説明

ALTER SCHEMAはスキーマ定義を変更します。

ALTER SCHEMAを使用するにはスキーマの所有者でなければなりません。スキーマ名を変更するには、そのデータベースのCREATE権限を持たなければなりません。所有者を変更するには、直接または間接的に新しい所有者ロールのメンバでなければなりません。また、そのロールがデータベースにおいてCREATEを持たなければなりません。（スーパーユーザがこれらの権限をすべて自動的に持つことに注意してください。）

パラメータ

name

既存のスキーマの名前です。

new_name

新しいスキーマの名前です。pg_から始まる名前は、システムスキーマとして予約されているため使用することができません。

new_owner

スキーマの新しい所有者です。

互換性

標準SQLにはALTER SCHEMA文はありません。

関連項目

CREATE SCHEMA, DROP SCHEMA

ALTER SEQUENCE

ALTER SEQUENCE — シーケンスジェネレータの定義を変更する

概要

```
ALTER SEQUENCE [ IF EXISTS ] name
    [ AS data_type ]
    [ INCREMENT [ BY ] increment ]
    [ MINVALUE minvalue | NO MINVALUE ] [ MAXVALUE maxvalue | NO MAXVALUE ]
    [ START [ WITH ] start ]
    [ RESTART [ [ WITH ] restart ] ]
    [ CACHE cache ] [ [ NO ] CYCLE ]
    [ OWNED BY { table_name.column_name | NONE } ]
ALTER SEQUENCE [ IF EXISTS ] name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER SEQUENCE [ IF EXISTS ] name RENAME TO new_name
ALTER SEQUENCE [ IF EXISTS ] name SET SCHEMA new_schema
```

説明

ALTER SEQUENCEは、既存のシーケンスジェネレータのパラメータを変更します。ALTER SEQUENCEで指定されなかったパラメータについては、以前の設定が保持されます。

ALTER SEQUENCEを使用するには、シーケンスの所有者でなければなりません。シーケンスのスキーマを変更するには、新しいスキーマにおけるCREATE権限も持たなければなりません。所有者を変更するには、新しく所有者となるロールの直接または間接的なメンバでなければなりません。またそのロールはシーケンスのスキーマ上にCREATE権限を持たなければなりません。(これらの制限は、シーケンスの削除および再作成によりユーザが実行できないことを、所有者の変更により実行されないようにするためのものです。しかし、スーパーユーザはすべてのシーケンスの所有者を変更することができます。)

パラメータ

name

変更するシーケンスの名前です(スキーマ修飾名も可)。

IF EXISTS

シーケンスが存在しない場合にエラーとしません。この場合、注意メッセージが発生します。

data_type

オプション句AS data_typeはシーケンスのデータ型を変更します。有効な型はsmallint、integer、bigintです。

データ型を変更したとき、以前の最小値と最大値は、古いデータ型の最小値と最大値に一致している場合に限り(別の言い方をすれば、暗示的にせよ明示的にせよ、シーケンスがNO MINVALUEまたはNO

MAXVALUEで作られていた場合に)、シーケンスの最小値および最大値が自動的に変更されます。そうでない場合、新しい値が同じコマンドの一部として指定されているのでなければ、最小値と最大値は保存されます。最小値と最大値が新しいデータ型に適合しない場合は、エラーが生成されます。

increment

INCREMENT BY increment句は省略可能です。正の値が指定された時は昇順のシーケンス、負の値が指定された時は降順のシーケンスにします。指定がない場合、以前の増分値が保持されます。

minvalue
NO MINVALUE

MINVALUE minvalue句はシーケンスジェネレータが生成する最小値を決定します。NO MINVALUEが指定された場合、昇順の時は1、降順の時はデータ型の最小値がデフォルトになります。どちらのオプションも指定されていなければ、現在の最小値が保持されます。

maxvalue
NO MAXVALUE

MAXVALUE maxvalue句はシーケンスが生成する最大値を決定します。NO MAXVALUEが指定された場合、昇順の時はデータ型の最大値、降順の時は-1がデフォルトになります。どちらのオプションも指定されていなければ、現在の最大値が保持されます。

start

START WITH start句(省略可能)は、記録されているシーケンスの開始値を変更します。これは現在のシーケンス値に影響しません。単に将来実行されるALTER SEQUENCE RESTARTコマンドが使用する値を設定するだけです。

restart

RESTART WITH restart句(省略可能)は、シーケンスの現在値を変更します。これはis_called = falseとしてsetval関数を呼び出すことと似ています。指定した値は次のnextval呼出時に返されます。restartを付けずにRESTARTと記述することは、CREATE SEQUENCEで記録、または前回ALTER SEQUENCE START WITHで設定された開始値を指定することと同じです。

setvalの呼び出しとは異なり、シーケンスに対するRESTARTの操作はトランザクション的で、同時実行中のトランザクションが同じシーケンスから値を取得するのをブロックします。それが期待する動作でないときは、setvalを使用してください。

cache

CACHE cache句を使用すると、アクセスを高速化するために、シーケンス番号を事前に割り当て、メモリに保存しておくことができます。最小値は1です(一度に生成する値が1つだけなので、キャッシュがない状態になります)。指定がなければ、以前のキャッシュ値が保持されます。

CYCLE

CYCLEキーワードを使用すると、シーケンスが限界値(昇順の場合はmaxvalue、降順の場合はminvalue)に達した時、そのシーケンスを周回させることができます。限界値に達した時、次に生成される番号は、昇順の場合はminvalue、降順の場合はmaxvalueになります。

NO CYCLE

NO CYCLEキーワードが指定されると、シーケンスの限界値に達した後のnextval呼び出しは全てエラーとなります。CYCLEもNO CYCLEも指定されていない場合は、以前の周回動作が保持されます。

OWNED BY table_name.column_name
OWNED BY NONE

OWNED BYオプションにより、シーケンスは指定されたテーブル列に関連付けられ、その列(やテーブル全体)が削除されると、自動的にシーケンスも一緒に削除されるようになります。指定があると、以前に指定されたシーケンスの関連は、指定された関連に置き換えられます。指定するテーブルは、シーケンスと同一所有者でなければならず、また、同一のスキーマ内に存在しなければなりません。OWNED BY NONEを指定することで、既存の関連は削除され、シーケンスは「独立」したものになります。

new_owner

シーケンスの新しい所有者のユーザ名です。

new_name

シーケンスの新しい名称です。

new_schema

シーケンスの新しいスキーマです。

注釈

ALTER SEQUENCEは、コマンドを実行したバックエンド以外のバックエンドにおけるnextvalに対しては、すぐには効力を発揮しません。これらのバックエンドに事前に割り当てられた(キャッシュされた)シーケンス値がある場合、この値を全て使い果たした後に、変更されたシーケンス生成パラメータを検知します。コマンドを実行したバックエンドには、即座に変更が反映されます。

ALTER SEQUENCEはシーケンスのcurrval状態には影響しません。(8.3より前のPostgreSQLでは影響を与える場合がありました。)

ALTER SEQUENCEは、同時に実行されるnextval、currval、lastval、setvalの呼び出しをブロックします。

歴史的な理由によりALTER TABLEはシーケンスにも使用することができます。しかし、シーケンスに対して許されるALTER TABLEの構文は、上で示した構文と等価なものだけです。

例

serialというシーケンスを105から再開します。

```
ALTER SEQUENCE serial RESTART WITH 105;
```

互換性

ALTER SEQUENCEは、PostgreSQLの拡張であるAS、START WITH、OWNED BY、OWNER TO、RENAME TO、SET SCHEMA構文を除いて、標準SQLに従っています。

関連項目

CREATE SEQUENCE, DROP SEQUENCE

ALTER SERVER

ALTER SERVER — 外部サーバの定義を変更する

概要

```
ALTER SERVER name [ VERSION 'new_version' ]
    [ OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ... ] ) ]
ALTER SERVER name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER SERVER name RENAME TO new_name
```

説明

ALTER SERVERは外部サーバの定義を変更します。第1の構文はサーバのバージョン文字列、またはサーバの一般的なオプションを変更します。（少なくとも1つの句が必要です。）第2の構文はサーバの所有者を変更します。

サーバを変更するためには、サーバの所有者でなければなりません。さらに所有者を変更するためには、サーバを所有し、かつ、直接または間接的に新しい所有者のロールのメンバでなければなりません。また、サーバの外部データラッパに対してUSAGE権限も必要です。（スーパーユーザはこれらの判定基準すべてを自動的に満たしていることに注意してください。）

パラメータ

name

既存のサーバの名前です。

new_version

新しいサーバのバージョンです。

OPTIONS ([ADD | SET | DROP] option ['value'] [, ...])

サーバのオプションを変更します。ADD、SET、DROPは行う動作を指定します。明示的な動作の指定がない場合ADDとみなされます。オプション名は一意でなければなりません。また、名前と値はサーバの外部データラッパのライブラリを使用して検証されます。

new_owner

外部サーバの新しい所有者のユーザ名です。

new_name

外部サーバの新しい名前です。

例

サーバfooを変更し、接続オプションを追加します。

```
ALTER SERVER foo OPTIONS (host 'foo', dbname 'foodb');
```

サーバfooを変更し、バージョンとhostオプションを変更します。

```
ALTER SERVER foo VERSION '8.4' OPTIONS (SET host 'baz');
```

互換性

ALTER SERVERはISO/IEC 9075-9 (SQL/MED)に従います。OWNER TOとRENAME構文はPostgreSQLの拡張です。

関連項目

CREATE SERVER, DROP SERVER

ALTER STATISTICS

ALTER STATISTICS — 拡張統計オブジェクトの定義を変更する

概要

```
ALTER STATISTICS name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER STATISTICS name RENAME TO new_name
ALTER STATISTICS name SET SCHEMA new_schema
ALTER STATISTICS name SET STATISTICS new_target
```

説明

ALTER STATISTICSは既存の拡張統計オブジェクトのパラメータを変更します。ALTER STATISTICSコマンドで明示的に設定されないパラメータは、以前の設定を保持します。

ALTER STATISTICSを使用するには、その統計オブジェクトを所有していなければなりません。統計オブジェクトのスキーマを変更するには、新しいスキーマに対するCREATE権限も持っていなければなりません。所有者を変更するには、新しい所有ロールの直接的あるいは間接的なメンバーでも無ければならず、またそのロールは統計オブジェクトのスキーマに対するCREATE権限を持っていなければなりません。(これらの制限は、統計オブジェクトを削除し、そして再作成することによって実現できないことを、所有者を変更することで実現できることがないことを強制するものです。しかし、スーパーユーザはどの統計オブジェクトの所有者も変更できます。)

パラメータ

name

変更の対象となる統計オブジェクトの名前(オプションでスキーマ修飾可)です。

new_owner

統計オブジェクトの新しい所有者のユーザ名です。

new_name

統計オブジェクトの新しい名前です。

new_schema

統計オブジェクトの新しいスキーマです。

new_target

後続のANALYZE操作でこの統計オブジェクトについて統計情報を集める目標です。目標は0から10000の範囲で設定できます。あるいは、設定されていれば、被参照列の統計目標の最大値を、もしくは、シ

ステムのデフォルトの統計目標(default_statistics_target)を使うように戻すために-1に設定します。PostgreSQL問い合わせプランナによる統計情報の使用に関する詳細な情報は14.2を参照してください。

互換性

標準SQLにはALTER STATISTICSコマンドはありません。

関連項目

CREATE STATISTICS, DROP STATISTICS

ALTER SUBSCRIPTION

ALTER SUBSCRIPTION — サブスクリプションの定義を変更する

概要

```
ALTER SUBSCRIPTION name CONNECTION 'conninfo'
ALTER SUBSCRIPTION name SET PUBLICATION publication_name [, ...] [ WITH ( set_publication_option
 [= value] [, ... ] ) ]
ALTER SUBSCRIPTION name REFRESH PUBLICATION [ WITH ( refresh_option [= value] [, ... ] ) ]
ALTER SUBSCRIPTION name ENABLE
ALTER SUBSCRIPTION name DISABLE
ALTER SUBSCRIPTION name SET ( subscription_parameter [= value] [, ... ] )
ALTER SUBSCRIPTION name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER SUBSCRIPTION name RENAME TO new_name
```

説明

ALTER SUBSCRIPTIONはCREATE SUBSCRIPTIONで指定できるサブスクリプションの属性のほとんどを変更できます。

ALTER SUBSCRIPTIONを使用するには、そのサブスクリプションを所有していなければなりません。所有者を変更するには、新しい所有ロールの直接的あるいは間接的メンバーでもなければなりません。新しい所有者はスーパーユーザである必要があります。（現在は、すべてのサブスクリプションの所有者はスーパーユーザでなければならず、そのため所有者のチェックは実際には回避されます。しかしこれは将来、変更されるかもしれません。）

パラメータ

name

属性の変更の対象となるサブスクリプションの名前です。

CONNECTION 'conninfo'

この句では、元はCREATE SUBSCRIPTIONにより設定された接続の属性を変更します。詳細な情報はそちらを参照してください。

SET PUBLICATION publication_name

サブスクライブするパブリケーションのリストを変更します。詳細はCREATE SUBSCRIPTIONを参照してください。デフォルトでは、このコマンドはREFRESH PUBLICATIONのような動作もします。

set_publication_optionは、この操作についての追加のオプションを指定します。以下のオプションがサポートされています。

refresh (boolean)

falseにすると、このコマンドはテーブルを情報を更新しません。後で別にREFRESH PUBLICATIONを実行することになります。デフォルトはtrueです。

さらにREFRESH PUBLICATIONの項で説明されているrefreshオプションを指定できます。

REFRESH PUBLICATION

不足しているテーブル情報をパブリッシャーから取得します。最後のREFRESH PUBLICATION、あるいはCREATE SUBSCRIPTIONの実行の後でサブスクライブ対象のパブリケーションに追加されたテーブルの複製が、これにより開始されます。

refresh_optionは更新(refresh)の操作について追加のオプションを指定します。以下のオプションがサポートされています。

copy_data (boolean)

サブスクライブ対象のパブリケーションにある既存のデータが、レプリケーションの開始時にコピーされるかどうかを指定します。デフォルトはtrueです。(以前にサブスクライブされたテーブルはコピーされません。)

ENABLE

以前に無効化されたサブスクリプションを有効化し、トランザクションの終了時に論理レプリケーションワーカを起動します。

DISABLE

実行中のサブスクリプションを無効化し、トランザクションの終了時に論理レプリケーションワーカを停止します。

SET (subscription_parameter [= value] [, ...])

この句では、元はCREATE SUBSCRIPTIONにより設定されたパラメータを変更します。詳細な情報はそちらを参照してください。使用できるオプションはslot_nameとsynchronous_commitです。

new_owner

サブスクリプションの新しい所有者のユーザ名です。

new_name

サブスクリプションの新しい名前です。

例

サブスクリプションがサブスクライブするパブリケーションをinsert_onlyに変更します。

```
ALTER SUBSCRIPTION mysub SET PUBLICATION insert_only;
```

サブスクリプションを無効化(停止)します。

```
ALTER SUBSCRIPTION mysub DISABLE;
```

互換性

ALTER SUBSCRIPTIONはPostgreSQLの拡張です。

関連項目

CREATE SUBSCRIPTION, DROP SUBSCRIPTION, CREATE PUBLICATION, ALTER PUBLICATION

ALTER SYSTEM

ALTER SYSTEM — サーバの設定パラメータを変更する

概要

```
ALTER SYSTEM SET configuration_parameter { TO | = } { value | 'value' | DEFAULT }

ALTER SYSTEM RESET configuration_parameter
ALTER SYSTEM RESET ALL
```

説明

ALTER SYSTEMはデータベースクラスタ全体にわたるサーバの設定パラメータを変更するのに使われます。手作業でpostgresql.confファイルを編集するという伝統的な方法よりも、この方が便利かもしれません。ALTER SYSTEMは指定されたパラメータ設定をpostgresql.auto.confファイルに書き出し、これがpostgresql.confに加えて読み込まれます。パラメータをDEFAULTに設定する、あるいはこれの別表記であるRESETを使うと、postgresql.auto.confファイルから、その設定のエントリを削除します。そのような設定のエントリをすべて削除するにはRESET ALLを使用してください。

ALTER SYSTEMで設定された値は、次回のサーバ設定の再ロードで、またサーバ開始時にのみ変更可能なパラメータについては次回のサーバ再起動で有効になります。サーバ設定の再ロードは、SQL関数pg_reload_conf()の呼び出し、pg_ctl reloadの実行、あるいはメインのサーバプロセスにSIGHUPを送信することで実行できます。

ALTER SYSTEMを実行できるのはスーパーユーザーだけです。このコマンドはファイルシステムに直接作用し、ロールバックできないため、トランザクションブロックや関数の内部で使うことはできません。

パラメータ

configuration_parameter

設定する設定パラメータの名前です。利用可能なパラメータに付いては第19章に記述されています。

value

パラメータの新しい値です。値は、対象のパラメータとして適切な文字列定数、識別子、数値あるいはそれらをカンマで区切ったリストで指定できます。DEFAULTと指定することができ、このとき、パラメータとその値をpostgresql.auto.confから削除します。

注釈

このコマンドではdata_directoryおよびpostgresql.confで設定できないパラメータ(例えばpreset options)を設定することはできません。

パラメータを設定するその他の方法については19.1を参照してください。

例

wal_levelを設定します。

```
ALTER SYSTEM SET wal_level = replica;
```

それを取り消して、postgresql.confで有効だった設定に戻します。

```
ALTER SYSTEM RESET wal_level;
```

互換性

ALTER SYSTEM文はPostgreSQLの拡張です。

関連項目

SET, SHOW

ALTER TABLE

ALTER TABLE — テーブル定義を変更する

概要

```
ALTER TABLE [ IF EXISTS ] [ ONLY ] name [ * ]
    action [, ... ]
ALTER TABLE [ IF EXISTS ] [ ONLY ] name [ * ]
    RENAME [ COLUMN ] column_name TO new_column_name
ALTER TABLE [ IF EXISTS ] [ ONLY ] name [ * ]
    RENAME CONSTRAINT constraint_name TO new_constraint_name
ALTER TABLE [ IF EXISTS ] name
    RENAME TO new_name
ALTER TABLE [ IF EXISTS ] name
    SET SCHEMA new_schema
ALTER TABLE ALL IN TABLESPACE name [ OWNED BY role_name [, ... ] ]
    SET TABLESPACE new_tablespace [ NOWAIT ]
ALTER TABLE [ IF EXISTS ] name
    ATTACH PARTITION partition_name { FOR VALUES partition_bound_spec | DEFAULT }
ALTER TABLE [ IF EXISTS ] name
    DETACH PARTITION partition_name


ここで、
      actionは以下のいずれかです。

    ADD [ COLUMN ] [ IF NOT EXISTS ] column_name data_type [ COLLATE collation ]
 [ column_constraint [ ... ] ]
    DROP [ COLUMN ] [ IF EXISTS ] column_name [ RESTRICT | CASCADE ]
    ALTER [ COLUMN ] column_name [ SET DATA ] TYPE data_type [ COLLATE collation ]
 [ USING expression ]
    ALTER [ COLUMN ] column_name SET DEFAULT expression
    ALTER [ COLUMN ] column_name DROP DEFAULT
    ALTER [ COLUMN ] column_name { SET | DROP } NOT NULL
    ALTER [ COLUMN ] column_name DROP EXPRESSION [ IF EXISTS ]
    ALTER [ COLUMN ] column_name ADD GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY
 [ ( sequence_options ) ]
    ALTER [ COLUMN ] column_name { SET GENERATED { ALWAYS | BY DEFAULT } | SET sequence_option |
 RESTART [ [ WITH ] restart ] } [...]
    ALTER [ COLUMN ] column_name DROP IDENTITY [ IF EXISTS ]
    ALTER [ COLUMN ] column_name SET STATISTICS integer
    ALTER [ COLUMN ] column_name SET ( attribute_option = value [, ... ] )
    ALTER [ COLUMN ] column_name RESET ( attribute_option [, ... ] )
    ALTER [ COLUMN ] column_name SET STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN }
```

```
    ADD table_constraint [ NOT VALID ]
    ADD table_constraint_using_index
    ALTER CONSTRAINT constraint_name [ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED |
 INITIALLY IMMEDIATE ]
    VALIDATE CONSTRAINT constraint_name
    DROP CONSTRAINT [ IF EXISTS ]  constraint_name [ RESTRICT | CASCADE ]
    DISABLE TRIGGER [ trigger_name | ALL | USER ]
    ENABLE TRIGGER [ trigger_name | ALL | USER ]
    ENABLE REPLICA TRIGGER trigger_name
    ENABLE ALWAYS TRIGGER trigger_name
    DISABLE RULE rewrite_rule_name
    ENABLE RULE rewrite_rule_name
    ENABLE REPLICA RULE rewrite_rule_name
    ENABLE ALWAYS RULE rewrite_rule_name
    DISABLE ROW LEVEL SECURITY
    ENABLE ROW LEVEL SECURITY
    FORCE ROW LEVEL SECURITY
    NO FORCE ROW LEVEL SECURITY
    CLUSTER ON index_name
    SET WITHOUT CLUSTER
    SET WITHOUT OIDS
    SET TABLESPACE new_tablespace
    SET { LOGGED | UNLOGGED }
    SET ( storage_parameter [= value] [, ... ] )
    RESET ( storage_parameter [, ... ] )
    INHERIT parent_table
    NO INHERIT parent_table
    OF type_name
    NOT OF
    OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
    REPLICA IDENTITY { DEFAULT | USING INDEX index_name | FULL | NOTHING }


また、
      partition_bound_specは以下のいずれかです。

IN ( partition_bound_expr [, ...] ) |
FROM ( { partition_bound_expr | MINVALUE | MAXVALUE } [, ...] )
  TO ( { partition_bound_expr | MINVALUE | MAXVALUE } [, ...] ) |
WITH ( MODULUS numeric_literal, REMAINDER numeric_literal )


また、
      column_constraintは以下の通りです。

[ CONSTRAINT constraint_name ]
```

```
{ NOT NULL |
  NULL |
  CHECK ( expression ) [ NO INHERIT ] |
  DEFAULT default_expr |
  GENERATED ALWAYS AS ( generation_expr ) STORED |
  GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY [ ( sequence_options ) ] |
  UNIQUE index_parameters |
  PRIMARY KEY index_parameters |
  REFERENCES reftable [ ( refcolumn ) ] [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ]
    [ ON DELETE referential_action ] [ ON UPDATE referential_action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]


また、
      table_constraintは以下の通りです。

[ CONSTRAINT constraint_name ]
{ CHECK ( expression ) [ NO INHERIT ] |
  UNIQUE ( column_name [, ... ] ) index_parameters |
  PRIMARY KEY ( column_name [, ... ] ) index_parameters |
  EXCLUDE [ USING index_method ] ( exclude_element WITH operator [, ... ] ) index_parameters
 [ WHERE ( predicate ) ] |
  FOREIGN KEY ( column_name [, ... ] ) REFERENCES reftable [ ( refcolumn [, ... ] ) ]
    [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ] [ ON DELETE referential_action ] [ ON
 UPDATE referential_action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]


またtable_constraint_using_indexは以下の通りです。

    [ CONSTRAINT constraint_name ]
    { UNIQUE | PRIMARY KEY } USING INDEX index_name
    [ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]


UNIQUE、
      PRIMARY KEY、
      および、
      EXCLUDE制約でのindex_parametersは以下の通りです。

[ INCLUDE ( column_name [, ... ] ) ]
[ WITH ( storage_parameter [= value] [, ... ] ) ]
[ USING INDEX TABLESPACE tablespace_name ]


EXCLUDE制約でのexclude_elementは以下の通りです。
```

```
{ column_name | ( expression ) } [ opclass ] [ ASC | DESC ] [ NULLS { FIRST | LAST } ]
```

説明

ALTER TABLEは既存のテーブルの定義を変更します。以下のようにいくつかの副構文があります。要求されるロックレベルはそれぞれの副構文によって異なることに注意してください。特に記述がなければACCESS EXCLUSIVEロックを取得します。複数のサブコマンドが使われるときは、それらのサブコマンドが要求するうち、もっとも高いレベルのロックを取得します。

ADD COLUMN [IF NOT EXISTS]

この構文を使用すると、CREATE TABLEと同じ構文を使って新しい列をテーブルに追加できます。IF NOT EXISTSが指定され、その名前の列が既に存在している場合は、エラーが発生しません。

DROP COLUMN [IF EXISTS]

この構文を使用すると、テーブルから列を削除できます。削除する列を含んでいるインデックスおよびテーブル制約も自動的に削除されます。削除する列を参照する多変量統計がある場合、列の削除の結果、その統計が1つの列のデータしか含まないようになるなら、それも削除されます。また、削除する列にテーブル以外が依存(例えば、外部キー制約、ビューなど)している場合、CASCADEを付ける必要があります。IF EXISTSが指定されている場合、もしその列がなかったとしてもエラーにはなりません。この場合は代わりに注意が出力されます。

SET DATA TYPE

この構文を使用すると、テーブルの列の型を変更できます。その列を含むインデックスと簡単なテーブル制約は、元々与えられた式を再解析し、新しい型を使用するように自動的に変換されます。COLLATE句を使うと、新しい列の照合順を指定できます。省略時の照合順は新しい列の型のデフォルトになります。USING句を使うと、古い列値をどのように新しい値に計算するかを指定できます。省略された場合、デフォルトの変換は、古いデータ型から新しいデータ型への代入キャストと同じになります。古いデータ型から新しいデータ型への暗黙キャストあるいは代入キャストがない場合、USING句を指定しなければなりません。

SET/DROP DEFAULT

これらの構文を使用すると、列のデフォルト値を設定または削除できます(ここで、削除はデフォルト値をNULLに設定することと等価です)。新しいデフォルト値は、変更後に行われるINSERTまたはUPDATEコマンドにのみ適用されます。テーブル内の既存の行は変更されません。

SET/DROP NOT NULL

これらの構文は、列の値としてNULL値を認めるか拒絶するかを変更します。

SET NOT NULLは、テーブルの項目でその列がNULL値であるものが1つもない場合にのみ、その列に設定可能です。通常これはALTER TABLEがテーブル全体をスキャンする際に確認されます。しかしながら、NULLが存在できないことを示す有効なCHECK制約が見つかれば、テーブルスキャンは省略されます。

このテーブルがパーティションの場合、親テーブルでNOT NULLの印がつけられている列についてDROP NOT NULLを実行することはできません。すべてのパーティションからNOT NULL制約を削除するには、親テーブルでDROP NOT NULLを実行してください。親テーブルにNOT NULL制約がない場合でも、望むなら

各パーティションにそのような制約を追加することができます。つまり、親テーブルがNULLを許していても子テーブルでNULLを禁止することができますが、その逆はできません。

DROP EXPRESSION [IF EXISTS]

この構文は、格納された生成列を通常の基本列に変換します。列の既存のデータは保持されますが、以後の変更はもはや生成式を適用しません。

DROP EXPRESSION IF EXISTSが指定され、その列が格納された生成列でない場合は、エラーを発生させません。この場合、注意メッセージが発行されます。

ADD GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY
SET GENERATED { ALWAYS | BY DEFAULT }
DROP IDENTITY [IF EXISTS]

この構文では、列がIDENTITY列であるかどうか、または既存のIDENTITY列の生成属性を変更することができます。詳細はCREATE TABLEを参照してください。SET DEFAULT同様に、この構文は、変更後に行われるINSERTまたはUPDATEコマンドにのみ適用されます。テーブル内の既存の行は変更されません。

DROP IDENTITY IF EXISTSが指定され、その列がIDENTITY列でない場合は、エラーを発生させません。この場合、注意メッセージが発行されます。

SET sequence_option
RESTART

この構文では、既存のIDENTITY列に紐付けられているシーケンスを変更します。sequence_optionはINCREMENT BYなどALTER SEQUENCEがサポートするオプションです。

SET STATISTICS

この構文は、以後のANALYZE操作において、列単位での統計情報収集対象を設定します。対象として、0から10000までの範囲の値を設定可能です。また、対象を-1に設定すると、システムのデフォルト統計情報対象(default_statistics_target)が使用されます。PostgreSQLの問い合わせプランナによる統計情報の使用に関する詳細は、14.2を参照してください。

SET STATISTICSはSHARE UPDATE EXCLUSIVEロックを取得します。

SET (attribute_option = value [, ...])
RESET (attribute_option [, ...])

この構文は属性単位のオプションの設定または設定解除を行います。現時点では属性単位のオプションで定義されているのはn_distinctおよびn_distinct_inheritedのみです。これらのオプションは、その後のANALYZE操作により生成される個別値数の推定値を上書きします。n_distinctはテーブル自身の統計情報に影響を与え、n_distinct_inheritedはテーブルとそれを継承した子テーブルから集めた統計情報に影響を与えます。正の値の場合、ANALYZEは、その列には、正確に指定された個数の非NULLの個別値が存在するものとみなします。負の値の場合、この値は-1以上でなければなりませんが、ANALYZEは、その列内の非NULLの個別値はテーブルのサイズに線形であるとみなし、推定テーブルサイズに指定した値の絶対値を乗じた値が個別値数であるとみなします。たとえば、-1という値は、列内のすべての値に重複がないことを意味し、-0.5という値は個々の値は平均して2回現れることを意味します。テーブルの行数との乗算は問い合わせ計画を作成するまで行われませんので、テーブルサイズが変わり続けるような場合にこれは有用かもしれません。0という値を指定することで、個別値数の推定を通常

に戻します。PostgreSQL問い合わせプランナにおける統計情報の使用に関しては14.2を参照してください。

属性単位のオプションの変更はSHARE UPDATE EXCLUSIVEロックを取得します。

SET STORAGE

この構文は、列の保管モードを設定します。列をインラインで保持するか補助TOASTテーブルに保持するか、また、データを圧縮するかどうかを制御できます。PLAINは、integerのような固定長の値に対して使用します。インラインで保持され、圧縮されません。MAINは、インラインで保持されていて、圧縮可能なデータに使用します。EXTERNALは圧縮されていない外部データに使用します。EXTENDEDは圧縮された外部データに使用します。EXTENDEDは、PLAIN以外の保管をサポートするほとんどのデータ型におけるデフォルトです。EXTERNALを使用すると、非常に長いtextおよびbytea列に対する部分文字列操作の処理速度が向上しますが、必要な保管容量が増えるというデメリットがあります。SET STORAGE自体はテーブルをまったく変更しないことに注意してください。以後のテーブルの更新時に遂行する戦略を設定するだけです。詳細は68.2を参照してください。

ADD table_constraint [NOT VALID]

この構文は、CREATE TABLEと同じ制約構文に加え、現時点では外部キー制約と検査制約でのみ許されるNOT VALIDオプションを使って新しい制約をテーブルに追加します。

通常この構文は、テーブルの既存の行が新しい制約を満たすか確認するため、テーブルのスキャンの原因となります。しかし、NOT VALIDオプションが使われていれば、時間がかかるかもしれないこのスキャンは省略されます。それでも、制約はその後の挿入や更新に対して強制されます（つまり、外部キー制約の場合、被参照テーブルに一致する行が存在しない限り失敗します。指定された検査制約に一致する新しい行が存在しない限り失敗します）。しかしデータベースは、VALIDATE CONSTRAINTオプションを使用して検証されるまで、テーブル内のすべての行で制約が保持されているとみなしません。NOT VALIDオブションを使うことに関する更なる情報はNotes以下を参照してください。

ADD table_constraintのほとんどの構文ではACCESS EXCLUSIVEロックが必要ですが、ADD FOREIGN KEYではSHARE ROW EXCLUSIVEロックだけが必要です。ADD FOREIGN KEYは、制約を宣言したテーブルでのロックに加えて、被参照テーブルのSHARE ROW EXCLUSIVEロックも取得することに注意してください。

一意性制約や主キー制約がパーティションテーブルに追加されるときには、追加的な制限が適用されます。CREATE TABLEを参照してください。また、今のところ、パーティションテーブルでの外部キー制約ではNOT VALIDと宣言できません。

ADD table_constraint_using_index

この構文は、既存の一意性インデックスに基づき、テーブルにPRIMARY KEYまたはUNIQUE制約を新たに追加します。インデックスのすべての列がこの制約に含まれます。

このインデックスは式列を持つことはできず、また部分インデックスであってはいけません。またこれはデフォルトのソート順序を持つB-Treeインデックスでなければなりません。これらの制限により、このインデックスが通常のADD PRIMARY KEYまたはADD UNIQUEコマンドにより構築されたインデックスと等価であることを確実にします。

PRIMARY KEYが指定され、インデックスの列がNOT NULLと印付けされていない場合、このコマンドはこうした列のそれぞれに対してALTER COLUMN SET NOT NULLの実施を試みます。これは列にNULLが含まれ

ないことを検証するために完全なテーブルスキャンを必要とします。この他の場合においては、これが高速な操作です。

制約名が提供された場合、インデックスの名前は制約名に合うように変更されます。提供されない場合は制約にはインデックスと同じ名前が付けられます。

このコマンドの実行後、インデックスは、制約により「所有」され、それはインデックスが通常のADD PRIMARY KEYまたはADD UNIQUEにより構築された場合と同様です。特にこの制約を削除するとインデックスも消えてしまいます。

この形式は今のところパーティションテーブルではサポートされません。

> **注記**
>
> 既存のインデックスを使用した制約の追加は、テーブル更新を長時間ブロックすることなく新しい制約を追加しなければならない場合に有用になる可能性があります。このためには、CREATE INDEX CONCURRENTLYを用いてインデックスを作成し、この構文を使用して正式の制約としてインストールしてください。後述の例を参照してください。

ALTER CONSTRAINT

この構文は以前に作成された制約の属性を変更します。現在は外部キー制約のみを変更できます。

VALIDATE CONSTRAINT

この構文は、以前にNOT VALIDとして作成された外部キー制約または検査制約を、これらの制約を満たさない行が存在しないことを確認するためにテーブルをスキャンして、検証します。制約がすでに有効であると記録されている場合は何も起こりません。(このコマンドの有用性の説明はNotes以下を参照してください。)

DROP CONSTRAINT [IF EXISTS]

この構文はテーブル上の指定した制約を、制約の基となるインデックスと共に削除します。IF EXISTSが指定された場合、その制約がなくてもエラーになりません。この場合は代わりに注意が出力されます。

DISABLE/ENABLE [REPLICA | ALWAYS] TRIGGER

この構文を使用すると、テーブルに属するトリガの発行について設定することができます。無効にされたトリガはシステム上に存在し続けますが、トリガイベントが発生したとしても実行されません。遅延トリガの場合、有効無効状態の確認は、トリガ関数を実際に実行しようとする時ではなく、イベントの発生時に行われます。名前でトリガを1つ指定して有効または無効にすることもできますし、テーブル上のすべてのトリガを有効または無効にすることもできます。また、ユーザトリガのみを有効または無効にすることも可能です（このオプションは、外部キー制約、遅延可能な一意性および排他制約を実装するために使用される内部向けに生成される制約トリガを除外します。）内部向けに生成される制約トリガを有効または無効にするにはスーパーユーザ権限が必要です。トリガが実行されなかった場合は当然ながら制約の整合性が保証されませんので、制約トリガの無効化は注意して実行しなければなりません。

トリガ発行機構は設定変数session_replication_roleの影響も受けます。単に有効としたトリガ（デフォルト）では、レプリケーションロールが「origin」（デフォルト）または「local」の場合に発行されます。ENABLE REPLICAと設定されたトリガでは、セッションが「replica」モードである場合のみ発行されます。

そして、ENABLE ALWAYSと設定されたトリガでは、現在のレプリケーションロールに関係なく発行されます。

この仕組みの効果はデフォルト設定ではレプリカ上でトリガが発行しないことです。トリガがオリジンでテーブル間でデータを伝播するのに使われている場合にレプリケーションシステムは伝播したデータもレプリケーションします。レプリカ上でトリガが再度発動すべきではありませんので、これは有用です。しかしながら、トリガが外部的な警告を発するなどの他の意図で使われている場合、レプリカでもトリガが発行されるようにENABLE ALWAYSを設定するのが適切と言えます。

このコマンドはSHARE ROW EXCLUSIVEを取得します。

DISABLE/ENABLE [REPLICA | ALWAYS] RULE

この構文を使用すると、テーブルに属する書き換えルールの実行について設定することができます。ルールは無効にしてもシステムに残りますが、問い合わせ書き換え時に適用されません。この意味はトリガの有効化、無効化と同じです。この設定はON SELECTルールでは無視されます。現在のセッションがデフォルト以外のレプリケーションモードであったとしても、ビュー操作を維持するために常に適用されます。

前述のトリガと同様に、ルール発行機構は設定変数session_replication_roleの影響を受けます。

DISABLE/ENABLE ROW LEVEL SECURITY

これらの構文は、テーブルの行セキュリティポリシーの適用を制御します。有効にされ、かつテーブルにポリシーが存在しない場合は、デフォルトの拒絶ポリシーが適用されます。行単位セキュリティが無効になっている場合でも、テーブルのセキュリティが存在し得ることに注意してください。この場合、ポリシーは適用されず、無視されます。CREATE POLICYも参照してください。

NO FORCE/FORCE ROW LEVEL SECURITY

これらの構文は、ユーザがテーブルの所有者である場合について、テーブルの行セキュリティポリシーの適用を制御します。有効の場合、ユーザがテーブルの所有者であれば、行セキュリティポリシーが適用されます。無効（デフォルト）の場合、ユーザがテーブルの所有者であれば、行セキュリティポリシーは適用されません。CREATE POLICYも参照してください。

CLUSTER ON

この構文は、以後のCLUSTER操作用のデフォルトインデックスを選択します。テーブルの再クラスタ化は実際には行いません。

clusterオプションの変更はSHARE UPDATE EXCLUSIVEロックを取得します。

SET WITHOUT CLUSTER

この構文は、テーブルから、一番最後に適用されたCLUSTERインデックス指定を削除します。以後のインデックスを指定しないクラスタ操作に影響を及ぼします。

clusterオプションの変更はSHARE UPDATE EXCLUSIVEロックを取得します。

SET WITHOUT OIDS

システム列oidを削除する、後方互換のための構文です。システム列oidは今では追加できませんので、これは効果がありません。

SET TABLESPACE

この構文を使用すると、テーブルのテーブル空間を指定したテーブル空間に変更し、テーブルに関連するデータファイルを新しいテーブル空間に移動することができます。テーブルにインデックスがあっても移動されません。インデックスを移動するには、別途SET TABLESPACEコマンドを実行します。パーティションテーブルに適用された場合には何も移動されませんが、以後CREATE TABLE PARTITION OFで作られるパーティションは、TABLESPACE句により上書きされない限り、そのテーブル空間を使うようになります。

ALL IN TABLESPACE構文を使うことで、テーブル空間内の現在のデータベースのすべてのテーブルを移動することができます。この場合、移動されるすべてのテーブルがまずロックされ、それから一つずつ移動されます。この構文はOWNED BYもサポートしており、これを使うと、指定のロールが所有しているテーブルだけを移動します。NOWAITを指定した場合、必要とするすべてのロックを即座に獲得できなければ、このコマンドは失敗します。このコマンドではシステムカタログは移動されないことに注意し、必要なら代わりにALTER DATABASEを使うか、あるいはALTER TABLEで明示的に指定してください。information_schemaのリレーションはシステムカタログとはみなされないので、移動されます。CREATE TABLESPACEも参照してください。

SET { LOGGED | UNLOGGED }

この構文は、テーブルをログを取らないテーブルからログを取るテーブルに変更、あるいはその逆を行います(UNLOGGED参照)。これは一時テーブルに対して使うことはできません。

SET (storage_parameter [= value] [, ...])

この構文は、1つ以上のテーブルの格納パラメータを変更します。設定可能なパラメータの詳細に関してはCREATE TABLE文書のStorage Parametersを参照してください。このコマンドによってテーブルの内容が即座に変更されない点に注意してください。パラメータによりますが、期待する効果を得るためにテーブルを書き換える必要がある場合があります。このためには、VACUUM FULL、CLUSTERまたはテーブルを強制的に書き換えるALTER TABLEの構文のいずれかを使用してください。プランナに関連するパラメータについては、次にテーブルがロックされた時に変更が有効になるため、現在実行中の問い合わせは影響を受けません。

fillfactor、TOAST、およびautovacuumのストレージパラメータおよびプランナに関連するパラメータparallel_workersについてはSHARE UPDATE EXCLUSIVEロックが獲得されます。

RESET (storage_parameter [, ...])

この構文は、1つ以上の格納パラメータをデフォルト値に再設定します。SET同様、テーブル全体を更新するためにテーブルの書き換えが必要になる場合があります。

INHERIT parent_table

この構文は、対象テーブルを指定した親テーブルの子テーブルとして追加します。その後に行われる親テーブルへの問い合わせには対象テーブルの項目も含まれます。子テーブルとして追加するためには、対象テーブルには親テーブルと同じ列がすべて含まれていなければなりません。(この他の列を持つこともできます。) これらの列のデータ型は一致している必要があり、親テーブルでNOT NULL制約がある場合は、子テーブルでも同様にNOT NULL制約を持たなければなりません。

また、親テーブルのCHECK制約すべてについても、一致する制約が子テーブルに存在しなければなりません。ただし、親テーブルにおいて継承不可と印付けされている(つまりALTER TABLE ... ADD

CONSTRAINT ... NO INHERIT付きで作成された)制約は除きます(これらは無視されます)。一致する子テーブルの制約はすべて継承不可であってはなりません。現時点ではUNIQUE、PRIMARY KEY、FOREIGN KEY制約は無視されますが、将来変更されるかもしれません。

NO INHERIT parent_table

この構文は、指定した親テーブルの子テーブル群から対象のテーブルを削除します。親テーブルへの問い合わせでは、対象としたテーブルからのデータが含まれなくなります。

OF type_name

この構文は、CREATE TABLE OFで形成されたかのように、テーブルと複合型とを関連付けします。テーブルの列名とその型のリストは、複合型のものと正確に一致していなければなりません。テーブルはどのテーブルも継承していてはいけません。これらの制限によりCREATE TABLE OFにより作成できるテーブル定義と同等になります。

NOT OF

この構文は型と型付けされたテーブルの関連を取り除きます。

OWNER TO

この構文を使用すると、テーブル、シーケンス、ビュー、マテリアライズドビュー、または外部テーブルの所有者を、指定したユーザに変更できます。

REPLICA IDENTITY

この構文を使用すると、更新あるいは削除された行を特定できるよう、先行書き込みログに書き込まれる情報を変更します。このオプションは、論理レプリケーションが使われている場合以外は何の効果もありません。DEFAULTはシステムテーブル以外についてのデフォルトで、主キー列があれば、その古い値を記録します。USING INDEXは指定したインデックスに含まれる列の古い値を記録しますが、このインデックスは一意であり、部分インデックスや遅延可能インデックスではなく、またNOT NULLの列のみを含まなければなりません。FULLは行のすべての列の古い値を記録します。NOTHINGは古い行の情報を何も記録しません(これがシステムテーブルについてのデフォルトです)。どの場合についても、行の古いバージョンと新しいバージョンの間で、ログに記録される列のうち少なくとも1つが変わっていなければ、古い値はログに記録されません。

RENAME

RENAME構文を使用すると、テーブル(もしくは、インデックス、シーケンス、ビュー、マテリアライズドビュー、外部テーブル)の名前、テーブルの個々の列名、テーブルの制約名を変更できます。元となるインデックスを持つ制約名を変更するとき、インデックス名も同様に変更されます。格納されているデータへの影響はありません。

SET SCHEMA

この構文を使用して、テーブルを別のスキーマに移動することができます。関連するインデックスや制約、テーブル列により所有されるシーケンスも同様に移動されます。

ATTACH PARTITION partition_name { FOR VALUES partition_bound_spec | DEFAULT }

この構文は、既存のテーブル(それ自体がパーティションテーブルのこともあります)を対象テーブルのパーティションとして追加します。テーブルは、FOR VALUESを使って指定の値のパーティションとして、

あるいは、DEFAULTを使ってデフォルトパーティションとして追加できます。対象テーブルの各インデックスについて、対応するインデックスが付加されるテーブルに作られます。また、同等のインデックスが既にある場合には、そのインデックスが、ALTER INDEX ATTACH PARTITIONが実行された場合と同様に、対象テーブルのインデックスに付加されます。既存のテーブルが外部テーブルの場合、今のところ対象テーブルにUNIQUEインデックスがあるときにはテーブルを対象テーブルのパーティションとして追加することはできない点に注意してください（CREATE FOREIGN TABLEも参照してください）。対象テーブルにある各ユーザ定義の行レベルのトリガに対しては、対応するものが付加されるテーブルに作られます。

FOR VALUESを使ったパーティションはpartition_bound_specでCREATE TABLEと同じ構文を使います。パーティション境界の指定は、対象テーブルのパーティション戦略とパーティションキーと対応していなければなりません。付加されるテーブルは、対象と全て同じ列を持ち、それ以上の列は持たず、列の型も一致していなければなりません。また、対象テーブルにある全てのNOT NULLおよびCHECK制約を持たなければなりません。今のところ、FOREIGN KEY制約は考慮されません。親テーブルのUNIQUEおよびPRIMARY KEY制約は、既に在るのでなければ、パーティションに作られます。もし、アタッチされるテーブルのいずれかのCHECK制約がNO INHERITと印付けされていたなら、コマンドは失敗します。このような制約はNO INHERIT句なしに再作成しなければなりません。

新しいパーティションが通常のテーブルの場合、テーブルに存在する行がパーティションの制約に違反しないことを確認するため、テーブルの全件走査が行われます。このコマンドを実行するより前に、望まれるパーティションの制約を満たす行だけしか許さないような有効なCHECK制約をテーブルに追加すれば、この全件走査を避けることができます。CHECK制約は、パーティションの制約を確認するためにテーブルをスキャンする必要がないか決めるために使われます。しかし、パーティションキーに式が一つでもあり、パーティションがNULL値を受け付けないときは、この仕組みは機能しません。NULL値を受け付けないリストパーティションに追加するときも、それが式でないなら、パーティションキーの列にNOT NULL制約を追加してください。

新しいパーティションが外部テーブルの場合、外部テーブルのすべての行がパーティションの制約に従うかどうかの確認は何も行われません。（外部テーブルの制約についてはCREATE FOREIGN TABLEの議論を参照してください。）

テーブルがデフォルトパーティションを持っている場合、新たなパーティションの定義はデフォルトパーティションに対するパーティション制約を変更します。デフォルトパーティションは新パーティションに移動すべきいかなる行を含むことができず、そのような行が無いことを確認するためスキャンが行われます。このスキャンは、新パーティションのスキャンと同様に、適切なCHECK制約があれば回避できます。やはり、新パーティションのスキャンと同様に、デフォルトパーティションが外部テーブルであるときは、このスキャンは常に省略されます。

パーティションの追加は、追加されるテーブルと（もしあれば）デフォルトパーティションでのACCESS EXCLUSIVEロックに加えて、親テーブルでSHARE UPDATE EXCLUSIVEロックを取得します。

DETACH PARTITION partition_name

この構文は、指定したパーティションを対象のテーブルから切り離します。切り離されたパーティションは単独のテーブルとして存在し続けますが、切り離される前のテーブルとの紐付けはなくなります。対象テーブルのインデックスに付加されていた全てのインデックスも切り離されます。対象テーブルのものの複製として作られたトリガは削除されます。

RENAME、SET SCHEMA、ATTACH PARTITION、DETACH PARTITIONは、複数の変更リストに結合して、まとめて処理することができますが、それらを除き、ALTER TABLEのすべての構文は1つだけのテーブルに対して作用

します。例えば、複数の列の追加、型の変更を単一のコマンドで実行することができます。これは特に巨大なテーブルでは便利です。変更のために必要なテーブル全体の走査が1回で済むからです。

ALTER TABLEコマンドを使用するには、変更するテーブルを所有している必要があります。テーブルのスキーマあるいはテーブル空間を変更するには、新しいスキーマあるいはテーブル空間におけるCREATE権限も持っていなければなりません。テーブルを親テーブルの新しい子テーブルとして追加するには、親テーブルも所有している必要があります。またテーブルをテーブルのパーティションとして追加する場合、追加されるテーブルを所有している必要があります。また、所有者を変更するには、新しい所有ロールの直接あるいは間接的なメンバでなければならず、かつ、そのロールがテーブルのスキーマにおけるCREATE権限を持たなければなりません（この制限により、テーブルの削除と再作成を行ってもできないことが、所有者の変更によってもできないようにしています。ただし、スーパーユーザはすべてのテーブルの所有者を変更することができます）。列の追加、列の型の変更、OF句の使用を行うためには、データ型に対するUSAGE権限を持たなければなりません。

パラメータ

IF EXISTS

テーブルが存在しない場合でもエラーとしません。この場合は注意メッセージが発行されます。

name

変更対象となる既存のテーブルの名前です（スキーマ修飾名も可）。テーブル名の前にONLYが指定された場合、そのテーブルのみが変更されます。ONLYが指定されていない場合、そのテーブルおよび（もしあれば）そのテーブルを継承する全てのテーブルが更新されます。オプションで、テーブル名の後に*を指定することで、明示的に継承するテーブルも含まれることを示すことができます。

column_name

新規または既存の列の名前です。

new_column_name

既存の列の新しい名前です。

new_name

テーブルの新しい名前です。

data_type

新しい列のデータ型、もしくは既存の列に対する新しいデータ型です。

table_constraint

テーブルの新しいテーブル制約です。

constraint_name

新しい、あるいは既存の制約の名前です。

CASCADE

削除された列や制約に依存しているオブジェクト（例えば、削除された列を参照しているビューなど）を自動的に削除し、さらにそれらのオブジェクトに依存するすべてのオブジェクトも削除します（5.14参照）。

RESTRICT

依存しているオブジェクトがある場合、列または制約の削除要求を拒否します。これがデフォルトの動作です。

trigger_name

有効または無効にする単一のトリガの名前です。

ALL

テーブルに属するすべてのトリガを有効または無効にします。（外部キー制約、遅延可能な一意性および排他制約を実装するために使用される、内部向けに生成される制約トリガが含まれる場合、スーパーユーザ権限が必要です。）

USER

外部キー制約、遅延可能な一意性および排他制約を実装するために使用される、内部向けに生成されるトリガを除く、テーブルに属するトリガすべてを有効または無効にします。

index_name

既存のインデックスの名前です。

storage_parameter

テーブルの格納パラメータの名前です。

value

テーブルの格納パラメータの新しい値です。パラメータによりこれは数値となることも文字列となることもあります。

parent_table

このテーブルに関連付ける、または、このテーブルから関連付けを取り除く親テーブルです。

new_owner

テーブルの新しい所有者のユーザ名です。

new_tablespace

テーブルを移動する先のテーブル空間の名前です。

new_schema

テーブルを移動する先のスキーマの名前です。

partition_name

新しいパーティションとして追加する、またはテーブルから切り離すテーブルの名前です。

partition_bound_spec

新しいパーティションのパーティション境界の指定です。その構文の詳細についてはCREATE TABLEを参照してください。

注釈

COLUMNキーワードには意味がなく、省略可能です。

ADD COLUMNで列が追加され、非変動性のDEFAULTが指定されたときには、デフォルトは宣言時に評価されてテーブルのメタデータに格納された結果です。この値は全ての既存行の列に使われます。DEFAULTが指定されなかった場合にはNULLが使われます。どちらの場合もテーブルを書き直す必要はありません。

変動性のDEFAULT句を持つ列を追加したり、既存の列の型を変更するには、テーブルとインデックス全体の書き換えが必要になります。例外として、既存列の型を変更するとき、USING句が列の内容を変更せず、かつ、古い型が新しい型とバイナリ変換可能であるか新しい型全体に対する制約のないドメインである場合、テーブルの書き換えは必要ありません。しかし、影響を受ける列に対するインデックスはすべて再構築されなければなりません。テーブルが巨大な場合、テーブル、インデックスまたはその両方の再構築には非常に時間がかかる可能性があります。また、一時的に2倍のディスク容量が必要とされます。

CHECKあるいはNOT NULL制約を追加する時は、既存の行が制約に従うかどうかを検証するためにテーブルの走査が必要になりますが、テーブルの書き換えは必要ありません。

同様に、新しいパーティションを追加するときは、既存の行がパーティションの制約を満たすかどうかを確認するため、テーブルが走査されるかもしれません。

単一のALTER TABLE内に複数の変更を指定できるオプションを提供する主な理由は、複数のテーブル走査や書き換えを1回のテーブル走査にまとめることができるようにすることです。

新しい外部キーや検査制約を検証するために大きなテーブルを走査するのは長い時間が掛かる可能性があり、ALTER TABLE ADD CONSTRAINTコマンドがコミットされるまで、そのテーブルのその他の更新は締め出されます。NOT VALID制約オプションの主な目的は、同時実行中の更新に制約を追加する影響を減らすことです。NOT VALIDを付ければ、ADD CONSTRAINTコマンドはテーブルを走査せず、すぐにコミットされます。その後で、VALIDATE CONSTRAINTコマンドを発行して、既存の行が制約を満たすか検証できます。他のトランザクションが挿入したり更新したりする行に対しては制約が強制されていることは分かっていますので、この検証操作では同時実行中の更新を締め出す必要はありません。既に存在する行だけ確認する必要があります。それゆえ、検証には変更するテーブルのSHARE UPDATE EXCLUSIVEロックのみが必要です。(制約が外部キーなら、制約が参照するテーブルのROW SHAREロックも必要です。) 同時実行性をさらに向上させるため、テーブルに既に制約違反が存在することを知っている場合にNOT VALIDとVALIDATE CONSTRAINTを使うことは有用かもしれません。一度制約が設定されれば、新しい違反は挿入されることはありませんし、既存の問題は、VALIDATE CONSTRAINTを最終的に成功するまで使って、余裕のある時に修正できます。

DROP COLUMN構文は、列を物理的には削除せず、SQLの操作に対して不可視にします。このコマンドを実行した後、テーブルに挿入または更新が行われると、削除した列にはNULLが格納されます。したがって、列の削除は短時間で行えます。しかし、削除された列が占めていた領域がまだ回収されていないため、テーブルの

ディスク上のサイズはすぐには小さくなりません。この領域は、その後既存の行が更新されるにつれて回収されます。

削除した列が占有していたスペースを即座に再利用できるようにするには、テーブル全体を書き換える構文のALTER TABLEを使用することができます。この結果、各行の削除される列がNULL値で再構成されます。

テーブルを書き換える構文のALTER TABLEはMVCC的に安全ではありません。同時実行中のトランザクションが、テーブル書き換えが発生する前に取得したスナップショットを使っている場合、テーブルの書き換え後はそのトランザクションにはテーブルが空であるように見えます。詳しくは13.5を参照して下さい。

SET DATA TYPEのUSINGオプションでは、その行の古い値を含め、どのような式でも指定できます。つまり、変換対象の列と同様に、その他の列も参照することができます。そのため、一般的な変換をSET DATA TYPE構文で行うことができます。この柔軟性のため、USING式は列のデフォルト値には(仮に存在していても)適用されません。結果が定数式にならない可能性があるためです(デフォルト値は定数式でなければいけません)。したがって、古い型から新しい型への暗黙キャストや代入キャストが存在しない場合、USINGが指定されていても、SET DATA TYPEがデフォルト値の変換に失敗する可能性があります。この場合は、DROP DEFAULTでデフォルト値を削除し、ALTER TYPEを実行した後で、SET DEFAULTを使用して再度適切なデフォルト値を指定してください。変更対象の列を含むインデックスと制約も同様の配慮が必要です。

そのテーブルを継承するテーブルがある場合、子テーブルに同じ処理を実行しなければ、親テーブルに対する列の追加、列の名前、列の型の変更を実行することはできません。この制限により、子テーブルの列が常に親テーブルと一致していることが保証されます。同様に、すべての子テーブルでCHECK制約の名前を変更し、それが親と子の間で一致するようにしなければ、親テーブルのCHECK制約の名前を変更することはできません。(しかし、この制限はインデックスの基づく制約にはあらわれません。) また、親テーブルからSELECTすると、その子テーブルからもSELECTすることになるため、親テーブルの制約は、それが子テーブルでも有効であると印を付けられるまで、有効であると印を付けられません これらのすべての場合において、ALTER TABLE ONLYは受け付けられません。

再帰的なDROP COLUMN操作では、子テーブルが他の親テーブルからその列を継承しておらず、かつ、その列について独立した定義を持っていない場合のみ、その子テーブルの列を削除します。再帰的でないDROP COLUMN(つまり、ALTER TABLE ONLY ... DROP COLUMN)操作では、継承された列は削除されません。削除する代わりに、その列は継承されておらず独立して定義されているという印を付けます。再帰的でないDROP COLUMNコマンドは、パーティションテーブルでは失敗します。テーブルのすべてのパーティションは、パーティションの最上位と同じ列を持っていなければならないからです。

IDENTITY列についての操作(ADD GENERATED、SET、DROP IDENTITYなど)およびTRIGGER、CLUSTER、OWNERおよびTABLESPACEの操作は子テーブルに再帰的に伝わりません。つまり、常にONLYが指定されているかのように動作します。制約の追加は、NO INHERIT印がないCHECK制約に関してのみ再帰的に伝わります。

システムカタログテーブルについては、いかなる部分の変更も許可されていません。

有効なパラメータの詳しい説明はCREATE TABLEを参照してください。第5章に、継承に関するさらに詳しい情報があります。

例

varchar型の列をテーブルに追加します。

```
ALTER TABLE distributors ADD COLUMN address varchar(30);
```

これは表の既存の行すべてで、新しい列をNULL値で埋めることになります。

デフォルトが非NULLの列を追加します。

```
ALTER TABLE measurements
  ADD COLUMN mtime timestamp with time zone DEFAULT now();
```

既存の行では、新しい列の値として現在時刻が入ります。また、新しい行では挿入時刻を受け取ります。

列を追加して、後で使われるデフォルトとは異なる値で埋めます。

```
ALTER TABLE transactions
  ADD COLUMN status varchar(30) DEFAULT 'old',
  ALTER COLUMN status SET default 'current';
```

既存の行はoldで埋められますが、後続のコマンドに対するデフォルトはcurrentになります。別々のALTER TABLEコマンドで2つの副コマンドを発行する場合と、効果は同じです。

テーブルから列を削除します。

```
ALTER TABLE distributors DROP COLUMN address RESTRICT;
```

1つの操作で既存の2つの列の型を変更します。

```
ALTER TABLE distributors
    ALTER COLUMN address TYPE varchar(80),
    ALTER COLUMN name TYPE varchar(100);
```

USING句を使用して、Unixタイムスタンプを持つinteger型の列をtimestamp with time zoneに変更します。

```
ALTER TABLE foo
    ALTER COLUMN foo_timestamp SET DATA TYPE timestamp with time zone
    USING
        timestamp with time zone 'epoch' + foo_timestamp * interval '1 second';
```

上と同じことをします。ただし、その列は、自動的に新しいデータ型にキャストされないデフォルト式を持つ場合についてです。

```
ALTER TABLE foo
    ALTER COLUMN foo_timestamp DROP DEFAULT,
    ALTER COLUMN foo_timestamp TYPE timestamp with time zone
    USING
        timestamp with time zone 'epoch' + foo_timestamp * interval '1 second',
    ALTER COLUMN foo_timestamp SET DEFAULT now();
```

既存の列の名前を変更します。

```
ALTER TABLE distributors RENAME COLUMN address TO city;
```

既存のテーブルの名前を変更します。

```
ALTER TABLE distributors RENAME TO suppliers;
```

既存の制約の名前を変更します。

```
ALTER TABLE distributors RENAME CONSTRAINT zipchk TO zip_check;
```

列に非NULL制約を付与します。

```
ALTER TABLE distributors ALTER COLUMN street SET NOT NULL;
```

列から非NULL制約を削除します。

```
ALTER TABLE distributors ALTER COLUMN street DROP NOT NULL;
```

テーブルとそのすべての子テーブルにCHECK制約を付与します。

```
ALTER TABLE distributors ADD CONSTRAINT zipchk CHECK (char_length(zipcode) = 5);
```

そのテーブルのみに適用され、その子テーブルには適用されない検査制約を追加します。

```
ALTER TABLE distributors ADD CONSTRAINT zipchk CHECK (char_length(zipcode) = 5) NO INHERIT;
```

（この検査制約はこの後作成される子テーブルにも継承されません。）

テーブルとそのすべての子テーブルからCHECK制約を削除します。

```
ALTER TABLE distributors DROP CONSTRAINT zipchk;
```

1つのテーブルのみから検査制約を削除します。

```
ALTER TABLE ONLY distributors DROP CONSTRAINT zipchk;
```

（この検査制約はすべての子テーブルで残ったままです。）

テーブルに外部キー制約を付与します。

```
ALTER TABLE distributors ADD CONSTRAINT distfk FOREIGN KEY (address) REFERENCES addresses
 (address);
```

テーブルへの外部キーの追加で、他の作業への影響を最小限にするには、以下のようにします。

```
ALTER TABLE distributors ADD CONSTRAINT distfk FOREIGN KEY (address) REFERENCES addresses
 (address) NOT VALID;
```

```
ALTER TABLE distributors VALIDATE CONSTRAINT distfk;
```

テーブルに(複数列の)一意性制約を付与します。

```
ALTER TABLE distributors ADD CONSTRAINT dist_id_zipcode_key UNIQUE (dist_id, zipcode);
```

自動的に命名される主キー制約をテーブルに付与します。1つのテーブルが持てる主キーは1つだけであることに注意してください。

```
ALTER TABLE distributors ADD PRIMARY KEY (dist_id);
```

テーブルを別のテーブル空間に移動します。

```
ALTER TABLE distributors SET TABLESPACE fasttablespace;
```

テーブルを別のスキーマに移動します。

```
ALTER TABLE myschema.distributors SET SCHEMA yourschema;
```

インデックスを再構築している間の更新をブロックすることなく、主キー制約を再作成します。

```
CREATE UNIQUE INDEX CONCURRENTLY dist_id_temp_idx ON distributors (dist_id);
ALTER TABLE distributors DROP CONSTRAINT distributors_pkey,
    ADD CONSTRAINT distributors_pkey PRIMARY KEY USING INDEX dist_id_temp_idx;
```

範囲パーティションテーブルにパーティションを追加します。

```
ALTER TABLE measurement
    ATTACH PARTITION measurement_y2016m07 FOR VALUES FROM ('2016-07-01') TO ('2016-08-01');
```

リストパーティションテーブルにパーティションを追加します。

```
ALTER TABLE cities
    ATTACH PARTITION cities_ab FOR VALUES IN ('a', 'b');
```

ハッシュパーティションテーブルにパーティションを追加します。

```
ALTER TABLE orders
    ATTACH PARTITION orders_p4 FOR VALUES WITH (MODULUS 4, REMAINDER 3);
```

パーティションテーブルにデフォルトパーティションを追加します。

```
ALTER TABLE cities
    ATTACH PARTITION cities_partdef DEFAULT;
```

パーティションテーブルからパーティションを切り離します。

```
ALTER TABLE measurement
    DETACH PARTITION measurement_y2015m12;
```

互換性

(USING INDEXがない)ADD、DROP [COLUMN]、DROP IDENTITY、RESTART、SET DEFAULT、(USINGのない)SET DATA TYPE、SET GENERATED、SET sequence_option構文は標準SQLに従います。他の構文は標準SQLに対するPostgreSQLの拡張です。また、単一のALTER TABLEコマンド内に複数の操作を指定する機能もPostgreSQLの拡張です。

ALTER TABLE DROP COLUMNを使って、1つしか列がないテーブルから列を削除して、列がないテーブルを作成することができます。これはPostgreSQLの拡張です。SQLでは、列を持たないテーブルは認められていません。

関連項目

CREATE TABLE

ALTER TABLESPACE

ALTER TABLESPACE — テーブル空間の定義を変更する

概要

```
ALTER TABLESPACE name RENAME TO new_name
ALTER TABLESPACE name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER TABLESPACE name SET ( tablespace_option = value [, ... ] )
ALTER TABLESPACE name RESET ( tablespace_option [, ... ] )
```

説明

ALTER TABLESPACEはテーブル空間の定義を変更するのに使うことができます。

テーブル空間の定義を変更するにはテーブル空間の所有者でなければなりません。所有者を変更するには、直接または間接的に新しい所有者ロールのメンバでなければなりません。(スーパーユーザがこれらの権限を自動的に持つことに注意してください。)

パラメータ

name

既存のテーブル空間の名前です。

new_name

テーブル空間の新しい名前です。pg_から始まる名前は、システムのテーブル空間用に予約されているため使用することができません。

new_owner

テーブル空間の新しい所有者です。

tablespace_option

設定または設定解除するテーブル空間パラメータです。現時点ではseq_page_cost、random_page_cost、effective_io_concurrency、maintenance_io_concurrencyパラメータのみが利用可能です。特定のテーブル空間にこの値を設定することにより、同一の名前の設定パラメータ(seq_page_cost、random_page_cost、effective_io_concurrency、maintenance_io_concurrency参照)で決定される、そのテーブル空間内のテーブルからページを読み取るプランナの通常のコスト推定値とエクゼキュータの先読みの振る舞いが上書きされます。これはあるテーブル空間が他のI/Oサブシステムに比べ低速または高速なディスク上にある場合に有用となるかもしれません。

例

テーブル空間index_spaceをfast_raidという名前に変更します。

```
ALTER TABLESPACE index_space RENAME TO fast_raid;
```

テーブル空間index_spaceの所有者を変更します。

```
ALTER TABLESPACE index_space OWNER TO mary;
```

互換性

標準SQLにはALTER TABLESPACE文はありません。

関連項目

CREATE TABLESPACE, DROP TABLESPACE

ALTER TEXT SEARCH CONFIGURATION

ALTER TEXT SEARCH CONFIGURATION — テキスト検索設定の定義を変更する

概要

```
ALTER TEXT SEARCH CONFIGURATION name
    ADD MAPPING FOR token_type [, ... ] WITH dictionary_name [, ... ]
ALTER TEXT SEARCH CONFIGURATION name
    ALTER MAPPING FOR token_type [, ... ] WITH dictionary_name [, ... ]
ALTER TEXT SEARCH CONFIGURATION name
    ALTER MAPPING REPLACE old_dictionary WITH new_dictionary
ALTER TEXT SEARCH CONFIGURATION name
    ALTER MAPPING FOR token_type [, ... ] REPLACE old_dictionary WITH new_dictionary
ALTER TEXT SEARCH CONFIGURATION name
    DROP MAPPING [ IF EXISTS ] FOR token_type [, ... ]
ALTER TEXT SEARCH CONFIGURATION name RENAME TO new_name
ALTER TEXT SEARCH CONFIGURATION name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER TEXT SEARCH CONFIGURATION name SET SCHEMA new_schema
```

説明

ALTER TEXT SEARCH CONFIGURATIONはテキスト検索設定の定義を変更します。トークン型から辞書への対応付けの変更、または、設定名称の変更、設定の所有者の変更を行うことができます。

ALTER TEXT SEARCH CONFIGURATIONを使用するためには、設定の所有者でなければなりません。

パラメータ

name

既存のテキスト検索設定の名称（スキーマ修飾可）です。

token_type

設定のパーサが発行するトークン型の名称です。

dictionary_name

指定したトークン型（複数可）で考慮されるテキスト検索辞書の名称です。複数の辞書が列挙された場合、指定された順序で参照されます。

old_dictionary

対応付けにて置換されるテキスト検索辞書の名称です。

new_dictionary

old_dictionaryを置き換えるテキスト検索辞書の名称です。

new_name

テキスト検索設定の新しい名称です。

new_owner

テキスト検索設定の新しい所有者です。

new_schema

テキスト検索設定の新しいスキーマです。

ADD MAPPING FOR構文は指定したトークン型で参照される辞書のリストをインストールします。既にそのトークン型に対する対応付けが存在する場合はエラーになります。ALTER MAPPING FOR構文は、まず既存の対象トークン型に対する対応付けを削除する点を除き、同一です。ALTER MAPPING REPLACE構文は、すべてのold_dictionaryをnew_dictionaryで置き換えます。FORがあれば、これは指定したトークン型に対してのみ行われ、なければ、設定におけるすべての対応付けに対して行われます。DROP MAPPING構文は指定したトークン型（複数可）に対するすべての辞書を削除します。この結果、このテキスト検索設定ではこれらの型のトークンが無視されるようになります。IF EXISTSがない限り、トークン型に対する対応付けが存在しない場合はエラーになります。

例

次の例は、my_config内でenglishが使用されるすべてに対し、english辞書をswedish辞書で置換します。

```
ALTER TEXT SEARCH CONFIGURATION my_config
  ALTER MAPPING REPLACE english WITH swedish;
```

互換性

標準SQLにはALTER TEXT SEARCH CONFIGURATION文はありません。

関連項目

CREATE TEXT SEARCH CONFIGURATION, DROP TEXT SEARCH CONFIGURATION

ALTER TEXT SEARCH DICTIONARY

ALTER TEXT SEARCH DICTIONARY — テキスト検索辞書の定義を変更する

概要

```
ALTER TEXT SEARCH DICTIONARY name (
    option [ = value ] [, ... ]
)
ALTER TEXT SEARCH DICTIONARY name RENAME TO new_name
ALTER TEXT SEARCH DICTIONARY name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER TEXT SEARCH DICTIONARY name SET SCHEMA new_schema
```

説明

ALTER TEXT SEARCH DICTIONARYはテキスト検索辞書の定義を変更します。辞書のテンプレート固有のオプションの変更、辞書の名称、辞書の所有者を変更することができます。

ALTER TEXT SEARCH DICTIONARYを使用するには、辞書の所有者でなければなりません。

パラメータ

name

既存のテキスト検索辞書の名称（スキーマ修飾可）です。

option

この辞書に設定される、テンプレート固有のオプションの名称です。

value

テンプレート固有のオプションで使用される、新しい値です。等号記号と値が省略された場合、そのオプションの以前の設定は辞書から削除され、デフォルト値が使用されます。

new_name

テキスト検索辞書の新しい名称です。

new_owner

テキスト検索辞書の新しい所有者です。

new_schema

テキスト検索辞書の新しいスキーマです。

テンプレート固有のオプションは任意の順序で記述することができます。

例

次の例は、雪だるま式に増加する辞書のストップワードを変更します。他のパラメータはそのまま変更されません。

```
ALTER TEXT SEARCH DICTIONARY my_dict ( StopWords = newrussian );
```

次の例は、言語オプションをdutch(オランダ語)に変更し、ストップワードオプションを完全に消去します。

```
ALTER TEXT SEARCH DICTIONARY my_dict ( language = dutch, StopWords );
```

次の例は、実際には何も変更しませんが、辞書の定義を「更新」します。

```
ALTER TEXT SEARCH DICTIONARY my_dict ( dummy );
```

(無効なオプションが存在しても、オプションを消去するコードでエラーとしないため、これは動作します。) このトリックは、辞書用の設定ファイルを変更する際に有用です。このALTERにより既存のデータベースセッションは強制的に設定ファイルを再度読み込みます。こうしないと、以前に読み込んだ設定ファイルを再読み込みすることはありません。

互換性

標準SQLにはALTER TEXT SEARCH DICTIONARY文はありません。

関連項目

CREATE TEXT SEARCH DICTIONARY, DROP TEXT SEARCH DICTIONARY

ALTER TEXT SEARCH PARSER

ALTER TEXT SEARCH PARSER — テキスト検索パーサの定義を変更する

概要

```
ALTER TEXT SEARCH PARSER name RENAME TO new_name
ALTER TEXT SEARCH PARSER name SET SCHEMA new_schema
```

説明

ALTER TEXT SEARCH PARSERはテキスト検索パーサの定義を変更します。現時点では、パーサ名称の変更機能のみがサポートされています。

ALTER TEXT SEARCH PARSERを使用するためにはスーパーユーザでなければなりません。

パラメータ

name

既存のテキスト検索パーサの名称（スキーマ修飾可）です。

new_name

新しいテキスト検索パーサの名称です。

new_schema

全文検索パーサの新しいスキーマです。

互換性

標準SQLにはALTER TEXT SEARCH PARSER文はありません。

関連項目

CREATE TEXT SEARCH PARSER, DROP TEXT SEARCH PARSER

ALTER TEXT SEARCH TEMPLATE

ALTER TEXT SEARCH TEMPLATE — テキスト検索テンプレートの定義を変更する

概要

```
ALTER TEXT SEARCH TEMPLATE name RENAME TO new_name
ALTER TEXT SEARCH TEMPLATE name SET SCHEMA new_schema
```

説明

ALTER TEXT SEARCH TEMPLATEはテキスト検索テンプレートの定義を変更します。現時点では、テンプレート名称の変更機能のみがサポートされています。

ALTER TEXT SEARCH TEMPLATEを使用するためにはスーパーユーザでなければなりません。

パラメータ

name

既存のテキスト検索テンプレートの名称（スキーマ修飾可）です。

new_name

新しいテキスト検索テンプレートの名称です。

new_schema

テキスト検索テンプレートの新しいスキーマです。

互換性

標準SQLにはALTER TEXT SEARCH TEMPLATE文はありません。

関連項目

CREATE TEXT SEARCH TEMPLATE, DROP TEXT SEARCH TEMPLATE

ALTER TRIGGER

ALTER TRIGGER — トリガ定義を変更する

概要

```
ALTER TRIGGER name ON table_name RENAME TO new_name
ALTER TRIGGER name ON table_name [ NO ] DEPENDS ON EXTENSION extension_name
```

説明

ALTER TRIGGERを使用すると、既存のトリガのプロパティを変更できます。RENAME句を使用すると、トリガ定義を変更せずに、指定されたテーブルのトリガ名を変更することができます。DEPENDS ON EXTENSION句を使用すると、トリガが拡張に依存するものとして印をつけられます。これにより、拡張が削除されると、トリガも自動的に削除されるようになります。

トリガのプロパティを変更するには、トリガで処理されるテーブルを所有している必要があります。

パラメータ

name

変更の対象となる既存のトリガの名前です。

table_name

このトリガで処理されるテーブルの名前です。

new_name

トリガの新しい名前です。

extension_name

トリガが依存する(もしくはNOが指定された場合にはもはや依存していない)拡張の名前です。拡張に依存していると印をつけられたトリガは、拡張が削除されると自動的に削除されます。

注釈

トリガを一時的に有効または無効にする機能はALTER TABLEが提供します。ALTER TRIGGERではありません。ALTER TRIGGERには、一度にテーブルのトリガを有効または無効にするオプションを表現する、簡便な方法がないからです。

例

既存のトリガの名前を変更します。

```
ALTER TRIGGER emp_stamp ON emp RENAME TO emp_track_chgs;
```

トリガが拡張に依存するという印を付けます。

```
ALTER TRIGGER emp_stamp ON emp DEPENDS ON EXTENSION emplib;
```

互換性

ALTER TRIGGERは、標準SQLに対するPostgreSQLの拡張です。

関連項目

ALTER TABLE

ALTER TYPE

ALTER TYPE — 型定義を変更する

概要

```
ALTER TYPE name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER TYPE name RENAME TO new_name
ALTER TYPE name SET SCHEMA new_schema
ALTER TYPE name RENAME ATTRIBUTE attribute_name TO new_attribute_name [ CASCADE | RESTRICT ]
ALTER TYPE name action [, ... ]
ALTER TYPE name ADD VALUE [ IF NOT EXISTS ] new_enum_value [ { BEFORE |
 AFTER } neighbor_enum_value ]
ALTER TYPE name RENAME VALUE existing_enum_value TO new_enum_value
ALTER TYPE name SET ( property = value [, ... ] )



ここでactionは以下のいずれかです。



    ADD ATTRIBUTE attribute_name data_type [ COLLATE collation ] [ CASCADE | RESTRICT ]
    DROP ATTRIBUTE [ IF EXISTS ] attribute_name [ CASCADE | RESTRICT ]
    ALTER ATTRIBUTE attribute_name [ SET DATA ] TYPE data_type [ COLLATE collation ] [ CASCADE |
 RESTRICT ]
```

説明

ALTER TYPEは既存の型の定義を変更します。複数の副構文があります。

OWNER

この構文は型の所有者を変更します。

RENAME

この構文は型の名前を変更します。

SET SCHEMA

この構文は型を他のスキーマに移動します。

RENAME ATTRIBUTE

この構文は複合型に対してのみ利用可能です。型の個々の属性の名前を変更します。

ADD ATTRIBUTE

この構文は、CREATE TYPEと同じ構文を用いて、複合型に新しい属性を追加します。

DROP ATTRIBUTE [IF EXISTS]

この構文は複合型から属性を削除します。IF EXISTSが指定された時はその属性が存在しなくてもエラーにはなりません。この場合は代わりに注意が発せられます。

ALTER ATTRIBUTE ... SET DATA TYPE

この構文は複合型の属性の型を変更します。

ADD VALUE [IF NOT EXISTS] [BEFORE | AFTER]

この構文は列挙型に新しい値を追加します。列挙型の順序中での新しい値の場所は、既存の値のBEFOREまたはAFTERという形式で指定することができます。指定がなければ新しい項目は値のリストの最後に追加されます。

IF NOT EXISTSが指定されている場合、型の中に新しい値が既に含まれていたとしてもエラーになりません。注意が発生されますが、他の動作は行われません。そうでなければ、新しい値がすでに存在しているとエラーが起こります。

RENAME VALUE

この構文では列挙型の値の名前を変更します。列挙型の順序における値の位置は変更されません。指定の値が存在しない、あるいは新しい名前が既に存在する場合はエラーが発生します。

SET (property = value [, ...])

この構文は基本型に対してのみ適用可能です。CREATE TYPEで設定できる基本型属性のサブセットを調整できます。特に、以下の属性が変更できます。

- RECEIVEでバイナリ入力関数の名前を設定できます。NONEはその型のバイナリ入力関数を削除します。このオプションを使うにはスーパーユーザ権限が必要です。
- SENDでバイナリ出力関数の名前を設定できます。NONEはその型のバイナリ出力関数を削除します。このオプションを使うにはスーパーユーザ権限が必要です。
- TYPMOD_INで型修飾子入力関数の名前を設定できます。NONEはその型の型修飾子入力関数を削除します。このオプションを使うにはスーパーユーザ権限が必要です。
- TYPMOD_OUTで型修飾子出力関数の名前を設定できます。NONEはその型の型修飾子出力関数を削除します。このオプションを使うにはスーパーユーザ権限が必要です。
- ANALYZEは型固有の統計情報収集関数の名前を設定できます。NONEはその型の統計情報収集関数を削除します。このオプションを使うにはスーパーユーザ権限が必要です。
- STORAGE STORAGEはplain、extended、external、mainに設定できます(それぞれが何を意味するかの詳細は68.2を参照してください)。しかしながら、plainからその他の設定へ変更するにはスーパーユーザ権限が必要であり(その型のC関数がすべてTOASTの準備ができていることが必要だからです)、plainへその他の設定から変更することは全く許されていません(その型に、既にTOASTされた値がデータベース内にあるかもしれないためです)。このオプションを変更することは、それだけでは

格納されたデータを変更せず、今後作成されるテーブル列で使われるデフォルトのTOAST戦略を設定するだけであることに注意してください。既存のテーブル列のTOAST戦略を変更するにはALTER TABLEを参照してください。

この型属性についての詳細はCREATE TYPEを参照してください。基本型に対する属性の変更は、適切な場合その型に基づくドメインに自動的に伝播することに注意してください。

ADD ATTRIBUTE、DROP ATTRIBUTE、ALTER ATTRIBUTE操作は複数の変更リストにまとめて、並行して適用することができます。例えば、複数の属性の追加、複数の属性の変更、またはその両方を1つのコマンドで実行することができます。

ALTER TYPEを使用するには型の所有者でなければなりません。型のスキーマを変更するには、新しいスキーマにおけるCREATE権限も必要です。所有者を変更するには、直接または間接的に新しい所有者ロールのメンバでなければなりません。また、そのロールが型のスキーマにおいてCREATEを持たなければなりません。(この制限により、型の削除と再作成で行うことができない処理は所有者の変更で行えないようになります。しかし、スーパーユーザはすべての型の所有者を変更することができます。) 属性を追加または属性の型の変更を行うためには、その属性のデータ型に対するUSAGE権限を持たなければなりません。

パラメータ

name

変更対象の既存の型の名前です(スキーマ修飾名も可)。

new_name

新しい型の名前です。

new_owner

新しい型の所有者のユーザ名です。

new_schema

型の新しいスキーマです。

attribute_name

追加、変更または削除する属性の名前です。

new_attribute_name

変名する属性の新しい名前です。

data_type

追加する属性のデータ型、または、変更する属性の新しい型です。

new_enum_value

列挙型リストの値に追加する新しい値、あるいは既存の値につける新しい名前です。すべての列挙型のリテラル同様、引用符を付けなければなりません。

neighbor_enum_value

列挙型の並び順序において新しい値をその直前または直後に追加する、既存の列挙型の値です。すべての列挙型のリテラル同様、引用符を付けなければなりません。

existing_enum_value

名前の変更の対象となる既存の列挙型の値です。すべての列挙型のリテラルと同様、引用符を付ける必要があります。

property

修正する基本型属性の名前です。可能な値については上を参照してください。

CASCADE

変更される型で型付けされたテーブルとその子テーブルに、この操作を自動的に伝播します。

RESTRICT

変更対象の型がテーブルの型付けに使用されている場合に操作を拒絶します。これがデフォルトです。

注意

ALTER TYPE ... ADD VALUE（列挙型に新しい値を追加する構文）がトランザクションブロック内で実行された場合、トランザクションがコミットされるまで新しい値は使えません。

列挙型に追加された値を含む比較は、列挙型の元々の要素のみを含む比較よりも低速になることがあります。通常これは、新しい値のソート位置がリストの最後ではなくBEFOREまたはAFTERを用いて設定された場合のみで起こります。しかし最後に新しい値が追加された場合であっても起こる可能性があります。（これは、OIDカウンタが元の列挙型を作成してから「周回」した場合に起こります。）この速度の低下は通常は大きくありません。しかしこれが問題であれば、列挙型を削除し再作成する、あるいはデータベースをダンプし再ロードすることで最適な性能まで戻すことができます。

例

データ型の名前を変更します。

```
ALTER TYPE electronic_mail RENAME TO email;
```

email型の所有者をjoeに変更します。

```
ALTER TYPE email OWNER TO joe;
```

email型のスキーマをcustomersに変更します。

```
ALTER TYPE email SET SCHEMA customers;
```

複合型に新しい属性を追加します。

```
ALTER TYPE compfoo ADD ATTRIBUTE f3 int;
```

列挙型の特定のソート位置に新しい値を追加します。

```
ALTER TYPE colors ADD VALUE 'orange' AFTER 'red';
```

列挙型の値の名前を変更します。

```
ALTER TYPE colors RENAME VALUE 'purple' TO 'mauve';
```

既存の基本型に対するバイナリI/O関数を作成します。

```
CREATE FUNCTION mytypesend(mytype) RETURNS bytea ...;
CREATE FUNCTION mytyperecv(internal, oid, integer) RETURNS mytype ...;
ALTER TYPE mytype SET (
    SEND = mytypesend,
    RECEIVE = mytyperecv
);
```

互換性

属性の追加および削除を行う構文は標準SQLの一部です。他の構文はPostgreSQLの拡張です。

関連項目

CREATE TYPE, DROP TYPE

ALTER USER

ALTER USER — データベースロールを変更する

概要

```
ALTER USER role_specification [ WITH ] option [ ... ]


optionは次の通りです。

      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | REPLICATION | NOREPLICATION
    | BYPASSRLS | NOBYPASSRLS
    | CONNECTION LIMIT connlimit
    | [ ENCRYPTED ] PASSWORD 'password' | PASSWORD NULL
    | VALID UNTIL 'timestamp'

ALTER USER name RENAME TO new_name

ALTER USER { role_specification | ALL } [ IN DATABASE database_name ] SET configuration_parameter
 { TO | = } { value | DEFAULT }
ALTER USER { role_specification | ALL } [ IN DATABASE database_name ] SET configuration_parameter
 FROM CURRENT
ALTER USER { role_specification | ALL } [ IN DATABASE database_name ]
 RESET configuration_parameter
ALTER USER { role_specification | ALL } [ IN DATABASE database_name ] RESET ALL


ここでrole_specificationは以下の通りです。

    role_name
  | CURRENT_USER
  | SESSION_USER
```

説明

ALTER USERはALTER ROLEの別名になりました。

互換性

ALTER USER文は、PostgreSQLの拡張です。標準SQLでは、ユーザの定義は実装に任されています。

関連項目

ALTER ROLE

ALTER USER MAPPING

ALTER USER MAPPING — ユーザマップの定義を変更する

概要

```
ALTER USER MAPPING FOR { user_name | USER | CURRENT_USER | SESSION_USER | PUBLIC }
    SERVER server_name
    OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ... ] )
```

説明

ALTER USER MAPPINGはユーザマップの定義を変更します。

外部サーバの所有者は任意のユーザに対するそのサーバ向けのユーザマップを変更することができます。また、サーバ上でUSAGE権限がユーザに付与されていた場合、ユーザは自身の持つユーザ名に対応するユーザマップを変更することができます。

パラメータ

user_name

対応付けするユーザ名です。CURRENT_USERとUSERは現在のユーザ名に対応します。PUBLICは現在および将来にシステム上に存在するすべてのユーザに対応させるために使用します。

server_name

ユーザマップのサーバ名です。

OPTIONS ([ADD | SET | DROP] option ['value'] [, ...])

ユーザマップのオプションを変更します。新しいオプションは過去に指定されたオプションをすべて上書きします。ADD、SET、DROPは実行する動作を指定します。明示的な動作指定がない場合、ADDとみなされます。オプション名は一意でなければなりません。またオプションはサーバの外部データラッパにより検証されます。

例

サーバfooのユーザマップbobのパスワードを変更します。

```
ALTER USER MAPPING FOR bob SERVER foo OPTIONS (SET password 'public');
```

互換性

ALTER USER MAPPINGはISO/IEC 9075-9(SQL/MED)に従います。小さな構文上の問題があります。標準ではFORキーワードを省略します。CREATE USER MAPPINGとDROP USER MAPPINGではFORを似たような位置で使用し、またIBM DB2（他の主なSQL/MED実装になっています）ではALTER USER MAPPINGで必要としていますので、PostgreSQLは、一貫性と相互運用性を目的に、標準と違いを持たせています。

関連項目

CREATE USER MAPPING, DROP USER MAPPING

ALTER VIEW

ALTER VIEW — ビュー定義を変更する

概要

```
ALTER VIEW [ IF EXISTS ] name ALTER [ COLUMN ] column_name SET DEFAULT expression
ALTER VIEW [ IF EXISTS ] name ALTER [ COLUMN ] column_name DROP DEFAULT
ALTER VIEW [ IF EXISTS ] name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER VIEW [ IF EXISTS ] name RENAME [ COLUMN ] column_name TO new_column_name
ALTER VIEW [ IF EXISTS ] name RENAME TO new_name
ALTER VIEW [ IF EXISTS ] name SET SCHEMA new_schema
ALTER VIEW [ IF EXISTS ] name SET ( view_option_name [= view_option_value] [, ... ] )
ALTER VIEW [ IF EXISTS ] name RESET ( view_option_name [, ... ] )
```

説明

ALTER VIEWはビューの各種補助属性を変更します。(ビューを定義する問い合わせを変更したい場合はCREATE OR REPLACE VIEWを使用してください。)

ALTER VIEWを使用するためには、ビューの所有者でなければなりません。またビューのスキーマを変更するためには、新しいスキーマ上にCREATE権限を持たなければなりません。さらに所有者を変更するためには、新しく所有者となるロールの直接あるいは間接的なメンバでなければならず、かつ、そのロールはビューのスキーマ上にCREATE権限を持たなければなりません。(これらの制限は、ビューの削除および再作成によりユーザが実行できないことを、所有者の変更により実行できないようにするためのものです。しかし、スーパーユーザはすべてのビューの所有者を変更することができます。)

パラメータ

name

既存のビューの名前(スキーマ修飾可)です。

column_name

既存の列の名前です。

new_column_name

既存の列に対する新しい名前です。

IF EXISTS

ビューが存在する場合にエラーとしません。この場合には注意メッセージが発行されます。

SET/DROP DEFAULT

この構文は列のデフォルト値を設定または削除します。ビューの列のデフォルト値は、ビューに対するルールやトリガが適用される前にビューを対象とした任意のINSERTまたはUPDATEコマンド内に代入されます。したがってビューのデフォルトは基となるリレーションのデフォルト値よりも優先度が高くなります。

new_owner

ビューの新しい所有者のユーザ名です。

new_name

ビューの新しい名前です。

new_schema

ビューの新しいスキーマです。

```
SET ( view_option_name [= view_option_value] [, ... ] )
RESET ( view_option_name [, ... ] )
```

ビューのオプションを設定またはクリアします。現在、サポートされるオプションは以下の通りです。

check_option (enum)

ビューのcheck optionを変更します。値はlocalまたはcascadedのいずれかでなければなりません。

security_barrier (boolean)

ビューのsecurity-barrier属性を変更します。値はtrueやfalseのような論理値でなければなりません。

注釈

歴史的な理由により、ALTER TABLEをビューに対して使用することができます。ただし、使用可能な構文は上記のビューに対して許される構文に対応するALTER TABLEの構文のみです。

例

ビューfooの名前をbarに変更します。

```
ALTER VIEW foo RENAME TO bar;
```

更新可能ビューにデフォルトの列値を付与します。

```
CREATE TABLE base_table (id int, ts timestamptz);
CREATE VIEW a_view AS SELECT * FROM base_table;
ALTER VIEW a_view ALTER COLUMN ts SET DEFAULT now();
INSERT INTO base_table(id) VALUES(1);  -- ts will receive a NULL
```

```
INSERT INTO a_view(id) VALUES(2);  -- ts will receive the current time
```

互換性

ALTER VIEWは標準SQLに対するPostgreSQLの拡張です。

関連項目

CREATE VIEW, DROP VIEW

ANALYZE

ANALYZE — データベースに関する統計を収集する

概要

```
ANALYZE [ ( option [, ...] ) ] [ table_and_columns [, ...] ]
ANALYZE [ VERBOSE ] [ table_and_columns [, ...] ]


optionには以下のいずれかが入ります。

    VERBOSE [ boolean ]
    SKIP_LOCKED [ boolean ]


また、
      table_and_columnsは以下の通りです。

    table_name [ ( column_name [, ...] ) ]
```

説明

ANALYZEはデータベース内のテーブルの内容に関する統計情報を収集し、その結果をpg_statisticシステムカタログに保存します。問い合わせプランナが最も効率の良い問い合わせの実行計画を決定する際、この統計情報が使用されます。

table_and_columnsリストがない場合、ANALYZEは現在のデータベース内で現在のユーザが解析する権限のあるすべてのテーブルとマテリアライズドビューを処理します。リストがある場合、ANALYZEは指定されたテーブルのみを処理します。さらにテーブルの列名のリストを与え、その列の統計情報のみを収集することも可能です。

オプションリストが括弧で括られていた場合、オプションは任意の順序で書けます。括弧付きの構文はPostgreSQL 11で追加されました。括弧のない構文は廃止予定です。

パラメータ

VERBOSE

進行状況の表示を有効にします。

SKIP_LOCKED

リレーション上で動作を開始する時に、ANALYZEは衝突しているロックが解放されるのを待たないことを指定します。リレーションを待つことなく即時にロックできない場合、そのリレーションは飛ばされます。こ

のオプションを指定しても、リレーションのインデックスを開く時、パーティションやテーブル継承の子、ある種類の外部テーブルからサンプル行を取得する時には、ANALYZEがブロックするかもしれないことに注意してください。また、通常ANALYZEは指定されたパーティションテーブルのパーティションをすべて処理しますが、このオプションによりANALYZEは、パーティションテーブル上に衝突するロックがあれば、パーティションをすべて飛ばします。

boolean

選択したオプションをオンにするかオフにするか指定します。オプションを有効にする場合にはTRUE、ONまたは1と書くことができ、無効にする場合にはFALSE、OFFまたは0と書くことができます。booleanの値は省略することもでき、その場合にはTRUEとみなされます。

table_name

解析の対象とするテーブルの名前です（スキーマ修飾名も可）。省略された場合、現在のデータベースの中のすべての通常のテーブル、パーティションテーブル、マテリアライズドビュー（外部テーブルは除く）が解析されます。指定したテーブルがパーティションテーブルの場合、パーティションテーブル全体としての継承の統計と、個々のパーティションの統計の両方が更新されます。

column_name

解析の対象とする列名です。デフォルトは全ての列です。

出力

VERBOSEが指定された場合、ANALYZEは進捗メッセージとして処理中のテーブルを表示します。さらに、テーブルについての各種統計情報も表示されます。

注釈

テーブルを解析するためには、通常そのテーブルの所有者であるかスーパーユーザでなければなりません。しかしながら、データベースの所有者は、共有カタログを除いて、そのデータベースのテーブルをすべて解析できます。(共有カタログの制限は、データベース全体のANALYZEはスーパーユーザだけが実施できるということを意味します。) ANALYZEは呼び出したユーザが解析する権限のないテーブルを飛ばします。

外部テーブルは明示的に選択された場合にのみ解析されます。すべての外部データラッパがANALYZEをサポートしているとは限りません。テーブルのラッパがANALYZEをサポートしない場合、コマンドは警告を出力し、何も行いません。

デフォルトのPostgreSQLの設定では、自動バキュームデーモン（24.1.6参照）が、データが最初にロードされた時や通常の操作を通して変更された時にテーブルの自動解析まで面倒をみます。もし自動バキュームが無効にしているならばANALYZEは定期的に、もしくは、テーブルの内容に大きな変更がある度に行うことを推奨します。統計情報が正確であれば、プランナが最も適切な問い合わせ計画を選択できるようになります。これによって、問い合わせ処理の速度が向上します。読み取りの多いデータベースでは、VACUUMとANALYZEは、1日1回、データベースがあまり使用されていない時間帯に実行することが一般的です。（非常に更新が激しい場合、これでは十分ではありません。）

ANALYZEは、対象とするテーブルの読み取りロックのみを必要とします。したがって、そのテーブルに対する他の操作と並行して実行することができます。

通常、ANALYZEによって収集される統計情報には、各列の典型的な値と各列のデータ分布の概要を示す度数分布が含まれます。ANALYZEによってあまり意味がないとみなされた場合（例えば、一意性制約が付加された列では、典型的な値というものは存在しません）や、列のデータ型が適切な演算子をサポートしていない場合は、片方もしくは両方の情報を省略することがあります。第24章に、統計情報についての詳細が記載されています。

巨大なテーブルでは、ANALYZEは、全ての行を検査するのではなく、テーブルの中からランダムにサンプルを取り出して使用します。これによって、非常に巨大なテーブルであっても短時間で解析することが可能です。しかし、このようにして得られた統計情報はおおよそのものでしかなく、テーブルの内容に変更がなくてもANALYZEを実行する度に変化することに注意してください。これにより、EXPLAINが表示する、プランナの推定コストも多少変化する可能性があります。稀に、このような不確定要素のせいで、プランナがANALYZEを実行した後に異なる問い合わせ計画を選択してしまうことがあります。これを防止するには、以下に示すようにANALYZEで収集される統計情報の量を増やしてください。

設定パラメータ変数default_statistics_targetを調整するか、もしくはALTER TABLE ... ALTER COLUMN ... SET STATISTICS(ALTER TABLE参照)を使用して列単位の統計目標を列毎に設定することで、解析の範囲を制御することができます。目標値として設定するのは、典型的な値のリストにおけるエントリ数の最大値と度数分布のビンの最大数です。デフォルトの目標値は100です。しかし、この値は、プランナの推定精度とANALYZEの処理時間、pg_statisticの占める容量とのトレードオフによって大きくも小さくも調整されることがあります。目標値を0に設定すると、その列に関する統計情報の収集は無効になります。決してWHERE句、GROUP BY句、ORDER BY句に使用されない列に対しては、このような設定が有用です。プランナにとってそのような列の統計情報は不要だからです。

解析対象列の統計情報目標値の最大値によって、統計情報を作成するためにテーブルから抽出する行数が決定します。目標値を大きくすると、比例して、ANALYZEに要する時間とディスク容量が増加します。

ANALYZEで推定される値の1つに各列に出現する個別値の個数があります。行の部分集合のみしか検査されませんので、統計情報の対象をできる限り大きくしたとしても、この推定値はかなり不正確になることが時々あり得ます。この不正確性のために悪い問い合わせ計画となる場合、より正確な値を手作業で求めて、ALTER TABLE ... ALTER COLUMN ... SET (n_distinct = ...)(ALTER TABLE参照)で設定することができます。

解析中のテーブルが1つ以上の子テーブルを持つ場合、ANALYZEは2回統計情報を収集します。1回目は親テーブルのみのテーブル行を対象とし、2回目では親テーブルの行とそのすべての子テーブルの行を対象とします。継承ツリー全体をたどる問い合わせの計画作成では、この2回目の統計情報群が必要とされます。しかし自動バキュームデーモンでは、自動的に解析を行うかどうかを決定する際に親テーブル上の挿入や更新のみを考慮します。このテーブルへの挿入や更新がほとんどなければ、継承関係に対する統計情報は手作業でANALYZEを実行しない限り最新状態にはなりません。

子テーブルに外部テーブルがあり、その外部データラッパがANALYZEをサポートしない場合、その子テーブルは継承の統計を取得する際に無視されます。

解析しようとするテーブルが完全に空である場合、ANALYZEはそのテーブルに対する新しい解析情報を記録しません。これまでの統計情報はすべて保持されます。

互換性

標準SQLにはANALYZE文はありません。

関連項目

VACUUM, vacuumdb, 19.4.4, 24.1.6

BEGIN

BEGIN —トランザクションブロックを開始する

概要

```
BEGIN [ WORK | TRANSACTION ] [ transaction_mode [, ...] ]


transaction_modeは以下のいずれかです。

    ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED }
    READ WRITE | READ ONLY
    [ NOT ] DEFERRABLE
```

説明

BEGINはトランザクションブロックを初期化します。BEGINコマンド以降の文は全て、明示的なCOMMITもしくはROLLBACKが与えられるまで、単一のトランザクションの中で実行されます。デフォルト(BEGINがない場合)は、PostgreSQLはトランザクションを「自動コミット」で実行します。つまり、各文はそれぞれ固有のトランザクションの中で実行され、文の終わりで暗黙的にコミットが実行されます(これは実行が成功した場合です。失敗した場合はロールバックされます)。

トランザクションブロック内では、文はより迅速に実行されます。なぜなら、トランザクションの開始やコミットは、CPUとディスクにかなり高い負荷をかけるからです。また、1つのトランザクションで複数の文を実行することは、複数の関連するテーブルを更新する際、確実に一貫性を保つためにも役立ちます。関連する更新の中に完了していないものが存在する中間的な状態が、他のセッションから参照できなくなるからです。

分離レベル、読み書きモード、または遅延モードが指定されている場合、新しいトランザクションは、SET TRANSACTIONが実行された時と同様の特性を持ちます。

パラメータ

WORK
TRANSACTION

省略可能なキーワードです。これらは何も効果がありません。

BEGIN文のこの他のパラメータについては、SET TRANSACTIONを参照してください。

注釈

START TRANSACTIONにはBEGINと同じ機能があります。

トランザクションを終了させるにはCOMMITまたはROLLBACKを使用してください。

トランザクションブロック内でBEGINを発行すると、警告メッセージが表示されます。ただし、トランザクションの状態には影響ありません。トランザクションブロック内でトランザクションを入れ子にするには、セーブポイントを使用してください（詳しくはSAVEPOINTを参照してください）。

後方互換性の保持のため、連続するtransaction_modes間のカンマは省略することができます。

例

トランザクションブロックを開始します。

```
BEGIN;
```

互換性

BEGINはPostgreSQLの拡張です。標準SQLのSTART TRANSACTIONコマンドと等価です。このマニュアルページには互換性に関する詳細な情報が含まれます。

DEFERRABLE transaction_modeはPostgreSQLの言語拡張です。

埋め込みSQLでは、BEGINというキーワードを異なった目的に使用しています。データベースアプリケーションを移植する時は、トランザクションの意味で使用されているのかどうかに注意してください。

関連項目

COMMIT, ROLLBACK, START TRANSACTION, SAVEPOINT

CALL

CALL — プロシージャを呼び出す

概要

```
CALL name ( [ argument ] [, ...] )
```

説明

CALLは、プロシージャを実行します。

プロシージャがいくつかの出力パラメータを持っている場合、それらの出力パラメータの値を含んだ結果の行を返します。

パラメータ

name

プロシージャの名前です（スキーマ修飾名も可）。

argument

プロシージャの呼び出しに対する入力引数です。名前付きパラメータの使い方を含む、関数とプロシージャの呼び出し構文の完全な詳細は4.3を参照してください。

注釈

プロシージャの呼び出しを許可されるためには、ユーザがプロシージャに対するEXECUTE権限を持つ必要があります。

関数（プロシージャではなく）の呼び出しには、代わりにSELECTを使用します。

トランザクションブロック内でCALLが実行される場合、呼び出されたプロシージャはトランザクション制御文を実行できません。トランザクション制御文は、CALLが自身のトランザクション内で実行された場合のみ許可されます。

PL/pgSQLではCALLコマンド内の出力パラメータの扱いが異なります。42.6.3を参照してください。

例

```
CALL do_db_maintenance();
```

互換性

CALLは標準SQLに準拠しています。

関連項目

CREATE PROCEDURE

CHECKPOINT

CHECKPOINT — 先行書き込みログのチェックポイントを強制的に実行する

概要

```
CHECKPOINT
```

説明

チェックポイントとは、ログ内の情報を反映するために全てのデータファイルを更新する、先行書き込みログのある一時点を指します。チェックポイントによって、全てのデータファイルがディスクに書き出されます。チェックポイントの間に何が起きるかについては、29.4を参照してください。

CHECKPOINTコマンドは、コマンドが発行された時、(19.5.2によって制御される)システムによって予定された通常のチェックポイントを待たず、即座に強制的にチェックポイント処理を行います。通常の運用時にCHECKPOINTが使用されることは想定していません。

リカバリ中に実行された場合、CHECKPOINTは新しくチェックポイントを書き出さずにリスタートポイント(29.4参照)を強制します。

CHECKPOINTを実行できるのはスーパーユーザのみです。

互換性

CHECKPOINTコマンドは、PostgreSQLの拡張です。

CLOSE

CLOSE — カーソルを閉じる

概要

```
CLOSE { name | ALL }
```

説明

CLOSEは、開いたカーソルに関連するリソースを解放します。カーソルが閉じられた後は、そのカーソルに対する操作はできません。カーソルは必要がなくなった時点で閉じるべきです。

トランザクションがCOMMITもしくはROLLBACKで終了された時点で、開いている保持不可能カーソルは全て暗黙的に閉じられます。ROLLBACKにより保持可能カーソルを作成したトランザクションをアボートした場合、この保持可能カーソルは暗黙的に閉じられます。作成したトランザクションが正常にコミットされた場合、保持可能カーソルは明示的にCLOSEが実行されるまで、あるいは、クライアントとの接続が切断されるまで、開いたままになります。

パラメータ

name

閉じる対象となる、現在開いているカーソルの名前です。

ALL

すべてのカーソルを閉じます。

注釈

PostgreSQLには明示的なカーソルのOPEN文がありません。カーソルは宣言された時に開いたとみなされます。カーソルの宣言にはDECLARE文を使用してください。

pg_cursorsシステムビューを問い合わせることにより利用可能なすべてのカーソルを確認することができます。

カーソルがセーブポイントの後に閉じられ、後にロールバックした場合には、CLOSEはロールバックされません。つまり、そのカーソルは閉じたままとなります。

例

カーソルliahonaを閉じます。

```
CLOSE liahona;
```

互換性

CLOSEは標準SQLと完全な互換性を持ちます。ただし、CLOSE ALLはPostgreSQLの拡張です。

関連項目

DECLARE, FETCH, MOVE

CLUSTER

CLUSTER — インデックスに従ってテーブルをクラスタ化する

概要

```
CLUSTER [VERBOSE] table_name [ USING index_name ]
CLUSTER [VERBOSE]
```

説明

CLUSTERは、index_nameで指定されたインデックスに基づき、table_nameで指定されたテーブルをクラスタ化するように、PostgreSQLに指示します。このインデックスは前もってtable_name上に定義されていなければなりません。

テーブルがクラスタ化されると、それぞれのテーブルはインデックス情報に基づいて物理的に並べ直されます。クラスタ化は、1回限りの操作です。クラスタ化後にテーブルが更新されても、その変更はクラスタ化されません。つまり、新規に追加された行や更新された行は、インデックス順には保管されません。(インデックス順に保管したい場合は、コマンドを再度入力し、定期的に再クラスタ化を行います。また、更新される行は十分な領域が利用可能ならば同一ページ内に保持されますので、テーブルのfillfactor格納パラメータを100%より小さく設定することで、更新処理中のクラスタ順序付けを保護するのに役に立ちます。)

テーブルがクラスタ化されると、PostgreSQLはクラスタ化に使用されたインデックスを記録します。CLUSTER table_nameという構文によって、以前と同じインデックスを使用してテーブルを再クラスタ化します。またALTER TABLEのCLUSTERもしくはSET WITHOUT CLUSTER構文を使用して、将来のクラスタ化操作で使用するインデックスを設定したり、過去の設定を取り消すことができます。

パラメータを指定しないでCLUSTERを実行した場合、現在のデータベース内の以前にクラスタ化されたテーブルのうち、呼び出したユーザが所有する全てのテーブルを(スーパーユーザが実行する場合は全てのテーブルを)再クラスタ化します。このパラメータを指定しないCLUSTERを、トランザクションブロック内で実行することはできません。

クラスタ化を行っているテーブルでは、ACCESS EXCLUSIVEロックが獲得されています。これにより、CLUSTERが終わるまで、そのテーブルに対するデータベース操作(読み書き両方)はできません。

パラメータ

table_name

テーブルの名前です(スキーマ修飾名も可)。

index_name

インデックスの名前です。

VERBOSE

各テーブルのクラスタ化を行う時に進行状況報告を出力します。

注釈

テーブル内の個々の行にランダムにアクセスする場合、テーブル内のデータの順序は重要でありません。しかし、テーブル内の特定のデータにアクセスが集中していて、それらのデータをひとまとめにしているインデックスが存在する時は、CLUSTERによる利益を享受できます。テーブルからインデックスの値の範囲や、一致する複数の行を保有する1つのインデックスの値を要求する場合、CLUSTERが役に立ちます。一度インデックスが一致する最初の行に対するテーブルページを認識すると、一致する他の全ての行も同じテーブルページに存在する可能性が高いので、ディスクアクセスを減らして問い合わせ処理の速度を向上することができるからです。

CLUSTERは、指定されたインデックスによるインデックススキャン、または（インデックスがB-Treeの場合）シーケンシャルスキャン後のソートのいずれかを用いて、テーブルを再ソートすることができます。プランナのコストパラメータと利用可能な統計情報に基づき、より高速な方式の選択を試みます。

インデックススキャンが使用される場合、インデックス順にテーブルデータを並べた、テーブルの一時コピーが作成されます。同様に、テーブルの各インデックスの一時コピーも作成されます。したがって、ディスクには、少なくともテーブルとインデックスの合計サイズと同じ容量の空き領域が必要です。

シーケンシャルスキャンとソートが使用される場合も一時的なソートファイルが作成されます。一時的に必要となるサイズの最大値はテーブルサイズの倍のサイズにインデックスサイズを加えた値となります。この方式はインデックススキャンより高速になることが多いのですが、必要なディスク容量に耐えられない場合は、一時的にenable_sortをoffにすることで、この方式を無効にすることができます。

クラスタ処理の前にmaintenance_work_memを程良く大きな値に設定することを勧めます。（しかしCLUSTER操作専用に割り当てられるRAMの容量を超えないようにしてください。）

プランナはテーブルの順序付けに関する統計情報を記録しているため、新しくクラスタ化されたテーブルでは、ANALYZEを実行することが推奨されます。そうしないと、プランナが問い合わせ計画を適切に選択できない可能性があります。

CLUSTERはどのインデックスでクラスタ化したかを記録していますので、対象のテーブルを定期的に再クラスタ化できるように、最初にクラスタ化したいテーブルを手作業でクラスタ化し、その後にパラメータをまったく持たないCLUSTERを実行する定期的な保守用スクリプトを設定することができます。

例

インデックスemployees_indに基づいて、テーブルempをクラスタ化します。

```
CLUSTER employees USING employees_ind;
```

以前に使用したインデックスを使用して、テーブルemployeesをクラスタ化します。

```
CLUSTER employees;
```

データベース内の、以前にクラスタ化されたテーブルを全てクラスタ化します。

```
CLUSTER;
```

互換性

標準SQLにはCLUSTER文はありません。

```
CLUSTER index_name ON table_name
```

という構文も、8.3より前のバージョンのPostgreSQLとの互換性のためサポートされます。

関連項目

clusterdb

COMMENT

COMMENT — オブジェクトのコメントを定義する、または変更する

概要

```
COMMENT ON
{
  ACCESS METHOD object_name |
  AGGREGATE aggregate_name ( aggregate_signature ) |
  CAST (source_type AS target_type) |
  COLLATION object_name |
  COLUMN relation_name.column_name |
  CONSTRAINT constraint_name ON table_name |
  CONSTRAINT constraint_name ON DOMAIN domain_name |
  CONVERSION object_name |
  DATABASE object_name |
  DOMAIN object_name |
  EXTENSION object_name |
  EVENT TRIGGER object_name |
  FOREIGN DATA WRAPPER object_name |
  FOREIGN TABLE object_name |
  FUNCTION function_name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ] |
  INDEX object_name |
  LARGE OBJECT large_object_oid |
  MATERIALIZED VIEW object_name |
  OPERATOR operator_name (left_type, right_type) |
  OPERATOR CLASS object_name USING index_method |
  OPERATOR FAMILY object_name USING index_method |
  POLICY policy_name ON table_name |
  [ PROCEDURAL ] LANGUAGE object_name |
  PROCEDURE procedure_name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ] |
  PUBLICATION object_name |
  ROLE object_name |
  ROUTINE routine_name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ] |
  RULE rule_name ON table_name |
  SCHEMA object_name |
  SEQUENCE object_name |
  SERVER object_name |
  STATISTICS object_name |
  SUBSCRIPTION object_name |
  TABLE object_name |
  TABLESPACE object_name |
```

```
  TEXT SEARCH CONFIGURATION object_name |
  TEXT SEARCH DICTIONARY object_name |
  TEXT SEARCH PARSER object_name |
  TEXT SEARCH TEMPLATE object_name |
  TRANSFORM FOR type_name LANGUAGE lang_name |
  TRIGGER trigger_name ON table_name |
  TYPE object_name |
  VIEW object_name
} IS 'text'


ここでaggregate_signatureは以下の通りです。

* |
[ argmode ] [ argname ] argtype [ , ... ] |
[ [ argmode ] [ argname ] argtype [ , ... ] ] ORDER BY [ argmode ] [ argname ] argtype [ , ... ]
```

説明

COMMENTは、データベースオブジェクトに関するコメントを保存します。

各オブジェクトに保存できるコメント文字列は1つだけです。ですので、コメントを編集するためには、同一オブジェクトに対して新しくCOMMENTコマンドを発行してください。コメントを削除するには、テキスト文字列の部分にNULLを記述してください。オブジェクトが削除された時、コメントは自動的に削除されます。

ほとんどの種類のオブジェクトでは、オブジェクトの所有者のみがコメントを設定することができます。
ロールには所有者がありませんので、COMMENT ON ROLEにおける規則は、スーパーユーザロールに対するコメント付けはスーパーユーザでなければならず、スーパーユーザ以外のロールに対するコメント付けはCREATEROLEを持たなければならないとなります。同様に、アクセスメソッドには所有者がいないため、アクセスメソッドにコメントをつけるにはスーパーユーザでなければなりません。当然ながらスーパーユーザは何にでもコメントを付けることができます。

コメントは、psqlの\d系のコマンドで表示することができます。obj_description()、col_description()、shobj_descriptionという名前の、psqlが使用する組み込み関数を使うように構築することで、他のユーザインタフェースを使ってコメントを取り出せるようになります（表 9.73を参照してください）。

パラメータ

object_name
relation_name.column_name
aggregate_name
constraint_name
function_name
operator_name
policy_name
procedure_name
routine_name
rule_name
trigger_name

コメントを付加するオブジェクトの名前です。テーブル、集約、照合順、変換、ドメイン、外部テーブル、関数、インデックス、演算子、演算子クラス、演算子族、プロシージャ、ルーチン、シーケンス、統計、テキスト検索オブジェクト、データ型、ビューの名前は、スキーマ修飾することができます。列にコメントを付与する場合、relation_nameはテーブル、ビュー、複合型、外部テーブルを参照するものでなければなりません。

table_name
domain_name

制約、トリガー、ルール、ポリシーにコメントを作成する場合、これらのパラメータはオブジェクトが定義されているテーブルまたはドメインの名前を指定します。

source_type

キャストの変換元データ型の名前です。

target_type

キャストの変換先のデータ型の名前です。

argmode

関数、プロシージャまたは集約の引数のモードで、IN、OUT、INOUT、VARIADICのいずれかです。省略時のデフォルトはINです。関数を識別するには入力引数のみが必要ですので、COMMENTが実際にはOUT引数を無視することに注意してください。したがって、IN、INOUTおよびVARIADIC引数を列挙することで十分です。

argname

関数、プロシージャまたは集約の引数の名前です。関数の識別には引数データ型のみが必要ですので、COMMENTが実際には引数の名前を無視することに注意してください。

argtype

関数、プロシージャまたは集約の引数のデータ型です。

large_object_oid

ラージオブジェクトのOIDです。

left_type
right_type

演算子の引数のデータ型（スキーマ修飾も可）です。右単項演算子、左単項演算子における存在しない引数についてはNONEと記述してください。

PROCEDURAL

これには意味はありません。

type_name

変換のデータ型の名前です。

lang_name

変換の言語の名前です。

text

追加するコメントです。文字列リテラルとして記述します。コメントを削除する場合はNULLを記述します。

注釈

現在、コメントの閲覧に関するセキュリティ機構は存在しません。データベースに接続したユーザは誰でも、そのデータベース内のオブジェクトのコメントを参照することができます。データベース、ロール、テーブル空間などの共有オブジェクトに対するコメントは大域的に格納され、クラスタ内の任意のデータベースに接続した任意のユーザが共有オブジェクトに対するコメントをすべて見ることができます。そのため、コメントにはセキュリティ的に重大な情報を記載してはいけません。

例

テーブルmytableにコメントを付けます。

```
COMMENT ON TABLE mytable IS 'This is my table.';
```

先ほどのコメントを削除します。

```
COMMENT ON TABLE mytable IS NULL;
```

その他の例をいくつか示します。

```
COMMENT ON ACCESS METHOD gin IS 'GIN index access method';
COMMENT ON AGGREGATE my_aggregate (double precision) IS 'Computes sample variance';
COMMENT ON CAST (text AS int4) IS 'Allow casts from text to int4';
COMMENT ON COLLATION "fr_CA" IS 'Canadian French';
```

```
COMMENT ON COLUMN my_table.my_column IS 'Employee ID number';
COMMENT ON CONVERSION my_conv IS 'Conversion to UTF8';
COMMENT ON CONSTRAINT bar_col_cons ON bar IS 'Constrains column col';
COMMENT ON CONSTRAINT dom_col_constr ON DOMAIN dom IS 'Constrains col of domain';
COMMENT ON DATABASE my_database IS 'Development Database';
COMMENT ON DOMAIN my_domain IS 'Email Address Domain';
COMMENT ON EVENT TRIGGER abort_ddl IS 'Aborts all DDL commands';
COMMENT ON EXTENSION hstore IS 'implements the hstore data type';
COMMENT ON FOREIGN DATA WRAPPER mywrapper IS 'my foreign data wrapper';
COMMENT ON FOREIGN TABLE my_foreign_table IS 'Employee Information in other database';
COMMENT ON FUNCTION my_function (timestamp) IS 'Returns Roman Numeral';
COMMENT ON INDEX my_index IS 'Enforces uniqueness on employee ID';
COMMENT ON LANGUAGE plpython IS 'Python support for stored procedures';
COMMENT ON LARGE OBJECT 346344 IS 'Planning document';
COMMENT ON MATERIALIZED VIEW my_matview IS 'Summary of order history';
COMMENT ON OPERATOR ^ (text, text) IS 'Performs intersection of two texts';
COMMENT ON OPERATOR - (NONE, integer) IS 'Unary minus';
COMMENT ON OPERATOR CLASS int4ops USING btree IS '4 byte integer operators for btrees';
COMMENT ON OPERATOR FAMILY integer_ops USING btree IS 'all integer operators for btrees';
COMMENT ON POLICY my_policy ON mytable IS 'Filter rows by users';
COMMENT ON PROCEDURE my_proc (integer, integer) IS 'Runs a report';
COMMENT ON PUBLICATION alltables IS 'Publishes all operations on all tables';
COMMENT ON ROLE my_role IS 'Administration group for finance tables';
COMMENT ON ROUTINE my_routine (integer, integer) IS 'Runs a routine (which is a function or
 procedure)';
COMMENT ON RULE my_rule ON my_table IS 'Logs updates of employee records';
COMMENT ON SCHEMA my_schema IS 'Departmental data';
COMMENT ON SEQUENCE my_sequence IS 'Used to generate primary keys';
COMMENT ON SERVER myserver IS 'my foreign server';
COMMENT ON STATISTICS my_statistics IS 'Improves planner row estimations';
COMMENT ON SUBSCRIPTION alltables IS 'Subscription for all operations on all tables';
COMMENT ON TABLE my_schema.my_table IS 'Employee Information';
COMMENT ON TABLESPACE my_tablespace IS 'Tablespace for indexes';
COMMENT ON TEXT SEARCH CONFIGURATION my_config IS 'Special word filtering';
COMMENT ON TEXT SEARCH DICTIONARY swedish IS 'Snowball stemmer for Swedish language';
COMMENT ON TEXT SEARCH PARSER my_parser IS 'Splits text into words';
COMMENT ON TEXT SEARCH TEMPLATE snowball IS 'Snowball stemmer';
COMMENT ON TRANSFORM FOR hstore LANGUAGE plpythonu IS 'Transform between hstore and Python dict';
COMMENT ON TRIGGER my_trigger ON my_table IS 'Used for RI';
COMMENT ON TYPE complex IS 'Complex number data type';
COMMENT ON VIEW my_view IS 'View of departmental costs';
```

互換性

標準SQLにはCOMMENTはありません。

COMMIT

COMMIT — 現在のトランザクションをコミットする

概要

```
COMMIT [ WORK | TRANSACTION ] [ AND [ NO ] CHAIN ]
```

説明

COMMITは現在のトランザクションをコミットします。そのトランザクションで行われた全ての変更が他のユーザから見えるようになり、クラッシュが起きても一貫性が保証されるようになります。

パラメータ

WORK
TRANSACTION

省略可能なキーワードです。何も効果はありません。

AND CHAIN

AND CHAINが指定されていれば、新しいトランザクションは、直前に終了したものと同じトランザクションの特性(SET TRANSACTIONを参照してください)で即時に開始されます。そうでなければ、新しいトランザクションは開始されません。

注釈

トランザクションをアボートするにはROLLBACKを使用してください。

トランザクションの外部でCOMMITを発行しても特に問題は発生しません。ただし、警告メッセージが表示されます。トランザクションの外部でCOMMIT AND CHAINを発行するとエラーになります。

例

現在のトランザクションをコミットし、全ての変更を永続化します。

```
COMMIT;
```

互換性

コマンドCOMMITは標準SQLに準拠しています。COMMIT TRANSACTIONの構文はPostgreSQLでの拡張です。

関連項目

BEGIN, ROLLBACK

COMMIT PREPARED

COMMIT PREPARED — 二相コミット用に事前に準備されたトランザクションをコミットする

概要

```
COMMIT PREPARED transaction_id
```

説明

COMMIT PREPAREDは準備された状態のトランザクションをコミットします。

パラメータ

transaction_id

コミット対象のトランザクションのトランザクション識別子です。

注釈

準備されたトランザクションをコミットするには、元のトランザクションを実行したユーザと同じユーザか、スーパーユーザでなければなりません。しかし、トランザクションを実行したセッションと同じセッションである必要はありません。

このコマンドはトランザクションブロックの内側では実行できません。準備されたトランザクションは即座にコミットされます。

利用可能な準備されたトランザクションはすべて、pg_prepared_xactsシステムビューで列挙されます。

例

foobarトランザクション識別子で識別されるトランザクションをコミットします。

```
COMMIT PREPARED 'foobar';
```

互換性

COMMIT PREPAREDはPostgreSQLの拡張です。これは外部のトランザクション管理システムによる利用を意図したものです。トランザクション管理システムの一部（X/Open XAなど）は標準化されていますが、こうしたシステムのSQL側は標準化されていません。

関連項目

PREPARE TRANSACTION, ROLLBACK PREPARED

COPY

COPY — ファイルとテーブルの間でデータをコピーする

概要

```
COPY table_name [ ( column_name [, ...] ) ]
    FROM { 'filename' | PROGRAM 'command' | STDIN }
    [ [ WITH ] ( option [, ...] ) ]
    [ WHERE condition ]

COPY { table_name [ ( column_name [, ...] ) ] | ( query ) }
    TO { 'filename' | PROGRAM 'command' | STDOUT }
    [ [ WITH ] ( option [, ...] ) ]


ここでoptionは以下のいずれかです。

    FORMAT format_name
    FREEZE [ boolean ]
    DELIMITER 'delimiter_character'
    NULL 'null_string'
    HEADER [ boolean ]
    QUOTE 'quote_character'
    ESCAPE 'escape_character'
    FORCE_QUOTE { ( column_name [, ...] ) | * }
    FORCE_NOT_NULL ( column_name [, ...] )
    FORCE_NULL ( column_name [, ...] )
    ENCODING 'encoding_name'
```

説明

COPYコマンドは、PostgreSQLのテーブルと標準のファイルシステムのファイル間でデータを移動します。COPY TOコマンドはテーブルの内容をファイルにコピーします。また、COPY FROMコマンドは、ファイルからテーブルへとデータをコピーします（この時、既にテーブルにあるデータにコピーした内容を追加します）。また、COPY TOによりSELECT問い合わせの結果をコピーすることができます。

列リストが指定されている場合、COPY TOは指定された列のデータのみをファイルへコピーします。COPY FROMでは、ファイルの各フィールドが順に指定された列に挿入されます。COPY FROMの列リストに含まれていないテーブル列には、デフォルト値が挿入されます。

ファイル名付きのCOPYコマンドは、PostgreSQLサーバに対して直接ファイルへの読み書きをするように命じます。指定したファイルは必ずPostgreSQLユーザ(サーバを実行しているユーザID)からアクセスできる必要があります。また、ファイル名はサーバから見たように指定されなければなりません。PROGRAMが指定された

場合、サーバは指定したコマンドを実行しその標準出力を読み取る、または、プログラムの標準入力に書き出します。コマンドはサーバからの視点で指定しなければならず、また、PostgreSQLユーザによって実行できなければなりません。STDINやSTDOUTが指定された場合、データはクライアントとサーバ間を流れます。

パラメータ

table_name

既存のテーブルの名前です（スキーマ修飾名も可）。

column_name

コピー対象の列リストで、省略可能です。列リストが指定されていない場合は、生成列を除いてテーブルの全ての列がコピーされます。

query

SELECT、VALUES、INSERT、UPDATEあるいはDELETEコマンドで、その結果がコピーされます。問い合わせを括弧でくくる必要があることに注意してください。

INSERT、UPDATEおよびDELETEについてはRETURNING句を付けなければならず、また、対象のリレーションには、複数の文に展開される条件付きルール、ALSOルール、INSTEADルールがあってはなりません。

filename

入出力ファイルのパス名です。入力ファイル名は絶対パスでも相対パスでも記述することができますが、出力ファイル名は絶対パスでなければなりません。Windowsユーザの場合、E''文字列を使用し、パス名内のバックスラッシュを二重にする必要があるかもしれません。

PROGRAM

実行するコマンドです。COPY FROMでは、入力はコマンドの標準出力から読み取られ、COPY TOでは、出力はコマンドの標準入力に書き出されます。

コマンドはシェルから呼び出されることに注意してください。このためシェルコマンドに信頼できない入力元からの任意の引数を渡す必要がある場合、シェルにとって特殊な意味を持つかもしれない特殊文字の除去やエスケープを注意深く実施してください、セキュリティ上の理由のため、固定のコマンド文字列を使用することが最善です。または少なくともユーザからの入力が渡されることを防止してください。

STDIN

入力がクライアントアプリケーションからであることを指定します。

STDOUT

出力がクライアントアプリケーションへであることを指定します。

boolean

指定のオプションを有効とするか無効とするかを指定します。オプションを有効にするには、TRUE、ONまたは1と、無効にするにはFALSE、OFFまたは0と記述します。またboolean値は省略可能であり、省略時はTRUEとみなされます。

FORMAT

読み取りまたは書き込みに使用するデータ形式を選択します。text、csv（カンマ区切り値）、またはbinaryです。デフォルトはtextです。

FREEZE

あたかもVACUUM FREEZEコマンドを実行した後のように、行を凍結した状態のデータコピー処理を要求します。これは、初期データロード処理用の性能オプションとしての利用を意図しています。ロード元のテーブルが現在の副トランザクションで作成または切り詰めされ、開いているカーソルは存在せず、またこのトランザクションで保持される古めのスナップショットが存在しない場合のみ、行は凍結されます。今のところ、パーティションテーブルではCOPY FREEZEを実行できません。

データのロードに成功すると、他のすべてのセッションから即座にデータが参照可能になることに注意してください。これはMVCC可視性に関する一般的な規則に違反しますので、指定するユーザはこれが引き起こすかもしれない潜在的な問題に注意しなければなりません。

DELIMITER

ファイルの各行内の列を区切る文字を指定します。テキスト形式でのデフォルトはタブ文字、CSV形式ではカンマです。これは単一の1バイト文字でなければなりません。このオプションはbinary形式を使用する場合は許されません。

NULL

NULL値を表す文字列を指定します。デフォルトは、テキスト形式では\N（バックスラッシュN）、CSV形式では引用符のない空文字です。NULL値と空文字列を区別する必要がない場合は、テキスト形式であっても空文字列を使用した方が良いかもしれません。このオプションはbinary形式を使用する場合は許されません。

> **注記**
>
> COPY FROMの場合、この文字列と一致するデータ要素はNULL値として格納されます。COPY TO実行時に使用した同じ文字列を使用するようにしてください。

HEADER

ヘッダ行を含むファイルを指定します。このファイルには各列の名前が記載されています。出力では、先頭行にテーブルの列名が入り、入力では先頭行は無視されます。このオプションはCSV形式を使用する場合にのみ許されます。

QUOTE

データ値を引用符付けする際に使用される引用符文字を指定します。デフォルトは二重引用符です。これは単一の1バイト文字でなければなりません。このオプションはCSV形式を使用する場合にのみ許されます。

ESCAPE

データ内の文字がQUOTEの値と一致する場合に、その前に現れなければならない文字を指定します。デフォルトはQUOTEの値と同じです（このためデータ内に引用符用文字があるときは二つ続けます）。これは単一の1バイト文字でなければなりません。このオプションはCSV形式を使用する場合のみ許されます。

FORCE_QUOTE

指定された各列内にある全ての非NULL値を強制的に引用符で囲みます。NULL出力は引用符で囲まれません。*が指定された場合、非NULL値はすべての列で引用符付けされます。このオプションはCOPY TOにおいて、かつ、CSV形式を使用する場合のみ許されます。

FORCE_NOT_NULL

指定された列の値をNULL文字列に対して比較しません。NULL文字列が空であるデフォルトでは、空の値は引用符付けされていなくてもNULLではなく長さが0の文字列として読み取られることを意味します。このオプションはCOPY FROMにおいて、かつ、CSV形式を使用する場合のみで許されます。

FORCE_NULL

指定された列の値を、それが引用符付きであったとしても、NULL文字列と比較し、一致した場合は値をNULLにセットします。NULL文字列が空であるデフォルトでは、引用符付きの空文字列をNULLに変換します。このオプションはCOPY FROMで、かつCSV形式を使用する場合のみ許されます。

ENCODING

ファイルがencoding_nameで符号化されていることを指定します。このオプションが省略された場合、現在のクライアント符号化方式が使用されます。後述の注釈を参照してください。

WHERE

省略可能なWHERE句の一般的な形は以下の通りです。

```
WHERE condition
```

ここでconditionは、評価の結果がboolean型になる任意の式です。この条件を満たさない行はテーブルに挿入されません。変数参照を実際の行の値で置き換えた時に真を返す場合に、行は条件を満たします。

今のところ、WHERE式の中での副問い合わせは認められていませんし、評価はCOPY自身により行われた変更を見ることはありません(これは、式がVOLATILE関数の呼び出しを含む場合に問題になります)。

出力

正常に完了した場合、COPYコマンドは以下の形式のコマンドタグを返します。

```
COPY count
```

countはコピーされた行数です。

> **注記**
>
> psqlはコマンドがCOPY ... TO STDOUTであった場合、および、それと同等なpsqlのメタコマンド\copy ... to stdoutであった場合は、このコマンドタグを表示しません。これは、コマンドタグが表示されたデータと混同されないようにするためです。

注釈

COPY TOは通常のテーブルに対してのみ使用することができます。ビューに対して使用することはできません。しかし、COPY (SELECT * FROM viewname) TO ...と記述して、ビューの現在の内容をコピーすることはできます。

COPY FROMは通常のテーブル、外部テーブル、パーティションテーブルおよびINSTEAD OF INSERTトリガを持つビューに対して使用することができます。

COPYは指定された特定のテーブルのみを扱います。つまり子テーブルへまたは子テーブルからのデータのコピーは行いません。したがって例えば、COPY table TOはSELECT * FROM ONLY tableと同じデータを示します。しかしCOPY (SELECT * FROM table) TO ...を使用して、継承階層内のすべてのデータをダンプすることができます。

COPY TOの場合は値を読み込むテーブルに対するSELECT権限が、COPY FROMの場合は値を挿入するテーブルに対するINSERT権限が必要です。コマンド内で列挙された列に対する列権限があれば十分です。

テーブルの行単位セキュリティが有効な場合、適切なSELECTポリシーがCOPY table TO文に適用されます。現在のところ、COPY FROMは行単位セキュリティが有効なテーブルに対してはサポートされません。代わりにそれと等価なINSERTを使ってください。

COPYコマンドで指定するファイルは、クライアントアプリケーションではなく、サーバが直接読み込み/書き込みを行います。したがって、それらのファイルは、クライアントではなく、データベースサーバマシン上に存在するか、または、データベースサーバマシンからアクセス可能である必要があります。さらに、クライアントではなく、PostgreSQLユーザ(サーバを実行しているユーザID)が、アクセス権限と読み書き権限を持っている必要があります。同様に、PROGRAMで指定されたコマンドは、クライアントアプリケーションではなくサーバにより直接実行されるため、PostgreSQLユーザによって実行可能でなければなりません。ファイル名またはコマンドを指定したCOPYの実行は、データベースのスーパーユーザとデフォルトロールpg_read_server_files、pg_write_server_files、pg_execute_server_programの内の1つの権限を許可されたユーザのみに許可されています。このコマンドによって、サーバがアクセス権限を持つ全てのファイルの読み込み、書き込みやプログラムの実行が可能になってしまうためです。

COPYをpsqlの\copyと混同しないでください。\copyはCOPY FROM STDINやCOPY TO STDOUTを呼び出し、psqlクライアントからアクセスできるファイルにデータの書き込み/読み込みを行います。したがって、\copyコマンドでは、ファイルへのアクセスが可能かどうかと、ファイルに対するアクセス権限の有無は、サーバではなくクライアント側に依存します。

COPYでファイル名を指定する時は、常に絶対パスで記述することをお勧めします。COPY TOコマンドの場合はサーバによって絶対パス指定に変更させられますが、COPY FROMコマンドでは相対パスで指定されたファイルを読み込むことも可能となっています。後者では、クライアントの作業ディレクトリではなく、サーバプロセスの作業ディレクトリ(通常はクラスタのデータディレクトリ)からの相対的なディレクトリとして解釈されます。

PROGRAMを用いたコマンド実行は、SELinuxなどのオペレーティングシステムのアクセス制御機構によって制限されるかもしれません。

COPY FROMは、宛先テーブル上で任意のトリガと検査制約を呼び出しますが、ルールは呼び出しません。

IDENTITY列については、COPY FROMコマンドはINSERTのオプションOVERRIDING SYSTEM VALUEと同じように、必ず入力データが提供した列の値を書き込みます。

COPYの入出力はDateStyleの影響を受けます。デフォルト以外のDateStyleが設定された可能性があるPostgreSQLインストレーションとの移植を確実に行いたい場合は、COPYを使う前にDateStyleをISOに設定しなければなりません。また、IntervalStyleをsql_standardとしてデータをダンプすることは避けることを勧めます。負の時間間隔値が別のIntervalStyle設定を持つサーバで誤解釈される可能性があるためです。

たとえデータがクライアント経由ではなくサーバにより直接ファイルから読み書きされるとしても、入力データはENCODINGオプションまたは現在のクライアント符号化方式にしたがって解釈され、出力データはENCODINGオプションまたは現在のクライアント符号化方式で符号化されます。

COPYでは、エラーが発生するとすぐに処理を停止します。COPY TOコマンドの実行では何ら問題ありませんが、COPY FROMの場合は、対象となるテーブルは初めの方の行を既に受け取っています。これらの行は不可視となり、アクセスすることもできませんが、ディスク領域を占有します。したがって、大きなコピー処理がかなり進んだ後で失敗した場合には、それなりの量の無駄なディスク領域が使われてしまいます。この無駄な領域を取り戻すには、VACUUMを行う必要があります。

FORCE_NULLとFORCE_NOT_NULLは同じ列について同時に使うことができます。その場合の結果は、引用符付きのNULL文字列をNULL値に変換し、引用符なしのNULL文字列を空文字列に変換します。

ファイルの形式

テキスト形式

text形式を使用する場合、読み書きされるデータはテーブルの1つの行を1行で表したテキストファイルとなります。行内の列は区切り文字で区切られます。列の値自体は、その属性のデータ型の出力関数で生成された、または、その入力関数で受け付け可能な文字列です。値がNULLの列では、代わりに指定されたNULL値を表す文字列が使用されます。入力ファイルのいずれかの行にある列数が予期された数と違う場合、COPY FROMはエラーを発生します。

データの終了は、バックスラッシュとピリオド(\.)のみから構成される1行で表されます。ファイルの終了により同じ動作になるので、ファイルからの読み込みの場合はデータ終了マークは不要です。しかし、3.0以前のクライアントプロトコルを使用したクライアントアプリケーションとデータのコピーを行う場合だけは、読み込み、書き込みを問わず、終了マークが必要です。

バックスラッシュ文字(\)は、COPY対象データ内で、行や列の区切り文字と判定される可能性があるデータ文字列の引用符付けに使用します。特に、バックスラッシュ自体、改行、復帰、使用中の区切り文字などの文字が列の値に含まれている場合は、必ず前にバックスラッシュを付けなければなりません。

指定されたNULL文字列はバックスラッシュを付けずにCOPY TOに送られます。一方、COPY FROMでは、バックスラッシュを削除する前にNULL文字列と入力を比較します。したがって、\NといったNULL文字列が実際の\Nというデータ値と混乱することはあり得ません。(これは\\Nとして表現されます。)

COPY FROMは、バックスラッシュで始まる次のような文字の並びを識別します。

文字の並び	表現
\b	バックスペース(ASCII 8)

文字の並び	表現
\f	改ページ（ASCII 12）
\n	改行（ASCII 10）
\r	復帰（ASCII 13）
\t	タブ（ASCII 9）
\v	垂直タブ（ASCII 11）
\数字	バックスラッシュに続き1から3個の8進数の数字をコード番号として指定すると、そのコード番号が表す文字を指定できます。
\xdigits	バックスラッシュ、x という並びに続き1から2個の16進数の数字を指定すると、そのコード番号が表す文字を指定できます。

現在、COPY TOは、バックスラッシュの後ろに8進数や16進数を付けた形式で文字を出力することはありませんが、上記一覧にある制御文字については、バックスラッシュの文字並びを使用します。

上表で記載されていないバックスラッシュ付きの文字はすべて、その文字自体として解釈されます。しかし、不要なバックスラッシュの追加には注意してください。偶然にデータの終わりを示す印（\.）やヌル文字列（デフォルトでは\N）と合致する文字列を生成してしまうかもしれないためです。これらの文字列は他のバックスラッシュの処理を行う前に解釈されます。

COPYデータを生成するアプリケーションは、データ内の改行と復帰をそれぞれ、\nと\rに変換することを強く推奨されています。現在のところ、バックスラッシュと復帰文字でデータ内の復帰を表したり、バックスラッシュと改行文字でデータ内の改行を表すことが可能です。しかし、こういった表現は今後のリリースでは、受け付けられなくなる可能性があります。また、COPYファイルが異なるマシンをまたがって転送される場合、破損するおそれがかなりあります（例えば、UnixからWindowsあるいはその逆）。

COPY TOは各行の行末にUnix形式の改行（「\n」）を出力します。なお、Microsoft Windowsで稼働するサーバの場合は、サーバ上のファイルへのCOPYの場合にのみ復帰/改行（「\r\n」）を出力します。プラットフォームをまたがる一貫性のために、サーバのプラットフォームにかかわらず、COPY TO STDOUTは常に「\n」を送信します。COPY FROMは、改行、復帰、復帰/改行を行末として扱うことができます。データを意図したバックスラッシュのない改行や復帰によるエラーの危険性を減らすために、COPY FROMは、入力行の行末が全て共通でない場合に警告を発します。

CSV形式

この形式オプションは、スプレッドシートなど他の多くのプログラムで使用されるカンマ区切り値（CSV）ファイル形式をインポート、エクスポートするために使用されます。PostgreSQLの標準テキスト形式で使用されるエスケープ規則の代わりに、一般的なCSVのエスケープ機構を生成、認識します。

各レコードの値はDELIMITER文字で区切られます。区切り文字、QUOTE文字、NULL文字列、復帰、改行文字を含む値の場合、全体の値の前後にQUOTE文字が付与されます。値の中でQUOTE文字やESCAPE文字が現れる場合、その前にエスケープ用の文字が付与されます。また、FORCE_QUOTEを使用して、特定列内の非NULL値を出力する時に強制的に引用符を付与することもできます。

CSV形式にはNULL値と空文字列とを区別する標準的な方法はありません。PostgreSQLのCOPYでは引用符によってこれを区別しています。NULLはNULLパラメータの文字列として出力され、引用符で囲まれません。一方、NULLパラメータの文字列に一致する非NULL値は引用符で囲まれます。たとえばデフォルトの設定では、NULLは引用符付けのない空文字列として出力され、空文字列のデータ値は2つの引用符("")で出力されます。データの読み込みの際も同様の規則に従います。FORCE_NOT_NULLを使用して、特定列に対しNULL入力の比較を行わないようにすることもできます。またFORCE_NULLを使うことで、引用符付きのNULL文字列のデータの値をNULLに変換することもできます。

CSV形式ではバックスラッシュは特別な文字ではありませんので、データ終端記号\.がデータ値として現れることがあります。誤った解釈を防ぐために、行内の唯一の項目として\.というデータ値が現れる場合、出力に自動的に引用符が付けられます。また、入力では引用符で括られた場合データ終端記号として解釈されません。他のアプリケーションで作成されたファイルをロードしようとする場合、引用符で括られない列が1つあるだけで、それが\.という値を持つ可能性があるなら、入力ファイル内のこうした値を引用符で括る必要があります。

注記

CSV形式では文字はすべて意味を持ちます。空白文字で括られた引用符付きの値などDELIMITER以外のすべての文字がこうした文字に含まれます。これにより、固定長にするためにCSVの行に空白文字を埋めるシステムから取り出したデータをインポートする時にエラーが発生する可能性があります。このような状況になった場合、PostgreSQLにデータをインポートする前に、そのCSVファイルから余分な空白を除去する前処理が必要になります。

注記

CSV形式は、復帰文字や改行文字が埋め込まれ引用符で囲まれた値を含むCSVファイルを認識し、生成します。したがって、このファイルでは、テキスト形式とは異なり、1つのテーブル行が1行で表されているとは限りません。

注記

奇妙な(時には間違った)CSVファイルを生成するプログラムは多く存在するので、このファイル形式は標準というよりも慣習と言えるものです。したがって、この機能でインポートできないファイルが存在するかもしれませんし、COPYが他のプログラムで処理できないファイルを生成するかもしれません。

バイナリ形式

binary形式オプションにより、すべてのデータはテキストではなくバイナリ形式で書き込み/読み取りされるようになります。テキストやCSV形式よりも多少高速になりますが、バイナリ形式のファイルはマシンアーキテクチャやPostgreSQLのバージョンをまたがる移植性が落ちます。またバイナリ形式はデータ型に非常に依存します。たとえば、smallint列からバイナリデータを出力し、それをinteger列として読み込むことはできません。同じことをテキスト形式で実行すれば動作するのですが。

binaryファイルの形式は、ファイルヘッダ、行データを含む0以上のタプル、ファイルトレーラから構成されます。ヘッダとデータはネットワークバイトオーダです。

> **注記**
>
> 7.4以前のリリースのPostgreSQLでは異なるバイナリファイル形式を使用していました。

ファイルヘッダ

ファイルヘッダは15バイトの固定フィールドとその後に続く可変長ヘッダ拡張領域から構成されます。固定フィールドは以下の通りです。

署名

PGCOPY\n\377\r\n\0という11バイトの並びです。この署名の必須部分にNULLバイトが含まれていることに注意してください（この署名は、8ビットを通過させない転送方式によってファイルが破損した場合、これを容易に識別できるように設計されています。署名は、改行コード変換やNULLバイトの削除、上位ビット落ち、パリティの変更などによって変化します）。

フラグフィールド

このファイル形式の重要な部分となる32ビット整数のビットマスクです。ビットには0（LSB）から31（MSB）までの番号が付いています。このフィールドは、このファイル形式で使用される他の全ての整数フィールドも同様、ネットワークバイトオーダ（最上位バイトが最初に現れる）で保存されていることに注意してください。ファイル形式上の致命的な問題を表すために、16–31ビットは予約されています。この範囲に想定外のビットが設定されていることが判明した場合、読み込み先は処理を中断しなければなりません。後方互換における形式の問題を通知するために、0–15ビットは予約されています。この範囲に想定外のビットが設定されていても、読み込み先は無視すべきです。現在、1つのビットだけがフラグビットとして定義されており、残りは0でなければなりません。

ビット16

1ならば、OIDがデータに含まれています。0ならば、含まれていません。OIDシステム列は今はもうPostgreSQLでサポートされていませんが、フォーマットには指標が含まれています。

ヘッダ拡張領域長

自分自身を除いた、ヘッダの残り部分のバイト長を示す32ビットの整数です。現在、これは0となっており、すぐ後に最初のタプルが続きます。今後、ヘッダ内に追加データを格納するような形式の変更があるかもしれません。読み込み側では、ヘッダ拡張データの扱いがわからない場合、そのデータをスキップしなければなりません。

ヘッダ拡張領域は、それ自身で認識することができる塊の並びを保持するために用意されています。フラグフィールドは読み込み先に拡張領域の内容を知らせるものではありません。ヘッダ拡張内容の個々の設計は今後のリリースのために残してあります。

この設計によって、後方互換性を維持するヘッダの追加（ヘッダ拡張チャンクの追加や下位フラグビットの設定）と後方互換性のない変更（変更を通知するための高位フラグビットの設定や必要に応じた拡張領域へのサポート情報追加）の両方に対応できます。

タプル

全てのタプルはタプル内のフィールド数を表す16ビットの整数から始まります（現時点では、テーブル内の全てのタプルは同一のフィールド数を持つことになっていますが、今後、これは変更される可能性があります）。その後に、タプル中のそれぞれのフィールドが続きます。これらのフィールドには、先頭にフィールドデータが何バイトあるかを表す32ビット長のワードが付けられています（このワードが表す長さには自分自身は含まれません。したがって、0になることもあります）。特殊な値としてNULLフィールドを表す-1が用意されています。このNULLが指定された場合、値用のバイトはありません。

フィールド間には整列用のパッドやその他の余計なデータはありません。

現在、バイナリ形式のファイル内の全てのデータ値は、バイナリ形式（形式コード1）であると想定されています。将来の拡張によって、列単位に形式コードを指定するヘッダフィールドが追加される可能性があります。

実際のタプルデータとして適切なバイナリ形式を決定するためには、PostgreSQLのソース、特に各列のデータ型用の*send 関数と*recv関数（通常はソースの配布物内のsrc/backend/utils/adtディレクトリにあります）を調べなければなりません。

このファイルにOIDが含まれる場合、OIDフィールドがフィールド数ワードの直後に続きます。これは、フィールド数に含まれない点を除いて、通常のフィールドです。OIDシステム列はPostgreSQLの現在のバージョンではサポートされていないことに注意してください。

ファイルトレーラ

ファイルトレーラは、16ビットの整数ワードで構成され、-1が入っています。タプルのフィールド数ワードとは、容易に区別できます。

読み込み側は、フィールドカウントワードが-1でも、想定した列数でもなかった場合はエラーを報告しなければなりません。これにより、何らかの理由でデータと一致しなかったことを判定する特別な検査を行うことが可能になります。

例

次の例では、フィールド区切り文字として縦棒（|）を使用してテーブルをクライアントにコピーします。

```
COPY country TO STDOUT (DELIMITER '|');
```

ファイルからcountryテーブルにデータをコピーします。

```
COPY country FROM '/usr1/proj/bray/sql/country_data';
```

名前が'A'から始まる国のみをファイルにコピーします。

```
COPY (SELECT * FROM country WHERE country_name LIKE 'A%') TO '/usr1/proj/bray/sql/
a_list_countries.copy';
```

圧縮したファイルにコピーするためには、以下のように出力を外部の圧縮プログラムにパイプで渡すことができます。

```
COPY country TO PROGRAM 'gzip > /usr1/proj/bray/sql/country_data.gz';
```

これはSTDINからテーブルにコピーするのに適したデータの例です。

```
AF      AFGHANISTAN
AL      ALBANIA
DZ      ALGERIA
ZM      ZAMBIA
ZW      ZIMBABWE
```

各行の空白文字は実際にはタブ文字であることに注意してください。

以下は同一のデータをバイナリ形式で出力したものです。データをUnixユーティリティod -cを使ってフィルタしたものを示します。テーブルには3列あり、最初のデータ型はchar(2)、2番目はtext、3番目はintegerです。全ての行の3列目はNULL値です。

```
0000000   P   G   C   O   P   Y  \n 377  \r  \n  \0  \0  \0  \0  \0
0000020  \0  \0  \0  \0 003  \0  \0  \0 002   A   F  \0  \0  \0 013   A
0000040   F   G   H   A   N   I   S   T   A   N 377 377 377 377  \0 003
0000060  \0  \0  \0 002   A   L  \0  \0  \0 007   A   L   B   A   N   I
0000100   A 377 377 377 377  \0 003  \0  \0  \0 002   D   Z  \0  \0  \0
0000120 007   A   L   G   E   R   I   A 377 377 377 377  \0 003  \0  \0
0000140  \0 002   Z   M  \0  \0  \0 006   Z   A   M   B   I   A 377 377
0000160 377 377  \0 003  \0  \0  \0 002   Z   W  \0  \0  \0  \b   Z   I
0000200   M   B   A   B   W   E 377 377 377 377 377 377
```

互換性

標準SQLにはCOPY文はありません。

以下の構文は、PostgreSQLバージョン9.0より前に使用されていたもので、まだサポートされています。

```
COPY table_name [ ( column_name [, ...] ) ]
    FROM { 'filename' | STDIN }
    [ [ WITH ]
          [ BINARY ]
          [ DELIMITER [ AS ] 'delimiter_character' ]
          [ NULL [ AS ] 'null_string' ]
          [ CSV [ HEADER ]
                [ QUOTE [ AS ] 'quote_character' ]
                [ ESCAPE [ AS ] 'escape_character' ]
                [ FORCE NOT NULL column_name [, ...] ] ] ]
```

```
COPY { table_name [ ( column_name [, ...] ) ] | ( query ) }
    TO { 'filename' | STDOUT }
    [ [ WITH ]
          [ BINARY ]
          [ DELIMITER [ AS ] 'delimiter_character' ]
          [ NULL [ AS ] 'null_string' ]
          [ CSV [ HEADER ]
                [ QUOTE [ AS ] 'quote_character' ]
                [ ESCAPE [ AS ] 'escape_character' ]
                [ FORCE QUOTE { column_name [, ...] | * } ] ] ]
```

この構文では、BINARYとCSVがFORMATオプションの引数ではなく、独立したキーワードとして扱われることに注意してください。

以下の構文は、PostgreSQLバージョン7.3より前に使用されていたもので、まだサポートされています。

```
COPY [ BINARY ] table_name
    FROM { 'filename' | STDIN }
    [ [USING] DELIMITERS 'delimiter_character' ]
    [ WITH NULL AS 'null_string' ]

COPY [ BINARY ] table_name
    TO { 'filename' | STDOUT }
    [ [USING] DELIMITERS 'delimiter_character' ]
    [ WITH NULL AS 'null_string' ]
```

CREATE ACCESS METHOD

CREATE ACCESS METHOD — 新しいアクセスメソッドを定義する

概要

```
CREATE ACCESS METHOD name
    TYPE access_method_type
    HANDLER handler_function
```

説明

CREATE ACCESS METHODは新しいアクセスメソッドを作成します。

アクセスメソッドの名前はデータベース内で一意でなければなりません。

スーパーユーザのみが新しいアクセスメソッドを定義できます。

パラメータ

name

作成するアクセスメソッドの名前です。

access_method_type

この句では定義するアクセスメソッドの型を指定します。現在のところ、TABLEとINDEXだけがサポートされています。

handler_function

handler_functionはアクセスメソッドを表す、事前に登録された関数の名前（スキーマ修飾可）です。ハンドラ関数はinternal型の引数を1つだけ取るものとして定義される必要があります。戻り値の型はアクセスメソッドの型に依存し、TABLEアクセスメソッドの場合はtable_am_handlerでなければならず、INDEXのアクセスメソッドの場合はindex_am_handlerでなければなりません。ハンドラ関数が実装しなければならないC言語でのAPIはアクセスメソッドの型によって変わります。TABLEのアクセスメソッドのAPIについては第60章で、INDEXのアクセスメソッドのAPIについては第61章で説明されています。

例

INDEXのアクセスメソッドheptreeをハンドラ関数heptree_handlerで作成するには、次のようにします。

```
CREATE ACCESS METHOD heptree TYPE INDEX HANDLER heptree_handler;
```

互換性

`CREATE ACCESS METHOD`はPostgreSQLの拡張です。

関連項目

DROP ACCESS METHOD, CREATE OPERATOR CLASS, CREATE OPERATOR FAMILY

CREATE AGGREGATE

CREATE AGGREGATE — 新しい集約関数を定義する

概要

```
CREATE [ OR REPLACE ] AGGREGATE name ( [ argmode ] [ argname ] arg_data_type [ , ... ] ) (
    SFUNC = sfunc,
    STYPE = state_data_type
    [ , SSPACE = state_data_size ]
    [ , FINALFUNC = ffunc ]
    [ , FINALFUNC_EXTRA ]
    [ , FINALFUNC_MODIFY = { READ_ONLY | SHAREABLE | READ_WRITE } ]
    [ , COMBINEFUNC = combinefunc ]
    [ , SERIALFUNC = serialfunc ]
    [ , DESERIALFUNC = deserialfunc ]
    [ , INITCOND = initial_condition ]
    [ , MSFUNC = msfunc ]
    [ , MINVFUNC = minvfunc ]
    [ , MSTYPE = mstate_data_type ]
    [ , MSSPACE = mstate_data_size ]
    [ , MFINALFUNC = mffunc ]
    [ , MFINALFUNC_EXTRA ]
    [ , MFINALFUNC_MODIFY = { READ_ONLY | SHAREABLE | READ_WRITE } ]
    [ , MINITCOND = minitial_condition ]
    [ , SORTOP = sort_operator ]
    [ , PARALLEL = { SAFE | RESTRICTED | UNSAFE } ]
)

CREATE [ OR REPLACE ] AGGREGATE name ( [ [ argmode ] [ argname ] arg_data_type [ , ... ] ]
                        ORDER BY [ argmode ] [ argname ] arg_data_type [ , ... ] ) (
    SFUNC = sfunc,
    STYPE = state_data_type
    [ , SSPACE = state_data_size ]
    [ , FINALFUNC = ffunc ]
    [ , FINALFUNC_EXTRA ]
    [ , FINALFUNC_MODIFY = { READ_ONLY | SHAREABLE | READ_WRITE } ]
    [ , INITCOND = initial_condition ]
    [ , PARALLEL = { SAFE | RESTRICTED | UNSAFE } ]
    [ , HYPOTHETICAL ]
)


または以下の旧構文
```

```
CREATE [ OR REPLACE ] AGGREGATE name (
    BASETYPE = base_type,
    SFUNC = sfunc,
    STYPE = state_data_type
    [ , SSPACE = state_data_size ]
    [ , FINALFUNC = ffunc ]
    [ , FINALFUNC_EXTRA ]
    [ , FINALFUNC_MODIFY = { READ_ONLY | SHAREABLE | READ_WRITE } ]
    [ , COMBINEFUNC = combinefunc ]
    [ , SERIALFUNC = serialfunc ]
    [ , DESERIALFUNC = deserialfunc ]
    [ , INITCOND = initial_condition ]
    [ , MSFUNC = msfunc ]
    [ , MINVFUNC = minvfunc ]
    [ , MSTYPE = mstate_data_type ]
    [ , MSSPACE = mstate_data_size ]
    [ , MFINALFUNC = mffunc ]
    [ , MFINALFUNC_EXTRA ]
    [ , MFINALFUNC_MODIFY = { READ_ONLY | SHAREABLE | READ_WRITE } ]
    [ , MINITCOND = minitial_condition ]
    [ , SORTOP = sort_operator ]
)
```

説明

CREATE AGGREGATEは、新しい集約関数を定義します。CREATE OR REPLACE AGGREGATEは、新しい集約関数を定義するか、既存の定義を置き換えます。配布物には基本的、かつ、よく使用される集約関数がいくつか含まれています。これらの集約関数については、9.21に文書化されています。新しい型を定義する場合、またはまだ提供されていない集約関数が必要な場合、必要な機能を実現するためにCREATE AGGREGATEを使うことができます。

既存の定義を置き換える場合には、引数の型、結果の型、直接引数の数を変えることはできません。また、新しい定義は古いものと同じ種類(通常の集約、順序集合集約、仮想集合集約)でなければなりません。

スキーマ名が付けられている場合(例えば、CREATE AGGREGATE myschema.myagg ...)、集約関数は指定されたスキーマに作成されます。スキーマ名がなければ、集約関数は現在のスキーマに作成されます。

集約関数は名前と入力データ型(複数可)の組み合わせによって識別されます。演算の対象となる入力データ型が異なっていれば、同じスキーマ内に同じ名前の集約関数があっても構いません。1つのスキーマ内では、集約関数の名前と入力データ型の組み合わせは、通常の関数の名前と入力データ型の組み合わせと異なる必要があります。この動作は通常の関数名についてオーバーロードする時と同じです(CREATE FUNCTIONを参照してください)。

単純な集約関数は1つか2つの通常の関数から作られます。状態遷移関数sfuncと最終計算関数ffunc(省略可能)です。これらは以下のように使われます。

```
```

```
sfunc( 内部状態, 次のデータ値 ) ---> 次の内部状態
ffunc( 内部状態 ) ---> 集約の結果
```

PostgreSQLは、集約の現在の内部状態を保持するstypeデータ型の一時変数を作成します。それぞれの入力行に対して、集約引数の値が計算され、現在の状態値と新しい引数値で状態遷移関数が呼び出され、新しい内部状態変数が計算されます。全ての行が処理されると、最終関数が1回呼び出され、集約の出力値が計算されます。最終関数がない場合は、終了時の状態値がそのまま返されます。

集約関数は、初期条件、つまり内部状態値の初期値を提供することができます。これはtext型の値としてデータベースに格納されますが、状態値データ型の定数として有効な外部表現でなければいけません。初期状態が与えられていない場合、状態値はNULLから始まります。

状態遷移関数が「strict」と宣言されている場合、NULLを入力値にして呼び出すことはできません。そのような遷移関数では、集約は次のように実行されます。NULL入力値を持つ行は無視されます。(関数は呼び出されず、前回の状態値が保持されます。) 初期状態値がNULLである場合、初めて入力行がすべて非NULL入力値であった時にその最初の引数の値で状態値を置き換え、以後、すべて非NULL入力値のそれぞれの行について、遷移関数が呼び出されます。このような動作は、maxのような集約を実装するには便利です。ただし、state_data_typeが最初のarg_data_typeと同じ時にのみ有効であることに注意してください。これらの型が異なる時は、非NULL初期値を供給するか、strictでない遷移関数を使わなければいけません。

状態遷移関数がstrictでない場合は、それぞれの入力行に対してその関数が無条件に呼び出されるので、NULL入力とNULL状態値を自分で処理しなければいけません。これは、関数の作成者が、集約関数におけるNULL値の扱いを完全に制御できることを意味します。

最終関数が「strict」と宣言されていると、終了状態値がNULLの時は、最終関数が呼び出されません。その場合、NULLという結果が自動的に出力されます (もちろんこれは、strictな関数の一般的な動作に過ぎません)。いずれにせよ、最終関数はNULLを返すことができます。例えば、avgの最終関数は、入力が0行だとわかるとNULLを返します。

最終関数を、状態値だけでなく、集約の入力値に対応する追加パラメータも取るように宣言すると便利なことがあります。こうすることの主な理由に、最終関数が多様型で、状態値のデータ型だけでは結果の型を決定するのに不十分である、ということがあります。これらの追加パラメータは必ずNULLとして渡されます(従ってFINALFUNC_EXTRAオプションが使われている場合、最終関数はstrictであってはいけません)が、それでも有効なパラメータです。最終関数は、現在の呼び出しでの実際の引数の型を特定するために、例えばget_fn_expr_argtypeを使うことができます。

集約は、37.12.1に記述されているように移動集約モードをサポートすることができます。このためには、MSFUNC、MINVFUNC、MSTYPEのパラメータを指定する必要があり、またオプションでMSSPACE、MFINALFUNC、MFINALFUNC_EXTRA、MFINALFUNC_MODIFY、MINITCONDのパラメータを指定できます。MINVFUNCを除き、これらのパラメータは、Mのない単純集約の対応するパラメータのように動作しますが、集約について逆変換関数を含む別の実装を定義します。

パラメータのリストにORDER BYを含む構文は、順序集合集約と呼ばれる特別な種類の集約を作ります。またHYPOTHETICALが指定されている場合は、仮想集合集約が作られます。これらの集約は、ソートされた値のグループに対して、順序に依存した方法で作用するため、入力についてのソート順の指定は、呼び出しにおける本質的な部分になります。また、これらの集約は直接引数をとることができます。直接引数は、行毎に一度ではなく、集約に対して一度だけ評価されます。仮想集合集約は、順序集合集約のサブクラスで、直接引数のいくつかが、集約される引数の列と、数とデータ型についてマッチする必要があります。これにより、直接引数の値を、「仮想的な」行として、集約の入力行の集合に加えることができます。

37.12.4で説明されている通り、集約では部分集約をサポートすることができます。このためにはCOMBINEFUNCパラメータを指定する必要があります。state_data_typeがinternalの場合、通常はSERIALFUNCおよびDESERIALFUNCパラメータも提供して、並列集約を可能にするのが適切でしょう。並列集約を可能にするには、集約にPARALLEL SAFEの印をつける必要もあることに注意してください。

MINやMAXのような振舞いをする集約では、すべての入力行を走査せずにインデックスを検索することで最適化できることがあります。このように最適化される集約の場合、ソート演算子を指定することで明示してください。その演算子で生成されるソート順で集約の最初の要素が生成されなければならないということが基本的な必要条件です。言い換えると、

```
SELECT agg(col) FROM tab;
```

が

```
SELECT col FROM tab ORDER BY col USING sortop LIMIT 1;
```

と同じでなければならないということです。更に、集約がNULL入力を無視すること、および、NULL以外の入力がまったくなかった時にのみNULLという結果を返すことも前提となります。通常、データ型の<演算子はMINのソート演算子として、また、>演算子はMAXのソート演算子として適切です。指定した演算子がB-treeインデックス演算子クラスの「より小さい」ストラテジか「より大きい」ストラテジのメンバでない限り、最適化が実際には効果がないことに注意してください。

集約関数を作成するためには、引数の型、状態の型、戻り値の型に対するUSAGE権限およびサポート関数に対するEXECUTE権限を持たなければなりません。

パラメータ

name

作成する集約関数の名前です（スキーマ修飾名も可）。

argmode

引数のモードで、INまたはVARIADICです。(集約関数はOUTの引数をサポートしません。) 省略した場合のデフォルトはINです。VARIADICを指定できるのは、最後の引数だけです。

argname

引数の名前です。現在は、文書化を目的とする場合にのみ有効です。省略した場合、引数には名前がありません。

arg_data_type

集約関数が演算する入力データ型です。引数が存在しない集約関数を作成するには、引数指定のリストに*と記載してください（例えば count(*)などの集約です）。

base_type

CREATE AGGREGATEの旧構文では、入力データ型は集約の名前の次に記載されたものではなくbasetypeパラメータにより指定されます。この構文では入力パラメータを1つしかとれないことに注意

してください。この構文で引数を持たない集約を定義するためには、basetypeを"ANY" (*ではありません)と指定してください。順序集合集約関数は旧構文では定義できません。

sfunc

それぞれの入力行に対して呼び出される状態遷移関数の名前です。通常のN引数を持つ集約関数では、sfuncはN+1個の引数を取らなければなりません。最初の引数はstate_data_type型で、残りはその集約の入力データ型として宣言したものと一致していなければなりません。この関数はstate_data_type型の値を返さなければなりません。この関数は、現在の状態値と現在の入力データ値を受け取り、次の状態値を返します。

順序集合(仮想集合を含む)集約では、状態遷移関数は現在値と集約引数のみを受け取り、直接引数は受け取りません。それ以外の点は全く同じです。

state_data_type

集約の状態値のデータ型です。

state_data_size

集約の状態値のおおよその平均サイズ(単位はバイト)です。このパラメータを省略した場合、あるいはゼロを指定した場合、state_data_typeに基づいたデフォルトの推定が使われます。プランナは、グループ化された集約のクエリに必要なメモリを推定するのに、この値を使います。

ffunc

最終関数の名前です。最終関数は、全ての入力行に対する処理が終わった後、集約の結果を計算するために呼び出されます。通常の集約では、この関数はstate_data_type型の引数を1つ取らなければなりません。集約の出力データ型はこの関数の返り値として定義されます。ffuncが指定されない場合には、集約の結果として終了時の状態値が使われます。出力型はstate_data_typeになります。

順序集合(仮想集合を含む)集約では、最終関数は終了時の状態値だけでなく、すべての直接引数の値も受け取ります。

FINALFUNC_EXTRAが指定された場合、最終関数は、終了時の状態値と直接引数に加えて、集約の通常の(集約された)引数に対応する追加のNULL値を受け取ります。これは主に、多様型の集約が定義されているときに、集約の結果の型を正しく解決するのに役立ちます。

FINALFUNC_MODIFY = { READ_ONLY | SHAREABLE | READ_WRITE }

このオプションは、最終関数が引数を変更しない純粋な関数であるかどうかを指定します。READ_ONLYは変更しないことを示します。他の2つの値は遷移状態値を変更するかもしれないことを示します。さらなる詳細は以下のNotesをご覧ください。順序集合集約ではデフォルトがREAD_WRITEであることを除き、デフォルトはREAD_ONLYです。

combinefunc

集約関数が部分集約をサポートできるようにするために、combinefuncを指定することができます。これが指定されると、combinefuncは、入力値の何らかの部分集合に対する集約の結果を含む2つのstate_data_type値を結合し、両方の入力に対する集約結果を表す新しいstate_data_typeを生成しなければなりません。この関数は、個々の入力行に対して作用してそれを集約中の状態に追加するのではなく、実行中の状態に別の集約状態を追加するsfuncとして考えることができます。

combinefuncは、state_data_typeの引数を2つ取り、state_data_typeの値を返すものとして宣言されなければなりません。この関数は「strict」とすることもできます。その場合、入力状態の一方がNULLのときは関数が呼び出されず、他方の状態が正しい結果であると見なされます。

state_data_typeがinternalの集約関数では、combinefuncをSTRICTにすることはできません。この場合、combinefuncはNULL状態を正しく処理し、返される状態が集約のメモリコンテキスト内に適切に保存されることを確実にする必要があります。

serialfunc

state_data_typeがinternalの集約関数は、serialfunc関数がある場合に限り、並列集約に参加することができます。serialfuncは、集約の状態を他のプロセスに転送するためにbyteaの値にシリアライズしなければなりません。この関数はinternal型の引数を1つ取り、bytea型を返さなければなりません。これに対応するdeserialfuncも必要です。

deserialfunc

以前にシリアライズされた集約状態をデシリアライズしてstate_data_typeに戻します。この関数はbytea型およびinternal型の2つの引数を取り、internal型の結果を生成しなければなりません。(注意:2番目のinternalの引数は使用されませんが、型の安全性の理由から必要となっています。)

initial_condition

状態値の初期設定です。データ型state_data_typeとして受け取り可能な文字列定数でなければいけません。このパラメータが指定されない場合、状態値はNULLから始まります。

msfunc

移動集約モードにおいて、それぞれの入力行に対して呼び出される前方状態遷移関数の名前です。これは最初の引数と結果がmstate_data_type型で、state_data_typeとは異なるかもしれないことを除けば、通常の遷移関数と全く同じです。

minvfunc

移動集約モードで使われる逆状態遷移関数の名前です。この関数はmsfuncと同じ引数および結果型を持ちますが、現在の集約状態に対して、値を追加するのではなく、取り除くために使われます。逆遷移関数は前方状態遷移関数と同じstrictさの属性を持っていなければなりません。

mstate_data_type

移動集約モードを使うときの、集約状態値のデータ型です。

mstate_data_size

移動集約モードを使うときの、集約状態値のおおよその平均サイズ(バイト単位)です。state_data_sizeと同じように作用します。

mffunc

移動集約モードを使うときに、すべての入力行が走査された後で、集約結果を計算するために呼び出される最終関数の名前です。これは、最初の引数の型がmstate_data_typeであり、MFINALFUNC_EXTRAにより追加のダミー引数を指定できることを除けば、ffuncと同じように作用します。mffuncまた

はmstate_data_typeによって決定される集約結果の型は、集約の通常の実装によって決定される型と適合しなければなりません。

MFINALFUNC_MODIFY = { READ_ONLY | SHAREABLE | READ_WRITE }

このオプションはFINALFUNC_MODIFYと似ていますが、移動集約最終関数の挙動を記述します。

minitial_condition

移動集約モードを使うときの、状態値の初期設定です。これは、initial_conditionと同じように作用します。

sort_operator

MINまたはMAXのような集約に対して関連付けされるソート演算子です。これは単なる演算子の名前です（スキーマ修飾可能）。この演算子は集約（これは単一引数の通常の集約でなければなりません）と同じ入力データ型を持つと前提されています。

PARALLEL = { SAFE | RESTRICTED | UNSAFE }

PARALLEL SAFE、PARALLEL RESTRICTED、PARALLEL UNSAFEの意味はCREATE FUNCTIONにおけるものと同じです。集約は、その印がPARALLEL UNSAFE（これがデフォルトです！）あるいはPARALLEL RESTRICTEDとなっている場合、並列処理での使用を考慮されません。プランナは集約のサポート関数の並列処理安全性の印を考慮せず、集約自体の印のみを参照することに注意してください。

HYPOTHETICAL

順序集合集約についてのみ、このフラグは、仮想集合集約の要求に従って集約の引数が処理されることを指定します。つまり、最後のいくつかの引数が、集約される(WITHIN GROUPの)引数と適合しなければなりません。HYPOTHETICALフラグは実行時の動作には何の影響もなく、集約の引数のデータ型と照合についての解析時の解決にのみ影響します。

CREATE AGGREGATEのパラメータは、任意の順番で記述することができます。上記の順番で記述する必要はありません。

注釈

サポート関数名を指定するパラメータでは、必要なら、SFUNC = public.sumのようにスキーマ名を書くことができます。しかし、引数の型をそこに書くことはできません。サポート関数の引数の型は、他のパラメータにより決定されるからです。

通常PostgreSQL関数は入力値を変更しない純粋な関数であることが期待されます。しかし集約のコンテキストで使用される場合には、集約遷移関数は、これを偽ってその状態遷移引数を直接変更することが許されます。これにより、遷移状態の新しいコピーを都度作るのに比べると、かなりの性能上の利点が期待できます。

同様に、集約最終関数は入力値を変更しない純粋な関数であることが期待されますが、状態遷移引数を変更するのを避けることが実用的でないことがあります。そうした挙動はFINALFUNC_MODIFY引数を使って宣言しなければなりません。READ_WRITE値は、最終関数が遷移状態を明示されていない方法で変更することを示します。この値はwindow関数として集約を使うことを回避します。また、同じ入力値と遷移関数を共有す

る集約の遷移状態をマージすることを回避します。SHAREABLE値は、遷移関数が最終関数の後に適用できないが、最終関数の複数回の呼び出しを終了遷移状態値に適用できることを示します。この値はwindow関数として集約を使うことを回避しますが、遷移状態のマージを許容します。(つまり、ここでの最適化の眼目は、同じ最終関数を繰り返し適用することではなく、異なる最終関数を同じ終了遷移状態値に適用することです。これは最終関数のうちREAD_WRITEと印付けられているものが一つもない限り許容されます。)

集約が移動集約モードをサポートしていると、移動フレームの開始のあるウィンドウ(つまり、UNBOUNDED PRECEDING以外のフレーム開始モード)として集約が使われる場合に計算の効率が向上します。概念的には、前方遷移関数はウィンドウフレームに下から入るときに集約の状態に入力値を追加し、逆遷移関数はフレームを上から抜けるときにそれを取り除きます。従って、値が取り除かれるときは、必ず追加された時と同じ順番で取り除かれます。従って、逆遷移関数が実行される時は、いつでも最も早く追加されたけれども、まだ取り除かれていない引き数値を受け取ります。逆遷移関数は、最も古い行を取り除いた後、現在の状態に少なくとも1行が残ることを前提とできます。(そうならない場合は、ウィンドウ関数の仕組みは、逆遷移関数を使うのでなく、単純に新しい集約を開始します。)

移動集約モードの前方遷移関数は、新しい集約値としてNULLを返すことは許されません。逆遷移関数がNULLを返した場合、それは、逆関数がその入力値について状態計算を元に戻すことができなかったと見なされるため、集約の計算は現在のフレームの開始位置からやり直しとなります。こうすることで、実行中の状態値を元に戻すのが現実的でないということが稀に起こるような場合でも、移動集約モードを使うことができます。

移動集約が実装されていない場合でも、集約は移動フレームを使うことができますが、フレームの開始が移動した場合、PostgreSQLは必ず集約全体を再計算します。集約が移動集約モードをサポートするかどうかに関わらず、PostgreSQLは移動フレームの終了を再計算なしに処理することができます。これは、集約の状態に新しい値を追加し続けることで処理できます。これがwindow関数として集約を利用するためには最終関数が読み出し専用でなければならない理由です。最終関数は集約の状態値を破壊しないものとされるので、フレームの境界の集合に対して集約結果の値が得られた後でも、集約を続行することが可能です。

順序集合集約の構文では、VARIADICを最後の直接パラメータと、最後の集約(WITHIN GROUP)パラメータの両方について指定することができます。しかし、現在の実装ではVARIADICの使用を2つの方法に制限しています。1つ目は、順序集合集約では、VARIADIC "any"のみが利用でき、他のvariadicの配列型は利用できないことです。2つ目は、最後の直接パラメータがVARIADIC "any"の場合、集約パラメータは1つだけしか使えず、かつそれもVARIADIC "any"でなければならない、ということです。(システムカタログで使われる表現において、これらの2つのパラメータは、1つのVARIADIC "any"要素に統合されています。なぜなら、pg_procは2つ以上のVARIADICパラメータがある関数を表現できないからです。) 仮想集合集約の場合、VARIADIC "any"パラメータに対応する直接引数は仮想的なパラメータで、それより前のパラメータは、集約引数に対応する制約のない、追加の直接引数となります。

現在は、順序集合集約は、ウィンドウ関数として使うことはできないので、移動集約モードをサポートする必要はありません。

部分集約(並列集約を含む)は現在のところ、順序集約ではサポートされません。また、部分集約はDISTINCTあるいはORDER BY句を含む集約の呼び出しでは決して使われることはありません。なぜなら、部分集約ではそれらを意味論的にサポートできないからです。

例

37.12を参照してください。

互換性

CREATE AGGREGATEはPostgreSQLの言語拡張です。標準SQLには、ユーザ定義の集約関数を使用する機能はありません。

関連項目

ALTER AGGREGATE, DROP AGGREGATE

CREATE CAST

CREATE CAST — 新しいキャストを定義する

概要

```
CREATE CAST (source_type AS target_type)
    WITH FUNCTION function_name [ (argument_type [, ...]) ]
    [ AS ASSIGNMENT | AS IMPLICIT ]

CREATE CAST (source_type AS target_type)
    WITHOUT FUNCTION
    [ AS ASSIGNMENT | AS IMPLICIT ]

CREATE CAST (source_type AS target_type)
    WITH INOUT
    [ AS ASSIGNMENT | AS IMPLICIT ]
```

説明

CREATE CASTを使用すると、新しいキャストを定義できます。キャストは、2つのデータ型間の変換処理方法を指定するものです。以下に例を示します。

```
SELECT CAST(42 AS float8);
```

この文を実行すると、事前に指定された関数（この場合float8(int4)）が呼び出され、整数定数42がfloat8型に変換されます（適切なキャストが定義されていない場合、変換処理は失敗します）。

2つのデータ型を*バイナリ強制互換*とすることができます。これは、関数をまったく呼び出さなくても、「自由に」変換を行うことができることを意味します。これには、対応する値は、同じ内部表現を使用している必要があります。例えば、データ型textとvarcharには、両方向でバイナリ互換性があります。バイナリ強制互換性は必ずしも対称関係ではありません。例えば、現在の実装ではxmlからtextへのキャストは自由に行うことができますが、逆方向では少なくとも構文検査を行う関数が必要です。（2つの型が両方向でバイナリ強制互換であることは、バイナリ互換性と呼ばれます。）

WITH INOUT構文を使用して*I/O変換キャスト*としてキャスト定義を行うことができます。I/O変換キャストは、元データ型の出力関数を呼び出し、その結果文字列を対象データ型の入力関数に渡すことで行われます。多くの一般的な場合では、この機能により変換用に別個のキャスト関数を作成する必要性がなくなります。I/O変換キャストは通常の関数を基にしたキャストと同様に動作します。ただ実装が異なるだけです。

デフォルトでは、キャストは明示的なキャスト要求があった場合のみ発生します。明示的なキャスト要求の構文は、CAST(x AS typename)、もしくは、x::typename式です。

キャストにAS ASSIGNMENTオプションを付けると、対象データ型の列に代入する際、暗黙的にそのキャストを発生させることができます。例えば、foo.f1がtext型の列であるとします。

```
INSERT INTO foo (f1) VALUES (42);
```

integer型をtext型に変換するキャストにAS ASSIGNMENTオプションが付けられていれば、上記のSQL文が実行できます。しかし、AS ASSIGNMENTオプションが付いていなければ、実行できません（一般的に、この種のキャストを*代入キャスト*と呼びます）。

キャストにAS IMPLICITオプションを付けると、代入の場合だけでなく、式の中にある場合でも、全てのコンテキストで暗黙的にそのキャストを呼び出すことができます。（一般的に、この種のキャストを*暗黙キャスト*と呼びます。）例えば次のような問い合わせを考えてみます。

```
SELECT 2 + 4.0;
```

パーサはまず定数にそれぞれintegerとnumericであると印を付けます。システムカタログには、integer + numericという演算子はありませんが、numeric + numericという演算子は存在します。したがって、integerからnumericへのキャストが利用可能であり、そのキャストにAS IMPLICITが付いていればこの問い合わせは成功します（実際このようになっています）。パーサは暗黙的なキャストを行い、問い合わせをあたかも次のように記載されたものとして解決します。

```
SELECT CAST ( 2 AS numeric ) + 4.0;
```

ここで、カタログはまたnumericからintegerへのキャストも提供しています。もしこのキャストにAS IMPLICITが付いていたら（実際は付いていません）、パーサは上のように解釈するか、それとも、numeric定数をintegerにキャストし、integer + integerという演算子を適用するかを選択しなければなりません。どちらがより良いかという知見がなければ、選択をあきらめ、問い合わせがあいまいであると宣告します。2つのキャストの内1つのみが暗黙的であるという事実が、パーサに、numericとintegerが混在する式をnumericとして扱うという適切な解決方法を知らせる方法です。これに関する組み込まれた知見は存在しません。

暗黙キャストは、多用しない方が賢明です。暗黙的キャストを使用し過ぎると、PostgreSQLがコマンドを思わぬ意味に解釈してしまう原因になります。また、複数の解釈が可能なため、コマンドをまったく解読できなくなってしまう可能性もあります。経験的には、2つのデータ型が同一の一般的なデータ型のカテゴリに属しており、変換によって情報が保持される場合のみ、暗黙キャストを呼び出し可能にするのが良い方法と思われます。例えば、int2型からint4型へのキャストは、暗黙キャストにするのが妥当ですが、float8型からint4型へのキャストは、おそらく代入キャストのみにすべきでしょう。text型からint4型への変換のような、カテゴリを越えるデータ型のキャストは、明示的にのみ使用するのが適切です。

注記

型の集合の中で複数の暗黙的なキャストを提供することが、有用性や標準との互換性上の理由により必要となることがあり、これにより、上で説明した通り防ぐことができないあいまいさが引き起こされます。パーサは、こうした状況でも望ましい動作の提供を補助できる*型カテゴリ*と*優先される型*に基づいた発見的手法を用意しています。詳細はCREATE TYPEを参照してください。

キャストを作成するためには、変換元または変換先（の内の一方）のデータ型を所有し、もう一方の型に対するUSAGE権限を持つ必要があります。また、バイナリ強制互換性を持つキャストを作成できるのは、スーパー

ユーザでなければなりません。(バイナリ強制互換性があるキャスト変換を誤って使用するとサーバがクラッシュしてしまう可能性が高いことから、この制限が付けられました)。

パラメータ

source_type

キャストする変換元のデータ型の名前です。

target_type

キャストする変換先のデータ型の名前です。

function_name[(argument_type [, ...])]

キャストを実行するために使用される関数です。関数名はスキーマ修飾することができます。スキーマ修飾されていない場合、関数はスキーマ検索パスから検索されます。関数の結果のデータ型は、キャストの変換先のデータ型と一致する必要があります。引数については後で説明します。引数リストが指定されない場合、関数名はスキーマ内で一意でなければなりません。

WITHOUT FUNCTION

変換元データ型から変換先データ型への間に、バイナリ強制互換性があることを示します。この場合、キャストを実行するのに関数は必要ありません。

WITH INOUT

キャストが、変換元データ型の出力関数を呼び出し、その結果の文字列を変換先データ型の入力関数に渡すことで行われる、I/O変換キャストであることを示します。

AS ASSIGNMENT

代入コンテキストで、暗黙的にキャストを呼び出せることを示します。

AS IMPLICIT

任意のコンテキストで、暗黙的にキャストを呼び出せることを示します。

キャストを実装する関数は1〜3個の引数を取ることができます。1番目の引数型はキャストの変換元データ型と同一、または、変換元データ型からのバイナリ強制互換を持つ型でなければなりません。2番目の引数(もしあれば)は、integer型でなければなりません。変換先の型に関連付けられた型修飾子を指定します。型修飾子がない場合は-1を指定します。3番目の引数(もしあれば)は、boolean型でなければなりません。キャストが明示的なキャストであればtrueを、それ以外であればfalseを指定します (奇妙な話ですが、標準SQLでは、明示的キャストと暗黙的キャストとの間で異なる振舞いを要求する場合があります。この引数はそのようなキャストを実装しなければならない関数用に提供されています。独自のデータ型をこの流儀に従うように設計することは勧められません)。

キャスト関数の戻り値は、キャストの対象型と同一またはバイナリ強制互換性を持たなければなりません。

通常、キャストにおける変換元データ型と変換先データ型は異なる必要があります。しかし、2つ以上の引数を持つ関数でキャストを実装した場合は、変換元と変換先とで同一のデータ型を持つキャストを宣言することができます。これは、システムカタログにおいて型固有の長さ強制関数を表現するために使用されていま

す。指定された関数は、型の値を強制的に2番目の引数で与えられた型修飾子の値にするために使用されます

キャストが変換元と変換先のデータ型が異なり、複数の引数を取る関数を持つ場合、あるデータ型から他のデータ型への変換と長さの強制を1つの操作にまとめたものをサポートします。引数を1つしか取らない場合は、型修飾子を使用して型を強制するために、データ型間の変換と修飾子の適用という2つのキャスト操作が必要となります。

ドメイン型へのキャスト、ドメイン型からのキャストは現在は効果がありません。ドメインへのキャスト、ドメインからのキャストは、基となる型と関連したキャストを使用します。

注釈

ユーザ定義のキャストを削除するにはDROP CASTを使用してください。

データ型を双方向に変更可能にするには、双方向のキャストを明示的に宣言する必要があることに注意してください。

ユーザ定義型と標準文字列型（text、varchar、char(n)）、および文字列カテゴリとして定義されたユーザ定義型との間のキャストを作成することは、通常必要ありません。PostgreSQLはこのために自動的なI/O変換キャストを提供します。この文字列への自動キャストは代入キャストとして扱われますが、文字列型からの入出力変換キャストは明示的なキャストのみです。この振舞いは独自のキャストを宣言して自動キャストを置き換えることで変更することができます。しかし、通常このようにするのは、この変換を標準の代入のみまたは明示的のみの設定よりもより呼び出しやすくしたい場合に限られます。他にも、型の入出力関数と異なる動作で変換したいという理由もあるかもしれません。しかし、これは非常に驚かされるものであり、そうすべきかどうか熟考すべきです。（組み込み型のごく一部は実際変換用に異なった振舞いをしますが、ほとんどは標準SQLの仕様のためのものです。）

必須ではありませんが、キャストを実装する関数には変換先のデータ型の名前を付けるという以前からの慣習に従っておくことを推奨します。多くのユーザはtypename(x)という関数スタイルの記法でデータ型のキャストを行っています。この記法は、キャストを実装している関数の呼び出しに他なりません。キャストとして特別に扱われるわけではないのです。ユーザが作成した変換関数の名前がこの慣習に従っていないと、他のユーザがとまどうことになります。PostgreSQLは引数として異なる型を取る同じ名前の関数をオーバーロードすることができるので、様々な型から特定の変換先型への変換関数の名前を全て変換先の型名にしても特に問題は発生しません。

> **注記**
>
> 実際のところ、前の段落は単純化しすぎたものです。関数呼び出し式が実際の関数と一致しない状態でキャスト要求として扱われる状況が2つ存在します。関数呼び出しname(x)が実際の関数に正確に一致せず、nameがデータ型の名前であり、pg_castがxの型からその型へのバイナリ強制互換のキャストを提供する場合、この呼び出しはバイナリ強制互換キャストとして処理されます。この例外は、実際の関数が存在しなくても、関数のような構文でバイナリ強制互換キャストを呼び出すことができるように作成されました。同様に、pg_castに項目がないが、文字列型との間のキャストが存在する場合、この呼び出しは入出力変換キャストとして処理されます。この例外により関数のような構文で入出力変換キャストができるようになります。

注記

この例外にも例外があります。複合型から文字列型へのI/O変換キャストでは関数構文を使用して呼び出すことができず、明示的なキャスト構文（CAST記法または::記法のいずれか）で記述しなければなりません この例外は、自動提供I/O変換キャストを導入した後、関数または列参照を意図した時に非常に簡単に間違って呼び出されることが判明したため追加されました。

例

関数int4(bigint)を使用したbigint型からint4型への代入キャストを作成します。

```
CREATE CAST (bigint AS int4) WITH FUNCTION int4(bigint) AS ASSIGNMENT;
```

（このキャストは、システムに既に定義されています。）

互換性

SQLではバイナリ強制互換性があるデータ型や実装関数の追加の引数について規定されていません。さらに、AS IMPLICITは、PostgreSQLの拡張です。これらの点以外では、CREATE CASTは標準SQLに準拠しています。

関連項目

CREATE FUNCTION, CREATE TYPE, DROP CAST

CREATE COLLATION

CREATE COLLATION — 新しい照合順序を定義する

概要

```
CREATE COLLATION [ IF NOT EXISTS ] name (
    [ LOCALE = locale, ]
    [ LC_COLLATE = lc_collate, ]
    [ LC_CTYPE = lc_ctype, ]
    [ PROVIDER = provider, ]
    [ DETERMINISTIC = boolean, ]
    [ VERSION = version ]
)
CREATE COLLATION [ IF NOT EXISTS ] name FROM existing_collation
```

説明

CREATE COLLATIONは指定したオペレーティングシステムのロケール設定を使用、または既存の照合順序をコピーすることで新しい照合順序を定義します。

照合順序を新しく作成するためには、格納先のスキーマにおけるCREATE権限が必要です。

パラメータ

IF NOT EXISTS

同じ名前の照合順序が既に存在する場合にエラーを発生させません。この場合、注意メッセージが発行されます。既存の照合順序が作られようとしていたものと類似したものかどうか、全く保証されないことに注意してください。

name

照合順序の名前です。照合順序の名前はスキーマ修飾することができます。スキーマ修飾されていない場合、照合順序は現在のスキーマ内に定義されます。照合順序の名前はそのスキーマ内で一意でなければなりません。(システムカタログでは異なる符号化方式に対して同じ名前の照合順序を含めることができます。しかしデータベース符号化方式が異なる場合には無視されます。)

locale

これは同時にLC_COLLATEおよびLC_CTYPEを設定する省略形です。これを指定した場合、これらのパラメータのどちらも指定することはできません。

lc_collate

LC_COLLATEロケールカテゴリに対して指定したオペレーティングシステムのロケールを使用します。

lc_ctype

LC_TYPEロケールカテゴリに対して指定したオペレーティングシステムのロケールを使用します。

provider

この照合順序に関連するロケールサービスで使用するプロバイダを指定します。取り得る値はicuとlibcです。libcがデフォルトです。利用可能な選択肢はオペレーティングシステムとビルド時のオプションに依存します。

DETERMINISTIC

照合順序が決定論的な比較を使うかどうかを指定します。デフォルトは真です。決定論的な比較では、たとえ論理的に比較すれば等しいとみなされるものであっても、バイト単位で等しくない文字列は等しくないとみなします。PostgreSQLはバイト単位の比較を使って分解します。決定論的でない比較では、例えば、照合順序で大文字小文字を区別しない、またはアクセントを区別しないようにできます。そのためには、適切なLC_COLLATEの設定を選んだ上で、ここで照合順序を非決定論的なものに設定することが必要です。

非決定論的な照合順序はICUプロバイダでのみサポートされています。

version

照合順序と一緒に保存するバージョン文字列を指定します。通常は省略すべきで、省略するとオペレーティングシステムが提供する照合順序の実際のバージョンから計算されます。このオプションはpg_upgradeが既存のインストレーションからバージョンをコピーする時に使われることを意図したものです。

照合順序のバージョン不適合を処理する方法については、ALTER COLLATIONも参照してください。

existing_collation

コピーする既存の照合順序の名前です。新しい照合順序は既存のものと同じ属性を持ちますが、独立したオブジェクトになります。

注意

CREATE COLLATIONはSHARE ROW EXCLUSIVEロックを使い、そのロックはpg_collationシステムカタログで衝突します。ですので、CREATE COLLATIONは一度に1つしか実行できません。

ユーザ定義の照合順序を削除するためにはDROP COLLATIONを使用してください。

照合順序の作成についての更なる情報については23.2.2.3を参照してください。

libc照合順序プロバイダを使う場合、ロケールは現在のデータベース符号化方式に適用可能でなければなりません。正確な規則についてはCREATE DATABASEを参照してください。

例

オペレーティングシステムのロケールfr_FR.utf8から照合順序を作成します（現在のデータベース符号化方式がUTF8であるとします）。

```
CREATE COLLATION french (locale = 'fr_FR.utf8');
```

ICUプロバイダを使い、ドイツの電話帳のソート順を使った照合順序を作成します。

```
CREATE COLLATION german_phonebook (provider = icu, locale = 'de-u-co-phonebk');
```

既存の照合順序から照合順序を作成します。

```
CREATE COLLATION german FROM "de_DE";
```

アプリケーションにおいてオペレーティングシステムに依存しない照合順序の名前を使用することができ、便利になるかもしれません。

互換性

標準SQLにはCREATE COLLATIONが存在しますが、既存の照合順序のコピーに限定されています。新しい照合順序を作成するための構文はPostgreSQLの拡張です。

関連項目

ALTER COLLATION, DROP COLLATION

CREATE CONVERSION

CREATE CONVERSION — 新しい符号化方式変換を定義する

概要

```
CREATE [ DEFAULT ] CONVERSION name
    FOR source_encoding TO dest_encoding FROM function_name
```

説明

CREATE CONVERSIONを使用すると、2つの文字セット符号化方式間の新しい変換を定義できます。

DEFAULTとして指定された変換は、クライアントとサーバの間での自動的な符号化方式の変換に使用できます。そのような使い方をサポートするためには、符号化方式Aから符号化方式Bへ、および、符号化方式Bから符号化方式Aへという2つの変換を定義する必要があります。

変換を作成するためには、その関数のEXECUTE権限、および、対象となるスキーマ上のCREATE権限を保持している必要があります。

パラメータ

DEFAULT

DEFAULT句により、この変換が、指定された変換元から対象となる符号化方式への変換のデフォルトであることが示されます。1つのスキーマ内でデフォルトとされる変換は、符号化方式の組み合わせ1組において1つだけです。

name

変換の名前です。変換名は、スキーマ修飾することができます。スキーマ修飾されていない場合、変換は現在のスキーマに定義されます。変換名は、スキーマ内で一意である必要があります。

source_encoding

変換元の符号化方式名です。

dest_encoding

変換先の符号化方式名です。

function_name

この関数は、変換の実行に使用されます。関数名は、スキーマ修飾することができます。スキーマ修飾されていない場合、関数はパスから検索されます。

関数は、下記のような形式で記述する必要があります。

```
conv_proc(

    integer,  -- 変換元符号化方式ID
    integer,  -- 変換先符号化方式ID
    cstring,  -- 変換元文字列（NULLで終わるC言語文字列）
    internal, -- 変換先文字列（NULLで終わるC言語文字列）
    integer   -- 変換元文字列長
) RETURNS void;
```

注釈

SQL_ASCII「符号化方式」を含む場合のサーバの振る舞いは組み込まれたものですので、変換元の符号化方式も対象となる符号化方式もSQL_ASCIIとすることはできません。

ユーザ定義の変換を削除するには、DROP CONVERSIONを使用します。

変換の作成に必要な権限は、今後のリリースで変更される可能性があります。

例

myfunc関数を使用して、UTF8からLATIN1への符号化方式の変換を作成します。

```
CREATE CONVERSION myconv FOR 'UTF8' TO 'LATIN1' FROM myfunc;
```

互換性

CREATE CONVERSIONは、PostgreSQLの拡張です。標準SQLにはCREATE CONVERSION文はありませんが、CREATE TRANSLATION文の目的および構文は非常に似たものです。

関連項目

ALTER CONVERSION, CREATE FUNCTION, DROP CONVERSION

CREATE DATABASE

CREATE DATABASE — 新しいデータベースを作成する

概要

```
CREATE DATABASE name
    [ [ WITH ] [ OWNER [=] user_name ]
           [ TEMPLATE [=] template ]
           [ ENCODING [=] encoding ]
           [ LOCALE [=] locale ]
           [ LC_COLLATE [=] lc_collate ]
           [ LC_CTYPE [=] lc_ctype ]
           [ TABLESPACE [=] tablespace_name ]
           [ ALLOW_CONNECTIONS [=] allowconn ]
           [ CONNECTION LIMIT [=] connlimit ]
           [ IS_TEMPLATE [=] istemplate ] ]
```

説明

CREATE DATABASEは新しいPostgreSQLデータベースを作成します。

データベースを作成するには、スーパーユーザ、もしくはCREATEDBという特別な権限を持つユーザである必要があります。CREATE ROLEを参照してください。

デフォルトでは、新しいデータベースは標準システムデータベースtemplate1を複製することによって作成されます。他のテンプレートを指定するには、TEMPLATE nameと記述します。特に、TEMPLATE template0と記述することで、そのバージョンのPostgreSQLによって定義済みの標準オブジェクトのみを持つ、(そこではユーザ定義オブジェクトは存在せず、システムオブジェクトは変更されていない)初期状態のデータベースを作ることができます。これは、template1に追加した独自オブジェクトをコピーしたくない場合に便利です。

パラメータ

name

作成するデータベースの名前です。

user_name

新しいデータベースを所有するユーザのロール名です。デフォルト設定(つまり、コマンドを実行したユーザ)を使用する場合はDEFAULTと指定します。他のロールによって所有されるデータベースを作成するためには、そのロールの直接的または間接的なメンバであるか、スーパーユーザでなければなりません。

template

新しいデータベースの作成元となるテンプレートの名前です。デフォルトテンプレート(template1)を使う場合は、DEFAULTと指定します。

encoding

新しいデータベースで使われる文字セット符号化方式です。文字列定数(例えば'SQL_ASCII')、整数の符号化方式番号、DEFAULTのいずれかを指定します。DEFAULTとすると、デフォルトの符号化方式(すなわちテンプレートデータベースの符号化方式)を使います。PostgreSQLサーバでサポートされる文字セットについては23.3.1で説明します。この他の制限については後述します。

locale

これはLC_COLLATEとLC_CTYPEを一度に設定する手っ取り早い方法です。これを指定した場合、上記のパラメータのどちらか1つを指定することはできません。

> **ヒント**
>
> その他のロケール設定lc_messages、lc_monetary、lc_numeric、lc_timeはデータベース毎には固定されず、このコマンドでは設定されません。特定のデータベースのデフォルトにしたい場合には、ALTER DATABASE ... SETが使えます。

lc_collate

新しいデータベースで使用する照合順(LC_COLLATE)です。これは、たとえばORDER BYを持つ問い合わせなどにおいて文字列に適用されるソート順やテキスト型の列に対するインデックスで使用される順序に影響します。

lc_ctype

新しいデータベースで使用する文字のクラス(LC_CTYPE)です。これは、たとえば小文字、大文字、数字といった文字の分類に影響します。デフォルトではテンプレートデータベースの文字クラスを使用します。さらなる制限に関しては後で説明します。

tablespace_name

新しいデータベースに関連付けされるデフォルトのテーブル空間名です。テンプレートデータベースのテーブル空間を使用する場合は、DEFAULTと指定します。このテーブル空間が、このデータベースで作成されるオブジェクトのデフォルトのテーブル空間となります。詳細はCREATE TABLESPACEを参照してください。

allowconn

falseの場合、誰もこのデータベースに接続できません。デフォルトはtrueで、接続が可能です(GRANT/REVOKE CONNECTなど他の仕掛けで制限されている場合を除きます)。

connlimit

このデータベースで確立できる同時接続数です。-1(デフォルト)は無制限を意味します。

istemplate

trueの場合、CREATEDB権限があれば、どのユーザでもこのデータベースを複製できます。false(デフォルト)の場合、スーパーユーザまたはデータベースの所有者だけが複製できます。

オプションのパラメータは、任意の順番で記述できます。上記の順番で記述しなくても構いません。

注釈

CREATE DATABASEはトランザクションブロックの内側では実行できません。

ほとんどの場合、「could not initialize database directory」という行が含まれるエラーは、データディレクトリの権限不足、ディスク容量不足などファイルシステムについての問題に関連するものです。

データベースを削除するにはDROP DATABASEを使用してください。

createdbプログラムは利便性のために提供される、このコマンドのラッパプログラムです。

データベースレベルの設定パラメータ(ALTER DATABASEによって設定されるもの)とデータベースレベルの権限(GRANTによって設定されるもの)はテンプレートデータベースからコピーされません。

template1以外のデータベースの名前をテンプレートに指定してデータベースをコピーすることは可能ですが、これは(まだ)一般的に使用する「COPY DATABASE」機能として意図されているわけではありません。主な制限は、コピー中に他のセッションからテンプレートデータベースへの接続ができないことです。CREATE DATABASEは、開始した時に他の接続があると失敗します。テンプレートデータベースへの新規接続はCREATE DATABASEが完了するまでできません。詳細は22.3を参照してください。

新しいデータベース用に指定される文字セット符号化方式は選択されたロケール設定(LC_COLLATEおよびLC_CTYPE)と互換性がなければなりません。ロケールがC(や同等のPOSIX)であれば、すべての符号化方式が許されますが、他のロケール設定では適切に動作する符号化方式は1つしかありません。(しかしWindowsではUTF-8符号化方式をすべてのロケールで使用することができます。) CREATE DATABASEでは、ロケール設定に関係なくスーパーユーザがSQL_ASCII符号化方式を指定することを許していますが、こうした選択は廃止予定であり、データベース内にロケールと互換性がない符号化方式でデータが格納された場合、文字列関数の誤動作を多く引き起こします。

符号化方式とロケール設定はテンプレートデータベースのこれらの設定と一致しなければなりません。ただしtemplate0がテンプレートとして使用される場合は例外です。他のデータベースには指定された符号化方式と一致しないデータを含む可能性やLC_COLLATEおよびLC_CTYPEがソート順序に影響するようなインデックスを含む可能性があることがこの理由です。こうしたデータをコピーしたものは、新しい設定から見ると破損したデータベースとなります。しかしtemplate0には影響を受けるデータやインデックスが含まれていないことが分かっています。

CONNECTION LIMITは厳密な制限ではありません。データベース向けの接続「スロット」が1つ残っていた時に同時に2つの新しいセッション開始要求があった場合、両方とも失敗する可能性があります。また、この制限はスーパーユーザおよびバックグラウンドのワーカプロセスには強制されません。

例

新しいデータベースを作成します。

```
CREATE DATABASE lusiadas;
```

ユーザsalesappを所有者、salesspaceをデフォルトのテーブル空間としてデータベースsalesを作成します。

```
CREATE DATABASE sales OWNER salesapp TABLESPACE salesspace;
```

別のロケールでデータベースmusicを作成します。

```
CREATE DATABASE music
    LOCALE 'sv_SE.utf8'
    TEMPLATE template0;
```

この例において、指定するロケールがtemplate1のロケールと異なる場合、TEMPLATE template0の句が必須となります。（それらが同じ場合、ロケールを明示的に指定することは必要ありません。）

別のロケールおよび別の文字セット符号化方式でデータベースmusic2を作成します。

```
CREATE DATABASE music2
    LOCALE 'sv_SE.iso885915'
    ENCODING LATIN9
    TEMPLATE template0;
```

指定するロケールと符号化方式の設定は対応するものでなければならず、そうでなければエラーが報告されます。

ロケール名はOSに固有のものであるため、上記のコマンドはすべての環境で同じように動作するとは限らないことに注意してください。

互換性

標準SQLにはCREATE DATABASE文はありません。データベースはカタログに相当するもので、その作成は実装依存です。

関連項目

ALTER DATABASE, DROP DATABASE

CREATE DOMAIN

CREATE DOMAIN — 新しいドメインを定義する

概要

```
CREATE DOMAIN name [ AS ] data_type
    [ COLLATE collation ]
    [ DEFAULT expression ]
    [ constraint [ ... ] ]


ここでconstraintは、
      以下の通りです。

[ CONSTRAINT constraint_name ]
{ NOT NULL | NULL | CHECK (expression) }
```

説明

CREATE DOMAINは新しいドメインを作成します。ドメインとは本質的には、特別な制約（使用可能な値集合に対する制限）を持ったデータ型です。ドメインを定義したユーザが、その所有者となります。

スキーマ名が付けられている場合（例えば、CREATE DOMAIN myschema.mydomain ...）、ドメインは指定されたスキーマに作成されます。スキーマ名が付けられていなければ、そのドメインは現在のスキーマに作成されます。ドメイン名は、そのスキーマ内に存在するデータ型およびドメインの間で、一意である必要があります。

ドメインを使用すると、共通な制約を1箇所に抽象化でき、メンテナンスに便利です。たとえば、E-mailアドレスを格納する列が複数のテーブルで使用されていて、アドレス構文の検証のためすべてが同一のCHECK制約を必要としているような場合です。このような場合、各テーブルに個別に制約を設定するよりも、ドメインを定義してください。

ドメインを作成するためには、基となる型に対するUSAGEを持たなければなりません。

パラメータ

name

作成するドメインの名前です（スキーマ修飾名でも可）。

data_type

ドメインの基となるデータ型です。配列指定子を含めることができます。

`collation`

ドメインの照合順(省略可能)です。照合順序の指定がなければ基となるデータ型のデフォルトの照合順序が使用されます。COLLATEが指定される場合、基となる型は照合順序が設定可能な型でなければなりません。

`DEFAULT expression`

DEFAULT句は、ドメインデータ型の列のデフォルト値を指定します。任意の無変数式を値とすることができます(ただし、副問い合わせは許可されません)。デフォルト式のデータ型は、そのドメインのデータ型と一致する必要があります。デフォルト値が指定されない場合、デフォルト値はNULL値となります。

デフォルト式は、挿入操作において該当する列に値が指定されなかった場合に使用されます。特定の列に対してデフォルト値が定義されている場合、それはドメインに関連するデフォルト値より優先します。反対に、基となるデータ型に関連するデフォルト値より、ドメインのデフォルト値が優先します。

`CONSTRAINT constraint_name`

制約の名前(省略可能)です。指定されなければ、システムが名前を生成します。

`NOT NULL`

このドメインの値としてNULLの使用を禁止します(ただし、以下の注釈を参照してください)。

`NULL`

ドメインの値としてNULLの使用を許可します。こちらがデフォルトです。

この句は非標準的なSQLデータベースとの互換性を持つためだけに用意されています。新しいアプリケーションでこの句を使用するのはお勧めできません。

`CHECK (expression)`

CHECK句は、ドメインの値が満たさなければならない整合性制約や検査を指定します。各制約は、Boolean型の結果を生成する式である必要があります。検査される値を参照するには、VALUEというキーワードを使用すべきです。TRUEまたはUNKNOWNとして評価される式は成功します。式の結果がFALSEになった場合、エラーが報告され、値はドメイン型に変換することができません。

現時点では、CHECK式に副問い合わせを含めたり、VALUE以外の変数を参照したりすることはできません。

ドメインに複数のCHECK制約がある場合、それらは名前のアルファベット順に評価されます。(PostgreSQLの9.5より前のバージョンでは、複数のCHECK制約がある場合について、特定の実行順序がありませんでした。)

注釈

ドメイン制約、特にNOT NULLは、値がドメイン型に変換されるときに検査されます。通常はドメイン型である列が、NOT NULLの制約にも関わらずNULLとして読み出される場合もあり得ます。例えば、外部結合の問い合わせにおいて、ドメインの列が外部結合のNULLになる側にあるときに、これが起こり得ます。より微妙な例は以下です。

```
INSERT INTO tab (domcol) VALUES ((SELECT domcol FROM tab WHERE false));
```

空でスカラーの副問い合わせにより、ドメイン型であると見なされるNULL値が生成されます。そのため、制約についてこれ以上の検証は行われず、挿入は成功します。

SQLではNULL値はすべてのデータ型で有効な値であると想定されているため、このような問題を回避するのは非常に難しいことです。したがって、最善の方法は、NOT NULL制約をドメイン型に直接適用するのではなく、NULL値が許されるようにドメインの制約を設計し、その上で、列のNOT NULL制約を、必要に応じて、ドメイン型の列に適用することです。

PostgreSQLはCHECK制約の条件はimmutableである、すなわち同じ入力値に対しては必ず同じ結果を与えると仮定します。この仮定は、値が初めてドメイン型に変換された時にのみCHECK制約を確認し、それ以外では確認しないことを正当化するものです。(これは、5.4.1で述べているテーブルのCHECK制約の扱いと本質的に同じです。)

この仮定を破るよくある例は、CHECK式内でユーザ定義関数を参照しており、後でその関数の振舞いを変更することです。PostgreSQLはそれを拒否しませんが、そのドメイン型の格納された値でCHECK制約に今や違反するものがあることには気付かないでしょう。これは、その後のデータベースのダンプとリロードが失敗する原因になるかもしれません。そのような変更を扱うお勧めの方法は、(ALTER DOMAINを使って)制約を削除し、関数の定義を調整し、その制約を再び追加することです。それにより格納されたデータに対して再確認が行われます。

例

この例では、データ型us_postal_codeを作成し、その型をテーブル定義の中で使用します。データが有効なUS郵便番号であるかどうかを検証するために正規表現検査が使用されます。

```
CREATE DOMAIN us_postal_code AS TEXT
CHECK(
   VALUE ~ '^\d{5}$'
OR VALUE ~ '^\d{5}-\d{4}$'
);

CREATE TABLE us_snail_addy (
  address_id SERIAL PRIMARY KEY,
  street1 TEXT NOT NULL,
  street2 TEXT,
  street3 TEXT,
  city TEXT NOT NULL,
  postal us_postal_code NOT NULL
);
```

互換性

CREATE DOMAINコマンドは標準SQLに準拠しています。

関連項目

ALTER DOMAIN, DROP DOMAIN

CREATE EVENT TRIGGER

CREATE EVENT TRIGGER — 新しいイベントトリガを定義する

概要

```
CREATE EVENT TRIGGER name
    ON event
    [ WHEN filter_variable IN (filter_value [, ... ]) [ AND ... ] ]
    EXECUTE { FUNCTION | PROCEDURE } function_name()
```

説明

CREATE EVENT TRIGGERは新しいイベントトリガを作成します。指定されたイベントが発生し、トリガに関連するWHEN条件がもしあればそれを満たす場合、トリガ関数が実行されます。イベントトリガの一般的な紹介については、第39章を参照してください。イベントトリガを作成したユーザがその所有者となります。

パラメータ

name

新しいトリガに付ける名前です。この名前はデータベース内で一意でなければなりません。

event

指定関数を呼び出すきっかけとなるイベントの名前です。イベント名の詳細については39.1を参照してください。

filter_variable

イベントをフィルタするために使用される変数の名前です。これにより、サポートしている状況の一部に対してのみにトリガの発行を制限することができます。現在filter_variableでサポートされているものはTAGのみです。

filter_value

どこでトリガを発行すべきかについて、関連するfilter_variable用の値のリストです。TAGの場合、これはコマンドタグ（例えば'DROP FUNCTION'）のリストを意味します。

function_name

引数を取らずevent_trigger型を返すと宣言された、ユーザが提供する関数です。

CREATE EVENT TRIGGERの構文では、キーワードFUNCTIONとPROCEDUREは等価ですが、参照されている関数はどちらの場合でも関数でなければならず、プロシージャであってはなりません。ここでキーワードPROCEDUREを使うことは、歴史的なものであり廃止予定です。

注釈

スーパーユーザのみがイベントトリガを作成することができます。

シングルユーザモード（postgres参照）ではイベントトリガは無効になります。エラーのあるイベントトリガが原因でデータベースの動作がおかしくなり、トリガを削除することもできない状態になった場合は、シングルユーザモードで再起動してください。削除できるようになります。

例

すべてのDDLコマンドの実行を禁じます。

```
CREATE OR REPLACE FUNCTION abort_any_command()
  RETURNS event_trigger
 LANGUAGE plpgsql
  AS $$
BEGIN
  RAISE EXCEPTION 'command % is disabled', tg_tag;
END;
$$;

CREATE EVENT TRIGGER abort_ddl ON ddl_command_start
   EXECUTE FUNCTION abort_any_command();
```

互換性

標準SQLにはCREATE EVENT TRIGGER文はありません。

関連項目

ALTER EVENT TRIGGER, DROP EVENT TRIGGER, CREATE FUNCTION

CREATE EXTENSION

CREATE EXTENSION — 拡張をインストールする

概要

```
CREATE EXTENSION [ IF NOT EXISTS ] extension_name
    [ WITH ] [ SCHEMA schema_name ]
             [ VERSION version ]
             [ CASCADE ]
```

説明

CREATE EXTENSIONは現在のデータベース内に新しい拡張を読み込みます。読み込み済みの拡張と同じ名前の拡張を読み込むことはできません。

拡張の読み込みは基本的に拡張のスクリプトファイルを実行することと同じです。スクリプトは通常、関数、 データ型、演算子、インデックスサポートメソッドなどのSQLオブジェクトを新しく作成するものです。CREATE EXTENSIONはさらに作成したすべてのオブジェクト識別子を記録して、DROP EXTENSIONが発行された時に削除できるようにします。

CREATE EXTENSIONを実行するユーザは、後で実施される権限検査のためにその拡張の所有者となります。また通常このユーザは拡張のスクリプトにより作成されたすべてのオブジェクトの所有者となります。

拡張の読み込みでは、通常その要素オブジェクトを作成するために必要となるいくつかの権限が必要です。多くの拡張では、これはスーパーユーザの権限が必要であることを意味します。しかしながら、制御ファイルで*trusted*と印付けされた拡張は、現在のデータベースに対してCREATE権限を持つユーザであれば誰でもインストールできます。この場合拡張オブジェクト自身は呼び出したユーザが所有しますが、そこに含まれるオブジェクトは(拡張スクリプトが明示的に呼び出したユーザに対して割り当てない限り)ブートストラップスーパーユーザが所有します。この設定は呼び出したユーザに拡張を削除する権限を与えますが、その中の個々のオブジェクトを修正する権限は与えません。

パラメータ

IF NOT EXISTS

同じ名前の拡張がすでに存在していてもエラーにしません。この場合注意が発せられます。既存の拡張が、現在利用可能なスクリプトファイルより作成されるものと何かしら似たものであることは保証されません。

extension_name

インストールする拡張の名前です。PostgreSQLはファイルSHAREDIR/extension/extension_name.controlから詳細を読み取り、拡張を作成します。

schema_name

拡張の内容を再配置させることができる場合に、拡張のオブジェクトをインストールするスキーマの名前です。指定されたスキーマは事前に存在していなければなりません。指定がなく、拡張の制御ファイルでもスキーマを指定していない場合、現在のデフォルトのオブジェクト生成用スキーマが使用されます。

拡張がその制御ファイルでschemaパラメータを指定している場合、そのスキーマをSCHEMA句で上書きすることはできません。SCHEMA句が指定され、それが拡張のschemaパラメータと相容れない場合、通常はエラーが発生します。しかし、CASCADE句も指定されている場合は、schema_nameが相容れなければ、それを無視します。必要なすべての拡張のインストールにおいて、それぞれの制御ファイルでschemaが指定されていなければ、指定されたschema_nameが使用されます。

拡張自体が任意のスキーマの中にあるとみなされていないことを思い出してください。拡張は修飾がない名前を持ちますので、データベース全体で一意でなければなりません。しかし拡張に属するオブジェクトはスキーマの中に置くことができます。

version

インストールする拡張のバージョンです。これは識別子あるいは文字列リテラルのいずれかとして記述できます。デフォルトのバージョンは拡張の制御ファイル内で規定したものになります。

CASCADE

この拡張が依存し、まだインストールされていないすべての拡張を自動的にインストールします。それらが依存するものも同様に再帰的にインストールされます。SCHEMA句が指定されている場合は、これによってインストールされるすべての拡張に適用されます。この文の他のオプションは自動的にインストールされる拡張には適用されません。特に、そのデフォルトバージョンは常に選択されます。

注釈

拡張をデータベースにロードするためにCREATE EXTENSIONを使用できるようになる前に、拡張のサポートファイルがインストールされていなければなりません。PostgreSQLが提供する拡張のインストールに関する情報は、追加で提供されるモジュールで説明します。

現在ロード可能な拡張はpg_available_extensionsまたはpg_available_extension_versionsシステムビューで識別できます。

注意

スーパーユーザとして拡張をインストールするには、拡張の作者が安全な方法で拡張のインストールスクリプトを書いたと信用することが必要です。悪意のあるユーザが、不注意に書かれた拡張スクリプトの以降の実行を危険に晒すトロイの木馬オブジェクトを作り、そのユーザがスーパーユーザの権限を得るようにするのは、それほど難しいことではありません。しかしながら、トロイの木馬オブジェクトはスクリプト実行時にsearch_pathにある場合にのみ危険です。これは拡張のインストール対象スキーマや依存する拡張のスキーマにあるということを意味します。そのため、スクリプトが注意深く検査されていない拡張を扱う優れた経験則は、信頼できないユーザにCREATE権限を許可していないし、今後も許可することのないスキーマにのみ、その拡張をインストールすることです。その拡張が依存する拡張についても同様です。

> PostgreSQLが提供する拡張は、他の拡張に依存する少数のものを除いて、この種のインストール時の攻撃に対して安全であると信じられています。各拡張の文書で述べているように、拡張は安全なスキーマにインストールするか、依存する拡張と同じスキーマにインストールするか、あるいはその両方であるスキーマにインストールすべきです。

新しい拡張の作成に関しては37.17を参照してください。

例

そのオブジェクトをaddonsスキーマに配置して、現在のデータベースにhstore拡張をインストールします。

```
CREATE EXTENSION hstore SCHEMA addons;
```

以下は同様のことを行なう別の方法です。

```
SET search_path = addons;
CREATE EXTENSION hstore;
```

既存のhstoreオブジェクトをインストールしたスキーマを注意して指定してください。

互換性

CREATE EXTENSIONはPostgreSQLの拡張です。

関連項目

ALTER EXTENSION, DROP EXTENSION

CREATE FOREIGN DATA WRAPPER

CREATE FOREIGN DATA WRAPPER — 新しい外部データラッパを定義する

概要

```
CREATE FOREIGN DATA WRAPPER name
    [ HANDLER handler_function | NO HANDLER ]
    [ VALIDATOR validator_function | NO VALIDATOR ]
    [ OPTIONS ( option 'value' [, ... ] ) ]
```

説明

CREATE FOREIGN DATA WRAPPERは新しい外部データラッパを作成します。外部データラッパを定義したユーザがその所有者となります。

外部データラッパの名前はデータベース内で一意でなければなりません。

スーパーユーザのみが外部データラッパを作成することができます。

パラメータ

name

作成する外部データラッパの名前です。

HANDLER handler_function

handler_functionは、事前に登録された、外部テーブル向けの関数実行を受け付けるために呼び出される関数の名前です。ハンドラ関数は引数を取らず、fdw_handler型を返すものでなければなりません。

ハンドラ関数を持たない外部データラッパを作成することもできますが、こうしたラッパを使用する外部テーブルは宣言することができるだけでアクセスできません。

VALIDATOR validator_function

validator_functionは、外部データラッパへ与える一般的なオプションと、その外部データラッパを使用する外部サーバ、ユーザマップおよび外部テーブルへ与えるオプションを検査するために呼び出される、前もって登録された関数の名前です。検証関数がない、またはNO VALIDATORが指定された場合、オプションは作成時に検査されません。(実装に依存しますが、実行時外部データラッパは無効なオプション指定を無視することも拒絶することもできます。) 検証関数は2つの引数を取らなければなりません。1つはtext[]型で、システムカタログ内に格納されたオプションの配列を含みます。もう1つはoid型で、オプションを含むシステムカタログのOIDです。戻り値の型は無視されます。関数はereport()関数を使用して無効なオプションを報告しなければなりません。

`OPTIONS ( option 'value' [, ... ] )`

この句は新しい外部データラッパ用のオプションを指定します。使用できるオプション名と値は外部データラッパごとに固有であり、外部データラッパの検証関数を使用して検証されます。オプション名は一意でなければなりません。

注釈

PostgreSQLの外部データ機能はまだ活発な開発がなされています。問い合わせの最適化がまだ開発が進んでいません（そしてほとんどがラッパに任せられています）。したがって将来の性能向上の余地が大きくあります。

例

無意味な外部データラッパdummyを作成します。

```
CREATE FOREIGN DATA WRAPPER dummy;
```

file_fdw_handlerハンドラ関数を持つ外部データラッパfileを作成します。

```
CREATE FOREIGN DATA WRAPPER file HANDLER file_fdw_handler;
```

いくつかオプションを付けた外部データラッパmywrapperを作成します。

```
CREATE FOREIGN DATA WRAPPER mywrapper
    OPTIONS (debug 'true');
```

互換性

CREATE FOREIGN DATA WRAPPERはISO/IEC 9075-9 (SQL/MED)に従います。ただし、HANDLER句とVALIDATOR句は拡張であり、PostgreSQLでは標準のLIBRARY句とLANGUAGE句は実装されていません。

しかし、SQL/MED機能は全体としてまだ従っていないことに注意してください。

関連項目

ALTER FOREIGN DATA WRAPPER, DROP FOREIGN DATA WRAPPER, CREATE SERVER, CREATE USER MAPPING, CREATE FOREIGN TABLE

CREATE FOREIGN TABLE

CREATE FOREIGN TABLE — 新しい外部テーブルを定義する

概要

```
CREATE FOREIGN TABLE [ IF NOT EXISTS ] table_name ( [
  { column_name data_type [ OPTIONS ( option 'value' [, ... ] ) ] [ COLLATE collation ]
 [ column_constraint [ ... ] ]
    | table_constraint }
    [, ... ]
] )
[ INHERITS ( parent_table [, ... ] ) ]
  SERVER server_name
[ OPTIONS ( option 'value' [, ... ] ) ]

CREATE FOREIGN TABLE [ IF NOT EXISTS ] table_name
  PARTITION OF parent_table [ (
  { column_name [ WITH OPTIONS ] [ column_constraint [ ... ] ]
    | table_constraint }
    [, ... ]
) ] partition_bound_spec
  SERVER server_name
[ OPTIONS ( option 'value' [, ... ] ) ]



ここでcolumn_constraintは以下の通りです。

[ CONSTRAINT constraint_name ]
{ NOT NULL |
  NULL |
  CHECK ( expression ) [ NO INHERIT ] |
  DEFAULT default_expr |
  GENERATED ALWAYS AS ( generation_expr ) STORED }



またtable_constraintは以下の通りです。

[ CONSTRAINT constraint_name ]
CHECK ( expression ) [ NO INHERIT ]
```

説明

CREATE FOREIGN TABLEは現在のデータベース内に新しい外部テーブルを作成します。このテーブルはコマンドを発行したユーザにより所有されます。

スキーマ名が指定された場合（例えばCREATE FOREIGN TABLE myschema.mytable ...）、テーブルは指定されたスキーマ内に作成されます。そうでなければ現在のスキーマ内に作成されます。外部テーブルの名前は同じスキーマ内にある他の外部テーブル、テーブル、シーケンス、インデックス、ビュー、マテリアライズドビューと異なるものでなければなりません。

CREATE FOREIGN TABLEはまた自動的に外部テーブルの1行に対応する複合型を表すデータ型を作成します。したがって外部テーブルは同じスキーマ内の既存のデータ型の名前と同じものを持つことができません。

PARTITION OF句が指定された場合、テーブルはparent_tableの指定された境界のパーティションとして作られます。

外部テーブルを作成するためには、外部サーバに対するUSAGE権限とテーブルで使用される列の型すべてに対するUSAGE権限を持たなければなりません。

パラメータ

IF NOT EXISTS

同じ名前のリレーションがすでに存在していてもエラーとしません。この場合注意が発せられます。既存のリレーションが作成しようとしたものと何かしら似たものであることは保証されません。

table_name

作成するテーブルの名前です（スキーマ修飾名でも可）。

column_name

新しいテーブルで作成される列の名前です。

data_type

列のデータ型です。これには、配列指定子を含めることができます。PostgreSQLでサポートされるデータ型の情報に関する詳細は第8章を参照してください。

COLLATE collation

COLLATE句は列（照合可能なデータ型でなければなりません）の照合順序を指定します。指定されなければ、列のデータ型のデフォルトの照合順序が使用されます。

INHERITS (parent_table [, ...])

オプションでINHERITS句を使い、新しい外部テーブルが自動的にすべての列を継承するテーブルのリストを指定できます。親テーブルには通常のテーブルまたは外部テーブルが指定できます。詳しくはCREATE TABLEの類似の構文を参照してください。

PARTITION OF parent_table FOR VALUES partition_bound_spec

この形式は、与えられた親テーブルのパーティションとして指定されたパーティション境界値を持つ外部テーブルを作成するために使うことができます。より詳細についてはCREATE TABLEの類似の形式を参照してください。現在のところに親テーブルにUNIQUEインデックスがある場合、親テーブルのパー

ティションとして外部テーブルを作成することは認められていないことに注意してください。(ALTER TABLE ATTACH PARTITIONも参照してください。)

CONSTRAINT constraint_name

列制約またはテーブル制約の名前(省略可)です。制約に違反した時、エラーメッセージ内に制約名が表示されるので、col must be positiveのような制約名を使って、クライアントアプリケーションに役立つ制約情報を通知することができます。(空白文字を含む制約名を指定するには二重引用符を使う必要があります。) 制約名が指定されなければ、システムが名前を生成します。

NOT NULL

その列がNULL値を持てないことを指定します。

NULL

その列がNULL値を持てることを指定します。これがデフォルトです。

この句は非標準的なSQLデータベースとの互換性のためだけに提供されています。新しいアプリケーションでこれを使用するのはお勧めしません。

CHECK (expression) [NO INHERIT]

CHECK句では、外部テーブルの各行が満たすと期待されるブーリアン結果を生成する式を指定します。つまり、式は外部テーブルのすべての行に対して、TRUEまたはUNKNOWNを生成し、決してFALSEにはなりません。列制約として指定したチェック制約はその列の値だけを参照しますが、テーブル制約として使われる式は複数の列を参照することができます。

現在のところ、CHECKの式は副問い合わせを含むことや、現在の行の列以外の変数を参照することはできません。システム列tableoidを参照することはできますが、それ以外のシステム列を参照することはできません。

NO INHERITと印を付けられた制約は、子テーブルに継承されません。

DEFAULT default_expr

DEFAULT句は、列定義の中に現れる、列に対するデフォルトデータ値を割り当てます。値は変数がない任意の式(副問い合わせおよび、現在のテーブル内の他の列へのクロス参照は許されません)です。デフォルト式のデータ型は列のデータ型とマッチしなければなりません。

デフォルト式は、列に対する値指定がないすべての挿入操作で使用されます。列に対するデフォルトがない場合、デフォルトはNULLです。

GENERATED ALWAYS AS (generation_expr) STORED

この句は、列を生成列として作成します。その列に書き込むことはできず、読み出された場合に指定された式の結果が返されます。

キーワードSTOREDは、列が書き込み時に計算されることを指定するのに必要です。(計算された値は保存用に外部データラッパへと送られ、読み込み時には返されなければなりません。)

生成式はテーブル内の他の列を参照できますが、他の生成列は参照できません。使われている関数や演算子はimmutableでなければなりません。他のテーブルへの参照はできません。

`server_name`

外部テーブル用に使用される既存の外部サーバの名前です。外部サーバの詳細についてはCREATE SERVERを参照してください。

`OPTIONS ( option 'value' [, ...] )`

新しい外部テーブルまたはその列の1つに関連するオプションです。設定可能なオプションの名前と値は外部データラッパそれぞれに固有なものであり、外部データラッパの検証関数を用いて検証されます。重複するオプション名は許されません(しかしテーブルオプションと列オプションでは同じ名前を持たせることはできます)。

注釈

外部テーブル上の制約(CHECK句やNOT NULL句など)はPostgreSQLのコアシステムによって強制されませんし、ほとんどの外部データラッパもそれを強制しようとはしません。つまり、制約は単にそれが成り立つと仮定されるものです。制約は外部テーブルの機能を使って行を挿入あるいは更新するときにのみ適用され、リモートサーバ上で直接更新するなど、他の手段による行の更新には適用されませんから、それを強制することにはあまり意味はありません。その代わりに、外部テーブルに指定する制約は、リモートサーバによって強制される制約を表現するものであるべきです。

一部の特別な目的の外部データラッパは、それがアクセス対象のデータにアクセスするための唯一の機構であり、またその場合、外部データラッパそれ自体にとって、制約の強制を実行することが適切なことがあります。ただし、ラッパのドキュメントにそのように書いてあるのでなければ、それを仮定しない方が良いでしょう。

PostgreSQLでは外部テーブルの制約を強制しませんが、問い合わせの最適化という目的のため、制約が正しいということを仮定します。外部テーブルで、宣言された制約を満たさない行が可視の状態で存在する場合、そのテーブルに対する問い合わせは誤った結果をもたらすかもしれません。制約の定義が現実に即したものであることを保証するのは、ユーザの責任です。

似たような配慮は生成列に適用されます。保存生成列は、ローカルのPostgreSQLサーバ上で挿入されたり更新されたりした時に計算され、外部データ保存領域へ書き出すために外部データラッパへと渡されますが、外部テーブルへの問い合わせが生成式と矛盾しない保存生成列の値を返すことは強制されていません。ここでも、問い合わせの結果が正しくないということになる可能性があります。

(外部データラッパがタプルルーティングをサポートしていれば)行はローカルパーティションから外部テーブルパーティションへ移動できますが、外部テーブルパーティションから別のパーティションには移動できません。

例

サーバfilm_serverを通してアクセスされる、外部テーブルfilmsを作成します。

```
CREATE FOREIGN TABLE films (
    code        char(5) NOT NULL,
    title       varchar(40) NOT NULL,
```

```
    did         integer NOT NULL,
    date_prod   date,
    kind        varchar(10),
    len         interval hour to minute
)
SERVER film_server;
```

範囲パーティションテーブルmeasurementのパーティションとして、サーバserver_07を通してアクセスされる外部テーブルmeasurement_y2016m07を作成します。

```
CREATE FOREIGN TABLE measurement_y2016m07
    PARTITION OF measurement FOR VALUES FROM ('2016-07-01') TO ('2016-08-01')
    SERVER server_07;
```

互換性

CREATE FOREIGN TABLEはおおよそ標準SQLに準拠します。しかしCREATE TABLEとほとんど同様、NULL制約とゼロ列の外部テーブルが許されます。列のデフォルト値を指定する機能もPostgreSQLの拡張です。PostgreSQLが定義する形式のテーブルの継承は標準とは異なります。

関連項目

ALTER FOREIGN TABLE, DROP FOREIGN TABLE, CREATE TABLE, CREATE SERVER, IMPORT FOREIGN SCHEMA

CREATE FUNCTION

CREATE FUNCTION — 新しい関数を定義する

概要

```
CREATE [ OR REPLACE ] FUNCTION
    name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ] [, ...] ] )
    [ RETURNS rettype
      | RETURNS TABLE ( column_name column_type [, ...] ) ]
  { LANGUAGE lang_name
    | TRANSFORM { FOR TYPE type_name } [, ... ]
    | WINDOW
    | IMMUTABLE | STABLE | VOLATILE | [ NOT ] LEAKPROOF
    | CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
    | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
    | PARALLEL { UNSAFE | RESTRICTED | SAFE }
    | COST execution_cost
    | ROWS result_rows
    | SUPPORT support_function
    | SET configuration_parameter { TO value | = value | FROM CURRENT }
    | AS 'definition'
    | AS 'obj_file', 'link_symbol'
  } ...
```

説明

CREATE FUNCTIONは新しい関数を定義します。CREATE OR REPLACE FUNCTIONは、新しい関数の作成、または、既存定義の置換のどちらかを行います。関数を定義するには、ユーザはその言語のUSAGE権限が必要です。

スキーマ名が含まれている場合、関数は指定されたスキーマに作成されます。スキーマ名がなければ、関数は現在のスキーマに作成されます。同じスキーマ内の同じ入力引数データ型を持つ既存の関数またはプロシージャの名前は、新しい関数の名前として使用できません。しかし、異なる引数データ型を持つ関数やプロシージャであれば、名前が重複しても構いません（これを、オーバーロードと言います）。

既存の関数定義を入れ替えるには、CREATE OR REPLACE FUNCTIONを使用してください。この方法では関数の名前や引数の型を変更することはできません（これを行った場合、新しく別の関数が作成されます）。また、CREATE OR REPLACE FUNCTIONでは、既存の関数の戻り値の型を変更することはできません。戻り値の型を変更したい場合は、その関数を削除し、再度作成してください。（これは、OUTパラメータを使用している場合、関数を削除しない限りOUTパラメータの型を変更できないことを意味します。）

CREATE OR REPLACE FUNCTIONを使用して既存の関数を置き換える場合、関数の所有者と権限は変わりません。他の関数に関するすべての属性には、そのコマンドで指定された値、または暗黙的な値が設定されま

す。関数を置き換えるためにはその関数を所有していなければなりません。(これには所有するロールのメンバであることが含まれています。)

関数を削除し再作成した場合、新しい関数は古いものと同じ実体にはなりません。古い関数を参照する、既存のルール、ビュー、トリガなどを削除しなければならないでしょう。関数を参照するオブジェクトを破壊せずに関数定義を変更するには、CREATE OR REPLACE FUNCTIONを使用してください。また、ALTER FUNCTIONを使用して、既存の関数の補助属性のほとんどを変更することができます。

関数を作成したユーザが、その関数の所有者となります。

関数を作成するためには、引数の型および戻り値の型に対するUSAGE権限を持たなければなりません。

パラメータ

name

作成する関数の名前です(スキーマ修飾名も可)。

argmode

引数のモードで、IN、OUT、INOUT、VARIADICのいずれかです。省略時のデフォルトはINです。OUT引数のみがVARIADICの後に続けることができます。また、RETURNS TABLE記法では、OUTとINOUT引数の両方を使用することはできません。

argname

引数の名前です。(SQLおよびPL/pgSQLを含む)言語の中にはこの名前を関数本体で使用できるものもあります。他の言語では、関数そのものに注目する限り、入力引数の名前は単なる追加ドキュメントとして扱われます。しかし関数呼び出し時に入力引数の名前を使用することで可読性を高めることができます。(4.3参照) どのような場合であっても、出力引数の名前は、結果の行型の列名となりますので重要です。(出力引数の名前を省略した場合、システムはデフォルトの列名を付与します。)

argtype

関数の引数のデータ型です(スキーマ修飾名も可)。基本データ型、複合データ型、ドメイン型、または、テーブル列の型の参照を使用することができます。

また、実装する言語に依存しますが、cstringといった「疑似型」を指定できる場合もあります。疑似型は、実引数の型の指定が不完全である、もしくは、通常のSQLデータ型の集合を越えていることを示します。

列の型を参照するには、table_name.column_name%TYPEと記述します。これを使用すると、テーブル定義が変更されても関数が影響を受けないようにするのに役に立つことがあります。

default_expr

パラメータが指定されなかった場合のデフォルト値として使用される式です。この式はパラメータの引数型と変換可能でなければなりません。入力パラメータ(INOUTを含みます)のみがデフォルト値を持つことができます。デフォルト値を持つパラメータの後ろにあるパラメータはすべて、同様にデフォルト値を持たなければなりません。

rettype

関数が返すデータの型です(スキーマ修飾名も可)。基本型、複合型、ドメイン型、または、テーブル列の型の参照を設定することができます。また、実装している言語によりますが、cstringのような「疑似型」も指定することが可能です。その関数が値を返すことを想定していない場合は、戻り値としてvoidを指定してください。

OUTもしくはINOUTパラメータが存在する場合、RETURNS句を省略することができます。省略しない場合は、出力用パラメータが意味する結果型に従ったもの、つまり、複数の出力用パラメータがあればRECORD、単一の出力用パラメータであればそれと同じ型、でなければなりません。

SETOF修飾子は、その関数が、1つではなく複数のアイテムの集合を返すことを示します。

列の型は、table_name.column_name%TYPEと記述することで参照されます。

column_name

RETURNS TABLE構文における出力列の名前です。これは実際名前付けされたOUTパラメータを宣言する別の方法ですが、RETURNS TABLEがRETURNS SETOFをも意味する点が異なります。

column_type

RETURNS TABLE構文における出力列のデータ型です。

lang_name

関数を実装している言語の名前です。このパラメータには、sql、c、internal、もしくはユーザ定義手続き言語(例:plpgsql)の名前を指定可能です。名前を単一引用符で囲むのは廃止予定で、大文字小文字の一致が必要になります。

TRANSFORM { FOR TYPE type_name } [, ...] }

関数呼び出しにどの変換を適用すべきかのリストです。変換はSQLの型と言語独自のデータ型の間の変換を行います(CREATE TRANSFORMを参照)。手続言語の実装では、通常、ビルトインの型についてハードコードされた知識があるので、それらをこのリストに含める必要はありません。手続言語の実装が型の処理について定めておらず、変換が提供されない場合は、データ型変換のデフォルトの動作によることになりますが、これは実装に依存します。

WINDOW

WINDOWは、この関数が普通の関数ではなくウィンドウ関数であることを示します。現在これはC言語で作成した関数のみに使用することができます。既存の関数定義を置き換える場合、WINDOW属性を変更することはできません。

IMMUTABLE
STABLE
VOLATILE

これらの属性は、関数の動作に関する情報を問い合わせオプティマイザに提供します。いずれか1つのキーワードのみ指定できます。指定がない場合は、デフォルトでVOLATILEと解釈されます。

IMMUTABLEは、関数がデータベースに対する変更を行わないこと、および、その関数に同じ引数値を与えた場合に常に同じ結果を返すことを示します。つまり、データベースを検索したり、引数リスト中に直接

存在しない情報を使用したりしないということです。このオプションが指定された場合、引数が全て定数である関数呼び出しは、即座に関数値と置き換えることができます。

STABLEは、関数がデータベースに対する変更を行わないこと、および、その関数に同じ引数値を与えた場合、常に同じ結果を返すが、SQL文が異なると結果が変わってしまう可能性があることを示します。これは、データベース検索や(現在の時間帯のような)パラメータ変数などに結果が依存する関数に適します。(これは現在のコマンドで変更された行を問い合わせたいAFTERトリガには不適切です。) また、current_timestamp系の関数は、1つのトランザクション内では値が変化しないため、STABLEであることに注意してください。

VOLATILEは、1つのテーブルスキャン内でも関数の値が変化する可能性があるため、最適化できないことを示します。このような意味で変動的(volatile)なデータベース関数は、比較的少数です。例えば、random()、currval()、timeofday()などは変動的な関数です。しかし、例えばsetval()などの副作用がある関数は、その結果を完全に予測できるとしても、呼び出しを最適化しないよう、VOLATILE(変動的)に分類する必要があることに注意してください。

詳細は37.7を参照してください。

LEAKPROOF

LEAKPROOFは、関数が副作用を持たないことを示します。その引数に関する情報を戻り値以外で漏らしません。例えば、一部の引数値に対してのみエラーメッセージを返す関数や何らかのエラーメッセージの中に引数の値を含める関数は漏洩防止(leakproof)とはいえません。これはsecurity_barrierオプション付きで作成されたビュー、あるいは行単位セキュリティが有効にされたテーブルに対して、システムが問い合わせを実行する方法に影響します。データが偶然に露見することを防ぐため、システムは、漏洩防止でない関数を含む問い合わせのユーザが提供した条件より前に、セキュリティポリシーおよびセキュリティバリアビューの条件を強制します。漏洩防止であるとされた関数および演算子は信頼できると見なされ、セキュリティポリシーおよびセキュリティバリアビューによる条件より先に実行されることがあります。なお、引数を取らない、あるいはセキュリティバリアビューやテーブルから引数を渡されない関数は、セキュリティ条件より前に実行するために漏洩防止とする必要はありません。CREATE VIEWおよび40.5を参照してください。このオプションはスーパーユーザによってのみ設定することができます。

CALLED ON NULL INPUT
RETURNS NULL ON NULL INPUT
STRICT

CALLED ON NULL INPUT(デフォルト)を指定すると、引数にNULLが含まれていても、関数が通常通り呼び出されます。その場合は、必要に応じてNULL値を確認し、適切な対応をすることは関数作成者の責任です。

RETURNS NULL ON NULL INPUTもしくはSTRICTを指定すると、関数の引数に1つでもNULLがある場合、常にNULLを返します。このパラメータが指定されると、NULL引数がある場合、関数は実行されません。代わりに、NULLという結果が自動的に与えられます。

[EXTERNAL] SECURITY INVOKER
[EXTERNAL] SECURITY DEFINER

SECURITY INVOKERを指定すると、関数を呼び出したユーザの権限で、その関数が実行されます。これがデフォルトです。SECURITY DEFINERを指定すると、関数を所有するユーザの権限で、その関数が実行されます。

EXTERNALキーワードは、SQLとの互換性を保つために許されています。しかし、SQLとは異なり、この機能は外部関数だけではなくすべての関数に適用されるため、このキーワードは省略可能です。

PARALLEL

PARALLEL UNSAFEは、その関数が並列モードでは実行できないこと、そしてそのような関数がSQL文の中にある場合は順次の実行プランが強制されることを意味します。これがデフォルトです。PARALLEL RESTRICTEDはその関数が並列モードで実行できますが、その実行は並列グループのリーダーに制限されることを意味します。PARALLEL SAFEはその関数が並列モードで制限なく実行することについて安全であることを意味します。

関数がデータベースの状態に何らかの変更を行う、サブトランザクションを使うなどトランザクションを変更する、シーケンスにアクセスするか設定に恒久的な変更をする(例えばsetval)という場合はparallel unsafe(並列は安全でない)という印をつけるべきです。一時テーブル、クライアントの接続状態、カーソル、プリペアド文、その他並列モードでシステムが同期できない様々なバックエンド独自の状態に関数がアクセスする場合、parallel restricted(並列は制限される)という印をつけるべきです(例えば、setseedはグループのリーダー以外では実行できません。なぜなら他のプロセスでなされた変更がリーダーに反映されないからです)。一般的に、restrictedあるいはunsafeな関数がsafeとラベル付けされた場合、あるいはunsafeな関数がrestrictedとラベル付けされた場合、それがパラレルクエリ内で使用されると、エラーが発生したり、誤った結果が生成されたりします。C言語の関数は、ラベルが間違っていると、理論的には全く予想できない動作をすることがあります。これは任意のCプログラムに対してシステムが自分を保護する手段がないからですが、多くの場合、その結果は他の関数と同程度の悪さでしょう。よくわからない場合は、デフォルトのUNSAFEで関数にラベル付けしてください。

COST execution_cost

この関数の推定実行コストを表す正数で、単位はcpu_operator_costです。関数が集合を返す場合、これは1行当たりのコストとなります。このコストが指定されない場合、C言語および内部関数では1、他のすべての言語では100となります。値をより大きくすると、プランナは必要以上に頻繁に関数を評価しないようになります。

ROWS result_rows

プランナが想定する、この関数が返す行数の推定値を表す正数です。これは、関数が集合を返すものと宣言された場合のみ使用可能です。デフォルト推定値は1000行です。

SUPPORT support_function

この関数のために使うプランナサポート関数の名前です(スキーマ修飾名も可)。詳細は37.11を参照してください。このオプションを使うにはスーパーユーザでなければなりません。

configuration_parameter
value

SET句により、関数が始まった時に指定した設定パラメータを指定した値に設定し、関数の終了時にそれを以前の値に戻すことができます。SET FROM CURRENTは、CREATE FUNCTIONの実行時点でのパラメータ値を、関数に入る時に適用する値として保管します。

関数にSET句が付いている場合、関数内部で実行されるSET LOCALコマンドの同一変数に対する効果はその関数に制限されます。つまり、設定パラメータの前の値は関数が終了する時に元に戻ります。しか

し、通常の(LOCALがない)SETコマンドはSET句を上書きします。これは過去に行われたSET LOCALコマンドに対してもほぼ同じです。つまり、このコマンドの効果は、現在のトランザクションがロールバックされない限り、関数が終了した後も永続化されます。

使用可能なパラメータと値については、SETおよび第19章を参照してください。

definition

関数を定義する文字列定数です。このパラメータの意味は言語に依存します。内部的な関数名、オブジェクトファイルへのパス、SQLコマンド、手続き言語で記述されたテキストなどを指定できます。

関数を定義する文字列を記述する際に、通常の単一引用符ではなく、ドル引用符(4.1.2.4参照)を使用すると便利なことが多くあります。ドル引用符を使用しなければ、関数定義内の単一引用符やバックスラッシュは必ず二重にしてエスケープしなければなりません。

obj_file, link_symbol

この構文のAS句は、動的にロードされるC言語関数において、C言語のソースコード中の関数名がSQL関数の名前と同じでない場合に使われます。obj_fileという文字列はコンパイルされたC関数を含む共有ライブラリファイルの名前で、LOADコマンドの場合と同じように解釈されます。文字列link_symbolはその関数のリンクシンボル、つまり、C言語ソースコード中の関数の名前です。リンクシンボルが省略された場合、定義されるSQL関数の名前と同じものであるとみなされます。全ての関数について、C言語における名前は、重複してはいけません。したがって、オーバーロードするC言語関数には、異なるC言語の名前を与える必要があります(例えば、C言語における名前の一部に引数の型を使用してください)。

同一オブジェクトファイルを参照する、CREATE FUNCTION呼び出しが繰り返された場合、そのファイルはセッション毎に一度だけロードされます。(おそらく開発段階で)ファイルをアンロードし再ロードするには、新しいセッションを開始してください。

さらに詳しい関数の作成方法については37.3を参照してください。

オーバーロード

PostgreSQLでは関数の*オーバーロード*が可能です。つまり、入力引数の型が異なっていれば、複数の関数に同じ名前を使用することができます。使うかどうかに関わりなく、この能力は、あるユーザが他のユーザを信用しないデータベースで関数を呼び出す時に、セキュリティの事前の対策を必要とします。10.3を参照してください。

同じ名前、同じ*入力用*パラメータ型を持つ場合、2つの関数は同一であるとみなされます。OUTパラメータは無視されます。したがって、例えば以下の宣言は競合しています。

```
CREATE FUNCTION foo(int) ...
CREATE FUNCTION foo(int, out text) ...
```

異なる引数型のリストを持つ関数は、作成時に競合するとはみなされませんが、デフォルト値が指定された場合使用時に競合する可能性があります。例えば以下を考えてみましょう。

```
CREATE FUNCTION foo(int) ...
```

```
CREATE FUNCTION foo(int, int default 42) ...
```

foo(10)という呼び出しは、どちらの関数を呼び出すべきかに関して曖昧さがあるために失敗します。

注釈

関数の引数と戻り値の宣言において、完全なSQL型の構文が使用できます。しかし、括弧付けされた型修飾子（例えばnumeric型の精度フィールド）は、CREATE FUNCTIONにより破棄されます。従って、CREATE FUNCTION foo (varchar(10)) ...はCREATE FUNCTION foo (varchar) ...とまったく同じになります。

既存の関数をCREATE OR REPLACE FUNCTIONを使って置き換える場合、パラメータ名の変更に関して制限があります。すでに何らかの入力パラメータに割り当てられた名前を変更することはできません。（しかし、これまで名前を持たなかったパラメータに名前を追加することは可能です。）複数の出力パラメータが存在する場合、関数の結果を表わす無名複合型の列名を変更することになるため、出力パラメータの名前を変更することはできません。既存の関数呼び出しが置き換わった時に動作しなくなることを確実に防ぐために、これらの制限がなされています。

関数がVARIADIC引数を持つSTRICTと宣言された場合、その厳密性検査では、variadic配列全体が非NULLかどうかを検査します。配列がNULL要素を持っていたとしても関数は呼び出されます。

例

ここでは、初心者向けの簡単な例をいくつか示します。37.3には、より多くの情報と例が記載されています。

```
CREATE FUNCTION add(integer, integer) RETURNS integer
    AS 'select $1 + $2;'
    LANGUAGE SQL
    IMMUTABLE
    RETURNS NULL ON NULL INPUT;
```

PL/pgSQLで、引数名を使用して、整数を1増やします。

```
CREATE OR REPLACE FUNCTION increment(i integer) RETURNS integer AS $$
        BEGIN
                RETURN i + 1;
        END;
$$ LANGUAGE plpgsql;
```

複数の出力用パラメータを持つレコードを返します。

```
CREATE FUNCTION dup(in int, out f1 int, out f2 text)
    AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
    LANGUAGE SQL;

SELECT * FROM dup(42);
```

上と同じことを、明示的な名前が付いた複合型を使用して、より冗長に行うことができます。

```
CREATE TYPE dup_result AS (f1 int, f2 text);

CREATE FUNCTION dup(int) RETURNS dup_result
    AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
    LANGUAGE SQL;

SELECT * FROM dup(42);
```

複数列を返す別の方法は、TABLE関数を使用することです。

```
CREATE FUNCTION dup(int) RETURNS TABLE(f1 int, f2 text)
    AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
    LANGUAGE SQL;

SELECT * FROM dup(42);
```

しかし、これは実際には、1つのレコードではなく、レコードの集合を返しますので、TABLE関数は上の例とは異なります。

SECURITY DEFINER関数の安全な作成

SECURITY DEFINER関数は関数を所有するユーザの権限で実行されますので、その関数を間違って使用できないことを確実にしなければなりません。安全上、search_pathは、信頼できないユーザが書き込み可能なスキーマを除去した形で設定すべきです。これは、悪意のあるユーザがその関数で使用されるオブジェクトを隠すようなオブジェクト（例えば、テーブル、関数、演算子など）を作成することを防ぎます。ここで特に重要なことは、一時テーブルスキーマです。このスキーマはデフォルトで最初に検索され、そして、通常誰でも書き込み可能です。一時スキーマの検索を強制的に最後にすることで、セキュリティを調整できます。このためには、pg_tempをsearch_pathの最後の項目として記載してください。安全な使用方法を以下の関数で示します。

```
CREATE FUNCTION check_password(uname TEXT, pass TEXT)
RETURNS BOOLEAN AS $$
DECLARE passed BOOLEAN;
BEGIN
        SELECT  (pwd = $2) INTO passed
        FROM    pwds
        WHERE   username = $1;

        RETURN passed;
END;
$$  LANGUAGE plpgsql
    SECURITY DEFINER

    -- 信頼できるスキーマ、その後にpg_tempという順でsearch_pathを安全に設定します。
```

```
    SET search_path = admin, pg_temp;
```

この関数の意図は、テーブルadmin.pwdsにアクセスすることです。しかしSET句がなければ、あるいはSET句がadminだけしか記述していなければ、pwdsという名前の一時テーブルを作成することで、この関数は無意味になってしまいます。

PostgreSQLバージョン8.3より前では、SET句は利用できません。このため古い関数には、search_pathを保管し、設定、そして元に戻すという、多少複雑なロジックが含まれているかもしれません。こうした目的にSET句を使用すると、かなり簡単になります。

この他に注意すべき点として、新しく作成された関数ではデフォルトで実行権限がPUBLICに付与されていることがあります。（詳細は5.7を参照してください。）SECURITY DEFINER関数の使用を一部のユーザのみに制限したいことはよくあります。このためには、デフォルトのPUBLIC権限を取り消し、そして、実行権限の付与を選択して行ってください。新しい関数がすべてのユーザに実行可能となる隙間が存在することを防ぐためには、単一トランザクション内で作成と権限設定を行ってください。以下に例を示します。

```
BEGIN;
CREATE FUNCTION check_password(uname TEXT, pass TEXT) ... SECURITY DEFINER;
REVOKE ALL ON FUNCTION check_password(uname TEXT, pass TEXT) FROM PUBLIC;
GRANT EXECUTE ON FUNCTION check_password(uname TEXT, pass TEXT) TO admins;
COMMIT;
```

互換性

CREATE FUNCTIONコマンドは標準SQLで定義されています。PostgreSQLにおけるCREATE FUNCTIONも類似の機能を持ちますが、完全な互換性はありません。属性には移植性がありません。また、使用可能な言語も異なります。

他のデータベースシステムとの互換性のために、argmodeはargnameの前に書くことも後に書くこともできます。しかし、1つ目の方法が標準に従っています。

パラメータのデフォルトに関しては、標準SQLではDEFAULTキーワードの構文のみを規定します。=を持つ構文はT-SQLおよびFirebirdで使用されています。

関連項目

ALTER FUNCTION, DROP FUNCTION, GRANT, LOAD, REVOKE

CREATE GROUP

CREATE GROUP — 新しいデータベースロールを定義する

概要

```
CREATE GROUP name [ [ WITH ] option [ ... ] ]


optionは次のようになります。

      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | REPLICATION | NOREPLICATION
    | BYPASSRLS | NOBYPASSRLS
    | CONNECTION LIMIT connlimit
    | [ ENCRYPTED ] PASSWORD 'password' | PASSWORD NULL
    | VALID UNTIL 'timestamp'
    | IN ROLE role_name [, ...]
    | IN GROUP role_name [, ...]
    | ROLE role_name [, ...]
    | ADMIN role_name [, ...]
    | USER role_name [, ...]
    | SYSID uid
```

説明

CREATE GROUPはCREATE ROLEの別名になりました。

互換性

標準SQLにはCREATE GROUPはありません。

関連項目

CREATE ROLE

CREATE INDEX

CREATE INDEX — 新しいインデックスを定義する

概要

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ [ IF NOT EXISTS ] name ] ON [ ONLY ] table_name
 [ USING method ]
    ( { column_name | ( expression ) } [ COLLATE collation ] [ opclass [ ( opclass_parameter
 = value [, ... ] ) ] ] [ ASC | DESC ] [ NULLS { FIRST | LAST } ] [, ...] )
    [ INCLUDE ( column_name [, ...] ) ]
    [ WITH ( storage_parameter [= value] [, ... ] ) ]
    [ TABLESPACE tablespace_name ]
    [ WHERE predicate ]
```

説明

CREATE INDEXは、指定したリレーションの指定した列(複数可)に対するインデックスを作ります。リレーションとしてテーブルまたはマテリアライズドビューを取ることができます。インデックスは主にデータベースの性能を向上するために使われます（しかし、インデックスの不適切な使用は性能の低下につながる可能性があります）。

インデックスのキーフィールドは、列名、または括弧に囲まれた式として指定されます。インデックスメソッドが複数列に対するインデックスをサポートする場合は、複数のフィールドを指定できます。

インデックスのフィールドとして、テーブル行の1つ以上の列の値から計算する式を指定できます。この機能は、元のデータに何らかの変換を加えた値を基とするデータへの高速なアクセスを行う手段として使用することができます。例えば、upper(col)という計算に基づくインデックスがあれば、WHERE upper(col) = 'JIM'という句ではインデックスを使用することができます。

PostgreSQLはB-tree、ハッシュ、GiST、SP-GiST、GIN、BRINのインデックスメソッドを用意しています。ユーザが独自にインデックスメソッドを定義することもできますが、これはかなり複雑です。

WHERE句が存在する場合、部分インデックスが作成されます。部分インデックスは、テーブルの一部、通常は、テーブルの中でよりインデックスが有用な部分のみのエントリを持つインデックスです。例えば、請求済みの注文と未請求の注文を情報として持つテーブルがあり、テーブル全体における未請求の注文の割合が小さく、かつ、頻繁に使用される場合、未請求の注文のみにインデックスを作成することで性能を向上できます。部分インデックスのその他の利用方法として、UNIQUE付きのWHEREを使用して、テーブルの部分集合に一意性を強制する例が考えられます。詳細は11.8を参照してください。

WHERE句内の式では、元となるテーブルの列のみを参照できます。しかし、インデックスを付加する列だけではなく、全ての列を使用することができます。また、現在、副問い合わせと集約式については、WHEREで使用することができません。同一の制限は、式で表されたインデックスのフィールドにも適用されます。

インデックスの定義で使用される全ての関数と演算子は、「不変」(immutable)でなければなりません。つまり、結果は入力引数にのみに依存し、(他のテーブルの内容や現時刻などの)外部からの影響を受けてはなりません。この制限によって、インデックスの動作が十分定義されていることが保証されます。インデックス式やWHERE句にユーザ定義の関数を使用する場合、関数を作成する際、IMMUTABLE(不変)オプションを付けることを忘れないでください。

パラメータ

UNIQUE

インデックスを(既にデータがある状態で)作成する時、およびテーブルにデータを追加する時に、テーブル内の値が重複していないかを検査します。重複エントリを生じるデータの挿入または更新はエラーとなります。

一意性インデックスがパーティションテーブルに適用されるときには、追加的な制限が適用されます。CREATE TABLEを参照してください。

CONCURRENTLY

このオプションを使用すると、PostgreSQLは、対象テーブルに対する同時挿入、更新、削除を防止するようなロックを獲得せずにインデックスを作成します。通常のインデックス作成処理では、完了するまで対象テーブルへの書き込みはできません(読み取りは可能です)。このオプションを使用する際に注意しなければならない点が複数あります。下記のBuilding Indexes Concurrentlyを参照してください。

一時テーブルに対してはCREATE INDEXは常に同時作成ではありません。他のセッションはアクセスできませんし、同時でないインデックス作成の方がより安価だからです。

IF NOT EXISTS

同じ名前のリレーションが既に存在している場合にエラーとしません。この場合、注意が発行されます。既存のインデックスが、作成されようとしていたものと類似のものである保証は全くないことに注意してください。IF NOT EXISTSを指定する場合はインデックス名が必須です。

INCLUDE

オプションのINCLUDE句は非キー列としてインデックスに含める列のリストを指定します。非キー列をインデックススキャンの検索条件に使うことはできません。また、インデックスで何であれ一意性制約や排他制約を強制する目的に対しても無視されます。しかしながら、インデックスオンリースキャンは、インデックスエントリから値を直接得ることができるので、インデックスのテーブルを見に行く必要なく、非キー列の内容を返すことができます。このように非キー列の追加は、そうでないとできないインデックスオンリースキャンを利用可能にします。

インデックスに非キー列を加えることには、特に幅広の列については、保守的であるのが賢明です。インデックス列がインデックス型で許される最大サイズを超えた場合、データ挿入は失敗してしまいます。いかなる場合でも、非キー列はインデックスのテーブルからデータを複製して、インデックスのサイズを膨張させます。よって、潜在的に検索を遅くします。さらに、非キー列を持つインデックスではB-tree重複排除は決して使われません。

INCLUDE句にある列リストは適合した演算子クラスを必要としません。ここには与えられたアクセスメソッドに対して定義された演算子クラスを持たないデータ型の列を含めることができます。

インデックスオンリースキャンで使うことができないため、INCLUDEする列に式は対応していません。

今のところ本機能はB-treeとGiSTインデックスアクセスメソッドに対応しています。B-treeとGiSTインデックスではINCLUDE句にリストされた列の値は、ヒープタプルに対応するリーフタプルに含まれますが、ツリーを辿るのに使われる上位レベルのインデックスエントリには含まれません。

name

作成するインデックスの名前です。この名前には、スキーマ名を含めることはできません。インデックスは、常にその親テーブルと同じスキーマに作成されます。この名前を省略すると、PostgreSQLはその親テーブルの名前とインデックス付けされる列名に基づいた適切な名前を選びます。

ONLY

テーブルがパーティションテーブルであっても、パーティションにインデックス作成を再帰的に実行しないことを示します。デフォルトでは再帰実行します。

table_name

インデックスを作成するテーブルの名前です（スキーマ修飾名の場合もあります）。

method

使用するインデックスメソッドの名前です。btree、hash、gist、spgist、gin、brinから選択します。デフォルトのメソッドはbtreeです。

column_name

テーブルの列の名前です。

expression

テーブル上の1つ以上の列を使用した式です。通常この式は、構文で示した通り括弧で囲む必要があります。しかし、式が関数呼び出し形式になっている場合は括弧を省略することができます。

collation

インデックスで使用する照合順序の名前です。デフォルトではインデックスはインデックス付け対象の列で宣言された照合順序またはインデックス付け対象の式の結果の照合順序を使用します。デフォルト以外の照合順序を使用する式を含む問い合わせで、デフォルト以外の照合順序を持つインデックスが有用になるかもしれません。

opclass

演算子クラスの名前です。詳細は下記を参照してください。

opclass_parameter

演算子クラスパラメータの名前です。詳細は下記を参照してください。

ASC

正方向のソート順を指定します(これがデフォルトです)。

DESC

逆方向のソート順を指定します。

NULLS FIRST

NULLを非NULLより前にソートすることを指定します。これはDESCが指定された場合のデフォルトです。

NULLS LAST

NULLを非NULLより後にソートすることを指定します。これはDESCが指定されない場合のデフォルトです。

storage_parameter

インデックスメソッド固有の格納パラメータの名前です。詳細は下記のIndex Storage Parametersを参照してください。

tablespace_name

インデックスを生成するテーブル空間です。指定されなかった場合、default_tablespace、もし一時テーブル上のインデックスであれば、temp_tablespacesが考慮されます。

predicate

部分インデックス用の制約式です。

インデックス格納パラメータ

WITH句を使うと、インデックスの格納パラメータを指定できます。インデックスメソッドはそれぞれ固有の設定可能な格納パラメータを持ちます。B-tree、ハッシュ、GiSTおよびSP-GiSTといったインデックスはすべて次のパラメータを受け付けます。

fillfactor (integer)

インデックス用のフィルファクタは割合（パーセント）で、インデックスメソッドがインデックスページをまとめ上げる時にどの程度ページを使用するかを決定するものです。B-treeでは、リーフページは初期インデックス構築時と右側（新しい最大キー値を追加する方向）にインデックスを拡張する時にこの割合分までページを使用します。その後ページすべてが完全に使用されると分割され、インデックスの効果が徐々に劣化します。B-treeのデフォルトのフィルファクタは90ですが、10から100までの任意の整数値を設定することができます。テーブルが静的な場合、100が最善でインデックスの物理サイズを最小化できます。更新が非常に多い場合は、ページ分割の頻度を少なくするために、より小さなフィルファクタを設定する方が良いです。この他のインデックスメソッドでは、フィルファクタを異なる意味で使用しますが、おおよそは同じです。メソッドによってフィルファクタのデフォルト値は異なります。

B-treeインデックスは以下パラメータも受け付けます。

deduplicate_items (boolean)

63.4.2に書かれているB-tree重複排除技法の使用を制御します。最適化を有効、無効にするにはON、OFFを設定します。(19.1に書かれているように、ONやOFFの他の綴りも認められています。) デフォルトはONです。

> **注記**
>
> ALTER INDEXでdeduplicate_itemsをオフにすると、その後の挿入で重複排除のトリガは発生しなくなりますが、それ自体は既存のポスティングリストタプルが標準のタプル表現を使うようにはしません。

vacuum_cleanup_index_scale_factor (floating point)

vacuum_cleanup_index_scale_factorのインデックス毎の値です。

GiSTインデックスではさらに以下のパラメータを受け付けます。

buffering (enum)

64.4.1で説明するバッファリング構築技術をインデックスを構築する時に使用するかどうかを決定します。OFFで無効に、ONで有効になります。またAUTOと指定すると、最初は無効ですが、インデックスサイズがeffective_cache_sizeに達した後はその場で有効になります。デフォルトはAUTOです。

GINインデックスでは以下の異なるパラメータを受け付けます。

fastupdate (boolean)

この設定は66.4.1で説明する高速更新技法を使用するかどうかを制御します。これは論理値パラメータであり、ONは高速更新を有効に、OFFは無効にします。デフォルトはONです。

> **注記**
>
> ALTER INDEXを使用してfastupdateを無効にすることにより、以後の挿入は待機中のインデックス項目リストに入らないようになります。しかし、このコマンド自体はこれまでの項目を吐き出しません。確実に待機中のリストを空にするためには、続いてテーブルをVACUUMするか、gin_clean_pending_list関数を呼び出すのが良いでしょう。

gin_pending_list_limit (integer)

gin_pending_list_limitのカスタムパラメータです。値はキロバイト単位で指定します。

BRINインデックスは別のパラメータを受け入れます。

pages_per_range (integer)

BRINインデックスの各エントリについて1つのブロックレンジを構成するテーブルブロックの数を定義します（詳しくは67.1参照）。デフォルトは128です。

autosummarize (boolean)

次のページへの挿入が検知された時に、いつでも直前のページに対してサマリー処理を起動するかどうかを定義します。

インデックスの同時作成

インデックスの作成が、通常のデータベース操作に影響を与えることがあります。通常PostgreSQLは、対象テーブルに対する書き込みをロックしてから、対象テーブル全体のインデックス作成を一度のスキャンで行います。他のトランザクションはテーブルを読み取ることはできますが、対象テーブル内の行を挿入、更新、削除しようとすると、インデックス作成が完了するまでブロックされます。実行中の運用状態のデータベースシステムの場合、これは重大な影響を与える可能性があります。非常に大規模なテーブルに対するインデックス作成は何時間もかかることがあり得ます。また小規模なテーブルであっても、インデックス作成により、運用状態のシステムとしては受け入れられないほど長い時間、書き込みロックがかかる可能性があります。

PostgreSQLは書き込みをロックしないインデックス作成もサポートしています。CREATE INDEXにCONCURRENTLYオプションをつけることでこの方式が行われます。このオプションを使うと、PostgreSQLはテーブルを2回スキャンしなければなりません。さらに、潜在的にそのインデックスを更新または使用する可能性がある、実行中のすべてのトランザクションが終わるまで待機しなければなりません。したがって、この方式は通常の方式よりも総作業時間がかかり、また、完了するまでの時間が非常に長くなります。しかし、インデックス作成中に通常の操作を行い続けることができますので、この方式は運用環境での新規インデックス作成に有用です。もちろん、インデックス作成によりCPUや入出力に余分に負荷がかかりますので、他の操作が低速になる可能性があります。

同時実行インデックス構築では実際には、1つのトランザクションでシステムカタログに登録され、さらに2つのトランザクションで2つのテーブルスキャンが起こります。各テーブルスキャンの前に、インデックス構築はテーブルを修正した実行中のトランザクションが終了するのを待たなければなりません。2回目のスキャンの後、インデックス構築は2回目のスキャンより前のスナップショット（第13章参照）を持つすべてのトランザクションが終了するのを待たなければなりません。その後でようやく、インデックスは利用可能であると印が付けられ、CREATE INDEXコマンドが終了します。しかし、それでもインデックスは問い合わせに対して即座に利用可能であるとは限りません。最悪の場合、インデックス構築開始前のトランザクションが存在する間は利用できません。

たとえばデッドロックや一意性インデックスにおける一意性違反など、テーブルスキャン中に問題が発生すると、CREATE INDEXは失敗しますが、「無効な」インデックスが残ってしまいます。こうしたインデックスは完全ではない可能性がありますので、問い合わせの際には無視されます。しかし、更新時にオーバーヘッドがかかります。psqlの\dコマンドでは、こうしたインデックスをINVALIDとして報告します。

```
postgres=# \d tab
       Table "public.tab"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 col    | integer |           |          |
Indexes:
    "idx" btree (col) INVALID
```

こうした場合の推奨復旧方法は、インデックスを削除し、再度CREATE INDEX CONCURRENTLYを実行することです。（他にもREINDEX INDEX CONCURRENTLYを使用したインデックスの再作成という方法もあります。）

この他に一意性インデックスを同時作成する場合の注意事項があります。2回目のテーブルスキャンが始まる時点で、他のトランザクションに対する一意性制約が既に有効になっているという点です。これは、インデックスが使用できるようになる前やインデックス作成が最終的に失敗したとしても、制約違反が他のトランザクションで報告されてしまうことを意味します。また、2回目のスキャン中に失敗した後も、「無効な」インデックスによる一意性制約は強制され続けます。

式インデックスや部分インデックスの同時作成もサポートされています。式の評価中にエラーが発生した場合も、上で説明した一意性制約違反と同様な状況が発生します。

同一テーブルに対する通常のインデックス作成処理は複数並行して行うことができます。しかし、あるテーブルに対するインデックスの同時作成は一度に1つしか行うことができません。また、どちらの場合でもインデックス作成中のテーブルのスキーマ変更はできません。この他に、通常のCREATE INDEXコマンドはトランザクションブロック内で実行させることができますが、CREATE INDEX CONCURRENTLYは実行させることができないという相違点があります。

今の所パーティションテーブルのインデックスの同時作成はサポートされていません。しかし、パーティションテーブルへの書き込みをロックしている時間を短くするために、各パーティション上のインデックスを個別に同時作成してから最後にパーティションインデックスを非同時的に作成することはできます。この場合、パーティションインデックスの作成はメタデータのみの操作になります。

注釈

インデックスが、どのような時に使用され、どのような時に使用されないか、また、どのような場合に有用かといった情報については第11章を参照してください。

現在は、B-tree、GiST、GIN、BRINインデックスメソッドのみが、複数列に対するインデックスをサポートしています。指定できる列は、デフォルトでは32個までです（この制限はPostgreSQLのコンパイル時に変更できます）。現在、B-treeのみが一意性インデックスをサポートしています。

オプションのパラメータのついた演算子クラスは、インデックスのそれぞれの列に指定することができます。演算子クラスは、その列のインデックスが使う演算子を識別します。例えば、4バイト整数に対するB-treeインデックスには、int4_opsクラスを使います。この演算子クラスには、4バイト整数の比較関数が含まれています。実際の使用では、通常、列のデータ型のデフォルト演算子クラスで十分です。演算子クラスを保持する主な理由は、データ型の中には有意な順序を2つ以上持つものがあるかもしれないからです。例えば、複素数のソートで絶対値または実数部のどちらかを使いたい場合がありえます。これを実現するには、データ型として2つの演算子クラスを定義し、インデックスを作る時に適切なクラスを選択します。演算子クラスについての詳細は、11.10と37.16を参照してください。

CREATE INDEXがパーティションテーブルに実行されたときのデフォルトの振る舞いは、全パーティションが一致するインデックスを持つようにする全パーティションへの再帰的な実行です。各パーティションは最初に同等のインデックスが既に存在するかの判断のために検査され、存在するなら作成するインデックスに対するパーティションインデックスとしてアタッチされます。新たに作成するインデックスが既存インデックスの親インデックスとなります。一致するインデックスが存在しない場合、新たなインデックスが作られて、自動的にアタッチされます。各パーティションの新たなインデックス名は、コマンドでインデックスが指定されなかった場合と同様に決定されます。ONLYオプションが指定された場合、再帰処理は行われず、そのインデックスは無効と印付けされます。（ALTER INDEX ... ATTACH PARTITIONは、ひとたび全パーティションが一致するインデックスを得たなら、インデックスを有効に印付けします）しかしながら、ONLYが指定されたとしても、将来にCREATE TABLE ... PARTITION OFを使って作成されるあらゆるパーティションは自動的に一致するインデックスを持つことに注意してください。

順序付きスキャンをサポートするインデックスメソッド（現時点ではB-Treeのみ）では、ASC、DESC、NULLS FIRST、NULLS LAST句(省略可能)をオプションで指定し、インデックスのソート順を変更することができます。順序付きインデックスは正方向にも逆方向にもスキャンすることができますので、単一列に対するDESCイン

デックスは通常は有用ではありません。このソート順序はすでに通常のインデックスを使用して実現できます。これらのオプションの価値は、SELECT ... ORDER BY x ASC, y DESCなどの順序指定が混在する問い合わせによって要求されるソート順に一致する、複数列に対するインデックスを作成できる点です。NULLSオプションは、インデックスに基づいた問い合わせにおいてソート処理を省略するために「NULLのソート順を低くする」動作をサポートする必要がある場合に有用です。デフォルトの動作は「NULLのソート順を高くする」です。

ほとんどのインデックスメソッドにおいて、インデックス作成速度はmaintenance_work_memの設定に依存します。より大きな値を設定すると、インデックス作成に必要となる時間が短縮されます。ただし、実際に使用できるメモリ量を超えるほど大きくすると、マシンがスワップ状態になり、遅くなります。

PostgreSQLはテーブルの行をより高速に処理するために複数CPUを効かせてインデックスを作成できます。この機能はパラレルインデックス作成と呼ばれています。パラレルでのインデックス作成をサポートしているインデックスメソッド（今のところB-Treeのみ）に対して、maintenance_work_memでは、いくつのワーカプロセスが実行されているかに拘らず、各インデックス作成操作で使うことができる全体のメモリの最大量を指定します。一般にコストモデルは、もしあるなら、どれだけの数のワーカプロセスを要求すべきかを自動的に決定します。

パラレルインデックス作成ではmaintenance_work_memを増やすことで、同様の逐次インデックス作成ではほとんど恩恵がみられない場合でも恩恵があるかもしれません。パラレルワーカはmaintenance_work_mem全体の内、少なくとも32MBの割り当て分を持たなければならないため、maintenance_work_memは要求されるワーカプロセス数に影響を及ぼすかもしれないことに注意してください。また、リーダープロセスに対しても32MBの割り当てを残さなければなりません。max_parallel_maintenance_workersを増やすことで、より多くのワーカが使用できるようになるかもしれません。これは、インデックス作成が既にI/Oバウンドであるのでない限り、インデックス作成の所要時間を減らすでしょう。もちろん、休止している十分なCPU容量もある前提です。

ALTER TABLEを通してparallel_workersの値を設定することで、テーブルに対してCREATE INDEXでどれだけのワーカプロセス数が要求されるかを、直接に調整できます。これはコストモデルを完全に無視して、maintenance_work_memがパラレルワーカの要求数に影響を与えることを回避します。ALTER TABLEを通してparallel_workersを0に設定することは、そのテーブルに対するパラレルインデックス作成を全ての場合に無効化します。

> **ヒント**
>
> インデックス作成のチューニング一部としてparallel_workersを設定した後、これをリセットしたいかもしれません。parallel_workersは全てのパラレルテーブルスキャンに影響を与えるので、これは不注意な問い合わせプランの変更を回避します。

CONCURRENTLYオプションを伴うCREATE INDEXは特に制限なくパラレル作成をサポートしますが、実際には最初のテーブルスキャンだけがパラレルに実行されます。

インデックスを削除するには、DROP INDEXを使用してください。

以前のPostgreSQLにはR-treeインデックスメソッドがありました。GiSTメソッドに比べて大きな利点がありませんでしたので、このメソッドは削除されました。古いデータベースからGiSTへの変換を簡単にするため、USING rtreeが指定された場合、CREATE INDEXはUSING gistと解釈します。

例

テーブルfilmsの列titleに一意性のB-treeインデックスを作成します。

```
CREATE UNIQUE INDEX title_idx ON films (title);
```

テーブルfilmsの列titleに、列directorと列ratingを含めて、一意性のB-treeインデックスを作成します。

```
CREATE UNIQUE INDEX title_idx ON films (title) INCLUDE (director, rating);
```

重複排除を無効にしたB-Treeを作成します。

```
CREATE INDEX title_idx ON films (title) WITH (deduplicate_items = off);
```

大文字小文字を区別しない検索が効率的になるように、式lower(title)に対してインデックスを作成します。

```
CREATE INDEX ON films ((lower(title)));
```

(この例では、インデックス名を省略することを選びました。このためシステムがfilms_lower_idxなどという名前を選ぶことになります。)

デフォルト以外の照合順序でインデックスを作成します。

```
CREATE INDEX title_idx_german ON films (title COLLATE "de_DE");
```

デフォルトと異なるNULLのソート順を指定したインデックスを作成します。

```
CREATE INDEX title_idx_nulls_low ON films (title NULLS FIRST);
```

デフォルトと異なるフィルファクタを持つインデックスを作成します。

```
CREATE UNIQUE INDEX title_idx ON films (title) WITH (fillfactor = 70);
```

高速更新を無効にしてGINインデックスを作成します。

```
CREATE INDEX gin_idx ON documents_table USING GIN (locations) WITH (fastupdate = off);
```

テーブルfilms上の列codeに対するインデックスを作成します。また、このインデックスをテーブル空間indexspace内に生成します。

```
CREATE INDEX code_idx ON films (code) TABLESPACE indexspace;
```

変換関数の結果に対するbox操作を効率的に使用できるようにpoint属性にGiSTインデックスを作成します。

```
CREATE INDEX pointloc
    ON points USING gist (box(location,location));
SELECT * FROM points
    WHERE box(location,location) && '(0,0),(1,1)'::box;
```

対象テーブルへの書き込みをロックせずにインデックスを作成します。

```
CREATE INDEX CONCURRENTLY sales_quantity_index ON sales_table (quantity);
```

互換性

CREATE INDEXはPostgreSQLの拡張です。標準SQLにはインデックスについての規定はありません。

関連項目

ALTER INDEX, DROP INDEX, REINDEX

CREATE LANGUAGE

CREATE LANGUAGE — 新しい手続き言語を定義する

概要

```
CREATE [ OR REPLACE ] [ TRUSTED ] [ PROCEDURAL ] LANGUAGE name
    HANDLER call_handler [ INLINE inline_handler ] [ VALIDATOR valfunction ]
CREATE [ OR REPLACE ] [ TRUSTED ] [ PROCEDURAL ] LANGUAGE name
```

説明

CREATE LANGUAGEは新しい手続き言語をPostgreSQLデータベースに登録します。この後で、関数とプロシージャをその新しい言語で定義できるようになります。

CREATE LANGUAGEは、言語名とその言語で作成された関数の実行に責任を持つハンドラ関数を関連付けます。言語ハンドラについての詳細は、第55章を参照してください。

CREATE OR REPLACE LANGUAGEは新しい言語を作成、または、既存の定義を置き換えます。言語がすでに存在する場合、パラメータはコマンドに従って更新されますが、言語の所有権と権限に関する設定は変更されません。また、その言語で作成された既存の関数も依然として有効であるものとみなされます。

新しい言語を登録する、または、既存の言語のパラメータを変更するには、ユーザはPostgreSQLのスーパーユーザ権限を持たなければなりません。しかし、言語が一度作成されれば、その所有権を非スーパーユーザに割り当てたり、そのユーザが削除したり、権限を変更したり、新しい所有者に割り当てたりすることは有効です。(しかしながら、基本的なC関数の所有権を非スーパーユーザに割り当てないでください。それは、 そのユーザに権限昇格パスを作ることになるでしょう。)

ハンドラ関数を指定しない形式のCREATE LANGUAGEは廃止されました。古いダンプファイルとの後方互換性のため、CREATE EXTENSIONと解釈されます。言語が同じ名前の拡張としてパッケージ化されていれば問題なく動作するでしょう、そしてそのようなパッケージ化は手続き言語を設定するのによく行なわれている方法です。

パラメータ

TRUSTED

TRUSTEDは、他の方法ではユーザがアクセスできないデータに対しては、その言語でのアクセスが許されないことを指定します。言語の登録時にこのキーワードを省略すると、PostgreSQLのスーパーユーザ権限を持つユーザのみが、この言語を使って新しい関数を作れるようになります。

PROCEDURAL

これには意味はありません。

name

新しい手続き言語の名前です。この名前はデータベースの言語の中で一意でなければなりません。

後方互換を保持するため、この名前を単一引用符で囲むこともできます。

HANDLER call_handler

call_handlerは手続き言語の関数を実行するために呼び出される関数の名前で、事前に登録しておく必要があります。このハンドラは、Version-1呼び出し規約に則って、C言語のようなコンパイル言語で書かれている必要があります。また、引数を取らずlanguage_handler型を返す関数として、PostgreSQLに登録されていなければなりません。language_handler型は、単に関数を呼び出しハンドラとして識別するのに使用するプレースホルダ型です。

INLINE inline_handler

inline_handlerはこの言語で無名コードブロックを実行（DOコマンド）するために呼び出される、事前に登録された関数の名前です。inline_handler関数が指定されない場合、その言語では無名コードブロックをサポートしません。このハンドラ関数は、DOコマンドの内部表現となるinternal型の引数を1つ取らなければならず、また、通常voidを返します。ハンドラの戻り値は無視されます。

VALIDATOR valfunction

valfunctionは、事前に登録された検証用関数の名前です。新しい関数が当該言語で作成された場合、その関数を検証するために呼び出されます。検証用関数が指定されていない場合、新しい関数は作成時にチェックされません。検証用関数は、oid型の引数を1つ取る必要があります。この引数は作成される関数のOIDになります。また、通常void型を返します。

検証用関数は、通常、関数本体が構文上正しいかどうかを検査するために使用されます。しかし、それだけでなく、関数のプロパティも検査可能です。例えば、その言語が処理できない特定のデータ型が引数に含まれていないかなどがチェックできます。エラーを通知するには、検証用関数でereport()関数を使用すべきです。関数の戻り値は無視されます。

注釈

手続き言語を削除するにはDROP LANGUAGEを使用してください。

システムカタログpg_language（51.29参照）には、現在インストールされている言語に関する情報が記録されています。またpsqlのコマンド\dLはインストールされた言語を一覧表示します。

手続き言語で関数を作成するには、ユーザはその言語に対するUSAGE権限を持たなければなりません。デフォルトでは、信頼された言語についてはPUBLICに（つまり全員に）USAGEが付与されています。これは必要に応じて取り消すことができます。

手続き言語は各データベースに局所的です。しかし、言語をtemplate1データベースにインストールすることができます。この場合、その後に作成されたすべてのデータベースで自動的にその言語は使用できるようになります。

例

新しい手続き言語を作成する最小の手順は以下の通りです。

```
CREATE FUNCTION plsample_call_handler() RETURNS language_handler
    AS '$libdir/plsample'
    LANGUAGE C;
CREATE LANGUAGE plsample
    HANDLER plsample_call_handler;
```

通常、これは拡張の作成スクリプト内に書かれており、ユーザは拡張をインストールすることでこれをすることになるでしょう。

```
CREATE EXTENSION plsample;
```

互換性

CREATE LANGUAGEはPostgreSQLの拡張です。

関連項目

ALTER LANGUAGE, CREATE FUNCTION, DROP LANGUAGE, GRANT, REVOKE

CREATE MATERIALIZED VIEW

CREATE MATERIALIZED VIEW — 新しいマテリアライズドビューを定義する

概要

```
CREATE MATERIALIZED VIEW [ IF NOT EXISTS ] table_name
    [ (column_name [, ...] ) ]
    [ USING method ]
    [ WITH ( storage_parameter [= value] [, ... ] ) ]
    [ TABLESPACE tablespace_name ]
    AS query
    [ WITH [ NO ] DATA ]
```

説明

CREATE MATERIALIZED VIEWは問い合わせからマテリアライズドビューを定義します。この問い合わせはコマンド発行時にビューにデータを投入する（WITH NO DATAが使用されていない場合）ために実行され、使用されます。また将来のREFRESH MATERIALIZED VIEWの使用で更新されるかもしれません。

CREATE MATERIALIZED VIEWはCREATE TABLE ASと似ていますが、必要に応じて後で更新できるように、ビューの初期化時に使用された問い合わせを記憶する点が異なります。マテリアライズドビューはテーブルと同じ属性を多く持ちますが、一時的なマテリアライズドビューをサポートしていません。

パラメータ

IF NOT EXISTS

同じ名前のマテリアライズドビューが既に存在する場合にエラーとしません。この場合、注意が発行されます。既存のマテリアライズドビューが作成されようとしていたものと類似のものであることは全く保証されないことに注意してください。

table_name

作成するマテリアライズドビューの名前（スキーマ修飾可）です。

column_name

新しいマテリアライズドビューの列の名前です。列名が提供されていない場合の列名は、問い合わせの出力列名から取られます。

USING method

この省略可能な句は、新しいマテリアライズドビューの内容を保存するのに使うテーブルアクセスメソッドを指定します。メソッドはTABLE型のアクセスメソッドであることが必要です。より詳

しい情報は第60章を参照してください。このオプションが指定されなければ、新しいマテリアライズドビューに対してはデフォルトのテーブルアクセスメソッドが選ばれます。より詳しい情報はdefault_table_access_methodを参照してください。

`WITH ( storage_parameter [= value] [, ... ] )`

この句は、新しいマテリアライズドビューの格納パラメータ(省略可能)を指定します。詳細についてはCREATE TABLEの文書のStorage Parametersを参照してください。CREATE TABLEでサポートされるすべてのパラメータはCREATE MATERIALIZED VIEWでもサポートされます。詳細についてはCREATE TABLEを参照してください。

`TABLESPACE tablespace_name`

tablespace_nameはマテリアライズドビューが作成されるテーブル空間の名前です。指定されていない場合はdefault_tablespaceを参照します。

`query`

SELECT、TABLEまたはVALUESコマンドです。この問い合わせはセキュリティ限定された操作の中で実行されます。具体的には一時テーブルを作成する関数の呼び出しは失敗します。

`WITH [ NO ] DATA`

この句は作成時にマテリアライズドビューにデータを投入するかどうかを指定します。投入しない場合、マテリアライズドビューはスキャン不可という印が付き、REFRESH MATERIALIZED VIEWが使用されるまで問い合わせることができません。

互換性

CREATE MATERIALIZED VIEWはPostgreSQLの拡張です。

関連項目

ALTER MATERIALIZED VIEW, CREATE TABLE AS, CREATE VIEW, DROP MATERIALIZED VIEW, REFRESH MATERIALIZED VIEW

CREATE OPERATOR

CREATE OPERATOR — 新しい演算子を定義する

概要

```
CREATE OPERATOR name (
    {FUNCTION|PROCEDURE} = function_name
    [, LEFTARG = left_type ] [, RIGHTARG = right_type ]
    [, COMMUTATOR = com_op ] [, NEGATOR = neg_op ]
    [, RESTRICT = res_proc ] [, JOIN = join_proc ]
    [, HASHES ] [, MERGES ]
)
```

説明

CREATE OPERATORは、新しい演算子nameを定義します。演算子を定義したユーザがその所有者となります。スキーマ名が指定されていた場合、その演算子は指定したスキーマに作成されます。スキーマ名が指定されなかった場合、現在のスキーマに作成されます。

演算子名として使用できるのは、以下に示す文字を使った、NAMEDATALEN-1（デフォルトでは63）文字までの文字列です。

```
+ - * / < > = ~ ! @ # % ^ & | ` ?
```

名前の選択には以下に示すいくつかの制約があります。

- --と/*はコメントの開始とみなされてしまうため、演算子名の一部として使うことができません。
- 複数の文字からなる演算子名は、下記の文字のうちの1つ以上を含まない限り、+または-で終わることができません。

  ```
  ~ ! @ # % ^ & | ` ?
  ```

  例えば、@-は演算子名として許されますが、*-は許されません。この制約により、PostgreSQLがSQLに準拠する問い合わせをトークン同士の間に空白を要求することなしに解析することが可能になります。
- 演算子名として=>を使用することは廃止される予定です。将来のリリースで完全に許されなくなるかもしれません。

演算子!=は入力時に<>に変換されるので、これらの2つの名前は常に等価です。

少なくともLEFTARGとRIGHTARGのどちらかは定義しなければなりません。二項演算子では、両方を定義しなければなりません。右単項演算子ではLEFTARGのみ、左単項演算子ではRIGHTARGのみが定義されなければなりません。

> **注記**
>
> 後置演算子とも呼ばれる右単項演算子は廃止予定であり、PostgreSQLバージョン14では削除されます。

function_name関数は、CREATE FUNCTIONを使って事前に定義されていなければなりません。また、指定された型の正しい数の引数(1つか2つ)を受け付けるよう定義する必要があります。

CREATE OPERATORの構文では、キーワードFUNCTIONとPROCEDUREは等価ですが、参照されている関数はどの場合も関数であって、プロシージャであってはなりません。ここで、キーワードPROCEDUREを使用することは歴史的なものであり、廃止予定です。

他の句は演算子最適化用の句(省略可能)です。これらの意味は37.15で説明されています。

演算子を作成するためには、引数の型と戻り値の型に対するUSAGE権限と背後にある関数に対するEXECUTE権限を持たなければなりません。交代演算子または否定子演算子を指定する場合は、これらの演算子を所有していなければなりません。

パラメータ

name

定義される演算子の名前です。使用できる文字は上を参照してください。この名前は、例えばCREATE OPERATOR myschema.+ (...)のように、スキーマ修飾可能です。修飾されていなければ、演算子は現在のスキーマに作成されます。異なるデータ型について処理するものであれば、同じスキーマ内の2つの演算子は同じ名前を持つことができます。これを*オーバーロード*と言います。

function_name

演算子を実装するために使用する関数です。

left_type

演算子の左オペランドのデータ型です(左オペランドが存在する場合のみ)。このオプションは左単項演算子では省略されます。

right_type

演算子の右オペランドのデータ型です(右オペランドが存在する場合のみ)。このオプションは右単項演算子では省略されます。

com_op

この演算子の交代演算子です。

neg_op

この演算子の否定子です。

res_proc

この演算子の制約選択評価関数です。

join_proc

この演算子の結合選択評価関数です。

HASHES

この演算子がハッシュ結合をサポートできることを示します。

MERGES

この演算子がマージ結合をサポートできることを示します。

スキーマ修飾された演算子名をcom_opまたは他のオプション引数に与えるには、以下の例のようにOPERATOR()構文を使用してください。

```
COMMUTATOR = OPERATOR(myschema.===) ,
```

注釈

詳細については37.14を参照してください。

CREATE OPERATORで演算子の語彙優先順位を指定することはできません。パーサの優先順位に関する動作は固定であるためです。詳細な優先順位については4.1.6を参照してください。

廃止されたオプションSORT1、SORT2、LTCMP、およびGTCMPは、マージ結合可能演算子に関連したソート演算子の名前を指定するために使用されていました。代わりにB-tree演算子族を検索することで関連する演算子の情報を見つけることができるようになりましたので、これは必要がなくなりました。これらの内のいずれかのオプションが指定された場合、暗黙的にMERGESを真に設定するだけで、それ以外は無視します。

データベースからユーザ定義の演算子を削除するにはDROP OPERATORを使用してください。データベース内の演算子を変更するにはALTER OPERATORを使用してください。

例

以下のコマンドは、データ型boxに対する領域等価性を判定する新しい演算子を定義します。

```
CREATE OPERATOR === (
    LEFTARG = box,
    RIGHTARG = box,
    FUNCTION = area_equal_function,
    COMMUTATOR = ===,
    NEGATOR = !==,
    RESTRICT = area_restriction_function,
    JOIN = area_join_function,
```

```
    HASHES, MERGES
);
```

互換性

CREATE OPERATORはPostgreSQLの拡張です。標準SQLには、ユーザ定義の演算子についての規定はありません。

関連項目

ALTER OPERATOR, CREATE OPERATOR CLASS, DROP OPERATOR

CREATE OPERATOR CLASS

CREATE OPERATOR CLASS — 新しい演算子クラスを定義する

概要

```
CREATE OPERATOR CLASS name [ DEFAULT ] FOR TYPE data_type
  USING index_method [ FAMILY family_name ] AS
  {  OPERATOR strategy_number operator_name [ ( op_type, op_type ) ] [ FOR SEARCH | FOR ORDER
 BY sort_family_name ]
   | FUNCTION support_number [ ( op_type [ , op_type ] ) ] function_name ( argument_type
 [, ...] )
   | STORAGE storage_type
  } [, ... ]
```

説明

CREATE OPERATOR CLASSは新しい演算子クラスを作成します。演算子クラスは、特定のデータ型がインデックスでどのように使用されるかを定義します。演算子クラスにより、データ型およびインデックスメソッドの特定の役割もしくは「戦略」に、どの演算子を使用するかが指定されます。また、インデックスの列に対して演算子クラスが選択される際、演算子クラスによってインデックスメソッドが使用するサポート関数が指定されます。演算子クラスで使用される全ての演算子および関数は、演算子クラスを作成できるようになる前に定義しておく必要があります。

スキーマ名が与えられている場合、その演算子クラスは指定されたスキーマに作成されます。スキーマ名がなければ、演算子クラスは現在のスキーマに作成されます。異なるインデックスメソッドに使用する場合のみ、同じスキーマ内の2つの演算子クラスに同じ名前を付けることができます。

演算子クラスは、それを定義したユーザが所有者となります。現在、演算子クラスを作成するには、スーパーユーザである必要があります。(誤った演算子クラスを定義すると、サーバを混乱させ、サーバクラッシュの原因とさえなり得るため、この制限が付けられています)。

現在、CREATE OPERATOR CLASSでは、インデックスメソッドに必要な全ての演算子および関数が演算子クラス定義に含まれているかどうか、また、演算子や関数の形式がそれ自身で整合性を持っているかを検査しません。ユーザの責任において、有効な演算子クラスを定義してください。

関連する演算子クラスを*演算子族*にまとめることができます。既存の演算子族に新しい演算子クラスを追加するためには、CREATE OPERATOR CLASSでFAMILYオプションを指定してください。このオプションを指定しないと、新しい演算子クラスはそのクラスと同じ名前の演算子族内に置かれます(この演算子族が存在しない場合は作成します)。

詳細については37.16を参照してください。

パラメータ

name

作成する演算子クラスの名前です。演算子クラス名は、スキーマ修飾することができます。

DEFAULT

DEFAULTを付けると、その演算子クラスが、そのデータ型のデフォルトの演算子クラスになります。特定のデータ型およびインデックスメソッドのデフォルトにできる演算子クラスは最大1つまでです。

data_type

この演算子クラスを使用する列のデータ型です。

index_method

この演算子クラスを使用するインデックスメソッドの名前です。

family_name

この演算子クラスの追加先となる既存の演算子族の名前です。指定されない場合、演算子クラスと同じ名前の演算子族が使用されます（演算子族が存在しない場合は作成します）。

strategy_number

演算子クラスに関連する演算子のインデックスメソッドの戦略番号です。

operator_name

演算子クラスに関連する演算子の名前です（スキーマ修飾名でも可）。

op_type

OPERATOR句では、演算子の入力データ型、もしくは、左単項演算子か右単項演算子を表すNONEを指定します。演算子クラスのデータ型と同じである通常の場合、入力データ型は省略可能です。

FUNCTION句では、関数の入力データ型（B-Tree比較関数およびハッシュ関数用）またはクラスのデータ型（B-treeソートサポート関数、B-tree等価イメージ関数とGiST、SP-GiST、GIN、BRIN演算子クラスのすべての関数用）と異なる場合、関数がサポートする予定の演算対象データ型です。これらのデフォルトは常に正確です。このため、データ型を跨がる比較をサポートする予定のB-treeソートサポート関数は除き、FUNCTION句でop_typeを指定する必要はありません。

sort_family_name

順序付け演算子に関連したソート順序を記述する、既存のbtree演算子族の名前（スキーマ修飾可）です。

FOR SEARCHもFOR ORDER BYも指定されていない場合、FOR SEARCHがデフォルトです。

support_number

演算子クラスに関連する関数用のインデックスメソッドのサポート関数の番号です。

function_name

演算子クラス用のインデックスメソッドのサポート関数となる関数の名前です（スキーマ修飾名でも可）。

argument_type

関数のパラメータのデータ型です。

storage_type

インデックスに実際に格納されるデータ型です。通常、このデータ型は列のデータ型と同じです。しかし、異なるデータ型を許可するインデックスメソッドも存在します（現時点ではGiST、GIN、BRIN）。インデックスメソッドが異なるデータ型の使用を許可していなければ、STORAGE句を指定してはいけません。列data_typeがanyarrayとして指定された場合、storage_typeをanyelementとして宣言し、インデックスのエントリが各インデックスが作成される実際の配列型に属する要素型のメンバーであることを示すことができます。

OPERATOR、FUNCTION、STORAGEは任意の順番で記述できます。

注釈

インデックス機構は、使用する前に関数に関するアクセス権限を検査しませんので、関数や演算子を演算子クラスに含めることは、PUBLICに実行権限を与えることと同じです。通常、演算子クラスで有用な種類の関数ではこれは問題になりません。

演算子はSQL関数で定義してはなりません。SQL関数は呼び出し元の問い合わせにインライン化されることが多いので、オプティマイザでその問い合わせがインデックスに一致するかどうかを認識できなくなってしまうからです。

PostgreSQL 8.4より前までは、OPERATOR句にRECHECKオプションを含めることができました。インデックス演算子に「損失がある」かどうかは実行時にその場で決定されるようになりましたので、これはサポートされなくなりました。これにより、演算子に損失があるかもしれないしないかもしれないような場合を効率的に扱うことができるようになりました。

例

以下のコマンド例では、_int4データ型（int4の配列）のGiSTインデックス演算子クラスを定義しています。この例の詳細については、intarrayモジュールを参照してください。

```
CREATE OPERATOR CLASS gist__int_ops
    DEFAULT FOR TYPE _int4 USING gist AS
        OPERATOR        3       &&,
        OPERATOR        6       = (anyarray, anyarray),
        OPERATOR        7       @>,
        OPERATOR        8       <@,
        OPERATOR        20      @@ (_int4, query_int),
        FUNCTION        1       g_int_consistent (internal, _int4, smallint, oid, internal),
        FUNCTION        2       g_int_union (internal, internal),
```

```
        FUNCTION        3       g_int_compress (internal),
        FUNCTION        4       g_int_decompress (internal),
        FUNCTION        5       g_int_penalty (internal, internal, internal),
        FUNCTION        6       g_int_picksplit (internal, internal),
        FUNCTION        7       g_int_same (_int4, _int4, internal);
```

互換性

CREATE OPERATOR CLASSはPostgreSQLの拡張です。標準SQLにはCREATE OPERATOR CLASS文はありません。

関連項目

ALTER OPERATOR CLASS, DROP OPERATOR CLASS, CREATE OPERATOR FAMILY, ALTER OPERATOR FAMILY

CREATE OPERATOR FAMILY

CREATE OPERATOR FAMILY — 新しい演算子族を定義する

概要

```
CREATE OPERATOR FAMILY name USING index_method
```

説明

CREATE OPERATOR FAMILYは演算子族を新規に作成します。演算子族は、関連する演算子クラスと、場合によっては、これらの演算子クラスと互換性があるが個々のインデックスの機能にとっては重要ではない、追加の演算子と関数の集合を定義します。(インデックスにとって重要な演算子と関数は、演算子族内で「自由」とするのではなく、対応する演算子クラスにまとめられなければなりません。通常、単一のデータ型に対する演算子は演算子クラスにまとめ、データ型を跨る演算子を両方のデータ型に対する演算子族内で自由とします。)

新しい演算子族の初期状態は空です。続いて、含むべき演算子クラスを追加するためにCREATE OPERATOR CLASSコマンドを発行してデータを投入します。必要なら、「自由」な演算子と対応するサポート関数を追加するためにALTER OPERATOR FAMILYコマンドを発行します。

スキーマ名が指定されると、演算子族は指定したスキーマ内に作成されます。さもなくば、現在のスキーマ内に作成されます。対象とするインデックスメソッドが異なる場合に限り、同一スキーマ内に同じ名前の2つの演算子族を持たせることができます。

演算子族を定義したユーザがその所有者となります。現時点では、作成者はスーパーユーザでなければなりません。演算子族を間違って定義すると、サーバが混乱し、クラッシュすることさえありますので、この制限が存在します。

詳細は37.16を参照してください。

パラメータ

name

作成する演算子族の名前です。この名前はスキーマ修飾可能です。

index_method

演算子族が対象とするインデックスメソッドの名前です。

互換性

CREATE OPERATOR FAMILYはPostgreSQLの拡張です。標準SQLにはCREATE OPERATOR FAMILY文はありません。

関連項目

ALTER OPERATOR FAMILY, DROP OPERATOR FAMILY, CREATE OPERATOR CLASS, ALTER OPERATOR CLASS, DROP OPERATOR CLASS

CREATE POLICY

CREATE POLICY — テーブルに新しい行単位のセキュリティポリシーを定義する

概要

```
CREATE POLICY name ON table_name
    [ AS { PERMISSIVE | RESTRICTIVE } ]
    [ FOR { ALL | SELECT | INSERT | UPDATE | DELETE } ]
    [ TO { role_name | PUBLIC | CURRENT_USER | SESSION_USER } [, ...] ]
    [ USING ( using_expression ) ]
    [ WITH CHECK ( check_expression ) ]
```

説明

CREATE POLICYはテーブルに新しい行単位のセキュリティポリシーを定義します。作成したポリシーを適用するには、(ALTER TABLE ... ENABLE ROW LEVEL SECURITYを使って)テーブルの行単位セキュリティを有効にしなければならないことに注意して下さい。

ポリシーは、それを定義する式にマッチした行をselect/insert/update/deleteする権限を与えます。テーブルの既存の行はUSINGで指定した式によって検査されます。INSERTまたはUPDATEによって作成される新しい行はWITH CHECKで指定した式によって検査されます。ある行についてUSING式がtrueを返した場合、その行はユーザに可視となりますが、falseまたはnullを返した場合は不可視となります。行に対してWITH CHECK式がtrueを返した場合、その行は挿入または更新されますが、falseまたはnullを返した場合はエラーが発生します。

INSERT文およびUPDATE文では、BEFOREトリガーが実行された後で、かつ、実際のデータ更新が行われるより前にWITH CHECK式が実行されます。従って、BEFORE ROWトリガーは挿入されるデータを変更する場合があり、これはセキュリティポリシーの検査の結果に影響を与えます。WITH CHECK式は他のいかなる制約よりも前に実行されます。

ポリシー名はテーブル毎につけられます。従って、1つのポリシー名を多くの異なるテーブルに使うことができます。また、その定義は各テーブル毎に異なった、適切な内容にできます。

ポリシーは特定のコマンドまたは特定のロールに対して適用することができます。新しく作成するポリシーのデフォルトは、特に指定しなければ、すべてのコマンドとロールに適用、となっています。複数のポリシーを単一のコマンドに適用できます。更なる詳細は以下を参照ください。表 272に、どのようにして、特定のコマンドに異なるタイプのポリシーが適用されるかがまとめられています。

USING式とWITH CHECK式の両方を持つことができるポリシー(ALLとUPDATE)についてWITH CHECK式が定義されていない場合、どの行が可視であるかの決定(通常のUSINGの対象)と新しい行のどれが追加可能であるかの決定(WITH CHECKの対象)の両方でUSING式が使用されます。

テーブルの行単位セキュリティが有効で、適用可能なポリシーが存在しない場合、「デフォルト拒否」のポリシーが適用され、すべての行が不可視で更新不能になります。

パラメータ

name

作成するポリシーの名前です。同じテーブルの他のポリシーとは異なる名前でなければなりません。

table_name

ポリシーが適用されるテーブルの名前（スキーマ修飾可）です。

PERMISSIVE

作成するポリシーが許容(permissive)ポリシーであることを指定します。問い合わせに適用可能なすべての許容ポリシーは論理演算子「OR」を使って結合されます。許容ポリシーを作成することで、管理者はアクセス可能なレコード集合を追加することができます。デフォルトではポリシーは許容ポリシーです。

RESTRICTIVE

作成するポリシーが制限(restrictive)ポリシーであることを指定します。問い合わせに適用可能なすべての制限ポリシーは論理演算子「AND」と使って結合されます。各行についてすべての制限ポリシーを満たさなければならなくなるため、制限ポリシーを作成することで、管理者はアクセス可能なレコード集合を減らすことができます。

制限ポリシーを有効に使ってアクセスを制限できるようにする前に、レコードへのアクセスを許可する許容ポリシーが少なくとも1つ必要であることに注意してください。制限ポリシーだけしか存在しない場合、レコードにアクセスすることはできません。許容ポリシーと制限ポリシーが混在している場合、少なくとも1つの許容ポリシーを満たし、さらに、すべての制限ポリシーを満たしている場合のみレコードにアクセスできます。

command

ポリシーが適用されるコマンドです。有効なオプションはALL、SELECT、INSERT、UPDATE、DELETEです。デフォルトはALLです。これらがどのように適用されるかの詳細は以下を参照して下さい。

role_name

ポリシーが適用されるロールです。デフォルトはPUBLICで、すべてのロールに対してポリシーが適用されます。

using_expression

任意のSQL条件式（戻り値はboolean）です。条件式に集約関数やウィンドウ関数を含めることはできません。行単位セキュリティが有効なときは、テーブルへの問い合わせにこの式が追加されます。この式がtrueを返す行が可視となります。この式がfalseまたはnullを返す行は、ユーザには（SELECTにおいて）不可視となり、また（UPDATEあるいはDELETEでは）更新の対象ではなくなります。そのような行は静かに無視され、エラーは報告されません。

check_expression

任意のSQL条件式（戻り値はboolean）です。条件式に集約関数やウィンドウ関数を含めることはできません。この式は、そのテーブルに対するINSERTおよびUPDATEの問い合わせで使用され、この式の評価

がtrueになる行のみが許されます。挿入されるレコード、あるいは更新の結果のレコードでこの式の評価がfalseまたはnullになるものについては、エラーが発生します。check_expressionは元の内容ではなく、予定される更新の後の新しい内容に対して評価されることに注意してください。

コマンド毎のポリシー

ALL

ポリシーにALLを使うのは、そのポリシーはコマンドの種類に関係なく、すべてのコマンドに適用されるという意味になります。ALLのポリシーと特定のコマンドに対するポリシーの両方が存在する場合、ALLのポリシーと特定のコマンドに対するポリシーの両方が適用されます。さらにALLのポリシーは、問い合わせの選択側と更新側の両方で適用されます。このとき、USING式だけが定義されていたら、両方の場合についてUSING式を使用します。

例えばUPDATEが実行されるとき、ALLのポリシーは、UPDATEが更新対象の行として選択できる行（USING式が適用されます）と、UPDATE文の結果としてできる行がテーブルに追加できるかどうかの検証（WITH CHECKが定義されていれば、それが適用され、なければUSING式が適用されます）の両方で適用可能です。INSERTまたはUPDATEコマンドがALLのWITH CHECK式に反する行をテーブルに追加しようとした場合、コマンド全体が中止されます。

SELECT

ポリシーにSELECTを使うのは、そのポリシーはSELECTの問い合わせの他に、そのポリシーが定義されているリレーションに対してSELECT権限が必要な時は常に適用されるという意味になります。その結果、SELECT問い合わせでは、SELECTポリシーに適うレコードだけが返されます。また、UPDATEなどSELECT権限が必要な問い合わせでも、SELECTポリシーによって許可されるレコードだけが見えます。SELECTポリシーはリレーションからレコードを取り出す場合にしか適用されないので、WITH CHECK式を持つことはできません。

INSERT

ポリシーにINSERTを使うのは、そのポリシーはINSERTコマンドに適用されるという意味になります。このポリシーに反する行を挿入しようとすると、ポリシー違反エラーを起こし、INSERTコマンド全体が中止されます。INSERTポリシーはリレーションにレコードを追加する場合にしか適用されないため、USING式を持つことはできません。

ON CONFLICT DO UPDATEのあるINSERTでは、INSERTポリシーのWITH CHECK式について、INSERTの部分でリレーションに追加されるすべての行についてのみ確認することに注意してください。

UPDATE

ポリシーにUPDATEを使うのは、そのポリシーはUPDATEコマンド、SELECT FOR UPDATEコマンド、SELECT FOR SHAREコマンド、および補助的にINSERTコマンドのON CONFLICT DO UPDATE句で適用されるという意味になります。UPDATEは既存のレコードを取り出すことと、レコードを新しい修正されたレコードで置換することが含まれるので、UPDATEポリシーはUSING式とWITH CHECK式の両方を受け付けます。USING式はUPDATEコマンドが操作の対象としてどのレコードを見ることができるかを決めるのに使われます。一方でWITH CHECKはどの修正した行をリレーションに戻すことができるかを定義します。

更新後の値がWITH CHECK式に反する行があればエラーを起こし、コマンド全体が中止されます。USING句だけが指定されていた場合は、それがUSINGとWITH CHECKの両方に対して使用されます。

通常は、UPDATEコマンドは更新対象のリレーションの列からデータを読む必要もあります（例えば、WHERE句の中、RETURNING句、あるいはSET句の右辺の式の中）。この場合、更新対象のリレーションのSELECT権限も必要となり、UPDATEポリシーに加えて、適切なSELECTまたはALLポリシーも適用されます。従って、ユーザはUPDATEまたはALLポリシーによって、行を更新する権限を付与されているのに加えて、SELECTまたはALLポリシーによって、更新対象の行にアクセスできなければなりません。

INSERTコマンドに補助的なON CONFLICT DO UPDATE句があり、UPDATEの部分が使われるとき、更新対象の行についてまず、すべてのUPDATEポリシーのUSING式が検査され、次いで、更新された新しい行がWITH CHECK式が検査されます。しかし、単独のUPDATEコマンドとは異なり、既存の行がUSING式を満たさないときは、エラーが発生します（UPDATEの部分が警告なしに回避されることは決してありません）。

DELETE

ポリシーにDELETEを使うのは、そのポリシーはDELETEコマンドに適用されるという意味になります。ポリシーを満たす行だけがDELETEコマンドから見えます。SELECTでは見えるけれど、削除の対象ではない、という行もあり得ます。それらの行がDELETEポリシーのUSING式を満たさない場合です。

ほとんどの場合、DELETEコマンドは削除対象のリレーションの列からデータを読む必要もあります（例えば、WHERE句の中やRETURNING句）。この場合、リレーション上のSELECT権限も必要となり、DELETEポリシーに加えて、適切なSELECTポリシーまたはALLポリシーも適用されます。従って、ユーザはDELETEまたはALLポリシーによって、行を削除する権限を付与されているのに加えて、SELECTまたはALLポリシーによって、削除対象の行にアクセスできなければなりません。

DELETEポリシーはリレーションからレコードが削除される場合にしか適用されず、確認すべき新しい行はないので、WITH CHECK式を持つことはできません。

表272 コマンドタイプにより適用されるポリシー

コマンド	SELECT/ALLポリシー	INSERT/ALLポリシー	UPDATE/ALLポリシー		DELETE/ALLポリシー
	USING式	WITH CHECK式	USING式	WITH CHECK式	USING式
SELECT	既存の行	—	—	—	—
SELECT FOR UPDATE /SHARE	既存の行	—	既存の行	—	—
INSERT	—	新しい行	—	—	—
INSERT ... RETURNING	新しい行 [a]	新しい行	—	—	—
UPDATE	既存の行と新しい行 [a]	—	既存の行	新しい行	—
DELETE	既存の行 [a]	—	—	—	既存の行
ON CONFLICT DO UPDATE	既存の行と新しい行	—	既存の行	新しい行	—

[a] 読み出しアクセスが既存の、あるいは新しい行（たとえば、リレーションのカラムを参照するWHEREあるいはRETURNING句）に要求された場合

複数のポリシーの適用

同じコマンドに対して、異なるコマンド種別の複数のポリシーを適用する場合（例えば、UPDATEコマンドに対してはSELECTとUPDATEのポリシーが適用されます）、ユーザは両方の種別の権限（例えば、リレーションから行を検索する権限と、それを更新する権限）を持っている必要があります。従って、ある種別のポリシーの式は、別の種別のポリシーの式とAND演算子を使って結合されます。

同じコマンドに対して同じコマンド種別の複数のポリシーが適用される場合、リレーションのアクセスを許可する少なくとも1つのPERMISSIVEポリシーがなければならず、さらにすべてのRESTRICTIVEポリシーを満たす必要があります。従って、すべてのPERMISSIVEポリシー式がORを使って結合され、すべてのRESTRICTIVEポリシー式がANDを使って結合され、その結果がANDを使って結合されます。PERMISSIVEポリシーがなければアクセスは拒絶されます。

複数のポリシーを結合するという目的において、ALLのポリシーは適用対象となっている他のすべてのポリシーと同じ種別であるとして扱われることに注意してください。

例えば、SELECTとUPDATEの両方の権限を必要とするUPDATEコマンドにおいて、それぞれの種別の適用可能な複数のポリシーがある場合、以下のように結合されます。

```
expression from RESTRICTIVE SELECT/ALL policy 1
AND
expression from RESTRICTIVE SELECT/ALL policy 2
AND
...
AND
(
  expression from PERMISSIVE SELECT/ALL policy 1
  OR
  expression from PERMISSIVE SELECT/ALL policy 2
  OR
  ...
)
AND
expression from RESTRICTIVE UPDATE/ALL policy 1
AND
expression from RESTRICTIVE UPDATE/ALL policy 2
AND
...
AND
(
  expression from PERMISSIVE UPDATE/ALL policy 1
  OR
  expression from PERMISSIVE UPDATE/ALL policy 2
  OR
  ...
)
```

注釈

ポリシーを作成あるいは変更するには、テーブルの所有者でなければなりません。

ポリシーは、データベース内のテーブルに対する明示的な問い合わせには適用されますが、システムが内部的な参照整合性のチェックや制約の検証をしている時には適用されません。この意味するところは、ある値が存在するかどうかを判定する間接的な方法がある、ということです。その例の1つは、主キーあるいは一意制約のある列に重複する値を挿入しようとすることです。挿入に失敗すれば、その値が既に存在すると推定することができます。(この例では、ユーザが見ることができないレコードを挿入することがポリシーにより許されていると仮定しています。) 別の例は、あるテーブルへの挿入は許されているが、そのテーブルが別の隠されているテーブルを参照している、という場合です。参照元のテーブルに値を挿入することで、値の存在が判断できます。この場合、挿入の成功はその値が参照先のテーブルに存在することを示唆します。これらの問題は、見ることができない値を示唆するかもしれないようなレコードの挿入、削除、更新が全くできないように注意深くポリシーを設計するか、あるいは外部的な意味を持つキーの代わりに生成された値(例:代理キー)を使うことで解決できます。

一般に、システムは問い合わせに記述される制限より前に、セキュリティポリシーを使ったフィルター条件を実行します。これは守られるべきデータが信頼できないかもしれないユーザ定義関数に偶然に意図せずに渡されることを防ぐためです。しかし、システム(またはシステム管理者)によってLEAKPROOFであるとされた関数や演算子については、信頼できるとみなして良いので、ポリシー式より先に評価される場合があります。

ポリシーの式はユーザの問い合わせに直接追加されるため、式は問い合わせ全体を実行しているユーザの権限によって実行されます。そのため、あるポリシーを使用するユーザは、その式が参照しているすべてのテーブルおよび関数にアクセスできる必要があります。そうでなければ、行単位セキュリティが有効になっているテーブルに問い合わせをしようとしたときに、単に権限なしのエラーを受け取ります。しかし、これによってビューの動作が変わることはありません。通常の問い合わせおよびビューと同じく、ビューによって参照されるテーブルに対する権限の確認とポリシーは、ビューの所有者の権限およびビューの所有者に適用されるポリシーを利用します。

更なる詳細と実践的な例については5.8に記述されています。

互換性

CREATE POLICYはPostgreSQLの拡張です。

関連項目

ALTER POLICY, DROP POLICY, ALTER TABLE

CREATE PROCEDURE

CREATE PROCEDURE — 新しいプロシージャを定義する

概要

```
CREATE [ OR REPLACE ] PROCEDURE
    name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ] [, ...] ] )
  { LANGUAGE lang_name
    | TRANSFORM { FOR TYPE type_name } [, ... ]
    | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
    | SET configuration_parameter { TO value | = value | FROM CURRENT }
    | AS 'definition'
    | AS 'obj_file', 'link_symbol'
  } ...
```

説明

CREATE PROCEDUREは新たなプロシージャを定義します。CREATE OR REPLACE PROCEDUREは新たなプロシージャを作るか、既存の定義を置きかえます。プロシージャを定義するにはユーザは言語にUSAGE権限が必要です。

スキーマ名が含まれている場合、プロシージャは指定されたスキーマに作られます。そうでなければ現在のスキーマに作られます。同スキーマ内で新たなプロシージャ名と入力引数型が既存のプロシージャや関数と一致してはなりません。しかしながら、プロシージャや関数が異なる引数型であれば同じ名前を共有できます（これは*オーバーロード*と呼ばれます）。

既存プロシージャの現在の定義を置き換えるには、CREATE OR REPLACE PROCEDUREを使用してください。この方法でプロシージャの名前や引数型を変更することはできません（試みれば、実際は新たな別プロシージャを作ることになるでしょう）。

既存プロシージャの置き換えにCREATE OR REPLACE PROCEDUREが使われたとき、プロシージャの所有者と権限設定は変更されません。その他全てのプロシージャ属性は、コマンドで指定された値または暗黙の値に設定されます。プロシージャを置き換えるには所有者（所有するロールのメンバであることも含みます）でなければなりません。

プロシージャを作ったユーザはプロシージャの所有者になります。

プロシージャを作るには、引数型に対してUSAGE権限を持っていなければなりません。

パラメータ

name

作成するプロシージャ名（スキーマ修飾も可）。

argmode

引数モードで、IN、INOUT、あるいは、VARIADICのいずれかです。省略した場合のデフォルトはINです。（OUT引数は今のところプロシージャに対してサポートされません。INOUTを代わりに使ってください。）

argname

引数名。

argtype

プロシージャ引数があるなら、そのデータ型（スキーマ修飾も可）です。引数型には基本型、複合型、ドメイン型、あるいは、テーブル列の型の参照が使えます。

実装言語によっては、cstringなどの「擬似データ型」を指定することができます。擬似データ型は実際の引数型が完全には特定されていないか、通常のSQLデータ型の枠外にあることを示しています。

列の型は以下のように参照されます。table_name.column_name%TYPE. この機能を使うことは時にプロシージャをテーブル定義の変更から独立させる助けとなります。

default_expr

パラメータが指定されなかった場合のデフォルト値として使用される式です。この式はパラメータの引数型と変換可能でなければなりません。デフォルト値を持つパラメータの後ろにあるパラメータはすべて、同様にデフォルト値を持たなければなりません。

lang_name

プロシージャを実装している言語の名前です。このパラメータには、sql、c、internal、もしくはユーザ定義手続き言語（例：plpgsql）の名前を指定可能です。名前を単一引用符で囲むのは廃止予定で、大文字小文字の一致が必要になります。

TRANSFORM { FOR TYPE type_name } [, ...] }

プロシージャ呼び出しにどの変換を適用すべきかのリストです。変換はSQLの型と言語独自のデータ型の間の変換を行います（CREATE TRANSFORMを参照）。手続言語の実装では、通常、ビルトインの型についてハードコードされた知識があるので、それらをこのリストに含める必要はありません。手続言語の実装が型の処理について定めておらず、変換が提供されない場合は、データ型変換のデフォルトの動作によることになりますが、これは実装に依存します。

[EXTERNAL] SECURITY INVOKER
[EXTERNAL] SECURITY DEFINER

SECURITY INVOKERを指定すると、プロシージャを呼び出したユーザの権限で、そのプロシージャが実行されます。これがデフォルトです。SECURITY DEFINERを指定すると、プロシージャを所有するユーザの権限で、そのプロシージャが実行されます。

EXTERNALキーワードは、SQLとの互換性を保つために許されています。しかし、SQLとは異なり、この機能は外部プロシージャだけではなくすべてのプロシージャに適用されるため、このキーワードは省略可能です。

SECURITY DEFINERプロシージャではトランザクション制御文（言語によりますが例えばCOMMITやROLLBACK）は実行できません。

configuration_parameter
value

SET句により、プロシージャが始まった時に指定した設定パラメータを指定した値に設定し、プロシージャの終了時にそれを以前の値に戻すことができます。SET FROM CURRENTは、CREATE PROCEDUREの実行時点でのパラメータ値を、プロシージャに入る時に適用する値として保管します。

プロシージャにSET句が付いている場合、プロシージャ内部で実行されるSET LOCALコマンドの同一変数に対する効果はそのプロシージャに制限されます。つまり、設定パラメータの前の値はプロシージャが終了する時に元に戻ります。しかし、通常の(LOCALがない)SETコマンドはSET句を上書きします。これは過去に行われたSET LOCALコマンドに対してもほぼ同じです。つまり、このコマンドの効果は、現在のトランザクションがロールバックされない限り、プロシージャが終了した後も永続化されます。

プロシージャにSET句が付いている場合、そのプロシージャではトランザクション制御文(言語によりますが例えばCOMMITとROLLBACK)を実行できません。

使用できるパラメータ名と値についての詳細はSETと第19章を参照してください。

definition

プロシージャを定義する文字列定数です。このパラメータの意味は言語に依存します。内部的なプロシージャ名、オブジェクトファイルへのパス、SQLコマンド、あるいは、手続き言語で記述されたテキストを指定できます。

プロシージャを定義する文字列を記述する際に、通常の単一引用符ではなく、ドル引用符(4.1.2.4参照)を使用すると便利なことが多くあります。ドル引用符を使用しなければ、プロシージャ定義内の単一引用符やバックスラッシュは必ず二重にしてエスケープしなければなりません。

obj_file, link_symbol

この構文のAS句は、動的にロードされるC言語プロシージャにおいて、C言語のソースコード中のプロシージャ名がSQLプロシージャの名前と同じでない場合に使われます。obj_fileという文字列はコンパイルされたCプロシージャを含む共有ライブラリファイルの名前で、LOADコマンドの場合と同じように解釈されます。文字列link_symbolはそのプロシージャのリンクシンボル、つまり、C言語ソースコード中のプロシージャの名前です。リンクシンボルが省略された場合、定義されるSQLプロシージャの名前と同じものであるとみなされます。

同じオブジェクトファイルを参照するCREATE PROCEDURE呼び出しが繰り返される場合、ファイルがセッションにつき一度だけロードされます。(おそらく開発中に)ファイルのアンロードと再ロードを行うには、新たなセッションを開始してください。

注釈

プロシージャにも該当する関数の作成についての詳細はCREATE FUNCTIONを参照してください。

プロシージャを実行するにはCALLを使います。

例

```
CREATE PROCEDURE insert_data(a integer, b integer)
```

```
LANGUAGE SQL
AS $$
INSERT INTO tbl VALUES (a);
INSERT INTO tbl VALUES (b);
$$;

CALL insert_data(1, 2);
```

互換性

CREATE PROCEDUREコマンドはSQL標準で定義されています。PostgreSQLのものは似ていますが、完全な互換ではありません。詳しくはCREATE FUNCTIONも参照してください。

関連項目

ALTER PROCEDURE, DROP PROCEDURE, CALL, CREATE FUNCTION

CREATE PUBLICATION

CREATE PUBLICATION — 新しいパブリケーションを定義する

概要

```
CREATE PUBLICATION name
    [ FOR TABLE [ ONLY ] table_name [ * ] [, ...]
      | FOR ALL TABLES ]
    [ WITH ( publication_parameter [= value] [, ... ] ) ]
```

説明

CREATE PUBLICATIONは現在のデータベースに新しいパブリケーションを追加します。パブリケーションの名前は現在のデータベースに存在するどのパブリケーションの名前とも異なるものでなければなりません。

パブリケーションは本質的にはテーブルの集合で、それらのテーブルのデータの変更が、論理レプリケーションを通じて複製されることが意図されているものです。論理レプリケーションの設定で、パブリケーションがどのように位置づけられるかの詳細については、30.1を参照してください。

パラメータ

name

新しいパブリケーションの名前です。

FOR TABLE

パブリケーションに追加するテーブルのリストを指定します。テーブル名の前にONLYが指定されているときは、そのテーブルだけがパブリケーションに追加されます。ONLYが指定されていないときは、そのテーブルと、そのすべての子テーブル（あれば）が追加されます。オプションで、テーブル名の後に*を指定して、子テーブルが含まれることを明示的に示すことができます。しかしながら、これはパーティションテーブルには適用されません。パーティションテーブルのパーティションは、パブリケーションに含まれると常に暗黙的にみなされますので、パブリケーションに明示的に追加されることは決してありません。

パブリケーションに含めることができるのは、永続的なベーステーブルとパーティションテーブルだけです。一時テーブル、ログを取らないテーブル、外部テーブル、マテリアライズドビュー、通常のビューはパブリケーションに含めることはできません。

パーティションテーブルがパブリケーションに追加された場合、既存のパーティションと将来のものすべてがパブリケーションに含められたと暗黙的にみなされます。ですので、パーティションに対して直接実行された操作であっても、その先祖を含むパブリケーション経由でパブリッシュされます。

FOR ALL TABLES

そのパブリケーションでは、将来作成されるテーブルも含め、そのデータベース内の全テーブルについての変更を複製するものとして印をつけます。

WITH (publication_parameter [= value] [, ...])

この句ではパブリケーションのオプションパラメータを指定します。以下のパラメータがサポートされています。

publish (string)

このパラメータは、新しいパブリケーションがどのDML操作をサブスクライバにパブリッシュするかを指定します。値はカンマで区切られた操作のリストです。使用できる操作はinsert、update、delete、truncateです。デフォルトではすべての動作をパブリッシュするので、このオプションのデフォルト値は'insert, update, delete, truncate'です。

publish_via_partition_root (boolean)

このパラメータは、パブリケーションに含まれるパーティションテーブル(またはそのパーティション)での変更を、実際に変更された個々のパーティションではなく、パーティションテーブルの識別とスキーマを使ってパブリッシュするかどうかを決めます。実際に変更された個々のパーティションのものでパブリッシュされるのがデフォルトです。これを有効にすると、パーティション化されていないテーブルやパーティションの異なる集合からなるパーティションテーブルへ変更を複製できるようになります。

これが有効な場合、パーティションに対して直接実行されたTRUNCATE操作は複製されません。

注釈

FOR TABLEとFOR ALL TABLESのどちらも指定されていない場合、パブリケーションは空のテーブルの集合で作られます。これは後でテーブルを追加したい場合に便利です。

パブリケーションを作るだけでは、レプリケーションは開始されません。これは単に将来のサブスクライバのためにグループとフィルタの論理を定義するだけです。

パブリケーションを作成するには、それを実行するユーザは現在のデータベースにCREATE権限を持っていなければなりません。(もちろん、スーパーユーザにはこの検査は適用されません。)

パブリケーションにテーブルを追加するには、それを実行するユーザがそのテーブルの所有権を持っていなければなりません。FOR ALL TABLES句は、それを実行するユーザがスーパーユーザである必要があります。

UPDATEまたはDELETEをパブリッシュするパブリケーションに追加されるテーブルにはREPLICA IDENTITYが定義されていなければなりません。そうでなければ、それらのテーブルに対して、それらの操作は禁止されることになります。

INSERT ... ON CONFLICTコマンドに対しては、パブリケーションはコマンドの結果として実際に起こった操作をパブリッシュします。従って、その結果に応じてINSERTあるいはUPDATEのいずれかとしてパブリッシュするか、あるいは何もパブリッシュしないかもしれません。

COPY ... FROMコマンドはINSERTの操作としてパブリッシュされます。

DDLの操作はパブリッシュされません。

例

2つのテーブルのすべての変更をパブリッシュするパブリケーションを作成します。

```
CREATE PUBLICATION mypublication FOR TABLE users, departments;
```

すべてのテーブルのすべての変更をパブリッシュするパブリケーションを作成します。

```
CREATE PUBLICATION alltables FOR ALL TABLES;
```

1つのテーブルのINSERTの操作のみをパブリッシュするパブリケーションを作成します。

```
CREATE PUBLICATION insert_only FOR TABLE mydata
    WITH (publish = 'insert');
```

互換性

CREATE PUBLICATIONはPostgreSQLの拡張です。

関連項目

ALTER PUBLICATION, DROP PUBLICATION

CREATE ROLE

CREATE ROLE — 新しいデータベースロールを定義する

概要

```
CREATE ROLE name [ [ WITH ] option [ ... ] ]


ここでoptionは以下の通りです。

      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | REPLICATION | NOREPLICATION
    | BYPASSRLS | NOBYPASSRLS
    | CONNECTION LIMIT connlimit
    | [ ENCRYPTED ] PASSWORD 'password' | PASSWORD NULL
    | VALID UNTIL 'timestamp'
    | IN ROLE role_name [, ...]
    | IN GROUP role_name [, ...]
    | ROLE role_name [, ...]
    | ADMIN role_name [, ...]
    | USER role_name [, ...]
    | SYSID uid
```

説明

CREATE ROLEは、PostgreSQLデータベースクラスタに新しいロールを加えます。ロールとは、データベースオブジェクトを所有することができ、データベース権限を持つことができる実体のことです。ロールは、使用状況に応じて「ユーザ」、「グループ」、もしくは、その両方であるとみなすことができます。ユーザの管理と認証に関する情報については、第21章と第20章を参照してください。このコマンドを使用するには、CREATEROLE権限を持つか、データベースのスーパーユーザでなければなりません。

ロールはデータベースクラスタのレベルで定義されるため、クラスタ内のすべてのデータベースで有効となることに注意してください。

パラメータ

name

新しいロールの名前です。

SUPERUSER
NOSUPERUSER

これらの句によって、新しいロールが「スーパーユーザ」となるかどうかが決まります。「スーパーユーザ」はデータベース内のすべてのアクセス制限より優先します。スーパーユーザという状態は危険ですので、本当に必要な場合にのみ使用しなければなりません。新しくスーパーユーザを作成するには、スーパーユーザでなければなりません。指定されなかった場合のデフォルトはNOSUPERUSERです。

CREATEDB
NOCREATEDB

これらの句はロールのデータベースの作成に関する権限を定義します。CREATEDBが指定された場合、定義されたロールは新しくデータベースを作成することができます。NOCREATEDBを使用した場合、そのロールにはデータベースを作成する権限が与えられません。指定されなかった場合のデフォルトはNOCREATEDBです。

CREATEROLE
NOCREATEROLE

これらの句は、ロールが新しいロールを作成（つまりCREATE ROLEを実行）できるかどうかを決定します。CREATEROLE権限を持つロールはロールを変更することも削除することもできます。指定されなかった場合のデフォルトはNOCREATEROLEです。

INHERIT
NOINHERIT

これらの句は、ロールがそのロールが属するロールの権限を「継承」するかどうかを決定します。INHERIT属性を持つロールは自動的に、直接的にまたは間接的にメンバとして割り当てられたすべてのデータベース権限を使用します。INHERITがないと、他のロール内のメンバ資格により得られる能力はそのロールへのSET ROLEだけです。他のロールの持つ権限は、SET ROLEを行った後にのみ利用可能です。指定されなかった場合のデフォルトはINHERITです。

LOGIN
NOLOGIN

これらの句により、ロールがログイン可能かどうか、つまり、そのロールをクライアント接続時にセッションを認証するための名前として使用することができるかどうかが決まります。LOGIN属性を持つロールはユーザとみなすことができます。この属性を持たないロールは、データベース権限を管理する際に有用ですが、普通の意味ではユーザとはいえません。指定されなかった場合のデフォルトはNOLOGINですが、CREATE ROLEがその別名であるCREATE USERで呼び出された場合は例外です。

REPLICATION
NOREPLICATION

これらの句は、ロールがレプリケーションロールかどうかを決定します。レプリケーションモード（物理または論理レプリケーション）のサーバに接続できるためには、またレプリケーションスロットを作成または削除できるためには、ロールはこの属性（またはスーパーユーザ）を持っている必要があります。REPLICATION属性を持つロールは、非常に高度な権限を持つものです。このため実際にレプリケーションで使用するロールでのみ使用しなければなりません。指定されない場合のデフォルト

はNOREPLICATIONです。REPLICATION属性を持つ新しいロールを作成するにはスーパーユーザでなければなりません。

BYPASSRLS
NOBYPASSRLS

これらの構文は、ロールがすべての行単位セキュリティ(RLS)ポリシーを無視するかどうかを決定します。NOBYPASSRLSがデフォルトです。BYPASSRLS属性を持つ新しいロールを作成するにはスーパーユーザでなければなりません。

pg_dumpはテーブルのすべての内容が確実にダンプされるようにするため、row_securityをデフォルトでOFFに設定することに注意してください。pg_dumpを実行するユーザが適切な権限を持っていなければ、エラーが返されます。しかしながら、スーパーユーザおよびダンプされるテーブルの所有者は、常にRLSを無視します。

CONNECTION LIMIT *connlimit*

ロールがログイン可能である場合、これは、ロールが確立できる最大同時接続数を指定します。-1(デフォルト)は無制限を意味します。通常の接続のみがこの制限の対象として数えられることに注意してください。準備されたトランザクションやバックグラウンドワーカの接続はこの制限の対象にはなりません。

[ENCRYPTED] PASSWORD '*password*'
PASSWORD NULL

ロールのパスワードを設定します。(パスワードはLOGIN属性を持つロールでのみ使用されますが、この属性を持たないロールにも定義することができます。) パスワード認証を行う予定がなければ、このオプションを省略することができます。パスワードの指定がなければ、パスワードがNULLに設定され、そのアカウントでのパスワード認証は常に失敗します。NULLというパスワードを明示的にPASSWORD NULLと記述することもできます。

> **注記**
>
> 空の文字列を指定した場合もパスワードをNULLに設定しますが、PostgreSQLのバージョン10より前はそのようになっていませんでした。以前のバージョンでは、認証方式と細かいバージョンによって空の文字列が使えたり使えなかったりし、また、いずれにせよlibpqは空の文字列を拒絶していました。この曖昧さを避けるためには、空の文字列の指定は避けるのが良いです。

パスワードは必ず暗号化されてシステムカタログに保存されます。ENCRYPTEDキーワードには何の効果もありませんが、後方互換性のために受け付けられます。暗号化の方法は設定パラメータpassword_encryptionによって決定されます。指定されたパスワード文字列が既にMD5またはSCRAMの暗号化形式になっている場合は、password_encryptionと関係なく、そのまま保存されます(指定のパスワードを暗号化した文字列を復号できないので、パスワードを別の形式で暗号化することができないからです)。これにより、ダンプ／リストア時に暗号化したパスワードを再ロードすることができます。

VALID UNTIL '*timestamp*'

VALID UNTIL句は、ロールのパスワードが無効になる日時を設定します。この句が省略された場合、パスワードは永遠に有効になります。

IN ROLE role_name

IN ROLE句には、新しく作成するロールを新規にメンバとして追加する既存の1つ以上のロールを列挙します。(新しく作成するロールを管理者として追加するオプションがないことに注意してください。このためには別途GRANTコマンドを使用してください。)

IN GROUP role_name

IN GROUPはIN ROLEの別名で、廃止予定です。

ROLE role_name

ROLEには、新しく作成するロールのメンバとして自動的に追加する既存の1つ以上のロールを列挙します。(これは実質的に新しく作成したロールを「グループ」とします。)

ADMIN role_name

ADMIN句はROLEと似ていますが、新しく作成されるロールに指定されたロールがWITH ADMIN OPTIONとして追加される点が異なります。つまり、新しく作成されるロールのメンバ資格を他者に与えることができる権利を、指定されたロールに与えます。

USER role_name

USER句はROLE句の別名で廃止予定です。

SYSID uid

SYSID句は無視されますが、後方互換性を維持するために受け付けられます。

注釈

ロールの属性を変更するにはALTER ROLEを、ロールを削除するにはDROP ROLEを使用してください。CREATE ROLEで指定したすべての属性は、後でALTER ROLEコマンドで変更可能です。

グループとして使用しているロールのメンバの追加、および、削除についての推奨方法は、GRANTとREVOKEを使用することです。

VALID UNTIL句は、パスワードの有効期限を定義するだけで、ロール自体の有効期限ではありません。特に、パスワードを使わない認証方式でログインを行う場合には、この有効期限は強制されません。

INHERIT属性は、許可可能な権限(つまり、データベースオブジェクトに対するアクセス権限とロールのメンバ資格)の継承を管理します。これは、CREATE ROLEやALTER ROLEで設定される特別なロール属性には適用されません。INHERITが設定されていたとしても、例えば、CREATEDB権限を持つロールのメンバであっても、データベース作成権限は即座に付与されません。データベースを作成する前にSET ROLEを使用してそのロールにならなければなりません。

後方互換性を維持するため、INHERIT属性はデフォルトです。以前のリリースのPostgreSQLでは、ユーザは常にメンバとなっているすべてのグループの権限でアクセスできました。しかし、NOINHERITの方が標準SQLの規定の意味により合ったものを提供します。

CREATEROLE権限には注意が必要です。CREATEROLE ロールという権限には継承という概念がありません。あるロールが特定の権限を持っていなくても、別のロールを作成できることを意味します。つまり、簡単に自身

の持つ権限と異なる権限（スーパーユーザ権限を持つロールは除きます）を持つ別のロールを作成できてしまいます。たとえば、CREATEROLE権限を持ち、CREATEDB権限を持たない「user」というロールが、CREATEDB権限を持つロールを新規に作成することができます。したがって、CREATEROLE権限を持つロールは、ほとんどスーパーユーザ権限を持つロールと同じであるものと考えてください。

PostgreSQLには、CREATE ROLEと同じ機能を持つプログラム（実際にこのコマンドを呼び出しています）createuserがあり、コマンドシェルから実行することができます。

CONNECTION LIMITオプションが加える制限は厳密ではありません。もしそのロールに1つだけ接続「スロット」が残っていた時に、ほぼ同時に2つのセッションが新しく始まった場合、両方とも失敗する可能性があります。また、この制限はスーパーユーザには適用されません。

このコマンドで暗号化しないパスワードを指定するときには注意しなければなりません。パスワードはサーバに平文で送信されます。クライアントのコマンド履歴やサーバのログにこれが残ってしまうかもしれません。しかし、createuserコマンドはパスワードを暗号化して送信します。また、psqlには\passwordコマンドがあり、これを使用して後でパスワードを安全に変更することができます。

例

ログイン可能なロールを作成します。ただし、パスワードはありません。

```
CREATE ROLE jonathan LOGIN;
```

パスワード付きのロールを作成します。

```
CREATE USER davide WITH PASSWORD 'jw8s0F4';
```

CREATE USERはLOGINを意味する点を除き、CREATE ROLEと同一です。

2004年まで有効なパスワードを持つロールを作成します。2005年に1秒でも入った時点でパスワードは無効になります。

```
CREATE ROLE miriam WITH LOGIN PASSWORD 'jw8s0F4' VALID UNTIL '2005-01-01';
```

データベースを作成でき、かつ、ロールを管理できるロールを作成します。

```
CREATE ROLE admin WITH CREATEDB CREATEROLE;
```

互換性

CREATE ROLE文は標準SQLで規定されています。しかしSQLでは以下の構文のみを要求しています。

```
CREATE ROLE name [ WITH ADMIN role_name ]
```

複数の初期管理者やそのほかのCREATE ROLEのオプションはPostgreSQLの拡張です。

標準SQLでは、ユーザとロールという概念を定義し、それらを別の概念としてみなしています。また、ユーザを定義するコマンドはすべて、各データベース実装で規定するものとしています。PostgreSQLでは、ユーザとロールを単一の実体に統一することを選択しています。したがって、ロールは標準よりも非常に多くのオプションの属性を持っています。

ユーザはNOINHERIT属性を与えること、ロールはINHERIT属性を与えることで、標準SQLで規定された振舞いに最も近くなります。

関連項目

SET ROLE, ALTER ROLE, DROP ROLE, GRANT, REVOKE, createuser

CREATE RULE

CREATE RULE — 新しい書き換えルールを定義する

概要

```
CREATE [ OR REPLACE ] RULE name AS ON event
    TO table_name [ WHERE condition ]
    DO [ ALSO | INSTEAD ] { NOTHING | command | ( command ; command ... ) }


ここでeventは以下のうちの1つです。

    SELECT | INSERT | UPDATE | DELETE
```

説明

CREATE RULEにより、指定したテーブルまたはビューに適用するルールを新しく定義できます。CREATE OR REPLACE RULEを使用すると、新しいルールの作成、または、同じテーブルの同じ名前の既存ルールの置き換えのいずれかを実行します。

PostgreSQLのルールシステムを使用すると、データベーステーブルに対する挿入、更新、削除時に本来の操作の代替として実行するアクションを定義できます。簡単に言えば、指定されたテーブルに対して指定されたコマンドが実行された時、ルールによって追加のコマンドが実行されるということです。その他にも、INSTEADルールによって指定されたコマンドを他のコマンドに置き換えたり、まったくコマンドを実行しないようにすることも可能です。また、ルールはSQLビューを実装するためにも使われます。重要なのは、ルールとは実際にコマンドを変換する仕組み、言い換えれば、コマンドのマクロであることです。ルールによる変換はコマンドの実行前に発生します。各物理行を個別に操作したい場合、ルールではなくトリガを使用する方が良いでしょう。ルールシステムについての詳細は、第40章に記載されています。

現時点では、ON SELECTルールは、条件を持たないINSTEADルールでなければなりません。また、ON SELECTルールが持つアクションは、単独のSELECTコマンドのみで構成される必要があります。したがって、ON SELECTルールを使えば、実質的にテーブルをビューにすることができます。このビューでは、元のテーブルの内容ではなく、ルールに含まれるSELECTコマンドが返す行が、ビューの内容として提示されます。ただし、実テーブルを作成し、それにON SELECTルールを定義するよりは、CREATE VIEWコマンドを使用する方をお勧めします。

ON INSERT、ON UPDATE、ON DELETEルールを必要に応じて定義し、ビューに対する更新操作を他のテーブルに対する適切な更新操作に置換することで、更新可能なビューという実在しないオブジェクトを作成することができます。INSERT RETURNINGなどをサポートしたければ、これらのルールに適切なRETURNING句を確実につけてください。

複雑なビューの更新に条件付きルールを使用しようとする場合、落とし穴があります。そのビューに許可するそれぞれの操作に、条件を持たないINSTEADルールを用意する必要があることです。ルールが条件付き

であったり、INSTEADでない場合、システムは更新操作を拒否します。その場合、システムが、場合によっては、ビューのダミーテーブルに対する操作になる可能性があるとみなすからです。使用する全てのケースについて条件付きルールを作成する場合は、条件を持たないDO INSTEAD NOTHINGルールを追加し、ダミーテーブルに対する更新は呼び出されないことをシステムに明示します。さらに、条件付きルールには、INSTEADと指定しないようにします。これらの条件が満たされた場合、デフォルトのINSTEAD NOTHINGアクションにルールに含まれるアクションが追加されます。(しかし現在、この方法ではRETURNING問い合わせのサポートがうまく動作しません。)

> **注記**
>
> 十分単純であり自動更新可能なビュー(CREATE VIEW参照)は、更新可能とするためにユーザ作成のルールを必要としません。とにかく明示的なルールを作成することもできますが、自動更新による変形は明示的なルールよりも通常高速です。
>
> 検討すべき別の方法は、ルールの代わりにINSTEAD OFトリガ(CREATE TRIGGER参照)を使うことです。

パラメータ

name

作成するルールの名前です。この名前は、同じテーブルの他のルールとは異なる名前にする必要があります。同一テーブルにイベントの種類が同じルールが複数あった場合、ルール名の順番(アルファベット順)に適用されます。

event

イベントとは、SELECT、INSERT、UPDATE、DELETEのいずれかです。ON CONFLICT句を含むINSERTは、INSERTルールまたはUPDATEルールのあるテーブルに対して使えないことに注意してください。代わりに、更新可能ビューを利用することを検討してください。

table_name

ルールを適用するテーブルまたはビューの名前です(スキーマ修飾名も可)。

condition

任意のSQL条件式です(boolean型を返します)。条件式では、NEWおよびOLD以外のテーブルは参照できません。また、集約関数を含めることもできません。

INSTEAD

INSTEADは、元のコマンドの代わりにこのコマンドが実行されることを示します。

ALSO

ALSOは元のコマンドに加えてこのコマンドが実行されることを示します。

ALSOもINSTEADも指定されなかった場合、ALSOがデフォルトです。

command

ルールのアクションを構成する単一または複数のコマンドです。有効なコマンドは、SELECT、INSERT、UPDATE、DELETE、NOTIFYです。

conditionとcommandの内部では、対象とするテーブルの値を参照するために、特別なテーブル名NEWとOLDを使用できます。NEWは、ON INSERTとON UPDATEルールで有効で、挿入または更新される新しい行を参照します。OLDは、ON UPDATEとON DELETEルールで有効で、更新または削除される既存の行を参照します。

注釈

テーブルにルールを定義する、または、そのルールを変更するためには所有者でなければなりません。

ビュー上のINSERT、UPDATE、DELETEルールでは、RETURNING句を追加して、ビューの列を返すことができます。ルールがINSERT RETURNING、UPDATE RETURNING、DELETE RETURNINGコマンドによって呼び出された場合、この句は出力を計算することに使用されます。RETURNINGなしでルールが呼び出された場合、ルールのRETURNING句は無視されます。現在の実装では、無条件のINSTEADルールのみがRETURNINGを含むことができます。さらに、同一イベント用のすべてのルールの中で多くても1つのRETURNING句を持つことができます。（これにより確実に、結果を計算するために使用されるRETURNING句の候補が1つのみになります。）ビュー上のRETURNING問い合わせは、利用可能なルールのどれにもRETURNINGが存在しない場合に拒絶されます。

ルールの循環は絶対に避けるよう注意してください。例えば、下記の2つのルール定義それぞれはPostgreSQLに受け入れられますが、ルールが再帰的に展開されるため、SELECTコマンドが、PostgreSQLにエラーを表示させます。

```
CREATE RULE "_RETURN" AS
    ON SELECT TO t1
    DO INSTEAD
        SELECT * FROM t2;

CREATE RULE "_RETURN" AS
    ON SELECT TO t2
    DO INSTEAD
        SELECT * FROM t1;

SELECT * FROM t1;
```

現在、ルールのアクションにNOTIFYコマンドが含まれる場合、NOTIFYコマンドは無条件に実行されます。つまり、ルールを適用すべき行が存在しなかったとしても、NOTIFYが発行されます。例えば、

```
CREATE RULE notify_me AS ON UPDATE TO mytable DO ALSO NOTIFY mytable;

UPDATE mytable SET name = 'foo' WHERE id = 42;
```

では、id = 42という条件に一致する行があってもなくても、UPDATEの際、1つのNOTIFYイベントが送信されます。これは実装上の制限であり、将来のリリースでは修正されるかもしれません。

互換性

CREATE RULEはPostgreSQLの言語拡張で、問い合わせ書き換えシステム全体が言語拡張です。

関連項目

ALTER RULE, DROP RULE

CREATE SCHEMA

CREATE SCHEMA — 新しいスキーマを定義する

概要

```
CREATE SCHEMA schema_name [ AUTHORIZATION role_specification ] [ schema_element [ ... ] ]
CREATE SCHEMA AUTHORIZATION role_specification [ schema_element [ ... ] ]
CREATE SCHEMA IF NOT EXISTS schema_name [ AUTHORIZATION role_specification ]
CREATE SCHEMA IF NOT EXISTS AUTHORIZATION role_specification


ここでrole_specificationは以下の通りです。

    user_name
  | CURRENT_USER
  | SESSION_USER
```

説明

CREATE SCHEMAを実行すると、現在のデータベースに新しいスキーマが登録されます。スキーマ名は、現在のデータベースにある既存のスキーマとは異なる名前にする必要があります。

スキーマは、本質的には名前空間です。スキーマには、名前付きオブジェクト（テーブル、データ型、関数、および演算子）が含まれます。これらのオブジェクトの名前は、他のスキーマに存在する他のオブジェクトの名前と重複しても構いません。名前付きオブジェクトには、スキーマ名を接頭辞としてオブジェクト名を「修飾」するか、必要なスキーマを含んだ検索パスを設定することによってアクセスできます。修飾なしのオブジェクト名を指定したCREATEコマンドは、そのオブジェクトの現在のスキーマ（current_schema関数で決定される検索パスの先頭部分）で作成されます。

CREATE SCHEMAには、オプションとして、新しいスキーマ内でオブジェクトを作成するためのサブコマンドを付加することができます。サブコマンドは、本質的にはスキーマ作成後に発行される別コマンドと同じように扱われます。ただし、AUTHORIZATION句を使用した場合、作成された全てのオブジェクトの所有者が指定したユーザになるという点で異なっています。

パラメータ

schema_name

作成するスキーマの名前です。省略された場合、user_nameがスキーマ名として使用されます。スキーマ名をpg_から始めることはできません。このような名前はシステムスキーマ用に予約されているためです。

user_name

新しいスキーマを所有するユーザのロール名です。省略された場合、デフォルトでは、コマンドを実行したユーザになります。他のロールを所有者とするスキーマを作成するためには、そのロールの直接的または間接的なメンバであるか、スーパーユーザでなければなりません。

schema_element

そのスキーマ内で作成されるオブジェクトを定義するSQL文です。現在、CREATE SCHEMA内では、CREATE TABLE、CREATE VIEW、CREATE INDEX、CREATE SEQUENCE、CREATE TRIGGER、およびGRANTのみが句として使用可能です。他の種類のオブジェクトは、スキーマ作成後に個別のコマンドを使えば作成できます。

IF NOT EXISTS

同じ名前のスキーマがすでに存在する場合に(注意を発生する以外)何も行いません。このオプションを使用する場合にはschema_element副コマンドを含めることはできません。

注釈

スキーマを作成するには、実行するユーザが現在のデータベースにおけるCREATE権限を持っている必要があります。(もちろん、スーパーユーザにはこの制限はありません。)

例

スキーマを作成します。

```
CREATE SCHEMA myschema;
```

joeユーザ用にスキーマを作成します。このスキーマの名前はjoeになります。

```
CREATE SCHEMA AUTHORIZATION joe;
```

testという名前のスキーマがすでに存在していなければ、joeユーザによって所有されるtestという名前のスキーマを作成します。(joeが既存のスキーマの所有者であるかどうかは関係ありません。)

```
CREATE SCHEMA IF NOT EXISTS test AUTHORIZATION joe;
```

スキーマを作成し、その中にテーブルとビューを作成します。

```
CREATE SCHEMA hollywood
    CREATE TABLE films (title text, release date, awards text[])
    CREATE VIEW winners AS
        SELECT title, release FROM films WHERE awards IS NOT NULL;
```

個々のサブコマンドがセミコロンで終わっていないことに注意してください。

以下は、上述のコマンドと等価であり、同じ結果をもたらします。

```
CREATE SCHEMA hollywood;
CREATE TABLE hollywood.films (title text, release date, awards text[]);
CREATE VIEW hollywood.winners AS
    SELECT title, release FROM hollywood.films WHERE awards IS NOT NULL;
```

互換性

標準SQLでは、CREATE SCHEMAでDEFAULT CHARACTER SET句を使用できます。また、現在PostgreSQLで使用できるよりも多くのサブコマンドを使用できます。

標準SQLでは、CREATE SCHEMAのサブコマンドを任意の順序で記述できます。現在のPostgreSQLの実装では、サブコマンドにおいて下方参照ができない場合があります。そのため、下方参照を避ける目的で、サブコマンドの順序を並べ替える必要が生じる可能性もあります。

標準SQLでは、スキーマの所有者は、常にそのスキーマ内の全てのオブジェクトを所有します。PostgreSQLでは、スキーマ所有者以外のユーザが所有するオブジェクトを、スキーマに含めることができます。このような状態は、スキーマ所有者が、そのスキーマでのCREATE権限を他のユーザに与えた場合やスーパーユーザがその中にオブジェクトを作成した場合にのみ発生します。

IF NOT EXISTSオプションはPostgreSQLの拡張です。

関連項目

ALTER SCHEMA, DROP SCHEMA

CREATE SEQUENCE

CREATE SEQUENCE — 新しいシーケンスジェネレータを定義する

概要

```
CREATE [ TEMPORARY | TEMP ] SEQUENCE [ IF NOT EXISTS ] name
    [ AS data_type ]
    [ INCREMENT [ BY ] increment ]
    [ MINVALUE minvalue | NO MINVALUE ] [ MAXVALUE maxvalue | NO MAXVALUE ]
    [ START [ WITH ] start ] [ CACHE cache ] [ [ NO ] CYCLE ]
    [ OWNED BY { table_name.column_name | NONE } ]
```

説明

CREATE SEQUENCEは、新しいシーケンス番号ジェネレータを作成します。これには、新しくnameという名前を持つ、1行だけの特殊なテーブルの作成と初期化が含まれます。シーケンスジェネレータは、このコマンドを実行したユーザによって所有されます。

スキーマ名が与えられている場合、そのシーケンスは指定されたスキーマに作成されます。スキーマ名がなければ、シーケンスは現在のスキーマに作成されます。また、一時シーケンスは特別なスキーマに存在するため、一時シーケンスの作成時にスキーマ名を与えることはできません。シーケンス名は、同じスキーマ内の他のシーケンス、テーブル、インデックス、ビュー、外部テーブルとは異なる名前にする必要があります。

シーケンスを作成した後、nextval、currval、setval関数を使用してシーケンスを操作します。これらの関数については9.17を参照してください。

シーケンスを直接更新することはできませんが、以下のような問い合わせは可能です。

```
SELECT * FROM name;
```

これを使用すると、シーケンスのパラメータと現在の状態を確認することができます。中でも、シーケンスのlast_valueフィールドは全てのセッションで割り当てられた最後の値を示します（もちろんこの値は、他のセッションのnextvalの実行により、表示された時点で既に最新ではない可能性があります）。

パラメータ

TEMPORARYもしくはTEMP

このパラメータが指定された場合、作成するシーケンスオブジェクトがそのセッション専用となり、セッション終了時に自動的に削除されます。一時シーケンスが存在する場合、同じ名前を持つ既存の永続シーケンスは、スキーマ修飾された名前で参照されない限り、（そのセッションでは）不可視になります。

IF NOT EXISTS

同じ名前のリレーションが既に存在する場合にエラーとしません。この場合、注意が発行されます。既存のリレーションが、作成されようとしていたシーケンスと類似のものであることは全く保証されないことに注意してください。それはシーケンスでさえ、ないかもしれません。

name

作成するシーケンスの名前です（スキーマ修飾名も可）。

data_type

オプション句AS data_typeではシーケンスのデータ型を指定します。有効な型はsmallint、integer、bigintです。bigintがデフォルトです。データ型によりシーケンスのデフォルトの最小値と最大値が決定されます。

increment

INCREMENT BY increment句は、現在のシーケンスの値から新しいシーケンス値を作成する際の値の増加量を設定します。この句は省略可能です。正の値が指定された時は昇順のシーケンス、負の値が指定された時は降順のシーケンスを作成します。指定がない場合のデフォルト値は1です。

minvalue
NO MINVALUE

MINVALUE minvalue句は、シーケンスとして作成する最小値を指定します。この句は省略可能です。この句が指定されなかった場合、もしくは、NO MINVALUEが指定された場合、デフォルトが使用されます。昇順のシーケンスでのデフォルト値は1です。降順のシーケンスでのデフォルト値は、そのデータ型の最小値です。

maxvalue
NO MAXVALUE

MAXVALUE maxvalue句は、シーケンスの最大値を決定します。この句は省略可能です。この句が指定されなかった場合、もしくはNO MAXVALUEが指定された場合、デフォルトが使用されます。昇順のシーケンスでのデフォルト値は、そのデータ型の最大値です。降順のシーケンスでのデフォルト値は-1です。

start

START WITH start句を使用すると、任意の数からシーケンス番号を開始することができます。この句は省略可能です。デフォルトでは、シーケンス番号が始まる値は、昇順の場合minvalue、降順の場合maxvalueになります。

cache

CACHE cacheオプションは、あらかじめ番号を割り当て、メモリに格納しておくシーケンス番号の量を指定します。これによりアクセスを高速にすることができます。最小値は1です（一度に生成する値が1つだけなので、キャッシュがない状態になります）。これがデフォルトになっています。

CYCLE
NO CYCLE

CYCLEオプションを使用すると、シーケンスが限界値（昇順の場合はmaxvalue、降順の場合はminvalue）に達した時、そのシーケンスを周回させることができます。限界値まで達した時、次に生成される番号は、昇順の場合はminvalue、降順の場合はmaxvalueになります。

NO CYCLEが指定された場合、シーケンスの限界値に達した後のnextval呼び出しは全てエラーになります。CYCLEもNO CYCLEも指定されていない場合は、NO CYCLEがデフォルトとなります。

OWNED BY table_name.column_name
OWNED BY NONE

OWNED BYオプションにより、シーケンスは指定されたテーブル列に関連付けられ、その列（やテーブル全体）が削除されると、自動的にシーケンスも一緒に削除されるようになります。指定するテーブルは、シーケンスと同一所有者でなければならず、また、同一のスキーマ内に存在しなければなりません。デフォルトはOWNED BY NONEであり、こうした関連付けがないことを示します。

注釈

シーケンスを削除するにはDROP SEQUENCEを使用してください。

シーケンスはbigint演算に基づいています。そのため、8バイト整数の範囲（-9223372036854775808から9223372036854775807まで）を越えることはできません。

nextvalとsetvalの呼び出しは決してロールバックされないので、シーケンスの番号について「欠番のない」割り当てが必要な場合には、シーケンスオブジェクトを使うことはできません。カウンターを含むテーブルに対して排他ロックを使うことで、欠番のない割り当てを構築することは可能ですが、この解決策はシーケンスオブジェクトに比べてずっと高価で、特に多くのトランザクションが同時にシーケンスの番号を必要とする場合は高価になります。

シーケンスオブジェクトのcacheとして1より大きな値を設定した場合、そのシーケンスを複数のセッションで同時に使用すると、予想外の結果になる可能性があります。各セッションは、シーケンスオブジェクトへの1回のアクセスの間に、連続するシーケンス値を取得し、キャッシュします。そして、キャッシュした数に応じて、シーケンスオブジェクトのlast_valueを増加させます。この場合、そのセッションは、その後のcache-1回に対しては、nextvalを使用してあらかじめ取得済みのシーケンス値を返し、シーケンスオブジェクトを変更しません。セッションに割り当てられたが使用されなかったシーケンス番号は、セッションの終了時に全て失われるため、結果としてシーケンスに「穴」ができます。

さらに、複数のセッションには異なるシーケンス値が割り当てられることが保証されていますが、全てのセッションが尊重されると、シーケンス値が順番通りにならないことがあります。例えば、cacheが10の場合を考えます。セッションAでは1から10までを確保し、nextval=1を返します。セッションBでは、セッションAがnextval=2を返す前に、11から20を確保し、nextval=11を返します。したがって、cacheを1に設定した場合はnextvalが順番に生成される値であると考えても問題ありませんが、cacheを1より大きな値に設定した場合は、nextvalの値が全て異なることのみが保証され、順番に生成される値であることは保証されません。また、last_valueは、値がnextvalによって返されたかどうかに関係なく、いずれかのセッションによって確保された最後の値となります。

この他、このようなシーケンスに対してsetvalが実行されても、他のセッションは、それぞれがキャッシュした取得済みの値を全て使い果たすまで、それがわからないことも考慮すべき問題です。

例

101から始まるserialという名前の昇順シーケンスを作成します。

```
CREATE SEQUENCE serial START 101;
```

このシーケンスから次の番号を選択します。

```
SELECT nextval('serial');

 nextval
---------
     101
```

このシーケンスから次の番号を選択します。

```
SELECT nextval('serial');

 nextval
---------
     102
```

このシーケンスをINSERTコマンドで使用します。

```
INSERT INTO distributors VALUES (nextval('serial'), 'nothing');
```

COPY FROMの後でシーケンス値を更新します。

```
BEGIN;
COPY distributors FROM 'input_file';
SELECT setval('serial', max(id)) FROM distributors;
END;
```

互換性

以下の例外を除き、CREATE SEQUENCEは標準SQLに従います。

- 次の値を取り出すには、標準のNEXT VALUE FOR式ではなくnextval()関数を使用します。
- OWNED BY句はPostgreSQLの拡張です。

関連項目

ALTER SEQUENCE, DROP SEQUENCE

CREATE SERVER

CREATE SERVER — 新しい外部サーバを定義する

概要

```
CREATE SERVER [ IF NOT EXISTS ] server_name [ TYPE 'server_type' ] [ VERSION 'server_version' ]
    FOREIGN DATA WRAPPER fdw_name
    [ OPTIONS ( option 'value' [, ... ] ) ]
```

説明

CREATE SERVERは新しい外部サーバを定義します。サーバを定義したユーザがその所有者となります。

外部サーバは通常、外部データラッパが外部データリソースにアクセスするために使用する接続情報をカプセル化します。さらに、ユーザマップによりユーザ指定の接続情報が指定される可能性があります。

サーバ名はデータベース内で一意でなければなりません。

サーバを作成するには、使用する外部データラッパ上にUSAGE権限が必要です。

パラメータ

IF NOT EXISTS

同じ名前のサーバが既に存在する時にエラーを発生させません。この場合、注意メッセージが発行されます。既存のサーバが、作成されようとしていたものと類似したものであるかどうか、全く保証されないことに注意してください。

server_name

作成する外部サーバの名前です。

server_type

サーバの種類(省略可能)です。外部データラッパで有用かもしれません。

server_version

サーババージョン(省略可能)です。外部データラッパで有用かもしれません。

fdw_name

このサーバを管理する外部データラッパの名前です。

OPTIONS (option 'value' [, ...])

この句はサーバのオプションを指定します。オプションは通常、サーバの接続の詳細を定義しますが、実際の名前とその値はサーバの外部データラッパに依存します。

注釈

dblinkモジュールを使用している場合、接続パラメータを表すために、外部サーバ名をdblink_connect関数の引数として使用することができます。この方法で利用できるようにするためには外部サーバ上にUSAGE権限が必要です。

例

外部データラッパpostgres_fdwを使用するmyserverサーバを作成します。

```
CREATE SERVER myserver FOREIGN DATA WRAPPER postgres_fdw OPTIONS (host 'foo', dbname 'foodb', port
 '5432');
```

詳細についてはpostgres_fdwを参照してください。

互換性

CREATE SERVERはISO/IEC 9075-9 (SQL/MED)に従います。

関連項目

ALTER SERVER, DROP SERVER, CREATE FOREIGN DATA WRAPPER, CREATE FOREIGN TABLE, CREATE USER MAPPING

CREATE STATISTICS

CREATE STATISTICS — 拡張統計情報を定義する

概要

```
CREATE STATISTICS [ IF NOT EXISTS ] statistics_name
    [ ( statistics_kind [, ... ] ) ]
    ON column_name, column_name [, ...]
    FROM table_name
```

説明

CREATE STATISTICSは指定したテーブル、外部テーブル、マテリアライズドビューのデータを追跡する新しい拡張統計オブジェクトを作成します。統計オブジェクトは現在のデータベースに作成され、コマンドを実行したユーザに所有されます。

スキーマ名が指定された場合（例:CREATE STATISTICS myschema.mystat ...）、統計オブジェクトは指定したスキーマ内に作成されます。スキーマ名を指定しなければ、現在のスキーマ内に作成されます。統計オブジェクトの名前は、同じスキーマ内のどの統計オブジェクトとも異なるものでなければなりません。

パラメータ

IF NOT EXISTS

同じ名前の統計オブジェクトが既に存在していてもエラーを発生させません。この場合、注意メッセージが発行されます。この場合、統計オブジェクトの名前だけが問題にされ、その定義の詳細は考慮されないことに注意してください。

statistics_name

作成する統計オブジェクトの名前（オプションでスキーマ修飾可）です。

statistics_kind

この統計オブジェクト内で計算する統計の種別です。現在サポートされる種別は、N個別値統計を有効にするndistinct、関数的依存統計を有効にするdependencies、最頻値の一覧を有効にするmcvです。この句を省略すると、統計オブジェクトのすべてのサポート対象の統計種別が含まれます。より詳細な情報は14.2.2および70.2を参照してください。

column_name

統計計算の対象となるテーブル列の名前です。少なくとも2つの列名を指定しなければなりません。列名の順序は重要ではありません。

table_name

統計情報が計算される列があるテーブルの名前（オプションでスキーマ修飾可）です。

注釈

テーブルを読み取る統計オブジェクトを作るには、そのテーブルの所有者でなければなりません。しかし、統計オブジェクトが作成された後は、その所有者と対象となるテーブルは無関係になります。

例

関数従属性のある2つの列を含むテーブルt1を作成します。つまり、第1の列の値を知っていれば、それだけでもう一方の列の値がわかる、というものです。その次に、これらの列の間に関数的依存統計を構築します。

```
CREATE TABLE t1 (
    a   int,
    b   int
);

INSERT INTO t1 SELECT i/100, i/500
                FROM generate_series(1,1000000) s(i);

ANALYZE t1;


-- マッチする行の数は非常に低く見積もられる
EXPLAIN ANALYZE SELECT * FROM t1 WHERE (a = 1) AND (b = 0);

CREATE STATISTICS s1 (dependencies) ON a, b FROM t1;

ANALYZE t1;


-- 行数の見積もりがより正確になる
EXPLAIN ANALYZE SELECT * FROM t1 WHERE (a = 1) AND (b = 0);
```

関数的依存統計がなければ、プランナは2つのWHERE条件を独立なものとみなすため、それらの選択性を掛け算して、非常に小さな行数見積もりを導きます。このような統計があれば、プランナはWHERE条件が冗長であることを認識し、行数を低く見積もりません。

(同一のデータの入った)完全に相関のある2つの列を持つテーブルt2を作成し、2つの列の最頻値(MCV)の一覧を作成します。

```
CREATE TABLE t2 (
    a   int,
    b   int
```

```
);

INSERT INTO t2 SELECT mod(i,100), mod(i,100)
                  FROM generate_series(1,1000000) s(i);

CREATE STATISTICS s2 (mcv) ON a, b FROM t2;

ANALYZE t2;


-- 有効な組み合わせ(MCV内で見つかる)
EXPLAIN ANALYZE SELECT * FROM t2 WHERE (a = 1) AND (b = 1);




-- 無効な組み合わせ(MCV内で見つからない)
EXPLAIN ANALYZE SELECT * FROM t2 WHERE (a = 1) AND (b = 2);
```

最頻値の一覧は、テーブルによく現れる特定の値に関するものだけでなく、テーブルに現れない値の組み合わせの選択の上限に関するより詳細な情報もプランナに与えますので、両方の場合に対してより良く見積もりができるようになります。

互換性

標準SQLにCREATE STATISTICSコマンドはありません。

関連項目

ALTER STATISTICS, DROP STATISTICS

CREATE SUBSCRIPTION

CREATE SUBSCRIPTION — 新しいサブスクリプションを定義する

概要

```
CREATE SUBSCRIPTION subscription_name
    CONNECTION 'conninfo'
    PUBLICATION publication_name [, ...]
    [ WITH ( subscription_parameter [= value] [, ... ] ) ]
```

説明

CREATE SUBSCRIPTIONは現在のデータベースに新しいサブスクリプションを追加します。サブスクリプションの名前は現在のデータベースに存在するどのサブスクリプションの名前とも異なるものでなければなりません。

サブスクリプションはパブリッシャーへのレプリケーション接続を表します。そのためこのコマンドはローカルのカタログに定義を追加するだけでなく、パブリッシャーのレプリケーションスロットも作成します。

このコマンドが実行されるトランザクションがコミットされた時点で、新しいサブスクリプションに対してデータを複製する論理レプリケーションワーカが開始されます。

サブスクリプションおよび論理レプリケーションの全体像についての追加情報は30.2および第30章に記述されています。

パラメータ

subscription_name

新しいサブスクリプションの名前です。

CONNECTION 'conninfo'

パブリッシャーへの接続文字列です。詳細は33.1.1を参照してください。

PUBLICATION publication_name

パブリッシャー上のパブリケーションで、サブスクリプションの対象となるものの名前です。

WITH (subscription_parameter [= value] [, ...])

この句ではサブスクリプションのオプションパラメータを指定します。以下のパラメータがサポートされています。

copy_data (boolean)

サブスクリプションの対象となるパブリケーションの既存データが、レプリケーションの開始時にコピーされるかどうかを指定します。デフォルトはtrueです。

create_slot (boolean)

このコマンドがパブリッシャー上にレプリケーションスロットを作るかどうかを指定します。デフォルトはtrueです。

enabled (boolean)

サブスクリプションが複製の動作をすぐに行うか、あるいは単に設定をするだけでまだ開始しないかを指定します。デフォルトはtrueです。

slot_name (string)

使用するレプリケーションスロットの名前です。デフォルトの挙動では、サブスクリプションの名前をスロット名として使用します。

slot_nameをNONEに設定すると、サブスクリプションに紐付けられたレプリケーションスロットがなくなります。これはレプリケーションスロットを後で手作業で作成する場合に使用できます。そのようなサブスクリプションは、enabledとcreate_slotの両方をfalseに設定しなければなりません。

synchronous_commit (enum)

このパラメータの値はsynchronous_commitの設定をオーバーライドします。デフォルト値はoffです。

論理レプリケーションではoffを使用するのが安全です。そうすることで、同期の失敗によりサブスクライバがトランザクションを失った場合でも、パブリッシャーからデータが再送されます。

同期論理レプリケーションを行う場合は別の設定が適切かもしれません。論理レプリケーションのワーカは書き込みおよび吐き出しの位置をパブリッシャーに報告しますが、同期レプリケーションを行っているときは、パブリッシャは実際に吐き出しがされるのを待ちます。これはつまり、サブスクリプションが同期レプリケーションで使われている時に、サブスクライバのsynchronous_commitをoffに設定すると、パブリッシャーでのCOMMITの遅延が増大するかもしれない、ということを意味します。この場合、synchronous_commitをlocalまたはそれ以上に設定することが有利になりえます。

connect (boolean)

CREATE SUBSCRIPTIONがパブリッシャーに接続するかどうかを指定します。これをfalseに設定すると、enabled、create_slot、copy_dataのデフォルト値をfalseに変更します。

connectをfalseに設定し、enabled、create_slotまたはcopy_dataをtrueに設定することは許されません。

このオプションがfalseに設定されると接続が行われないため、テーブルはサブスクライブされません。そのため、サブスクリプションを有効にしても、何も複製されません。テーブルをサブスクライブするには、ALTER SUBSCRIPTION ... REFRESH PUBLICATIONを実行する必要があります。

注釈

サブスクリプションとパブリケーションのインスタンスの間のアクセス制御をどのように設定するかの詳細については、30.7を参照してください。

レプリケーションスロットを作成する(デフォルトの動作です)場合、CREATE SUBSCRIPTIONをトランザクションブロックの内側で実行することはできません。

同じデータベースクラスタに接続するサブスクリプション(例えば、同一のクラスタ内のデータベース間で複製を行う、あるいは同一のデータベース内で複製を行う)の作成は、同じコマンド内でレプリケーションスロットが作成されない場合にのみ成功します。そうでない場合、CREATE SUBSCRIPTIONの呼び出しはハングアップします。これを動作させるには、(関数pg_create_logical_replication_slotをプラグイン名pgoutputで使って)レプリケーションスロットを別に作り、パラメータcreate_slot = falseを使ってサブスクリプションを作成してください。これは実装上の制限で、将来のリリースでは解決されるかもしれません。

例

パブリケーションmypublicationおよびinsert_onlyのテーブルを複製する、リモートサーバへのサブスクリプションを作成し、コミット後、すぐにレプリケーションを開始します。

```
CREATE SUBSCRIPTION mysub
         CONNECTION 'host=192.168.1.50 port=5432 user=foo dbname=foodb'
        PUBLICATION mypublication, insert_only;
```

パブリケーションinsert_onlyのテーブルを複製するリモートサーバへのサブスクリプションを作成しますが、後に有効化するまではレプリケーションを開始しません。

```
CREATE SUBSCRIPTION mysub
         CONNECTION 'host=192.168.1.50 port=5432 user=foo dbname=foodb'
        PUBLICATION insert_only
               WITH (enabled = false);
```

互換性

CREATE SUBSCRIPTIONはPostgreSQLの拡張です。

関連項目

ALTER SUBSCRIPTION, DROP SUBSCRIPTION, CREATE PUBLICATION, ALTER PUBLICATION

CREATE TABLE

CREATE TABLE — 新しいテーブルを定義する

概要

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE [ IF NOT EXISTS ] table_name
 ( [
  { column_name data_type [ COLLATE collation ] [ column_constraint [ ... ] ]
    | table_constraint
    | LIKE source_table [ like_option ... ] }
    [, ... ]
] )
[ INHERITS ( parent_table [, ... ] ) ]
[ PARTITION BY { RANGE | LIST | HASH } ( { column_name | ( expression ) } [ COLLATE collation ]
 [ opclass ] [, ... ] ) ]
[ USING method ]
[ WITH ( storage_parameter [= value] [, ... ] ) | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE tablespace_name ]

CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE [ IF NOT EXISTS ] table_name
    OF type_name [ (
  { column_name [ WITH OPTIONS ] [ column_constraint [ ... ] ]
    | table_constraint }
    [, ... ]
) ]
[ PARTITION BY { RANGE | LIST | HASH } ( { column_name | ( expression ) } [ COLLATE collation ]
 [ opclass ] [, ... ] ) ]
[ USING method ]
[ WITH ( storage_parameter [= value] [, ... ] ) | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE tablespace_name ]

CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE [ IF NOT EXISTS ] table_name
    PARTITION OF parent_table [ (
  { column_name [ WITH OPTIONS ] [ column_constraint [ ... ] ]
    | table_constraint }
    [, ... ]
) ] { FOR VALUES partition_bound_spec | DEFAULT }
[ PARTITION BY { RANGE | LIST | HASH } ( { column_name | ( expression ) } [ COLLATE collation ]
 [ opclass ] [, ... ] ) ]
[ USING method ]
[ WITH ( storage_parameter [= value] [, ... ] ) | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
```

```
[ TABLESPACE tablespace_name ]


ここでcolumn_constraintには、
      次の構文が入ります。

[ CONSTRAINT constraint_name ]
{ NOT NULL |
  NULL |
  CHECK ( expression ) [ NO INHERIT ] |
  DEFAULT default_expr |
  GENERATED ALWAYS AS ( generation_expr ) STORED |
  GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY [ ( sequence_options ) ] |
  UNIQUE index_parameters |
  PRIMARY KEY index_parameters |
  REFERENCES reftable [ ( refcolumn ) ] [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ]
    [ ON DELETE referential_action ] [ ON UPDATE referential_action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]


また、
      table_constraintには、
      次の構文が入ります。

[ CONSTRAINT constraint_name ]
{ CHECK ( expression ) [ NO INHERIT ] |
  UNIQUE ( column_name [, ... ] ) index_parameters |
  PRIMARY KEY ( column_name [, ... ] ) index_parameters |
  EXCLUDE [ USING index_method ] ( exclude_element WITH operator [, ... ] ) index_parameters
 [ WHERE ( predicate ) ] |
  FOREIGN KEY ( column_name [, ... ] ) REFERENCES reftable [ ( refcolumn [, ... ] ) ]
    [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ] [ ON DELETE referential_action ] [ ON
 UPDATE referential_action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]


またlike_optionは、
      以下の通りです。

{ INCLUDING | EXCLUDING } { COMMENTS | CONSTRAINTS | DEFAULTS | GENERATED | IDENTITY | INDEXES |
 STATISTICS | STORAGE | ALL }


またpartition_bound_specは、
      以下の通りです。
```

```
IN ( partition_bound_expr [, ...] ) |
FROM ( { partition_bound_expr | MINVALUE | MAXVALUE } [, ...] )
  TO ( { partition_bound_expr | MINVALUE | MAXVALUE } [, ...] ) |
WITH ( MODULUS numeric_literal, REMAINDER numeric_literal )


UNIQUE、
      PRIMARY KEYおよびEXCLUDE制約内のindex_parametersは以下の通りです。

[ INCLUDE ( column_name [, ... ] ) ]
[ WITH ( storage_parameter [= value] [, ... ] ) ]
[ USING INDEX TABLESPACE tablespace_name ]


EXCLUDE制約内のexclude_elementは以下の通りです。

{ column_name | ( expression ) } [ opclass ] [ ASC | DESC ] [ NULLS { FIRST | LAST } ]
```

説明

CREATE TABLEは、現在のデータベースに新しい空のテーブルを作成します。作成したテーブルはこのコマンドを実行したユーザが所有します。

スキーマ名が付けられている場合（例えば、CREATE TABLE myschema.mytable ...）、テーブルは指定されたスキーマに作成されます。スキーマ名がなければ、テーブルは現在のスキーマに作成されます。また、一時テーブルは特別なスキーマに存在するため、一時テーブルの作成時にスキーマ名を与えることはできません。テーブル名は、同じスキーマ内の他のテーブル、シーケンス、インデックス、ビュー、外部テーブルとは異なる名前にする必要があります。

さらに、CREATE TABLEは、作成するテーブルの1行に対応する複合型のデータ型を作成します。したがって、テーブルは、同じスキーマ内の既存のデータ型と同じ名前を持つことができません。

制約句には、挿入、更新操作を行うときに、新しい行、または更新する行が満たさなければならない制約（検査項目）を指定します。制約句は省略可能です。制約は、テーブル内で有効な値の集合を様々な方法で定義できるSQLオブジェクトです。

制約の定義にはテーブル制約と列制約という2種類があります。列制約は列定義の一部として定義されます。テーブル制約定義は、特定の列とは結びつけられておらず、複数の列を含有することができます。また、全ての列制約はテーブル制約として記述することができます。列制約は、1つの列にのみ影響する制約のための、簡便な記述方法に過ぎません。

テーブルを作成するためには、すべての列の型またはOF句中の型に対するUSAGE権限を持たなければなりません。

パラメータ

TEMPORARYまたはTEMP

このパラメータが指定された場合、テーブルは一時テーブルとして作成されます。一時テーブルは、そのセッションの終わり、場合によっては、現在のトランザクションの終わり(後述のON COMMITを参照)に自動的に削除されます。一時テーブルが存在する場合、同じ名前を持つ既存の永続テーブルは、スキーマ修飾名で参照されていない限り、現在のセッションでは非可視になります。一時テーブルに作られるインデックスも、全て自動的に一時的なものとなります。

自動バキュームデーモンは一時テーブルにアクセスできないため、一時テーブルのバキュームや解析を行うことはできません。このためセッションのSQLコマンドを用いて適切なバキュームと解析を実行しなければなりません。例えば、一時テーブルが複雑な問い合わせで使用される場合、一時テーブルにデータを投入した後にそれに対しANALYZEを実行することを勧めます。

オプションで、GLOBALまたはLOCALをTEMPORARYやTEMPの前に記述することができます。PostgreSQLでは、現在違いがなく、廃止予定です。Compatibilityを参照してください。

UNLOGGED

指定された場合、テーブルはログを取らないテーブルとして作成されます。ログを取らないテーブルに書き出されたデータは先行書き込みログ(第29章参照)には書き出されません。このため通常のテーブルより相当高速になります。しかしこれらはクラッシュ時に安全ではありません。クラッシュまたは異常停止の後、ログを取らないテーブルは自動的に切り詰められます。またログを取らないテーブルの内容はスタンバイサーバにコピーされません。ログを取らないテーブル上に作成されたインデックスはすべて同様に、ログを取らないようになります。

IF NOT EXISTS

同じ名前のリレーションがすでに存在していてもエラーとしません。この場合注意が発せられます。既存のリレーションが作成しようとしたものと何かしら似たものであることは保証されません。

table_name

作成するテーブルの名前です(スキーマ修飾名でも可)。

OF type_name

指定した複合型(スキーマ修飾可能)から構造を取り出した型付きテーブルを作成します。型付きテーブルはその型に束縛されます。例えば、型が(DROP TYPE ... CASCADEで)削除されるとそのテーブルは削除されます。

型付きテーブルが作成されると、その列のデータ型は背後の複合型により決定され、CREATE TABLEコマンドでは指定されません。しかしCREATE TABLEコマンドではテーブルにデフォルトと制約を追加できます。また、格納パラメータの指定も可能です。

column_name

新しいテーブルで作成される列の名前です。

data_type

列のデータ型です。これには、配列指定子を含めることができます。PostgreSQLでサポートされるデータ型の情報に関する詳細は第8章を参照してください。

COLLATE collation

COLLATE句は列（照合順の設定が可能なデータ型でなければなりません）に照合順を割り当てます。指定がなければ、列のデータ型のデフォルトの照合順が使用されます。

INHERITS (parent_table [, ...])

オプションのINHERITS句でテーブルの一覧を指定すると、新しいテーブルは指定されたテーブルの全ての列を自動的に継承します。親テーブルには通常のテーブルまたは外部テーブルを指定できます。

INHERITSを使用すると、新しい子テーブルとその親テーブル（複数可）との間に永続的な関連が作成されます。通常、親へのスキーマ変更は子にも伝播します。また、デフォルトでは、親テーブルの走査結果には子テーブルのデータが含まれます。

複数の親テーブルに同一名の列が存在する場合、それらのデータ型が一致していなければ、エラーとして報告されます。競合がなければ、これらの重複した列は新しいテーブルで1つの列の形に融合されます。新しいテーブルの列名の一覧に継承する列の名前が含まれる場合も、そのデータ型は継承する列のデータ型と一致していなければなりません。さらに、その列定義は1つに融合されます。新しいテーブルで明示的に列のデフォルト値を指定した場合、継承した列宣言における全てのデフォルト値は上書きされます。デフォルト値を指定しなかった場合、親側でデフォルト値が指定されている時は、それらのデフォルト値が全て同じ値でなければなりません。値が違う場合はエラーになります。

CHECK制約は、基本的には列と同様の方法でマージされます。複数の親テーブル、新しいテーブル、またはその両方の定義に同じ名前のCHECK制約が存在した場合、これらの制約はすべて同じ検査式を持たなければなりません。さもなくば、エラーが報告されます。同じ名前と式を持つ制約は1つのコピーにまとめられます。親テーブルでNO INHERITと印が付いた制約は考慮されません。新しいテーブル内の無名のCHECK制約は、一意な名前が必ず作られるため、マージされないことに注意してください。

列のSTORAGE設定もまた親テーブルからコピーされます。

親テーブルのある列がIDENTITY列の場合、その属性は継承されません。望むなら子テーブルの列をIDENTITY列と宣言することができます。

PARTITION BY { RANGE | LIST | HASH } ({ column_name | (expression) } [opclass] [, ...])

オプションのPARTITION BY句により、テーブルのパーティショニングの戦略を指定できます。このようにして作られたテーブルを*パーティション*テーブルと呼びます。括弧に囲まれた列や式のリストはテーブルの*パーティションキー*を構成します。範囲パーティションを使うときは、パーティションキーは複数の列または式にまたがることができます（最大で32ですが、この制限はPostgreSQLをビルドする時に変更できます）が、リストパーティションでは、パーティションキーは1つだけの列または式で構成されなければなりません。パーティションテーブルの作成時にBツリー演算子クラスを指定しない場合は、そのデータ型のデフォルトのBツリー演算子クラスが使用されます。Bツリー演算子クラスがない場合はエラーが報告されます。

範囲とリストのパーティショニングはBツリー演算子クラスを必要とし、ハッシュパーティションニングはハッシュ演算子クラスを必要とします。演算子クラスが明示的に指定されない場合、適当な型のデフォルトの演算子クラスが使われます。デフォルト演算子クラスがなければエラーが発生します。ハッシュパーティショニングが使われているとき、使われる演算子クラスはサポート関数2を実装していなければなりません（詳しくは37.16.3を参照）。

パーティションテーブルは（パーティションと呼ばれる）副テーブルに分割され、それらは別のCREATE TABLEコマンドにより作成されます。パーティションテーブルそれ自体は空になります。テーブルに挿入されるデータ行は、パーティションキーの列あるいは式の値に基づいて、1つのパーティションに回されます。新しい行の値に適合するパーティションが存在しないときは、エラーが報告されます。

パーティションテーブルはEXCLUDE制約をサポートしません。しかしながら、個々のパーティションにこの制約を定義することはできます。

テーブルパーティショニングに関するより詳しい議論は5.11を参照してください。

```
PARTITION OF parent_table { FOR VALUES partition_bound_spec | DEFAULT }
```

指定した親テーブルの*パーティション*としてテーブルを作成します。FOR VALUESを用いて特定の値のパーティションとして、あるいは、DEFAULTを用いてデフォルトパーティションとしてテーブルを作成できます。親テーブルにあるインデックス、制約、ユーザ定義の行レベルのトリガは新しいパーティションに複製されます。

*partition_bound_spec*は親テーブルのパーティショニング方法とパーティションキーに対応していなければならず、またそのテーブルのどの既存のパーティションとも重なり合ってはいけません。INの構文はリストパーティショニングで、FROMとTOの構文は範囲パーティショニングで、WITHの構文はハッシュパーティショニングで、使用されます。

*partition_bound_expr*は任意の無変数式です（サブクエリ、ウィンドウ関数、集約関数、複数行を返す関数は許されません）。式のデータ型は対応するパーティションキー列と一致しなければなりません。式はテーブル作成時に一度だけ評価されるため、CURRENT_TIMESTAMPなどの揮発性の式を含むことも可能です。

リストパーティションを作るときは、NULLを指定することができて、それはそのパーティションではパーティションキーの列をNULLにすることができるということを意味します。しかし、1つの親テーブルで2つ以上、そのようなリストパーティションを作ることはできません。範囲パーティションではNULLを指定することはできません。

範囲パーティションを作るとき、FROMで指定する下限はそれを含む境界、TOで指定する上限はそれを含まない境界になります。つまり、FROMリストで指定される値は、そのパーティションの対応するパーティションキー列において有効な値ですが、TOリストで指定される値はそうではない、ということです。この文の意味は行単位の比較の規則（9.24.5）に従って理解しなければならないことに注意してください。例えば、PARTITION BY RANGE (x,y)について、パーティション境界FROM (1, 2) TO (3, 4)には、x=1でy>=2の任意の値のもの、x=2でNULLでない任意のyのもの、x=3でy<4の任意の値のものが入ります。

範囲パーティションを作るとき、MINVALUEおよびMAXVALUEという特別な値を使用することができて、これらはそれぞれ列の値に下限と上限がないことを示します。例えば、FROM (MINVALUE) TO (10)で定義されたパーティションには10より小さいすべての値が入り、FROM (10) TO (MAXVALUE)で定義されたパーティションには10以上のすべての値が入ります。

2つ以上の列を含む範囲パーティションを作るとき、MAXVALUEを下限の一部として使うことや、MINVALUEを上限の一部として使うことも意味を持ちえます。例えば、FROM (0, MAXVALUE) TO (10, MAXVALUE)で定義されたパーティションには、パーティションキーの第1列が0より大きく、かつ10以下であるものが入ります。同様に、FROM ('a', MINVALUE) TO ('b', MINVALUE)で定義されたパーティションには、パーティションキーの第1列が"a"で始まるすべての行が入ります。

MINVALUEまたはMAXVALUEをパーティション境界の1つの列で使用する場合、それより後の列では同じ値を使用しなければならないことに注意してください。例えば、(10, MINVALUE, 0)は有効な境界ではありません。(10, MINVALUE, MINVALUE)とします。

timestamp,など一部の要素型では、"infinity"(無限)の概念があり、それも保存できる値であることにも注意してください。MINVALUEとMAXVALUEは保存できる真の値ではなく、値に境界がないということを表現するための方法に過ぎないため、これとは違います。MAXVALUEは"infinity"も含め、他のすべての値より大きいものと考えることができ、またMINVALUEは"minus infinity"も含め、他のすべての値より小さいものと考えることができます。従って、境界FROM ('infinity') TO (MAXVALUE)は空の範囲ではなく、たった1つの値、つまり"infinity"だけを保存します。

DEFAULTが指定された場合、テーブルは親テーブルのデフォルトパーティションとして作成されます。本オプションはハッシュパーティションされたテーブルには使用できません。親の他のどのパーティションにも当てはまらないパーティションキー値はデフォルトパーティションに送られます。

テーブルが既存のDEFAULTパーティションを持っていて、新たなパーティションが追加された場合、デフォルトパーティションは、新たなパーティションに属すのがふさわしい行が含まれていないことを確かめるために、検査されなければなりません。デフォルトパーティションに多数の行が含まれている場合、これは時間を要すかもしれません。デフォルトパーティションが、外部テーブルであるか、新パーティションに置くべき行を含むことができないことを証明する制約を持つ場合、この検査は省略されます。

ハッシュパーティションを作るときには法と残余を指定しなければなりません。法は正の整数でなければならず、残余は法よりも小さい非負整数でなければなりません。典型的にはハッシュパーティションテーブル初期設定をするとき、パーティションの数と等しい法を選び、全てのテーブルに同じ法と異なる残余を割り当てます(後述の例を参照)。しかしながら、全てのパーティションが同じ法を持つ必要はなく、あるハッシュパーティションテーブルのパーティションに存在する全ての法が次に大きい法の因子であることだけ必要です。このことは、全データを一度に移すことなくパーティション数を徐々に増やすことを可能にします。例えば、各々の法が8である8パーティションのハッシュパーティションテーブルがあるとして、パーティション数を16に増やさなければならなくなったとします。私たちは8を法とするパーティションの一つをデタッチして、新たに16を法とするキー空間の同じ部分(一つはデタッチしたパーティションと等しい残余を持ち、一つはその値に8を加えたのと等しい残余を持つ)を対象とする二つのパーティションを追加して、データを再配置することができます。これを(おそらくはより後に)8を法とする各パーティションがなくなるまで、繰り返すことができます。これは依然として各ステップで大きなデータ移動を伴いますが、全体の新テーブルを作って全データを一度に移さなければならないというよりは、まだ良いです。

パーティションは、それが属するパーティションテーブルと同じ列名および型を持っていなければなりません。パーティションテーブルの列名や型の変更は自動的にすべてのパーティションに反映されます。CHECK制約はすべてのパーティションで自動的に継承されますが、個々のパーティションで追加のCHECK制約を指定することができます。親の制約と同じ名前と条件を持つ追加制約は親の制約と統合されます。デフォルト制約は各パーティションで別々に指定できます。ですが、パーティションのデフォルト値は、パーティションテーブルを通してタプルを挿入する場合には適用されないことに注意してください。

パーティションテーブルに挿入された行は、自動的に正しいパーティションに回されます。適当なパーティションが存在しないときは、エラーが発生します。

TRUNCATEのように通常はテーブルとそれを継承するすべての子テーブルに影響を及ぼす操作は、すべてのパーティションに対しても適用されますが、個別のパーティションに対して操作することも可能です。DROP TABLEでパーティションを削除するには、親テーブルについてACCESS EXCLUSIVEのロックを取得する必要があることに注意してください。

LIKE source_table [like_option ...]

LIKE句にテーブルを指定すると、自動的にそのテーブルの全ての列名、そのデータ型、非NULL制約が新しいテーブルにコピーされます。

INHERITSとは違い、作成した後、新しいテーブルと元のテーブルが完全に分離されます。元のテーブルへの変更は新しいテーブルには適用されません。また、元のテーブルを走査しても新しいテーブルのデータは見つかりません。

INHERITSと異なり、LIKEによりコピーされた列や制約は類似の名前の列や制約にまとめられません。同じ名前が明示的に、あるいは他のLIKE句で指定された場合、エラーが通知されます。

オプションのlike_option句は元テーブルのどの追加属性をコピーするかを指定します。INCLUDING指定は属性をコピーし、EXCLUDING指定は属性を省きます。EXCLUDINGがデフォルトです。同種の対象に複数の指定がある場合には最後のものが使われます。指定可能なオプションは以下です。

INCLUDING COMMENTS

コピーされた列、制約、および、インデックスに対するコメントがコピーされます。デフォルトの振る舞いではコメントは除外されて、新しいテーブルのコピーされた列と制約にはコメントがありません。

INCLUDING CONSTRAINTS

CHECK制約がコピーされます。列制約とテーブル制約の区別はされません。非NULL制約は常に新しいテーブルにコピーされます。

INCLUDING DEFAULTS

コピーされた列定義に対するデフォルト式をコピーします。この指定が無い場合、デフォルト式はコピーされず、新しいテーブルのコピーされた列はNULLのデフォルトとなります。nextvalなどのデータベースを変更する関数を呼び出すデフォルトのコピーは、元のテーブルと新しいテーブルの間で関数の連鎖を引き起こすかもしれないことに注意してください。

INCLUDING GENERATED

コピーされた列定義の全ての生成式がコピーされます。デフォルトでは新たな列は通常の基底列となります。

INCLUDING IDENTITY

コピーされた列定義の全てのアイデンティティ指定がコピーされます。新しいテーブルの各識別列に対して新たなシーケンスが作られ、旧テーブルに関連付けられたシーケンスとは分離されます。

INCLUDING INDEXES

元テーブルのインデックス、および、PRIMARY KEY、UNIQUE、EXCLUDE制約が新しいテーブルに作成されます。新しいインデックスと制約の名前はデフォルトの規則に従って決められ、元テーブルでのどう名前付けされているかは考慮されません。(この振る舞いは新しいインデックスでの起こりうる名前重複エラーを回避します。)

INCLUDING STATISTICS

拡張統計情報が新しいテーブルにコピーされます。

INCLUDING STORAGE

コピーされた列定義に対するSTORAGE設定がコピーされます。デフォルトの振る舞いではSTORAGE設定は除外され、そのため新しいテーブルのコピーされた列はデータ型ごとのデフォルト設定を持ちます。STORAGE設定に関する詳細は68.2を参照してください。

INCLUDING ALL

INCLUDING ALLは全ての各オプションを選択することの短縮形式です。(一部オプションを除き全てを選択するために、個別のEXCLUDING句をINCLUDING ALLの後に書く場合におそらく有益です。)

またLIKE句をビュー、外部テーブル、複合型から列の定義をコピーするために使用することができます。適用できないオプション(ビューからのINCLUDING INDEXESなど)は無視されます。

CONSTRAINT constraint_name

列制約、テーブル制約の名前(省略可能)です。制約に違反すると、制約名がエラーメッセージに含まれます。ですので、col must be positive(正数でなければならない)といった名前の制約名を付与することで、クライアントアプリケーションへ有用な制約情報を渡すことができます。(空白を含む制約名を指定する場合、二重引用符が必要です。) 指定されなければ、システムが名前を生成します。

NOT NULL

その列がNULL値を持てないことを指定します。

NULL

その列がNULL値を持てることを指定します。これがデフォルトです。

この句は非標準的なSQLデータベースとの互換性のためだけに提供されています。新しいアプリケーションでこれを使用するのはお勧めしません。

CHECK (expression) [NO INHERIT]

CHECK句は、論理型の結果を生成する、新しい行または更新される行が挿入または更新処理を成功させるために満足しなければならない式を指定します。TRUEまたはUNKNOWNと評価される式は成功します。挿入または更新処理の行がFALSEという結果をもたらす場合はエラー例外が発生し、その挿入または更新によるデータベースの変更は行われません。列制約として指定された検査制約は列の値のみを参照しなければなりません。テーブル制約内の式は複数の列を参照できます。

現時点では、CHECK式には副問い合わせも現在の行の列以外の変数も含めることはできません(5.4.1を参照)。システム列tableoidを参照することはできますが、他のシステム列は参照できません。

NO INHERITと印が付いた制約は子テーブルには伝搬しません。

テーブルに複数のCHECK制約がある場合、それらはNOT NULL制約について検証した後で、各行について名前のアルファベット順に検証されます。(PostgreSQLの9.5より前のバージョンでは、CHECK制約の実行について特定の順序はありませんでした。)

DEFAULT default_expr

DEFAULT句を列定義に付けると、その列にデフォルトデータ値が割り当てられます。値として指定するのは任意の無変数式です(特に現在のテーブル内の他の列へクロス参照はできません)。副問い合わせも指定できません。デフォルト式のデータ型はその列のデータ型と一致する必要があります。

デフォルト式は、全ての挿入操作において、その列に値が指定されていない場合に使用されます。列にデフォルト値がない場合、デフォルト値はNULLになります。

GENERATED ALWAYS AS (generation_expr) STORED

この句は列を*生成列*として作成します。この列には書き込みできず、読むときには指定された式の結果が返されます。

キーワードSTOREDは列が書き込み時に計算されてディスクに格納されることをあらわすのに必要とされます。

生成式はそのテーブルの他の列を参照できますが、他の生成列は参照できません。使われる全ての関数と演算子はIMMUTABLEでなければなりません。他テーブルを参照することはできません。

GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY [(sequence_options)]

この句は列を*IDENTITY列*として作成します。それには暗示的なシーケンスが紐付けられ、新しい行のその列には紐付けられたシーケンスから取られた値が自動的に入ります。

ALWAYSとBY DEFAULTの句は、INSERTやUPDATEコマンドで明示的にユーザが指定した値をどのように扱うかを決定します。

INSERTコマンドでは、ALWAYSが選択された場合、ユーザの指定した値はINSERT文がOVERRIDING SYSTEM VALUEを指定している場合にのみ受け付けられます。BY DEFAULTが選択された場合、ユーザの指定した値が優先します。詳細はINSERTを参照してください。(COPYコマンドでは、この設定に関係なく、ユーザの指定した値が常に使われます。)

UPDATEでは、ALWAYSが選択された場合、DEFAULT以外の値への列の更新は拒絶されます。BY DEFAULTが選択された場合、列は普通に更新されます。(UPDATEコマンドにOVERRIDING句はありません。)

オプションでsequence_options句を指定することにより、シーケンスのオプションを変更できます。詳しくはCREATE SEQUENCEを参照してください。

UNIQUE (列制約)
UNIQUE (column_name [, ...]) [INCLUDE (column_name [, ...])] (表制約)

UNIQUE制約は、テーブルの1つまたは複数の列からなるグループが、一意な値のみを持つことができることを指定します。一意性テーブル制約の動作は一意性列制約と同じですが、さらに複数列にまたがる機能を持ちます。

一意性制約では、NULL値同士は等しいとはみなされなせん。

それぞれの一意性テーブル制約には、そのテーブルの他の一意性制約もしくは主キー制約によって指定された列の集合とは、異なる名前の列の集合を指定しなければなりません（同じ名前を指定すると、同じ制約が2回現れるだけになります）。

複数レベルのパーティション階層に一意性制約を設定するとき、対象パーティションテーブル、および全ての子孫のパーティションテーブルの、パーティションキー内の全ての列が制約定義に含まれなくてはなりません。

一意性制約を加えると、制約で使われている列や列のグループに一意性btreeインデックスが自動的に作られます。省略可能なINCLUDE句はインデックスに一意性を強要されない列を1つまたは複数、追加します。含めた（INCLUDEした）列に制約は強制されませんが、依存はしていることに注意してください。このため、これらの列に対する一部の操作（例えばDROP COLUMN）は制約の連鎖とインデックスの削除をひき起こすことがあります。

PRIMARY KEY（列制約）
PRIMARY KEY (column_name [, ...]) [INCLUDE (column_name [, ...])] (表制約)

PRIMARY KEY制約はテーブルの一列または複数の列が一意（重複がない）で、非NULLの値のみを持つことを指定します。列制約か表制約かに関わらず、1つのテーブルには主キーを1つだけ指定できます。

主キー制約では、同じテーブルに一意制約で指定した列の集合とは異なる列の集合を指定します。（そうでなければ、一意制約は冗長となり、捨てられます。）

PRIMARY KEYはUNIQUEとNOT NULLの組み合わせと同じデータ制約を課しますが、列の集合を主キーと指定することは、スキーマの設計についてのメタデータを提供することにもなります。なぜなら、主キーであることは、行を一意に特定するものとして、他のテーブルがその列の集合を当てにして良い、ということを意味するからです。

PRIMARY KEY制約は、パーティションテーブルに設定するときにUNIQUE制約が持つ制限を共有します。

PRIMARY KEY制約を追加すると、制約で使用する列や列のグループに一意性のbtreeインデックスが自動的に作られます。省略可能なINCLUDE句はインデックスの非キー部分に含める列のリストを指定できます。含めた列に一意性は強制されませんが、制約はこれらに依存はしています。このため、これらの含められた列に対する一部の操作（例えばDROP COLUMN）は制約の連鎖とインデックスの削除をひき起こすことがあります。

EXCLUDE [USING index_method] (exclude_element WITH operator [, ...]) index_parameters [WHERE (predicate)]

EXCLUDE句は排他制約を定義し、任意の2行について指定した列(複数可)または式(複数可)を指定した演算子(複数可)を使用して比較した場合、比較結果のすべてがTRUEを返さないことを保証します。指定した演算子のすべてが等価性を試験するものであれば、これはUNIQUE制約と同じですが、通常の一意性制約のほうが高速です。しかし、排他制約では単純な等価性よりも一般的な制約を指定することができます。例えば、テーブル内の2つの行が重複する円(8.8参照)を持たないといった制約を&&演算子を使用して指定することができます。

排他制約はインデックスを使用して実装されています。このため指定した演算子はそれぞれ適切な演算子クラス(11.10参照)でindex_methodインデックスアクセスメソッドと関連付けされていなければなりません。演算子は交換可能でなければなりません。オプションで、各exclude_elementは演算子クラス、

順序付けオプション、またはその両方を指定することができます。これらについてはCREATE INDEXで説明します。

アクセスメソッドはamgettupleをサポートしなければなりません(第61章参照)。現時点では、これはGINを使用できないことを意味します。B-treeやHashインデックスを排他制約で使用することは許容されますが、そうすることにあまり意味はありません。これが通常の一意性制約より良いことは何もないからです。このため現実的にはアクセスメソッドは常にGiSTもしくはSP-GiSTとなります。

predicateにより、排他制約をテーブルの部分集合に指定することができます。内部的には、これは部分インデックスを作成します。predicateの前後に括弧が必要であることに注意して下さい。

REFERENCES reftable [(refcolumn)] [MATCH matchtype] [ON DELETE action] [ON UPDATE action] (列制約)
FOREIGN KEY (column_name [, ...]) REFERENCES reftable [(refcolumn [, ...])]
[MATCH matchtype] [ON DELETE referential_action] [ON UPDATE referential_action] (表制約)

これらの句は、外部キー制約を指定します。外部キー制約は、新しいテーブルの1つまたは複数の列の集合が、被参照テーブルの一部の行の被参照列に一致する値を持たなければならないことを指定するものです。refcolumnリストが省略された場合、reftableの主キーが使用されます。被参照列は、被参照テーブルにおいて遅延不可の一意性制約もしくは主キー制約を持った列でなければなりません。ユーザは被参照テーブル(テーブル全体または特定の被参照列)についてREFERENCES権限を持っていなければなりません。外部キー制約の追加は被参照テーブルにSHARE ROW EXCLUSIVEロックを必要とします。一時テーブルと永続テーブルとの間で外部キー制約を定義できないことに注意してください。

参照列に挿入された値は、被参照テーブルと被参照列の値に対して、指定した照合型で照会されます。照合型には3種類があります。MATCH FULL、MATCH PARTIAL、MATCH SIMPLE(これがデフォルト)照合型です。MATCH FULLは全ての外部キー列がNULLとなる場合を除き、複数列外部キーのある列がNULLとなることを許可しません。それらがすべてNULLであれば、その行は被参照テーブル内で一致があることは要求されません。MATCH SIMPLEは、外部キーの一部がNULLであることを許可します。それらの一部がNULLであれば、その行は被参照テーブル内で一致があることは要求されません。MATCH PARTIALはまだ実装されていません。(当然ですが、NOT NULL制約を参照列に適用し、こうした状態が発生することを防止することができます。)

さらに、被参照列のデータが変更された場合、このテーブルの列のデータに何らかの動作が発生します。ON DELETE句は、被参照テーブルの被参照行が削除されようとした場合の動作を指定します。同様にON UPDATE句は、被参照テーブルの被参照列が新しい値に更新されようとした場合の動作を指定します。行の更新があった場合でも、被参照列が実際に変更されない場合は、動作は実行されません。制約が遅延可能と宣言されていても、NO ACTION検査以外の参照動作は遅延させられません。各句について、以下の動作を指定可能です。

NO ACTION

削除もしくは更新により外部キー制約違反が起こることを示すエラーを発生します。制約が遅延可能な場合、何らかの参照行が存在する限り、このエラーは制約の検査時点で発生します。これはデフォルトの動作です。

RESTRICT

削除もしくは更新が外部キー制約違反となることを示すエラーを発生します。検査が遅延できない点を除き、NO ACTIONと同じです。

CASCADE

削除された行を参照している行は全て削除します。また、参照している列の値を、被参照列の新しい値にします。

SET NULL

参照する列をNULLに設定します。

SET DEFAULT

参照する列をそのデフォルト値に設定します。(デフォルト値がNULLでない場合は被参照テーブルの中にデフォルト値に一致する行が存在しなければなりません。さもないと操作が失敗します。)

被参照列が頻繁に更新される場合、参照列にインデックスを付け、その外部キー制約に関連する参照動作がより効率的に実行できるようにする方が良いでしょう。

DEFERRABLE
NOT DEFERRABLE

制約を遅延させることが可能かどうかを制御します。遅延不可の制約は各コマンドの後すぐに検査されます。遅延可能な制約の検査は、(SET CONSTRAINTSコマンドを使用して)トランザクションの終了時まで遅延させることができます。NOT DEFERRABLEがデフォルトです。現在、UNIQUE、PRIMARY KEY、EXCLUDE、REFERENCES(外部キー)制約のみがこの句を受け付けることができます。NOT NULLおよび CHECK制約は遅延させることができません。遅延可能な制約はON CONFLICT DO UPDATE句を含むINSERT文において、競合解決のために使うことはできないことに注意してください。

INITIALLY IMMEDIATE
INITIALLY DEFERRED

制約が遅延可能な場合、この句は制約検査を行うデフォルトの時期を指定します。制約がINITIALLY IMMEDIATEの場合、各文の実行後に検査されます。これがデフォルトです。制約がINITIALLY DEFERREDの場合、トランザクションの終了時にのみ検査されます。制約検査の時期はSET CONSTRAINTSコマンドを使用して変更することができます。

USING method

このオプションの句は新しいテーブルの中身の格納に用いるテーブルアクセスメソッドを指定します。指定するものは、TABLEタイプのアクセスメソッドでなければなりません。より詳しい情報は第60章を参照してください。このオプションが指定されない場合、新しいテーブルにはデフォルトテーブルアクセスメソッドが選択されます。より詳しい情報はdefault_table_access_methodを参照してください。

WITH (storage_parameter [= value] [, ...])

この句はテーブルまたはインデックスに対するオプションの格納パラメータを指定します。詳しくはStorage Parametersを参照してください。後方互換性のため、テーブルに対するWITH句には新しいテーブルの行にOID(オブジェクト識別子)が含まれないことを示すためにOIDS=FALSEを含めることもできます。OIDS=TRUEはもはやサポートされません。

WITHOUT OIDS

これはWITHOUT OIDSのテーブルと宣言する後方互換性の構文です。WITH OIDSのテーブルを作ることはもはやサポートされません。

ON COMMIT

ON COMMITを使用して、トランザクションブロックの終了時点での一時テーブルの動作を制御することができます。以下の3つのオプションがあります。

PRESERVE ROWS

トランザクションの終了時点で、特別な動作は行われません。これがデフォルトの動作です。

DELETE ROWS

一時テーブル内の全ての行は、各トランザクションブロックの終わりで削除されます。実質的には、コミットの度に自動的にTRUNCATEが実行されます。パーティションテーブルに使われた場合、そのパーティションに連鎖適用はされません。

DROP

一時テーブルは現在のトランザクションブロックの終わりで削除されます。この動作は、パーティションテーブルに使われたときにそのパーティションを削除し、継承の子テーブルを伴うテーブルに使われたときに従属する子テーブルを削除します。

TABLESPACE tablespace_name

tablespace_nameは、新しいテーブルが作成されるテーブル空間名です。指定されていない場合、default_tablespaceが、また一時テーブルの場合はtemp_tablespacesが考慮されます。パーティションテーブルに対しては、そのテーブル自身ではストレージは必要としないため、他のテーブル空間が明示的に指定されていないときに新しく作成されたパーティションに使用するデフォルトのテーブル空間として、指定されたテーブル空間がdefault_tablespaceを上書きします。

USING INDEX TABLESPACE tablespace_name

この句により、UNIQUE、PRIMARY KEY、またはEXCLUDE制約に関連したインデックスを作成するテーブル空間を選択することができます。指定されていない場合、default_tablespaceが、また一時テーブルであればtemp_tablespacesが考慮されます。

格納パラメータ

WITH句により、テーブルおよびUNIQUE、PRIMARY KEY、またはEXCLUDE制約と関連づいたインデックスの格納パラメータを指定することができます。インデックスの格納パラメータについてはCREATE INDEXで説明します。現在テーブルで設定可能な格納パラメータの一覧を以下に示します。これらのパラメータの多くに対して、示した通り、さらにtoastという接頭辞のついた、同一の名前のパラメータがあります。これはもしあれば、テーブルの補助TOASTテーブルの動作を制御します。(TOASTに関する詳細については68.2を参照してください。) テーブルのパラメータ値が設定され、それと同等のtoast.パラメータが設定されていない場合、TOASTテーブルはテーブルのパラメータ値を利用します。これらのパラメータをパーティションテーブルについて指定することはサポートされませんが、個々の末端のパーティションについて指定することはできます。

fillfactor (integer)

テーブルのフィルファクタ(fillfactor)は10から100までの間の割合（パーセント）です。100（すべて使用）がデフォルトです。より小さな値を指定すると、INSERT操作は指定した割合までしかテーブルペー

ジを使用しません。各ページの残りの部分は、そのページ内の行の更新用に予約されます。これによりUPDATEは、元の行と同じページ上に更新済みの行を格納することができるようになります。これは別のページに更新済みの行を格納することよりも効率的です。項目の更新がまったくないテーブルでは、すべてを使用することが最善の選択ですが、更新が非常に多いテーブルではより小さめのフィルファクタが適切です。TOASTテーブルではこのパラメータを設定できません。

toast_tuple_target (integer)

toast_tuple_targetは、長い列値を圧縮したりTOASTテーブルに移動する前に必要とされる最小タプル長を指定します。また、これはTOAST化を開始したときに長さをそれ未満に減らそうとする目標にもなります。これはEXTERNAL(移動に対して)、MAIN(圧縮に対して)、または、EXTENDED(両方に対して)と印付けされた列に影響があり、また、新たなタプルにのみ適用されます。既存の行には影響ありません。デフォルトでは、このパラメータは1ブロックあたり少なくとも4タプルが可能であるように設定されます。これはデフォルトブロックサイズであれば2040バイトになります。有効な値は128バイトから、ブロックサイズ - ヘッダ（デフォルトでは8160バイト）の間です。非常に短いあるいは長い行に対して、この値を変更することはおそらく有用ではありません。時にはデフォルト設定が最適に近く、本パラメータを設定することで場合によっては悪影響があるかもしれないことに注意してください。

parallel_workers (integer)

このテーブルの並列スキャンを支援するために使用されるワーカの数を設定します。設定されなければ、リレーションのサイズに基づいてシステムが値を決定します。プランナやパラレルスキャンを使うユーティリティ文により選ばれるワーカの数は、例えばmax_worker_processesの設定によって、より少なくなるかもしれません。

autovacuum_enabled, toast.autovacuum_enabled (boolean)

特定のテーブルに対する自動バキュームデーモンを有効または無効にします。trueの場合、自動バキュームデーモンは、24.1.6に記述されたルールに従って、このテーブルに対して自動的にVACUUMあるいはANALYZEまたはその両方の操作を行います。falseの場合、トランザクションIDの周回問題を回避するためを除き自動バキュームは行われません。周回問題の回避については24.1.5を参照してください。autovacuumパラメータがfalseの場合、（トランザクションIDの周回問題を回避する場合を除き）自動バキュームデーモンはまったく実行されないことに注意して下さい。個々のテーブルの格納パラメータを設定しても、それは優先されません。従って、この格納パラメータを明示的にtrueに設定することにはほとんど意味はなく、falseに設定することのみが意味を持ちます。

vacuum_index_cleanup, toast.vacuum_index_cleanup (boolean)

このテーブルにVACUUMが実行されたときのインデックスのクリーンアップを有効または無効にします。デフォルト値はtrueです。インデックスのクリーンアップを無効にすることで、VACUUMを大幅に高速化できますが、テーブルの変更が頻繁である場合には深刻なインデックスの肥大化も生じさせるかもしれません。VACUUMのINDEX_CLEANUPパラメータは、指定されていたなら本オプションを上書きします。

vacuum_truncate, toast.vacuum_truncate (boolean)

バキュームがテーブル末尾の空ページの切り捨てを試みることを、有効または無効にします。デフォルト値はtrueです。trueの場合、VACUUMと自動バキュームは切り捨てを行い切り捨てられたページのディスク領域はオペレーティングシステムに返されます。切り捨てにはテーブルにACCESS EXCLUSIVEロックが必要であることに注意してください。VACUUMのTRUNCATEパラメータは、指定されていたなら本オプションを上書きします。

autovacuum_vacuum_threshold, toast.autovacuum_vacuum_threshold (integer)

autovacuum_vacuum_threshold パラメータについて、テーブル毎に設定する値です。

autovacuum_vacuum_scale_factor, toast.autovacuum_vacuum_scale_factor (floating point)

autovacuum_vacuum_scale_factor パラメータについて、テーブル毎に設定する値です。

autovacuum_vacuum_insert_threshold, toast.autovacuum_vacuum_insert_threshold (integer)

autovacuum_vacuum_insert_threshold パラメータについて、テーブル毎に設定する値です。特別な値である-1は、テーブルでのインサートバキュームを無効にするのに使われます。

autovacuum_vacuum_insert_scale_factor, toast.autovacuum_vacuum_insert_scale_factor (float4)

autovacuum_vacuum_insert_scale_factor パラメータについて、テーブル毎に設定する値です。

autovacuum_analyze_threshold (integer)

autovacuum_analyze_threshold パラメータについて、テーブル毎に設定する値です。

autovacuum_analyze_scale_factor (floating point)

autovacuum_analyze_scale_factor パラメータについて、テーブル毎に設定する値です。

autovacuum_vacuum_cost_delay, toast.autovacuum_vacuum_cost_delay (floating point)

autovacuum_vacuum_cost_delay パラメータについて、テーブル毎に設定する値です。

autovacuum_vacuum_cost_limit, toast.autovacuum_vacuum_cost_limit (integer)

autovacuum_vacuum_cost_limit パラメータについて、テーブル毎に設定する値です。

autovacuum_freeze_min_age, toast.autovacuum_freeze_min_age (integer)

vacuum_freeze_min_age パラメータについて、テーブル毎に設定する値です。テーブル単位のautovacuum_freeze_min_age パラメータをシステム全体のautovacuum_freeze_max_age 設定の1/2より大きく設定しても、自動バキュームが無視することに注意してください。

autovacuum_freeze_max_age, toast.autovacuum_freeze_max_age (integer)

autovacuum_freeze_max_age パラメータについて、テーブル毎に設定する値です。テーブル単位のautovacuum_freeze_max_age パラメータをシステム全体に対する設定より大きく設定しても、自動バキュームが無視することに注意してください（より小さな値しか設定できません）。

autovacuum_freeze_table_age, toast.autovacuum_freeze_table_age (integer)

vacuum_freeze_table_age パラメータについて、テーブル毎に設定する値です。

autovacuum_multixact_freeze_min_age, toast.autovacuum_multixact_freeze_min_age (integer)

vacuum_multixact_freeze_min_age パラメータについて、テーブル毎に設定する値です。テーブル単位のautovacuum_multixact_freeze_min_age パラメータをシステム全体

のautovacuum_multixact_freeze_max_ageの半分より大きく設定しても、自動バキュームが無視することに注意してください。

autovacuum_multixact_freeze_max_age, toast.autovacuum_multixact_freeze_max_age (integer)

autovacuum_multixact_freeze_max_ageパラメータについて、テーブル毎に設定する値です。テーブル単位のautovacuum_multixact_freeze_max_ageをシステム全体に対する設定より大きくしても、自動バキュームが無視することに注意してください(より小さな値しか設定できません)。

autovacuum_multixact_freeze_table_age, toast.autovacuum_multixact_freeze_table_age (integer)

vacuum_multixact_freeze_table_ageパラメータについて、テーブル毎に設定する値です。

log_autovacuum_min_duration, toast.log_autovacuum_min_duration (integer)

log_autovacuum_min_durationパラメータについて、テーブル毎に設定する値です。

user_catalog_table (boolean)

テーブルを論理レプリケーションのための追加のカタログテーブルとして宣言します。詳しくは48.6.2を参照してください。このパラメータはTOASTテーブルには設定できません。

Notes

PostgreSQLは自動的に各一意性制約と主キー制約に対してインデックスを作成し、その一意性を確実なものにします。したがって、主キーの列に明示的にインデックスを作成することは必要ありません (詳細についてはCREATE INDEXを参照してください)。

現在の実装では、一意性制約と主キーは継承されません。これは、継承と一意性制約を組み合わせると障害が発生するからです。

テーブルは1600列以上の列を持つことはできません (タプル長の制限により実際の制限はもっと小さくなります)。

例

filmsテーブルとdistributorsテーブルを作成します。

```
CREATE TABLE films (
    code        char(5) CONSTRAINT firstkey PRIMARY KEY,
    title       varchar(40) NOT NULL,
    did         integer NOT NULL,
    date_prod   date,
    kind        varchar(10),
    len         interval hour to minute
);

CREATE TABLE distributors (
     did    integer PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY,
```

```
     name   varchar(40) NOT NULL CHECK (name <> '')
);
```

2次元配列を持つテーブルを作成します。

```
CREATE TABLE array_int (
    vector  int[][]
);
```

filmsテーブルに 一意性テーブル制約を定義します。一意性テーブル制約はテーブルの1つ以上の列に定義することができます。

```
CREATE TABLE films (
    code        char(5),
    title       varchar(40),
    did         integer,
    date_prod   date,
    kind        varchar(10),
    len         interval hour to minute,
    CONSTRAINT production UNIQUE(date_prod)
);
```

検査列制約を定義します。

```
CREATE TABLE distributors (
    did     integer CHECK (did > 100),
    name    varchar(40)
);
```

検査テーブル制約を定義します。

```
CREATE TABLE distributors (
    did     integer,
    name    varchar(40),
    CONSTRAINT con1 CHECK (did > 100 AND name <> '')
);
```

filmsテーブルに主キーテーブル制約を定義します。

```
CREATE TABLE films (
    code        char(5),
    title       varchar(40),
    did         integer,
    date_prod   date,
    kind        varchar(10),
    len         interval hour to minute,
```

```
    CONSTRAINT code_title PRIMARY KEY(code,title)
);
```

distributorsテーブルに主キー制約を定義します。以下の2つの例は同等で、前者はテーブル制約構文を使用し、後者は列制約構文を使用します。

```
CREATE TABLE distributors (
    did     integer,
    name    varchar(40),
    PRIMARY KEY(did)
);

CREATE TABLE distributors (
    did     integer PRIMARY KEY,
    name    varchar(40)
);
```

以下では、name列のデフォルト値にリテラル定数を割り当てています。また、did列のデフォルト値として、シーケンスオブジェクトの次の値が生成されるように調整しています。modtimeのデフォルト値は、その行が挿入された時刻となります。

```
CREATE TABLE distributors (
    name      varchar(40) DEFAULT 'Luso Films',
    did       integer DEFAULT nextval('distributors_serial'),
    modtime   timestamp DEFAULT current_timestamp
);
```

2つのNOT NULL列制約をdistributorsテーブルに定義します。そのうち1つには明示的な名前を付けています。

```
CREATE TABLE distributors (
    did     integer CONSTRAINT no_null NOT NULL,
    name    varchar(40) NOT NULL
);
```

name列に対し、一意性制約を定義します。

```
CREATE TABLE distributors (
    did     integer,
    name    varchar(40) UNIQUE
);
```

上と同じですが、テーブル制約として指定します。

```
CREATE TABLE distributors (
    did     integer,
```

```
    name    varchar(40),
    UNIQUE(name)
);
```

テーブルとその一意性インデックスの両方に70%のフィルファクタを指定して、同じテーブルを作成します。

```
CREATE TABLE distributors (
    did     integer,
    name    varchar(40),
    UNIQUE(name) WITH (fillfactor=70)
)
WITH (fillfactor=70);
```

2つの円の重複を許さない排他制約を持つcirclesテーブルを作成します。

```
CREATE TABLE circles (
    c circle,
    EXCLUDE USING gist (c WITH &&)
);
```

diskvol1テーブル空間にcinemasテーブルを作成します。

```
CREATE TABLE cinemas (
        id serial,
        name text,
        location text
) TABLESPACE diskvol1;
```

複合型と型付きテーブルを作成します。

```
CREATE TYPE employee_type AS (name text, salary numeric);

CREATE TABLE employees OF employee_type (
    PRIMARY KEY (name),
    salary WITH OPTIONS DEFAULT 1000
);
```

範囲パーティションテーブルを作成します。

```
CREATE TABLE measurement (
    logdate         date not null,
    peaktemp        int,
    unitsales       int
) PARTITION BY RANGE (logdate);
```

パーティションキーに複数の列がある範囲パーティションテーブルを作成します。

```
CREATE TABLE measurement_year_month (
    logdate         date not null,
    peaktemp        int,
    unitsales       int
) PARTITION BY RANGE (EXTRACT(YEAR FROM logdate), EXTRACT(MONTH FROM logdate));
```

リストパーティションテーブルを作成します。

```
CREATE TABLE cities (
    city_id      bigserial not null,
    name         text not null,
    population   bigint
) PARTITION BY LIST (left(lower(name), 1));
```

ハッシュパーティションテーブルを作成します。

```
CREATE TABLE orders (
    order_id     bigint not null,
    cust_id      bigint not null,
    status       text
) PARTITION BY HASH (order_id);
```

範囲パーティションテーブルのパーティションを作成します。

```
CREATE TABLE measurement_y2016m07
    PARTITION OF measurement (
    unitsales DEFAULT 0
) FOR VALUES FROM ('2016-07-01') TO ('2016-08-01');
```

パーティションキーに複数の列がある範囲パーティションテーブルに、パーティションをいくつか作成します。

```
CREATE TABLE measurement_ym_older
    PARTITION OF measurement_year_month
    FOR VALUES FROM (MINVALUE, MINVALUE) TO (2016, 11);

CREATE TABLE measurement_ym_y2016m11
    PARTITION OF measurement_year_month
    FOR VALUES FROM (2016, 11) TO (2016, 12);

CREATE TABLE measurement_ym_y2016m12
    PARTITION OF measurement_year_month
    FOR VALUES FROM (2016, 12) TO (2017, 01);

CREATE TABLE measurement_ym_y2017m01
    PARTITION OF measurement_year_month
    FOR VALUES FROM (2017, 01) TO (2017, 02);
```

リストパーティションテーブルのパーティションを作成します。

```
CREATE TABLE cities_ab
    PARTITION OF cities (
    CONSTRAINT city_id_nonzero CHECK (city_id != 0)
) FOR VALUES IN ('a', 'b');
```

リストパーティションテーブルにパーティションを作成しますが、それ自体がさらにパーティションになり、それにパーティションを追加します。

```
CREATE TABLE cities_ab
    PARTITION OF cities (
    CONSTRAINT city_id_nonzero CHECK (city_id != 0)
) FOR VALUES IN ('a', 'b') PARTITION BY RANGE (population);

CREATE TABLE cities_ab_10000_to_100000
    PARTITION OF cities_ab FOR VALUES FROM (10000) TO (100000);
```

ハッシュパーティションテーブルのパーティションを作成します。

```
CREATE TABLE orders_p1 PARTITION OF orders
    FOR VALUES WITH (MODULUS 4, REMAINDER 0);
CREATE TABLE orders_p2 PARTITION OF orders
    FOR VALUES WITH (MODULUS 4, REMAINDER 1);
CREATE TABLE orders_p3 PARTITION OF orders
    FOR VALUES WITH (MODULUS 4, REMAINDER 2);
CREATE TABLE orders_p4 PARTITION OF orders
    FOR VALUES WITH (MODULUS 4, REMAINDER 3);
```

デフォルトのパーティションを作成します。

```
CREATE TABLE cities_partdef
    PARTITION OF cities DEFAULT;
```

互換性

CREATE TABLEは、以下に挙げるものを除いて、標準SQLに従います。

一時テーブル

CREATE TEMPORARY TABLEは標準SQLに類似していますが、その効果は同じではありません。標準では、一時テーブルは一度だけ定義され、それを必要とするセッションごとに自動的に(空の内容で始まる形で)出現します。PostgreSQLでは、これと異なり、各セッションで独自に、使用する一時テーブル用のCREATE TEMPORARY TABLEコマンドを発行しなければなりません。これにより、異なるセッションで同じ名前の一時

テーブルを異なる目的で使用することができます。一方、標準の方法では、ある一時テーブル名を持つインスタンスが、全て同一のテーブル構造を持つという制限があります。

標準における一時テーブルの動作定義の多くは無視されています。この点でのPostgreSQLの動作は、他の多くのSQLデータベースと似ています。

また標準SQLではグローバル一時テーブルとローカル一時テーブルを区別しています。ローカル一時テーブルは各セッション内のSQLモジュールそれぞれ用に内容の集合を分離しますが、その定義はセッション全体で共有されます。PostgreSQLはSQLモジュールをサポートしませんので、PostgreSQLではこの区別は適切ではありません。

互換性を保持するため、PostgreSQLは一時テーブルの宣言においてGLOBALとLOCALキーワードを受け付けますが、これらには現在、何の効果もありません。PostgreSQLの今後のバージョンでは、これらの意味についてより標準に近い実装を取り入れる可能性がありますので、これらのキーワードの使用は勧めません。

一時テーブル用のON COMMIT句もまた、標準SQLに類似していますが、いくつか違いがあります。ON COMMIT句が省略された場合、SQLでは、デフォルトの動作はON COMMIT DELETE ROWSであると規定しています。しかし、PostgreSQLでのデフォルトの動作はON COMMIT PRESERVE ROWSです。また、ON COMMIT DROPはSQLにはありません。

非遅延一意性制約

UNIQUEまたはPRIMARY KEY制約が非遅延の場合、PostgreSQLは行が挿入または変更されると即座に一意性を検査します。標準SQLでは一意性は文が完了した時にのみ強制されなければならないと記述しています。これにより、たとえば、1つのコマンドが複数のキー値を更新する時に違いが現れます。標準互換の動作をさせるためには、非遅延（つまりINITIALLY IMMEDIATE）ではなくDEFERRABLEとして制約を宣言してください。これが即座に行われる一意性検査よりかなり低速になる可能性があることに注意してください。

列検査制約

標準SQLでは、CHECK列制約はそれを適用する列のみを参照でき、複数の列を参照できるのはCHECKテーブル制約のみであるとされています。PostgreSQLにはこの制限はありません。列検査制約とテーブル検査制約を同様のものとして扱っています。

EXCLUDE制約

EXCLUDEという種類の制約はPostgreSQLの拡張です。

NULL「制約」

NULL「制約」（実際には非制約）は、標準SQLに対するPostgreSQLの拡張で、他のいくつかのデータベースシステムとの互換性（および NOT NULL制約との対称性）のために含まれています。どんな列に対してもデフォルトとなるため、これには意味はありません。

制約の命名

SQL標準ではテーブルとドメインの制約はテーブルやドメインを含むスキーマ中で一意な名前を持たなければなりません。PostgreSQLはより緩やかで、制約名は特定のテーブルやドメインに付加された制約の中で一意であることだけが求められます。しかしながら、この追加的な自由はインデックスに基づく制約（UNIQUE、PRIMARY KEY、およびEXCLUDE制約）にはありません。なぜなら、関連付けられたインデックスは制約と同じに命名されて、インデックス名は同スキーマ内の全てのリレーションの中で一意でなければならないからです。

今のところ、PostgreSQLはNOT NULL制約の名前を全く記録しませんので、これらは一意性の制限の対象ではありません。これは将来のリリースで変更されるかもしれません。

継承

INHERITS句による複数継承は、PostgreSQLの言語拡張です。SQL:1999以降では、異なる構文と意味体系による単一継承を定義しています。今のところ、SQL:1999方式の継承はPostgreSQLではサポートされていません。

列を持たないテーブル

PostgreSQLでは、列を持たないテーブルを作成することができます（例えば、CREATE TABLE foo();）。これは標準SQLからの拡張です。標準SQLでは列を持たないテーブルは許されません。列を持たないテーブルそれ自体は役に立ちませんが、これを無効とすると、ALTER TABLE DROP COLUMNに対して奇妙な特例を生成することになります。したがって、この仕様上の制限を無視する方が簡潔であると考えます。

複数のIDENTITY列

PostgreSQLではテーブルに2つ以上のIDENTITY列を持つことを許しています。標準SQLでは、1つのテーブルは最大で1つのIDENTITY列を持つことができると規定しています。主にスキーマの変更や移行でより柔軟性を持たせるために、この制約を緩和しています。INSERTコマンドはOVERRIDING句を1つだけしかサポートせず、これが文全体に適用されるため、複数のIDENTITY列があり、これらの動作が異なる場合は正しくサポートされないことに注意してください。

生成列

オプションSTOREDは標準ではありませんが他のSQL実装でも使われています。SQL標準は生成列の格納を規定していません。

LIKE句

LIKE句は標準SQLにありますが、PostgreSQLで利用可能な多くのオプションは標準にはなく、また標準のオプションの一部はPostgreSQLでは実装されていません。

WITH句

WITH句はPostgreSQLの拡張です。格納パラメータは標準にはありません。

テーブル空間

PostgreSQLのテーブル空間の概念は標準にはありません。したがって、TABLESPACEとUSING INDEX TABLESPACEは、PostgreSQLにおける拡張です。

型付きテーブル

型付きテーブルは標準SQLのサブセットを実装します。標準に従うと、型付きテーブルは背後の複合型に対応した列の他に「自己参照列」という列も持ちます。PostgreSQLは自己参照列を明示的にサポートしません。

PARTITION BY句

PARTITION BYはPostgreSQLの拡張です。

PARTITION OF句

PARTITION OF句はPostgreSQLの拡張です。

関連項目

ALTER TABLE, DROP TABLE, CREATE TABLE AS, CREATE TABLESPACE, CREATE TYPE

CREATE TABLE AS

CREATE TABLE AS — 問い合わせの結果によって新しいテーブルを定義する

概要

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE [ IF NOT EXISTS ] table_name
    [ (column_name [, ...] ) ]
    [ USING method ]
    [ WITH ( storage_parameter [= value] [, ... ] ) | WITHOUT OIDS ]
    [ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
    [ TABLESPACE tablespace_name ]
    AS query
    [ WITH [ NO ] DATA ]
```

説明

CREATE TABLE ASはテーブルを作成し、SELECT コマンドによって算出されたデータをそのテーブルに格納します。テーブルの列は、SELECTの出力列に結び付いた名前とデータ型を持ちます（ただし、新しい列名を明示したリストを渡すと、この列名を上書きすることができます）。

CREATE TABLE ASはビューの作成と似ていますが、実際にはまったく異なります。CREATE TABLE ASは新しいテーブルを作成し、新しいテーブルの内容を初期化するために一度だけ問い合わせを評価します。それ以降に行われた、問い合わせの元テーブルに対する変更は、新しいテーブルには反映されません。反対に、ビューは問い合わせの度に定義されたSELECT文を再評価します。

パラメータ

GLOBALまたはLOCAL

互換性を保持するためのキーワードで、無視されます。これらのキーワードの使用は廃止予定です。詳細についてはCREATE TABLEを参照してください。

TEMPORARYまたはTEMP

指定された場合、テーブルは一時テーブルとして作成されます。詳細についてはCREATE TABLEを参照してください。

UNLOGGED

指定された場合、テーブルはログを取らないテーブルとして作成されます。詳細についてはCREATE TABLEを参照してください。

IF NOT EXISTS

同じ名前のリレーションが既に存在する場合にエラーとしません。この場合、注意が発行されます。詳しくはCREATE TABLEを参照してください。

table_name

作成するテーブルの名前です(スキーマ修飾名も可)。

column_name

新しいテーブルにおける列の名前です。列名を指定しない場合は、問い合わせの出力列名を利用します。

USING method

この省略可能な句は、新しいテーブルの内容を保存するのに使うテーブルアクセスメソッドを指定します。メソッドはTABLE型のアクセスメソッドであることが必要です。より詳しい情報は第60章を参照してください。このオプションが指定されなければ、新しいテーブルに対してはデフォルトのテーブルアクセスメソッドが選ばれます。より詳しい情報はdefault_table_access_methodを参照してください。

WITH (storage_parameter [= value] [, ...])

この句は、新しいテーブル用の格納パラメータ(省略可能)を指定します。詳細はCREATE TABLEの文書のStorage Parametersを参照してください。後方互換のため、テーブルに対するWITH句には、OID(オブジェクト識別子)を持たないことを指定するOIDS=FALSEを含めることもできます。OIDS=TRUEは今はもうサポートされていません。

WITHOUT OIDS

これは、テーブルがWITHOUT OIDSであることを宣言する後方互換のための構文で、WITH OIDSであるテーブルを作成することは今はもうサポートされていません。

ON COMMIT

トランザクションブロックの終了時の一時テーブルの動作をON COMMITを使用して制御することができます。以下の3つのオプションがあります。

PRESERVE ROWS

トランザクションの終了時に特別な処理は何も行われません。これがデフォルトの動作です。

DELETE ROWS

各トランザクションブロックの終了時に、一時テーブルのすべての行が削除されます。本質的には、コミット毎に自動的にTRUNCATEが行われます。

DROP

現在のトランザクションブロックの終了時に一時テーブルは削除されます。

TABLESPACE tablespace_name

tablespace_nameは、新しいテーブルの作成先となるテーブル空間名です。指定がなければ、default_tablespace、一時テーブルの場合はtemp_tablespacesが考慮されます。

query

SELECT、TABLE、VALUESコマンドまたは、あらかじめ準備されたSELECT、TABLEまたはVALUES問い合わせを実行するEXECUTEコマンドです。

`WITH [ NO ] DATA`

この句は問い合わせで生成されるデータを新しいテーブルにコピーすべきかどうかを指定します。コピーしない場合はテーブル構造のみがコピーされます。デフォルトではデータをコピーします。

注釈

このコマンドは、SELECT INTOと同等の機能を持ちますが、`SELECT INTO`構文の他の使用方法と混乱する可能性が少ないため、こちらを使用する方が良いでしょう。さらに、`CREATE TABLE AS`は、`SELECT INTO`が提供する機能のスーパーセットを提供します。

例

`films`の最近の項目のみから構成される、新しいテーブル`films_recent`を作成します。

```
CREATE TABLE films_recent AS
  SELECT * FROM films WHERE date_prod >= '2002-01-01';
```

テーブルを完全に複製するために、`TABLE`コマンドを使った短縮形も使用することができます。

```
CREATE TABLE films2 AS
  TABLE films;
```

プリペアド文を使用して、`films`内の最近の項目のみから構成される一時テーブル`films_recent`を作成します。この新しいテーブルはコミット時に削除されます。

```
PREPARE recentfilms(date) AS
  SELECT * FROM films WHERE date_prod > $1;
CREATE TEMP TABLE films_recent ON COMMIT DROP AS
  EXECUTE recentfilms('2002-01-01');
```

互換性

`CREATE TABLE AS`は標準SQLに従います。以下は非標準の拡張です。

- 標準では副問い合わせ句を括弧で囲む必要がありますが、PostgreSQLではこの括弧は省略可能です。
- 標準では`WITH [ NO ] DATA`句は必須ですが、PostgreSQLでは省略可能です。
- PostgreSQLの一時テーブルの扱いは標準とは異なります。詳細はCREATE TABLEを参照してください。
- `WITH`句はPostgreSQLの拡張です。格納パラメータは標準にはありません。
- PostgreSQLのテーブル空間という概念は標準にはありません。したがって、`TABLESPACE`句は拡張です。

関連項目

CREATE MATERIALIZED VIEW, CREATE TABLE, EXECUTE, SELECT, SELECT INTO, VALUES

CREATE TABLESPACE

CREATE TABLESPACE — 新しいテーブル空間を定義する

概要

```
CREATE TABLESPACE tablespace_name
    [ OWNER { new_owner | CURRENT_USER | SESSION_USER } ]
    LOCATION 'directory'
    [ WITH ( tablespace_option = value [, ... ] ) ]
```

説明

CREATE TABLESPACEはクラスタ全体で使用できるテーブル空間を新規に登録します。このテーブル空間名は、データベースクラスタ内の既存のテーブル空間名と異なるものでなければなりません。

スーパーユーザはテーブル空間を使用することで、データベースオブジェクト（テーブルやインデックスなど）が入ったデータファイルを格納できる、ファイルシステム上の別の場所を定義できます。

適切な権限を持つユーザは、CREATE DATABASEやCREATE TABLE、CREATE INDEX、ADD CONSTRAINTコマンドにtablespace_nameを渡すことで、これらのオブジェクトを指定したテーブル空間に格納することができます。

> **警告**
>
> テーブル空間は、それが定義されているクラスタと独立して使うことはできません。22.6を参照してください。

パラメータ

tablespace_name

作成するテーブル空間の名前です。pg_から始まる名前はシステムのテーブル空間用に予約されているので使用することはできません。

user_name

テーブル空間を所有するユーザの名前です。省略時はデフォルトでコマンドを実行したユーザとなります。テーブル空間を作成できるのはスーパーユーザのみですが、テーブル空間の所有権を非スーパーユーザに割り当てることは可能です。

directory

テーブル空間用に使用するディレクトリです。このディレクトリは存在していなければなりませんし(CREATE TABLESPACE はディレクトリを作成しません)、空であるべきです。また、PostgreSQLのシステムユーザが所有していなければなりません。このディレクトリは絶対パス名で指定する必要があります。

tablespace_option

値を設定あるいはリセットするテーブル空間のパラメータです。現在、利用可能なパラメータはseq_page_cost、random_page_cost、effective_io_concurrency、maintenance_io_concurrencyだけです。特定のテーブル空間について、その値を設定すると、プランナがそのテーブル空間内のテーブルからページを読み込むコストの通常の推定値やエクゼキュータの先読みの振る舞いについて、通常参照する同じ名前の設定パラメータ(seq_page_cost、random_page_cost、effective_io_concurrency、maintenance_io_concurrencyを参照)よりも優先します。テーブル空間が、他のI/Oサブシステムより高速あるいは低速なディスク上にある時は、これが有効でしょう。

注釈

テーブル空間は、シンボリックリンクをサポートしているシステムでのみサポートされます。

トランザクションブロック内でCREATE TABLESPACEを実行することはできません。

例

ファイルシステムの/data/dbsにテーブル空間dbspaceを作成するためには、まずオペレーティングシステムの機能を使ってディレクトリを作成し、正しい所有権を設定します。

```
mkdir /data/dbs
chown postgres:postgres /data/dbs
```

次にPostgreSQL内でテーブル空間作成コマンドを発行します。

```
CREATE TABLESPACE dbspace LOCATION '/data/dbs';
```

他のデータベースユーザが所有するテーブル空間を作成するには、以下のようにコマンドを使います。

```
CREATE TABLESPACE indexspace OWNER genevieve LOCATION '/data/indexes';
```

互換性

CREATE TABLESPACEはPostgreSQLの拡張です。

関連項目

CREATE DATABASE, CREATE TABLE, CREATE INDEX, DROP TABLESPACE, ALTER TABLESPACE

CREATE TEXT SEARCH CONFIGURATION

CREATE TEXT SEARCH CONFIGURATION — 新しいテキスト検索設定を定義する

概要

```
CREATE TEXT SEARCH CONFIGURATION name (
    PARSER = parser_name |
    COPY = source_config
)
```

説明

CREATE TEXT SEARCH CONFIGURATIONは新しいテキスト検索設定を作成します。テキスト検索設定は、文字列をトークンに分割するためのテキスト検索パーサを指定すると共に、どのトークンを検索対象とするかを決定するために使用する辞書を特定します。

パーサのみが指定された場合、新しいテキスト検索設定の初期状態では、トークン型と辞書との対応付けが存在しません。そのため、すべての単語が無視されます。この設定を実用できるようにするためには、後でALTER TEXT SEARCH CONFIGURATIONコマンドを使用して、対応付けを作成しなければなりません。この他に、既存のテキスト検索設定を複製することも可能です。

スキーマ名が指定された場合、テキスト検索設定は指定されたスキーマ内に作成されます。そうでなければ、現在のスキーマ内に作成されます。

テキスト検索設定を定義したユーザが所有者になります。

詳細は第12章を参照してください。

パラメータ

name

作成するテキスト検索設定の名称です。名前をスキーマ修飾することができます。

parser_name

この設定で使用するテキスト検索パーサの名称です。

source_config

複製するテキスト検索設定の名称です。

注釈

PARSERとCOPYオプションを同時に使用することはできません。既存の設定が複製される場合は、パーサの設定も複製されるためです。

互換性

標準SQLにはCREATE TEXT SEARCH CONFIGURATION文はありません。

関連項目

ALTER TEXT SEARCH CONFIGURATION, DROP TEXT SEARCH CONFIGURATION

CREATE TEXT SEARCH DICTIONARY

CREATE TEXT SEARCH DICTIONARY — 新しいテキスト検索辞書を定義する

概要

```
CREATE TEXT SEARCH DICTIONARY name (
    TEMPLATE = template
    [, option = value [, ... ]]
)
```

説明

CREATE TEXT SEARCH DICTIONARYは新しいテキスト検索辞書を作成します。テキスト検索辞書は、検索の際に何を対象とし、何を対象としないのかについての方法を指定します。実際に作業を行う関数を指定するテキスト検索テンプレートに、辞書は依存します。通常辞書は、テンプレートの関数の動作の詳細を制御するいくつかのオプションを提供します。

スキーマ名が指定された場合、テキスト検索辞書は指定されたスキーマ内に作成されます。そうでなければ、現在のスキーマ内に作成されます。

テキスト検索辞書を定義したユーザが所有者になります。

詳細は第12章を参照してください。

パラメータ

name

作成するテキスト検索辞書の名称です。名前をスキーマ修飾することができます。

template

この辞書の基本動作を定義するテキスト検索テンプレートの名称です。

option

辞書に対して設定されるテンプレート固有のオプションの名称です。

value

テンプレート固有のオプションで使用される値です。値が単純な識別子または数値でない場合、引用符で括らなければなりません。（常に引用符で括ることもできます。）

オプションは任意の順序で指定することができます。

例

次の例で示すコマンドは、非標準のストップワードのリストを持つ、雪だるま式に増加する辞書を作成します。

```
CREATE TEXT SEARCH DICTIONARY my_russian (
    template = snowball,
    language = russian,
    stopwords = myrussian
);
```

互換性

標準SQLにはCREATE TEXT SEARCH DICTIONARY文はありません。

関連項目

ALTER TEXT SEARCH DICTIONARY, DROP TEXT SEARCH DICTIONARY

CREATE TEXT SEARCH PARSER

CREATE TEXT SEARCH PARSER — 新しいテキスト検索パーサを定義する

概要

```
CREATE TEXT SEARCH PARSER name (
    START = start_function ,
    GETTOKEN = gettoken_function ,
    END = end_function ,
    LEXTYPES = lextypes_function
    [, HEADLINE = headline_function ]
)
```

説明

CREATE TEXT SEARCH PARSERは新しいテキスト検索パーサを作成します。テキスト検索パーサは、テキスト文字列をトークンに分割し、トークンに型（カテゴリ）を割り当てる方法を定義します。パーサ自体は特別有用なものではありませんが、検索するためには、数個のテキスト検索辞書と共にテキスト検索設定と関連付けされなければなりません。

スキーマ名が指定された場合、テキスト検索パーサは指定されたスキーマ内に作成されます。そうでなければ、現在のスキーマに作成されます。

CREATE TEXT SEARCH PARSERを使用するには、スーパーユーザでなければなりません。（おかしなテキスト検索パーサ定義はサーバを混乱させ、クラッシュさせる可能性があるため、この制限があります。）

詳細は第12章を参照してください。

パラメータ

name

作成するテキスト検索パーサの名称です。この名前はスキーマ修飾することができます。

start_function

パーサの開始関数の名称です。

gettoken_function

次トークンを取り出すパーサの関数の名称です。

end_function

パーサの終了関数の名称です。

`lextypes_function`

パーサのLEXTYPE関数（生成するトークン型集合に関する情報を返す関数）の名称です。

`headline_function`

パーサの見出し関数（トークン集合を要約する関数）の名称です。

関数名は必要に応じてスキーマ修飾可能です。各種関数の引数リストは事前に決められているため、引数型の指定はありません。見出し関数以外の関数はすべて必要です。

引数は、上で示した順序だけではなく、任意の順序で記述することができます。

互換性

標準SQLには`CREATE TEXT SEARCH PARSER`文はありません。

関連項目

ALTER TEXT SEARCH PARSER, DROP TEXT SEARCH PARSER

CREATE TEXT SEARCH TEMPLATE

CREATE TEXT SEARCH TEMPLATE — 新しいテキスト検索テンプレートを定義する

概要

```
CREATE TEXT SEARCH TEMPLATE name (
    [ INIT = init_function , ]
    LEXIZE = lexize_function
)
```

説明

CREATE TEXT SEARCH TEMPLATEは新しいテキスト検索テンプレートを作成します。テキスト検索テンプレートは、テキスト検索辞書を実装する関数を定義します。テンプレートはそれ自体では有用ではありませんが、使用される辞書として実体化されなければなりません。通常この辞書はテンプレート関数に渡すパラメータを指定します。

スキーマ名が指定された場合、テキスト検索テンプレートは指定したスキーマに作成されます。そうでなければ、現在のスキーマに作成されます。

CREATE TEXT SEARCH TEMPLATEを使用するには、スーパーユーザでなければなりません。おかしなテキスト検索テンプレート定義はサーバを混乱させ、クラッシュさせる可能性があるため、この制限があります。辞書とテンプレートを分離させた理由は、テンプレートにより辞書定義の「安全でない」側面を隠蔽化することです。辞書を定義する時に設定できるパラメータは、非特権ユーザが設定しても安全なものです。このため、辞書の作成では特権操作は必要ありません。

詳細は第12章を参照してください。

パラメータ

name

作成するテキスト検索テンプレートの名称です。この名前はスキーマ修飾することができます。

init_function

テンプレートの初期化関数の名称です。

lexize_function

テンプレートの字句化関数の名称です。

関数名は必要に応じてスキーマ修飾可能です。各種関数の引数リストは事前に定められているので、引数型の指定はありません。字句化関数は必須ですが、初期化関数は省略可能です。

引数は、上で示した順序だけではなく、任意の順序で記述することができます。

互換性

標準SQLにはCREATE TEXT SEARCH TEMPLATE文はありません。

関連項目

ALTER TEXT SEARCH TEMPLATE, DROP TEXT SEARCH TEMPLATE

CREATE TRANSFORM

CREATE TRANSFORM — 新しい変換を定義する

概要

```
CREATE [ OR REPLACE ] TRANSFORM FOR type_name LANGUAGE lang_name (
    FROM SQL WITH FUNCTION from_sql_function_name [ (argument_type [, ...]) ],
    TO SQL WITH FUNCTION to_sql_function_name [ (argument_type [, ...]) ]
);
```

説明

CREATE TRANSFORMは新しい変換を定義します。CREATE OR REPLACE TRANSFORMは新しい変換を作成するか、あるいは既存の変換を置換します。

変換はデータ型を手続き言語にどのように適合させるかを定義します。例えばhstore型を使ってPL/Pythonの関数を書くとき、PL/Pythonはhstoreの値をPythonの環境でどのように表現するか、事前の知識がありません。言語の実装は通常、デフォルトでテキスト表現を使いますが、これは例えば連想配列やリストの方がより適切な場合には不便です。

変換では次の2つの関数を指定します。

- 「from SQL」関数では、型をSQL環境から言語へと変換します。この関数は、その言語で記述された関数の引数で呼び出されます。
- 「to SQL」関数では、型を言語からSQL環境へと変換します。この関数は、その言語で記述された関数の戻り値で呼び出されます。

これらの関数を両方とも提供する必要はありません。一方が指定されなければ、必要な時はその言語独自のデフォルトの動作が使われます。（ある方向への変換がまったく起きないようにするためには、必ずエラーを発生させる変換関数を作成することもできます。）

変換を作成するには、その型を所有し、そのUSAGE権限があること、言語のUSAGE権限があること、from-SQL関数あるいはto-SQL関数を指定する場合は、それらを所有し、そのEXECUTE権限があることが必要です。

パラメータ

type_name

変換の対象となるデータ型の名前です。

lang_name

変換の対象となる言語の名前です。

`from_sql_function_name[(argument_type [, ...])]`

型をSQL環境から言語に変換する関数の名前です。internal型の引数を1つとり、internal型の値を戻します。実引数は変換される型になり、関数はそうであるとしてコーディングされます。(しかし、少なくとも1つのinternal型の引数がなければ、internalを戻すSQLレベルの関数を宣言することができません。) 実際の戻り値は、言語の実装に依存したものになります。引数リストが指定されない場合、関数名はスキーマ内で一意でなければなりません。

`to_sql_function_name[(argument_type [, ...])]`

型を言語からSQL環境に変換する関数の名前です。internal型の引数を1つとり、変換の型であるデータ型を戻します。実引数の値は言語の実装に依存したものになります。引数リストが指定されない場合、関数名はスキーマ内で一意でなければなりません。

注釈

変換を削除するにはDROP TRANSFORMを使います。

例

hstore型で言語plpythonuの変換を作成するため、まず以下のように型と言語を設定します。

```
CREATE TYPE hstore ...;

CREATE EXTENSION plpythonu;
```

Then create the necessary functions:

```
CREATE FUNCTION hstore_to_plpython(val internal) RETURNS internal
LANGUAGE C STRICT IMMUTABLE
AS ...;

CREATE FUNCTION plpython_to_hstore(val internal) RETURNS hstore
LANGUAGE C STRICT IMMUTABLE
AS ...;
```

そして最後に、それらを互いに接続する変換を以下のように作成します。

```
CREATE TRANSFORM FOR hstore LANGUAGE plpythonu (
    FROM SQL WITH FUNCTION hstore_to_plpython(internal),
    TO SQL WITH FUNCTION plpython_to_hstore(internal)
);
```

現実には、これらのコマンドは拡張の中にまとめられているでしょう。

contribの下には変換を提供するいくつかの拡張が含まれており、それらは現実世界での例となります。

互換性

この構文のCREATE TRANSFORMはPostgreSQLの拡張です。標準SQLにはCREATE TRANSFORMコマンドがありますが、それはデータ型をクライアントの言語に適合させるためのものです。その使用法はPostgreSQLではサポートされていません。

関連項目

CREATE FUNCTION, CREATE LANGUAGE, CREATE TYPE, DROP TRANSFORM

CREATE TRIGGER

CREATE TRIGGER — 新しいトリガを定義する

概要

```
CREATE [ CONSTRAINT ] TRIGGER name { BEFORE | AFTER | INSTEAD OF } { event [ OR ... ] }
    ON table_name
    [ FROM referenced_table_name ]
    [ NOT DEFERRABLE | [ DEFERRABLE ] [ INITIALLY IMMEDIATE | INITIALLY DEFERRED ] ]
    [ REFERENCING { { OLD | NEW } TABLE [ AS ] transition_relation_name } [ ... ] ]
    [ FOR [ EACH ] { ROW | STATEMENT } ]
    [ WHEN ( condition ) ]
    EXECUTE { FUNCTION | PROCEDURE } function_name ( arguments )



ここでeventは以下のいずれかを取ることができます。



    INSERT
    UPDATE [ OF column_name [, ... ] ]
    DELETE
    TRUNCATE
```

説明

CREATE TRIGGERは新しいトリガを作成します。作成したトリガは指定したテーブル、ビューまたは外部テーブルと関連付けられ、そのテーブルに特定の操作が行われた時に指定した関数function_nameを実行します。

トリガでは、起動のタイミングとして、行への操作が開始される前（制約条件のチェックとINSERT、UPDATEまたはDELETEが行われる前）、操作が完了した後（制約条件がチェックされ、INSERT、UPDATEまたはDELETEが完了した後）、操作の代わり（ビューにおける挿入、更新、削除の場合）のいずれかを指定することができます。イベントの前または代わりにトリガが起動する場合、そのトリガは対象行に対する操作を省略したり、（INSERTとUPDATEの操作時のみ）挿入する行を変更したりすることができます。イベントの後にトリガが起動する場合、他のトリガの影響を含む全ての変更が、トリガに対して「可視」状態となります。

FOR EACH ROW付きのトリガは、その操作によって変更される行ごとに1回ずつ呼び出されます。例えば、10行に影響を与えるDELETE操作は、対象リレーション上のすべてのON DELETEトリガを、削除される各行について1回ずつ、個別に10回呼び出すことになります。反対に、FOR EACH STATEMENT付きのトリガは、その操作によって何行変更されたかにかかわらず、任意の操作ごとに1回のみ実行されます（変更対象が0行となる操作でも、適用できるすべてのFOR EACH STATEMENTトリガが実行されます）。

トリガイベントのINSTEAD OFとして発行されるように指定されたトリガはFOR EACH ROW印を付けなければなりません。またビュー上にのみ定義することができます。ビューに対するBEFOREおよびAFTERトリガはFOR EACH STATEMENT印を付けなければなりません。

さらに、FOR EACH STATEMENTのみですが、トリガをTRUNCATEに対して発行するように定義することができます。

以下の表にどの種類のトリガがテーブル、ビュー、外部テーブルに対して使用できるかをまとめます。

いつ	イベント	行レベル	文レベル
BEFORE	INSERT/UPDATE/DELETE	テーブル、および外部テーブル	テーブル、ビューおよび外部テーブル
	TRUNCATE	—	テーブル
AFTER	INSERT/UPDATE/DELETE	テーブルおよび外部テーブル	テーブル、ビューおよび外部テーブル
	TRUNCATE	—	テーブル
INSTEAD OF	INSERT/UPDATE/DELETE	ビュー	—
	TRUNCATE	—	—

またトリガ定義では、論理値のWHEN条件を指定することができ、これによってトリガを発行すべきかどうかが判定されます。行レベルのトリガでは、WHEN条件は行の列の古い値、新しい値、またはその両方で検証することができます。文レベルのトリガでもWHEN条件を持たせることができますが、条件としてテーブル内のどの値も参照することができませんので、この機能はあまり有用ではありません

同一イベントに同じ種類の複数のトリガが定義された場合、名前のアルファベット順で実行されます。

CONSTRAINTオプションが指定された場合、このコマンドは*制約トリガ*を作成します。これは、SET CONSTRAINTSを使用してトリガを発行するタイミングを調整することができるという点を除き、通常のトリガと同じです。制約トリガは（外部テーブルではない）普通のテーブルのAFTER ROWトリガでなければなりません。トリガイベントを引き起こした文の最後、またはそれを含むトランザクションの最後のいずれかで発行することができます。後者の場合、*遅延*と呼ばれます。SET CONSTRAINTSを使用することで、強制的に待機中の遅延トリガの発行を即座に行わせることができます。制約トリガは、実装する制約に違反した時に例外を発生するものと想定されています。

REFERENCINGオプションは*遷移リレーション*の収集を有効にします。遷移リレーションとは現在のSQL文によって挿入、削除または修正されたすべての行を含む行集合です。この機能により、トリガはSQL文によって行われたことを、一度に1行ずつだけではなく、全体のビューとして参照することができます。このオプションは、制約トリガではないAFTERトリガについてのみ使うことができます。また、トリガがUPDATEトリガの場合、column_nameのリストを指定してはいけません。OLD TABLEは一度だけ指定することができ、UPDATEまたはDELETEのときに実行されるトリガにのみ指定できます。これは文によって更新または削除されるすべての行の*更新前イメージ*を含む遷移リレーションを作成します。同様に、NEW TABLEは一度だけ指定することができ、UPDATEまたはINSERTのときに実行されるトリガにのみ指定できます。これは、文によって更新または挿入されるすべての行の*更新後イメージ*を含む遷移リレーションを作成します。

SELECTはまったく行を変更しないため、SELECTトリガを作成することはできません。SELECTトリガが必要に見える問題には、ルールやビューが現実的な解決策を提供できるでしょう。

トリガに関するより詳細については、第38章を参照してください。

パラメータ

name

新しいトリガに付与する名前です。同じテーブルの他のトリガと異なる名前にする必要があります。名前をスキーマ修飾することはできません。トリガはそのテーブルのスキーマを引き継ぎます。制約トリガの場合、この名前がSET CONSTRAINTSを使用してトリガの動作を変更する時に使用されます。

BEFORE
AFTER
INSTEAD OF

関数の呼び出しをイベントの前に行うか後に行うか、それとも代替として行うかを決定します。制約トリガではAFTERとしてしか指定することができません。

event

INSERT、UPDATE、DELETE、TRUNCATEのいずれかが入ります。このパラメータは、トリガを起動するイベントを指定します。遷移リレーションが要求される場合を除き、ORを使用して、複数のイベントを指定することができます。

UPDATEイベントでは、以下の構文を使用して列リストを指定することができます。

```
UPDATE OF column_name1 [, column_name2 ... ]
```

このトリガは、UPDATEコマンドの対象として列挙された列のいずれか少なくとも1つの列が指定された場合に、もしくは列挙された列の1つがUPDATEの対象の列に依存する生成列である場合に、発行されます。

INSTEAD OF UPDATEイベントでは列リストを使用できません。遷移リレーションを要求する場合も列リストを指定することはできません。

table_name

トリガを作成するテーブル、ビューまたは外部テーブルの名前です（スキーマ修飾名も可）。

referenced_table_name

制約で参照される他のテーブルの名前（スキーマ修飾可）です。このオプションは外部キー制約で使用されるものであり、一般利用を推奨しません。これは制約トリガでのみ指定することができます。

DEFERRABLE
NOT DEFERRABLE
INITIALLY IMMEDIATE
INITIALLY DEFERRED

トリガのデフォルトのタイミングです。これらの制約オプションについてはCREATE TABLE文書を参照してください。これは制約トリガでのみ指定することができます。

REFERENCING

このキーワードは、トリガの文の遷移リレーションへのアクセスを提供する1つまたは2つのリレーション名の宣言の直前に起きます。

OLD TABLE
NEW TABLE

この句は、それに続くリレーション名が更新前イメージの遷移リレーションなのか、更新後イメージの遷移リレーションなのかを示します。

transition_relation_name

この遷移リレーションについて、トリガ内で使用される(修飾されていない)名前です。

FOR EACH ROW
FOR EACH STATEMENT

このパラメータは、トリガ関数を、トリガイベントによって影響を受ける行ごとに1回起動するか、SQL文ごとに1回のみ起動するかを指定します。どちらも指定されない場合は、FOR EACH STATEMENTがデフォルトです。制約トリガはFOR EACH ROWのみ指定することができます。

condition

トリガ関数を実際に実行するか否かを決定する論理式です。WHENが指定された場合、conditionがtrueを返す場合のみ関数が呼び出されます。FOR EACH ROWトリガでは、WHEN条件で、それぞれOLD.column_name、NEW.column_nameと記述することで、古い行の値、新しい行の値、またはその両方の列を参照することができます。当然ながらINSERTトリガではOLDを参照することはできませんし、DELETEトリガではNEWを参照することはできません。

INSTEAD OFトリガはWHEN条件をサポートしません。

現時点ではWHEN条件に副問い合わせを含めることはできません。

制約トリガでは、WHEN条件の評価は遅延されず、行の更新操作が行われた直後に発生することに注意してください。この条件が真と評価されなかった場合、トリガは遅延実行用のキューに入りません。

function_name

ユーザが提供する関数です。この関数は、引数を取らずtrigger型を返すよう定義されます。トリガが起動した時に実行されます。

CREATE TRIGGERの構文では、キーワードFUNCTIONとPROCEDUREは等価ですが、参照されている関数はどちらの場合でも関数でなければならず、プロシージャであってはなりません。ここでキーワードPROCEDUREを使うことは、歴史的なものであり廃止予定です。

arguments

トリガ実行時に関数に渡される引数をカンマで区切ったリストで、省略可能です。引数として指定するのは、リテラル文字列定数です。単純な名前および数値定数を記述できますが、全て文字列に変換されます。関数内でこれらの引数にアクセスする方法について調べるためには、トリガ関数を実装した言語の説明を参照してください。通常の関数引数とは異なる場合があります。

注釈

テーブルにトリガを作成するには、ユーザがそのテーブルに対しTRIGGER権限を持っている必要があります。またユーザはトリガ関数に対しEXECUTE権限を持たなければなりません。

トリガを削除するためにはDROP TRIGGERを使用してください。

列指定のトリガ(UPDATE OF column_name構文で定義されたトリガ)は、列挙された列のいずれかがUPDATEコマンドのSETリスト内に対象として指定された場合に発行されます。BEFORE UPDATEトリガにより行の内容になされた変更は考慮されないため、トリガが発行されない場合であっても、列の値が変更されることはあります。反対に、UPDATE ... SET x = x ...のようなコマンドは、列の値が変更されませんが、x列に対するトリガが発行されます。

自身でトリガのコードを書かなくても、よくある問題を解決するために使うことのできる組み込みのトリガ関数が多少あります。9.28を参照してください。

BEFOREトリガにおいてWHEN条件は関数が実行される、またはされそうな直前に評価されます。このためWHENの使用はトリガ関数の先頭で同一の条件を試験することと実質的に違いはありません。この条件で確認できるNEW行が現在の値であり、それまでのトリガで変更されている可能性があることに、特に注意して下さい。またBEFOREトリガのWHEN条件では、NEW行のシステム列(ctidなど)はまだ設定されていないので、検査することができません。

AFTERトリガにおいて、WHEN条件は行の更新を行った直後に評価され、文の最後でトリガを発行するためにイベントを保持すべきかどうかを決定します。このためAFTERトリガのWHEN条件は真を返さない場合、イベントを保持する必要もありませんし、文の最後の行を再度取り出す必要もありません。これにより、トリガをわずかな行のみに対して発行する必要がある場合、多くの行を変更する文を非常に高速にすることができます。

場合によっては1つのSQLコマンドが2種類以上のトリガを発行することがあります。例えば、ON CONFLICT DO UPDATE句のあるINSERTでは、挿入と更新の両方の操作が発生するかもしれないので、必要に応じて両方の種類のトリガを発行します。トリガに提供される遷移リレーションはトリガのイベント種類毎に個別のものです。従って、INSERTトリガには挿入された行だけが見え、一方でUPDATEトリガには更新された行だけが見えます。

ON UPDATE CASCADEやON DELETE SET NULLなど外部キーを強制する動作によって起こる行の更新や削除は、それを起こしたSQLコマンドの一部であるとみなされます(このような動作は決して遅延実行されないことに注意してください)。影響を受けたテーブルの関連するトリガが発行されるため、これはSQLコマンドの種類と直接には一致しないトリガが発行される別のケースとなります。単純な場合、遷移リレーションを要求するトリガは、元となる1つのSQLコマンドによって起こされたテーブルへのすべての変更を、一つの遷移リレーションとして見ることになります。しかし、遷移リレーションを要求するAFTER ROWトリガの存在により、一つのSQLコマンドによって発生する外部キーを強制する動作が複数のステップに分割され、各ステップがそれぞれの遷移リレーションを持つという場合もあります。そのような場合、すべての文レベルのトリガは1つの遷移リレーションの集合の作成に対して1度ずつ呼び出され、それによりトリガが遷移リレーション内の変更された行をちょうど一度だけ見ることを確実にしています。

ビューに付けられている文レベルのトリガは、ビューに対する操作が行レベルのINSTEAD OFトリガによって取り扱われた時にのみ発行されます。ビューに対する操作がINSTEAD OFルールによって取り扱われる場合は、ビューを指定した元の文の代わりに、そのルールが出力した文が実行されます。それにより、発行されるトリガは、置き換えられた文によって指定されたテーブルに付けられたトリガとなります。同様に、ビューが自動更新可能ならば、操作は、ビューの基底テーブル上の操作に自動的に書き換えられる文によって取り扱われます。その結果、発行されるのは基底テーブルの文レベルのトリガとなります。

パーティションテーブルに行レベルのトリガを作ると、存在するパーティションすべてに同一のトリガがつくられます。そして、後から作られたり追加されるパーティションも同一のトリガを含みます。パーティションが親から切り離された場合、トリガは削除されます。パーティションテーブルのトリガはINSTEAD OFにはできません。

パーティションテーブルや継承した子テーブルがあるテーブルを変更したとき、明示的に指定されたテーブルに付けられている文レベルのトリガが発行されますが、パーティションや子テーブルに付けられている文レベルのトリガは発行されません。対照的に、問合せ中で明示的に指定されていなくても、行レベルのトリガはすべての変更されたパーティションや子テーブルに対して発行されます。REFERENCING句で指定された遷移リレーションのある文レベルのトリガが定義されている場合、行の変更前イメージおよび変更ごイメージは、変更されたすべてのパーティションおよび子テーブルから見ることができます。継承された子テーブルの場合、行イメージはトリガが付けられたテーブルに存在する列だけしか含みません。現在のところ、遷移リレーションのある行レベルトリガは、パーティションや継承した子テーブルには定義できません。

例

テーブルaccountsの行が更新される直前に関数check_account_updateを実行します。

```
CREATE TRIGGER check_update
    BEFORE UPDATE ON accounts
    FOR EACH ROW
    EXECUTE FUNCTION check_account_update();
```

上と同じです。しかし、列balanceがUPDATEコマンドの対象として指定された場合のみ実行されます。

```
CREATE TRIGGER check_update
    BEFORE UPDATE OF balance ON accounts
    FOR EACH ROW
    EXECUTE FUNCTION check_account_update();
```

以下の構文では、列balanceが実際に変更された場合のみ関数が実行されます。

```
CREATE TRIGGER check_update
    BEFORE UPDATE ON accounts
    FOR EACH ROW
    WHEN (OLD.balance IS DISTINCT FROM NEW.balance)
    EXECUTE FUNCTION check_account_update();
```

何か変更された場合のみにaccountsの更新のログを取る関数を呼び出します。

```
CREATE TRIGGER log_update
    AFTER UPDATE ON accounts
    FOR EACH ROW
    WHEN (OLD.* IS DISTINCT FROM NEW.*)
    EXECUTE FUNCTION log_account_update();
```

ビューの背後にあるテーブルに行を挿入するために、各行に対して関数view_insert_rowを実行します。

```
CREATE TRIGGER view_insert
    INSTEAD OF INSERT ON my_view
    FOR EACH ROW
```

```
    EXECUTE FUNCTION view_insert_row();
```

各文に対して関数check_transfer_balances_to_zeroを実行して、transferの行が相殺してゼロになることを確認します。

```
CREATE TRIGGER transfer_insert
    AFTER INSERT ON transfer
    REFERENCING NEW TABLE AS inserted
    FOR EACH STATEMENT
    EXECUTE FUNCTION check_transfer_balances_to_zero();
```

各行に対して関数check_matching_pairsを実行して、対応する組み合わせに対して同じ時に（同じ文により）変更されていることを確認します。

```
CREATE TRIGGER paired_items_update
    AFTER UPDATE ON paired_items
    REFERENCING NEW TABLE AS newtab OLD TABLE AS oldtab
    FOR EACH ROW
    EXECUTE FUNCTION check_matching_pairs();
```

38.4には、C言語で作成されたトリガ関数の完全な例があります。

互換性

PostgreSQLにおけるCREATE TRIGGER文は標準SQLのサブセットを実装したものです 現在は、PostgreSQLには、次の機能がありません。

- AFTERトリガの遷移テーブル名はREFERENCING句を使って標準SQLの方法で指定できますが、FOR EACH ROWトリガで使用される行変数はREFERENCING句で指定することができません。それはトリガ関数が書かれる言語に依存する方法で利用できますが、各言語によって決まった方法になります。一部の言語は、REFERENCING句がOLD ROW AS OLD NEW ROW AS NEWとなっているかのように動作します。
- 標準SQLでは列を指定したUPDATEトリガでも遷移テーブルを使うことができますが、その場合遷移テーブルで見ることができる行の集合はトリガの列リストに依存します。これは現在のところPostgreSQLでは実装されていません。
- PostgreSQLでは、トリガ動作として、ユーザ定義関数の実行しか認めていません。標準では、多数の他のSQLコマンドを実行させることができます。例えば、トリガ動作としてCREATE TABLEを実行させることも可能です。この制限を回避する方法は簡単です。必要なコマンドを実行するユーザ定義関数を作成すればよいのです。

SQLでは、複数のトリガは、作成時刻順に起動すべきであると規定しています。PostgreSQLでは名前順です。この方が便利だと考えられるからです。

SQLでは、数珠繋ぎの削除に対するBEFORE DELETEは、数珠繋ぎのDELETEが完了した後に発行するものと規定しています。PostgreSQLでは、BEFORE DELETEは常に削除操作よりも前に、それも起点となる削除よりも前に行われます。この方がより一貫性があると考えられいます。また、参照整合性に関する動作により引き起

こされる更新を実行している間に、BEFOREトリガが行を更新し、更新を妨げるような場合の動作も標準に従わないものがあります。これは、制約違反となるかもしれませんし、参照整合性制約に合わないデータを格納してしまうかもしれません。

ORを使用して単一トリガに複数の動作を指定する機能は、標準SQLに対するPostgreSQLの拡張です。

TRUNCATEでのトリガ発行機能、および、ビューに対する文レベルのトリガの定義機能は標準SQLに対するPostgreSQLの拡張です。

CREATE CONSTRAINT TRIGGERは標準SQLに対するPostgreSQLの拡張です。

関連項目

ALTER TRIGGER, DROP TRIGGER, CREATE FUNCTION, SET CONSTRAINTS

CREATE TYPE

CREATE TYPE — 新しいデータ型を定義する

概要

```
CREATE TYPE name AS
    ( [ attribute_name data_type [ COLLATE collation ] [, ... ] ] )

CREATE TYPE name AS ENUM
    ( [ 'label' [, ... ] ] )

CREATE TYPE name AS RANGE (
    SUBTYPE = subtype
    [ , SUBTYPE_OPCLASS = subtype_operator_class ]
    [ , COLLATION = collation ]
    [ , CANONICAL = canonical_function ]
    [ , SUBTYPE_DIFF = subtype_diff_function ]
)

CREATE TYPE name (
    INPUT = input_function,
    OUTPUT = output_function
    [ , RECEIVE = receive_function ]
    [ , SEND = send_function ]
    [ , TYPMOD_IN = type_modifier_input_function ]
    [ , TYPMOD_OUT = type_modifier_output_function ]
    [ , ANALYZE = analyze_function ]
    [ , INTERNALLENGTH = { internallength | VARIABLE } ]
    [ , PASSEDBYVALUE ]
    [ , ALIGNMENT = alignment ]
    [ , STORAGE = storage ]
    [ , LIKE = like_type ]
    [ , CATEGORY = category ]
    [ , PREFERRED = preferred ]
    [ , DEFAULT = default ]
    [ , ELEMENT = element ]
    [ , DELIMITER = delimiter ]
    [ , COLLATABLE = collatable ]
)

CREATE TYPE name
```

説明

CREATE TYPEは、現在のデータベースで使用できる新しいデータ型を登録します。型を定義したユーザがその所有者となります。

スキーマ名が与えられている場合、型は指定されたスキーマに作成されます。スキーマ名がなければ、その型は現在のスキーマに作成されます。型名は、同じスキーマにある既存の型もしくはドメインとは、異なる名前にする必要があります（さらに、テーブルはデータ型と関連しているため、データ型名は同じスキーマのテーブル名とも競合しないようにしてください）。

上の構文概要に示すように、CREATE TYPEには5つの構文があります。これらはそれぞれ、複合型, 列挙型、範囲型、基本型、シェル型を作成します。これらの内最初の4個についてはここで順番に説明します。シェル型は、後で定義される型用の単なるプレースホルダで、型名以外のパラメータをつけずにCREATE TYPEを実行することで作成されます。シェル型は、範囲型と基本型を作成するときの前方参照として必要となるもので、それぞれの節で説明します。

複合型

CREATE TYPEの最初の構文を使用すると、複合型を作成できます。複合型は、属性名およびデータ型のリストにより指定されます。データ型の照合順序が設定可能である場合、属性の照合順序も指定することができます。複合型は本質的にはテーブルの行型と同じです。しかし、型を定義することだけが必要なのであれば、CREATE TYPEを使用することで、実際のテーブルを作成する必要がなくなります。スタンドアローンの複合型は、例えば関数の引数や戻り値の型として有用です。

複合型を作成するためには、すべての属性型に対してUSAGE権限を持たなければなりません。

列挙型

CREATE TYPEの2つ目の構文を使用すると、8.7で説明する列挙型（enum）を作成します。列挙型は、引用符付きのラベルのリストを取ります。ラベルはNAMEDATALEN（PostgreSQLの標準のビルドでは64バイト）バイトよりも少ない長さでなければなりません。(ラベルのない列挙型を作成できますが、ALTER TYPEを使ってラベルを少なくとも1つ追加するまでは、そのような型は値を保持するのに使えません。)

範囲型

CREATE TYPEの三番目の構文は、8.17で説明する範囲型を新規に作成します。

範囲型のsubtypeは、関連する（範囲型の値を順序を決定するための）b-tree演算子クラスを持つ任意の型を取ることができます。通常、派生元型のデフォルトのb-tree演算子クラスが順序を決定するために使用されます。デフォルト以外の演算子クラスを使用するためには、subtype_opclassでその名前を指定してください。派生元型が照合順序変更可能であり、範囲の順序付けでデフォルト以外の照合順序を使用したい場合は、collationオプションで使用したい照合順序を指定してください。

canonical関数(省略可能)は、定義する範囲型の引数を1つ取り、同じ型の値を返さなければなりません。これは適切な時に範囲値を正規形式に変換するために使用されます。詳細については8.17.8を参照してくださ

い。canonical関数を作成することは多少厄介です、というのは、範囲型を定義できるようになる前に定義されている必要があるからです。このためには、まず、名前と所有者以外の属性を持たないプレースホルダであるシェル型を作成しなければなりません。これは、他にパラメータをつけずにCREATE TYPE nameを実行することで行われます。その後、このシェル型を引数と結果として使用する関数を宣言することができます。最後に同じ名前を用いて範囲型を宣言することができます。これは自動的にシェル型の項目を有効な範囲型に置き換えます。

subtype_diff関数(省略可能)は、subtype型の2つの値を引数として取り、与えられた2つの値の差異を表すdouble precision型を返さなければなりません。これは省略することができますが、提供することでその範囲型の列に対するGiSTインデックスの効率を大きく向上させることができます。詳細については8.17.8を参照してください。

基本型

CREATE TYPEの4つ目の構文を使用すると、基本型(スカラ型)を新しく作成できます。新しい基本型を作成するにはスーパーユーザでなければなりません。(エラーがある型定義が混乱を招き、サーバがクラッシュすることすらあるため、この制限がなされました。)

パラメータは、上述の順番である必要はなく、任意の順番で指定することができ、多くは省略可能です。型を定義する前に、(CREATE FUNCTIONを用いて)2つ以上の関数を登録しておく必要があります。サポート関数であるinput_functionとoutput_functionは必須です。receive_function関数、send_function関数、type_modifier_input_function関数、type_modifier_output_function関数、およびanalyze_function関数は省略可能です。通常、これらの関数は、C言語やその他の低レベル言語で作成されなければなりません。

input_functionは、型のテキストによる外部表現を内部表現形式に変換するものであり、その型用に定義される演算子や関数で使用されます。output_functionはこの逆の変換を行うものです。入力関数は、1つのcstring型の引数、あるいは、cstring型、oid型、integer型という3つの引数を取るように宣言されます。最初の引数にはC言語文字列の入力テキスト、2番目には型自体のOID(配列型の場合は例外で要素の型のOIDとなります)、3番目は、判明していれば対象列のtypmodを渡します(不明な場合は-1を渡します)。この入力関数では、データ型自身の値を返さなければなりません。通常入力関数はSTRICTとして宣言しなければなりません。そうしないと、NULLという入力値を読み取った時、NULLという最初のパラメータを持って呼び出されます。この場合でもエラーを発生させるのでなければ、関数はNULLを返さなければなりません。(こうした状況はほとんどの場合、ドメイン入力関数をサポートすることを意図しています。ドメイン入力関数ではNULL入力を拒絶しなければならない可能性があります。) 出力関数は、新しいデータ型の引数を1つ取るように宣言しなければなりません。出力関数は、cstring型を返さなければなりません。出力関数はNULL値に対して呼び出されることはありません。

receive_functionでは、型のバイナリによる外部表現を内部表現に変換します。この関数は省略可能です。この関数が与えられない場合、この型ではバイナリ入力を行うことができません。バイナリ表現の方法は、適度な可搬性を保ちつつ、内部表現への変換コストが小さくなるよう選択すべきです(例えば標準の整数データ型は、外部バイナリ表現としてはネットワークバイトオーダを使用し、内部表現ではマシン固有のバイトオーダを使用します)。この受信関数では、値が有効かどうかを判定するための適切な検査を行わなければなりません。受信関数は、internal型の引数1つ、または、internal型とoid、integer型の3つの引数を取るように宣言されます。最初の引数は受信したバイト文字列を保持するStringInfoバッファへのポインタ、省略可能な引数は、テキスト入力関数の説明と同じです。受信関数は、データ型自体の値を返す必要があります。通常受信関数はSTRICTとして宣言しなければなりません。そうしないと、NULLという入力値を読み取っ

た時、NULLという最初のパラメータを持って呼び出されます。この場合でも関数はエラーを発生させるのでなければNULLを返さなければなりません。（こうした状況はほとんどの場合、ドメイン受信関数をサポートすることを意図しています。ドメイン受信関数ではNULL入力を拒絶しなければならない可能性があります。）同様に、send_functionは、内部表現からバイナリによる外部表現に変換します。この関数も省略可能です。この関数が与えられない場合、この型ではバイナリ出力を行うことができません。この送信関数は、新しいデータ型の引数1つを取るように宣言しなければなりません。送信関数はbytea型を返さなければなりません。送信関数はNULL値に対して呼び出されません。

ここで、新しいデータ型を作成する前に入力関数と出力関数を作成する必要があるのに、どのようにしてそれらの関数で新しいデータ型を戻り値や入力として宣言できるのか、疑問に思うかもしれません。その答えは、まず型が最初にシェル型として定義されます。これは名称と所有者以外の属性を持たないプレースホルダ型です。これは、コマンドCREATE TYPE nameを他にパラメータをつけずに発行することで行われます。この後、Cの入出力関数をこのシェル型を参照するように定義することができます。最後に完全な定義を持ったCREATE TYPEによって、シェル型の項目が完全かつ有効な型定義に置き換わり、新しい型を普通に使用することができるようになります。

type_modifier_input_functionとtype_modifier_output_functionは必須ではありませんが、型が修飾子をサポートする場合は必要です。修飾子とは、char(5)やnumeric(30,2)などの型宣言に付与されるオプションの制約です。PostgreSQLでは、ユーザ定義型が1つ以上の整数定数または識別子を修飾子として取ることができます。しかし、この情報はシステムカタログに格納される時に0以上の整数1つにまとめられるものでなければなりません。type_modifier_input_functionには、cstring型配列の形で宣言された修飾子が渡されます。その値について妥当性を検査しなければなりません（不当な場合はエラーとします）。そして、正しい場合は、「typmod」列として格納される、0以上のinteger値を1つ返さなければなりません。型がtype_modifier_input_functionを持たない場合、型修飾子は拒否されます。type_modifier_output_functionは内部的な整数typmod値をユーザ側の表示に合わせて変換します。この関数は型名に付与する正確な文字列となるcstring値を返さなければなりません。たとえばnumeric用の関数では(30,2)を返すかもしれません。デフォルトの表示用書式が保管されたtypmod整数値を括弧で括ったものと一致している場合は、type_modifier_output_functionを省略することができます。

オプションのanalyze_functionは、このデータ型の列に対する、型固有の統計情報の収集を行います。その型用のデフォルトのB-tree演算子クラスがあれば、ANALYZEはデフォルトでは型の「等価」演算子と「小なり」演算子を使用して統計情報を集めようと試みます。非スカラ型には、この振舞いはあまり適していません。そのため、独自の解析関数を指定することで、この振舞いを上書きすることができます。この解析関数は、internal型の引数を1つ取り、戻り値としてbooleanを返すように宣言する必要があります。解析関数用のAPIの詳細は、src/include/commands/vacuum.hにあります。

新しい型の内部表現の詳細を理解しなければならないのは、これらのI/O関数とその型に関連して動作するユーザ定義の関数のみですが、内部表現には、PostgreSQLに対し宣言しなければならない複数の属性値があります。属性の中で最も重要なものはinternallengthです。基本データ型は、internallengthに正の整数を指定して固定長として作成するだけでなく、internallengthにVARIABLEと設定し可変長として作成することもできます（内部的には、これはtyplenを-1に設定することで表現されます）。全ての可変長型の内部表現は、型の値の全長を示す4バイトの整数値から始まらなければなりません。（長さフィールドは多くの場合68.2に記述されているようにエンコードされており、それに直接アクセスすることは賢明ではないことに注意してください。）

オプションのPASSEDBYVALUEフラグは、このデータ型の値が参照ではなく値によって渡されることを示します。値によって渡される型は固定長でなければならず、その内部表現はDatum型のサイズ（4バイトのマシンもあれば8バイトのマシンもあります）を超えてはいけません。

alignmentパラメータは、そのデータ型の格納の際に必要な整列を指定します。設定可能な値は、1、2、4、8バイト境界での整列です。可変長の型は最低でも4の整列を持たなければならないことに注意してください。最初の要素としてint4を持たなければならないからです。

storageパラメータを使用することで、可変長データ型を格納する際の戦略を選択することができます（固定長の型にはplainのみが使用できます）。plainを指定すると、その型のデータは常にインラインで格納され、圧縮されません。extendedを指定すると、システムはまず長いデータ値を圧縮しようとし、それでもまだ長過ぎる場合は値をメインテーブルの行から削除して移動します。externalはメインテーブルから値を削除して移動することを許しますが、システムはデータを圧縮しようとしません。mainはデータの圧縮を許しますが、できるだけ値をメインテーブルから削除しないようにします（行を収めるために他に方法がない場合にはメインテーブルから削除されてしまう可能性がありますが、extendedやexternalが指定されたアイテムよりも優先してメインテーブルに残されます）。

plainを除くすべてのstorageの値は、そのデータ型の関数が、68.2および37.13.1に記述されているように*toast*された値を処理できることを暗示します。その他の特定の値を指定するのは、TOAST可能なデータ型の列について、単にデフォルトのTOAST戦略を決めるだけです。ユーザは個々の列についてALTER TABLE SET STORAGEを使って他の戦略を選択できます。

like_typeパラメータは、何らかの既存のデータ型から複製するという、データ型の基本表現プロパティを指定する、別の方法を提供します。internallength、passedbyvalue、alignment、storageの値が指定された型から複製されます。（通常は望ましくありませんが、LIKE句と一緒にこれらの値を指定することで、値を上書きすることも可能です。）新しい型の低レベル実装にある流儀に従った既存の型を「移す」時に、この方法で表現を指定することが特に有用です。

categoryとpreferredパラメータは、あいまいな状況でどの暗黙的なキャストが適用されるかについての制御を補助するために使用することができます。各データ型は単一のASCII文字で命名されるカテゴリに属しており、各型はそのカテゴリ内で「優先される（preferred）」か否かです。オーバーロードされた関数または演算子の解決に、この規則が有用な場合には、パーサは優先される型へのキャストを優先します（ただし、同一のカテゴリ内の他の型からだけです）。より詳細は第10章を参照してください。他の型への、または、ほかの型からの暗黙的なキャストを持たない型では、これらの設定をデフォルトのままにしておくことで十分です。しかし、暗黙的なキャストを持つ関連する型のグループでは、それらすべてを1つのカテゴリに属すものとし、「最も汎用的な」型の1つまたは2つをカテゴリ内で優先されるものとして選択することが有用となる場合が多くあります。ユーザ定義型を、数値型や文字列型などの既存の組み込みカテゴリに追加する場合に、categoryパラメータは特に有用です。しかし、完全にユーザ定義の新しい型カテゴリを作成することもできます。そのようなカテゴリの命名には大文字以外の任意のASCII文字を選択してください。

ユーザがそのデータ型の列のデフォルトをNULL以外にしたい場合は、デフォルト値を指定することができます。デフォルト値はDEFAULTキーワードで指定してください（この方法で指定されたデフォルト値を、特定の列に付与された、明示的なDEFAULT句によって上書きすることができます）。

データ型が配列であることを示すには、ELEMENTキーワードを使用して配列要素の型を指定してください。例えば、4バイト整数（int4）の配列を定義するには、ELEMENT = int4と指定してください。配列型の詳細は後述します。

この型による配列の外部形式における値間の区切り文字を示すために、delimiterで特定の文字を設定することができます。デフォルトの区切り文字はカンマ（','）です。この区切り文字は、配列要素の型に関係するものであり、配列型自体に関係するものでないことに注意してください。

論理型のcollatableパラメータ(省略可能)が真の場合、COLLATE句を使用することによって、型の列定義と式は照合順序情報を持つことができます。照合順序情報を実際に使用するかどうかは、型に対する操作を

行う関数実装に任されています。照合順序を設定可能な型を作成することにより、これが自動的に行われることはありません。

配列型

ユーザ定義型が作成されると、PostgreSQLは、自動的に関連する配列型を作成します。その要素型の名前の前にアンダースコアを付け、必要に応じてNAMEDATALEN長より短くなるように切り詰められた名前になります。(こうして付けられた名前が既存の型名と競合する場合、競合する名称がなくなるまでこの処理が繰り返されます。) この暗黙的に作成される配列型は可変長で、組み込み入出力関数array_inとarray_outを使用します。配列型はその要素となる型の所有者とスキーマのなんらかの変更に追従し、また、要素となる型が削除された場合に削除されます。

「システムが自動的に配列型を正しく作成するのであれば、ELEMENTオプションはどうして存在するのだろう」と疑問に思うのも道理です。ELEMENTが意味を持つ、唯一の場合は次のような条件を満たす固定長の型を作成する時です。その条件とは、内部的に複数の同一の要素からなる配列となっていること、その配列に対して添字を指定して直接アクセスできること、加えて、今後作成する型全体に対する操作がどのようなものであっても、それらから直接アクセスできることです。例えば、point型は、2つの浮動小数点だけから構成され、それらはpoint[0]およびpoint[1]を用いてアクセスすることができます。この機能は、その内部形式が同一の固定長フィールドの正確な並びである、固定長の型でのみ動作することに注意してください。添字による指定が可能な可変長型は、array_inとarray_outを使用して、一般化された内部表現を持つ必要があります。歴史的な理由(明らかに間違いなのですが、変更するには遅すぎたため)により、固定長配列型への要素番号指定は0から始まり、可変長配列の場合は1から始まります。

パラメータ

name

作成するデータ型の名前です(スキーマ修飾名も可)。

attribute_name

複合型用の属性(列)名です。

data_type

複合型の列となる、既存のデータ型の名前です。

collation

複合型の列または範囲型に関連付けされる、既存の照合順序の名前です。

label

列挙型の1つの値に関連付けられるテキスト形式のラベルを表す、文字列リテラルです。

subtype

範囲型がその範囲の対象として表現する、要素型の名前です。

subtype_operator_class

派生元型のb-tree演算子クラスの名前です。

canonical_function

範囲型の正規化関数の名前です。

subtype_diff_function

派生元型の差異をとる関数の名前です。

input_function

指定された型のテキストによる外部形式のデータを内部形式に変換する関数の名前です。

output_function

指定された型の内部形式のデータをテキストによる外部形式に変換する関数の名前です。

receive_function

指定された型のバイナリによる外部形式のデータを内部形式に変換する関数の名前です。

send_function

指定された型の内部形式のデータをバイナリによる外部形式に変換する関数の名前です。

type_modifier_input_function

型に関する修飾子の配列を内部形式に変換する関数の名前です。

type_modifier_output_function

内部形式の型修飾子をテキストの外部形式に変換する関数の名前です。

analyze_function

指定したデータ型の統計情報解析を行う関数の名前です。

internallength

新しいデータ型の内部表現のバイト長を表す数値定数です。デフォルトでは、可変長であるとみなされます。

alignment

データ型の格納整列条件です。このオプションを指定する場合は、char、int2、int4、doubleのいずれかでなければなりません。デフォルトはint4です。

storage

データ型の格納戦略です。このオプションを指定する場合は、plain、external、extended、mainのいずれかでなければなりません。デフォルトはplainです。

like_type

新しい型に同じ表現を持たせる既存のデータ型の名前です。internallength、passedbyvalue、alignment、storageの値が、このCREATE TYPEコマンドのどこかで明示的な指定により上書きされない限り、型から複製されます。

category

この型用のカテゴリコード（単一のASCII文字）です。デフォルトは「ユーザ定義型」を表す'U'です。他の標準カテゴリコードを表 51.63に示します。独自のカテゴリを作成するために他のASCII文字を選択することもできます。

preferred

この型がカテゴリ内で優先される型である場合に真、さもなくば偽です。デフォルトは偽です。動作に予想外の変化を引き起こしますので既存の型カテゴリに新しく優先される型を作成することには十分注意してください。

default

そのデータ型のデフォルト値です。省略された場合、デフォルトはNULLです。

element

配列型を作成する場合、その配列の要素の型を指定します。

delimiter

このデータ型による配列で、値間の区切り文字として使われる文字です。

collatable

この型を操作する時に照合順序情報を使用することができる場合に真を取ります。デフォルトは偽です。

注釈

一度作成したデータ型の使用には制限はありませんので、基本型または範囲型の作成は型定義で言及した関数の実行権をPUBLICに対して付与することと同じです。この種の型定義において有用な関数では、これは通常問題になりません。しかし、外部形式から、または、外部形式への変換を行う時に、その関数が「秘密の」情報を必要とする場合、型を設計する前に熟考してください。

PostgreSQLバージョン8.3より前のバージョンでは、生成される配列型の名前は常に要素型の名前の前に1つのアンダースコア文字（_）を付けたものになりました。（このため型の名前は他の名前よりも1文字短く制限されていました。）通常はこのように名付けられることは変わりありませんが、最大長の名前の場合やアンダースコアから始まるユーザ定義の型と競合する場合、配列型の名前はこの変換とは変わることがあります。このため、この規則に依存したコードを書くことは避けてください。代わりに、pg_type.typarrayを使用して、指定した型に関連した配列型を特定してください。

アンダースコアから始まる型やテーブル名の使用を避けることが賢明です。サーバは生成された配列型名称をユーザ指定の名前と競合しないように変更しますが、混乱する危険があります。特に古いクライアントソ

フトウェアを使用する場合、名前がアンダースコアから始まる型を常に配列を表すものと想定しているかもしれません。

PostgreSQLバージョン8.2より前まででは、シェル型を作成するCREATE TYPE name構文は存在しません。新規に基本型を作成する方法は、最初に入力関数を作成することでした。この方法では、PostgreSQLは新しいデータ型の名称を、入力関数の戻り値型で初めて見ます。このときに、シェル型が暗黙的に作成され、残りの入出力関数の定義で参照することができます。この方法もまだ使用できますが、廃止予定であり、将来のリリースで禁止される可能性があります。また、関数定義における単純なタイプミスの結果作成されるシェル型によって起こるカタログの混乱を防止するため、入力関数がCで作成された場合にのみこの方法によってシェル型が作成されます。

例

次の例では、複合型を作成し、それを関数定義で使用します。

```
CREATE TYPE compfoo AS (f1 int, f2 text);

CREATE FUNCTION getfoo() RETURNS SETOF compfoo AS $$
    SELECT fooid, fooname FROM foo
$$ LANGUAGE SQL;
```

次の例では、列挙型を作成し、それをテーブル定義に使用します。

```
CREATE TYPE bug_status AS ENUM ('new', 'open', 'closed');

CREATE TABLE bug (
    id serial,
    description text,
    status bug_status
);
```

次の例では、範囲型を作成します。

```
CREATE TYPE float8_range AS RANGE (subtype = float8, subtype_diff = float8mi);
```

次の例では、基本データ型boxを作成し、その型をテーブル定義の中で使用しています。

```
CREATE TYPE box;

CREATE FUNCTION my_box_in_function(cstring) RETURNS box AS ... ;
CREATE FUNCTION my_box_out_function(box) RETURNS cstring AS ... ;

CREATE TYPE box (
    INTERNALLENGTH = 16,
    INPUT = my_box_in_function,
    OUTPUT = my_box_out_function
```

```
);

CREATE TABLE myboxes (
    id integer,
    description box
);
```

box型の内部構造がfloat4型が4つの配列の場合、このように書き換えることもできます。

```
CREATE TYPE box (
    INTERNALLENGTH = 16,
    INPUT = my_box_in_function,
    OUTPUT = my_box_out_function,
    ELEMENT = float4
);
```

このようにすると、box値の要素に要素番号でアクセスできます。その他は、上の例と同様の動作をします。

次の例では、ラージオブジェクト型を作成し、テーブル定義にてそれを使用します。

```
CREATE TYPE bigobj (
    INPUT = lo_filein, OUTPUT = lo_fileout,
    INTERNALLENGTH = VARIABLE
);
CREATE TABLE big_objs (
    id integer,
    obj bigobj
);
```

その他の例は、37.13を参照してください。ここには、入力関数、出力関数などを使った例があります。

互換性

複合型を作成する、最初のCREATE TYPEコマンドの構文は標準SQLに従います。他の構文はPostgreSQLの拡張です。標準SQLではまた他のCREATE TYPE構文を定義していますが、PostgreSQLでは実装されていません。

ゼロ個の要素を持つ複合型を作成する機能は標準から派生したPostgreSQL固有のもの（CREATE TABLEの場合と同様）です。

関連項目

ALTER TYPE, CREATE DOMAIN, CREATE FUNCTION, DROP TYPE

CREATE USER

CREATE USER — 新しいデータベースロールを定義する

概要

```
CREATE USER name [ [ WITH ] option [ ... ] ]


ここでoptionは以下の通りです。

      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | REPLICATION | NOREPLICATION
    | BYPASSRLS | NOBYPASSRLS
    | CONNECTION LIMIT connlimit
    | [ ENCRYPTED ] PASSWORD 'password' | PASSWORD NULL
    | VALID UNTIL 'timestamp'
    | IN ROLE role_name [, ...]
    | IN GROUP role_name [, ...]
    | ROLE role_name [, ...]
    | ADMIN role_name [, ...]
    | USER role_name [, ...]
    | SYSID uid
```

説明

CREATE USERはCREATE ROLEの別名になりました。唯一の違いは、CREATE USERという名前でコマンドが呼び出されると、デフォルトでLOGINになり、CREATE ROLEという名前でコマンドが呼び出されると、デフォルトでNOLOGINとなる点です。

互換性

CREATE USER文は、PostgreSQLの拡張です。標準SQLでは、ユーザの定義は実装に任されています。

関連項目

CREATE ROLE

CREATE USER MAPPING

CREATE USER MAPPING — 外部サーバのユーザマップを新しく定義する

概要

```
CREATE USER MAPPING [ IF NOT EXISTS ] FOR { user_name | USER | CURRENT_USER | PUBLIC }
    SERVER server_name
    [ OPTIONS ( option 'value' [ , ... ] ) ]
```

説明

CREATE USER MAPPINGは外部サーバとユーザの対応付けを定義します。ユーザマップは通常接続情報をカプセル化し、外部データラッパは外部データリソースにアクセスするためにこの情報と外部サーバによりカプセル化した情報を使用します。

外部サーバの所有者は任意のユーザに対するそのサーバ向けのユーザマップを作成することができます。また、サーバ上でUSAGE権限がユーザに付与されている場合、ユーザは自身の持つユーザ名に対応するユーザマップを作成することができます。

パラメータ

IF NOT EXISTS

指定のユーザから指定の外部サーバへのマッピングが存在する場合にエラーを発生させません。この場合、注意メッセージが発行されます。既存のユーザマッピングが、作成しようとしていたものと類似するものかどうか、全く保証されないことに注意してください。

user_name

外部サーバに対応付けされる既存のユーザ名です。CURRENT_USERとUSERは現在のユーザの名前に対応します。PUBLICが指定された場合、ユーザ指定がないマップが適用されたときに使用される公開マップと呼ばれるものが作成されます。

server_name

ユーザマップを作成する対象の既存のサーバの名前です。

OPTIONS (option 'value' [, ...])

この句はユーザマップのオプションを指定します。通常オプションはマップにおける実際のユーザ名とパスワードを定義します。オプション名は一意でなければなりません。使用できるオプションの名前と値は、サーバの外部データラッパにより異なります。

例

ユーザbobとサーバfooとのユーザマップを作成します。

```
CREATE USER MAPPING FOR bob SERVER foo OPTIONS (user 'bob', password 'secret');
```

互換性

CREATE USER MAPPINGはISO/IEC 9075-9 (SQL/MED)に従います。

関連項目

ALTER USER MAPPING, DROP USER MAPPING, CREATE FOREIGN DATA WRAPPER, CREATE SERVER

CREATE VIEW

CREATE VIEW — 新しいビューを定義する

概要

```
CREATE [ OR REPLACE ] [ TEMP | TEMPORARY ] [ RECURSIVE ] VIEW name [ ( column_name [, ...] ) ]
    [ WITH ( view_option_name [= view_option_value] [, ... ] ) ]
    AS query
    [ WITH [ CASCADED | LOCAL ] CHECK OPTION ]
```

説明

CREATE VIEWは問い合わせによるビューを定義します。ビューは物理的な実体として存在するものではありません。その代わり、問い合わせでビューが参照される度に、指定された問い合わせが実行されます。

CREATE OR REPLACE VIEWも同様の働きをしますが、このコマンドでは、同じ名前のビューが既に存在している場合、そのビューを置き換えます。新しい問い合わせは、既存のビュー問い合わせが生成する列と同じ列(つまり、同じ順序の同じデータ型の同じ列名)を生成しなければなりません。しかし、そのリストの最後に列を追加しても構いません。出力列を生成する計算をまったく異なるものにしても構いません。

スキーマ名が付けられている場合(例えば、CREATE VIEW myschema.myview ...)、ビューは指定されたスキーマに作成されます。スキーマ名がなければ、そのビューは現在のスキーマに作成されます。一時ビューは特別なスキーマに作成されます。そのため、一時ビューを作成する時にはスキーマ名を付けることはできません。ビュー名は、同じスキーマ内の他のビュー、テーブル、シーケンス、インデックス、外部テーブルとは異なる名前である必要があります。

パラメータ

TEMPORARYまたはTEMP

これが指定された場合、ビューは一時ビューとして作成されます。現在のセッションが終わった時、一時ビューは自動的に削除されます。一時ビューが存在する間、現在のセッションでは、これと同じ名前の永続リレーションはスキーマ修飾した名前で参照していない限り不可視です。

ビューで参照されるテーブルの一部が一時テーブルであった場合、(TEMPORARYの指定があってもなくても)ビューは一時ビューとして作成されます。

RECURSIVE

再帰的ビューを作成します。

```
CREATE RECURSIVE VIEW [ schema . ] view_name (column_names) AS SELECT ...;
```

という構文は

```
CREATE VIEW [ schema . ] view_name AS WITH RECURSIVE view_name (column_names) AS (SELECT ...)
 SELECT column_names FROM view_name;
```

と同等です。再帰的ビューではビューの列名リストを指定する必要があります。

name

作成するビューの名前です（スキーマ修飾名も可）。

column_name

ビューの列名として使用する名前のリストで、省略可能です。省略された場合、問い合わせに由来する名前が使用されます。

WITH (view_option_name [= view_option_value] [, ...])

この句はビュー用のオプションのパラメータを指定します。以下のパラメータがサポートされています。

check_option (enum)

このパラメータはlocalかcascadedのいずれかで、WITH [CASCADED | LOCAL] CHECK OPTIONを指定するのと同じです(以下を参照)。このオプションは、既存のビューについてALTER VIEWを使って変更することができます。

security_barrier (boolean)

行単位セキュリティを提供することを意図したビューでは、これを有効にしなければなりません。詳細については40.5を参照してください。

query

ビューの列と行を生成するSELECTまたはVALUESコマンドです。

WITH [CASCADED | LOCAL] CHECK OPTION

このオプションは、自動的に更新可能なビューの動作を制御します。このオプションが指定された場合、ビューに対するINSERTおよびUPDATEコマンドでは、新しい行がビュー定義の条件を満たすことが検査されます(つまり、新しい行がビューで見ることができるかどうか、検査されます)。条件を満たさない場合、更新は拒絶されます。CHECK OPTIONが指定されない場合、ビューに対するINSERTおよびUPDATEコマンドは、ビューで見ることができない行を作ることができます。以下のcheck optionがサポートされます。

LOCAL

新しい行は、そのビュー自体に直接定義されている条件に対してのみ検査されます。ビューが基にするビューについて定義されている条件は、(それらもCHECK OPTIONを指定しているのでなければ)検査されません。

CASCADED

新しい行は、そのビュー、およびそれが基にするすべてのビューの条件に対して検査されます。CHECK OPTIONが指定され、LOCALもCASCADEDも指定されていないときは、CASCADEDが指定されたとみなされます。

CHECK OPTIONはRECURSIVEなビューで使うことはできません。

CHECK OPTIONは、自動更新可能で、かつINSTEAD OFトリガーもINSTEADルールもないビューについてのみサポートされていることに注意してください。自動更新可能ビューがINSTEAD OFトリガーのあるビューに基づいて定義されている場合、LOCAL CHECK OPTIONを使って自動更新可能ビューの条件を検査することはできますが、INSTEAD OFトリガーを持つ基のビューの条件は検査されません(cascaded check optionはトリガーで更新されるビューにまでは伝わらず、またトリガーで更新可能なビューに直接定義されたcheck optionは無視されます)。ビューあるいはその基となるリレーションにINSTEADルールがあり、INSERTあるいはUPDATEの書き換えが生じる場合、その書き換えられたクエリでは(INSTEADルールのあるリレーションに基づく自動更新可能ビューのものも含めて)すべてのcheck optionが無視されます。

注釈

ビューを削除するには、DROP VIEW文を使用してください。

ビューの列の名前と型は指定通りに割り当てられることに注意してください。例えば、次のコマンドを見てください。

```
CREATE VIEW vista AS SELECT 'Hello World';
```

この例は列の名前がデフォルトの?column?になるので好ましくありません。また、列のデータ型もデフォルトのtextになりますが、これは求めるものと違うかもしれません。ビューの結果として文字リテラルを返したい場合は、次のように指定するのがよりよい方法です。

```
CREATE VIEW vista AS SELECT text 'Hello World' AS hello;
```

ビューが参照するテーブルにアクセスできるかどうかは、ビューの所有者の権限で決定されます。これは、場合によっては、背後のテーブルに対する安全で制限されたアクセスを提供します。しかしすべてのビューが不正な改変に対して安全ではありません。40.5を参照してください。ビュー内で実行される関数については、ビューを使用した問い合わせにおいて、その関数が直接呼び出された場合と同様に扱われます。したがって、ビューを使用するユーザには、ビュー内で使用される全ての関数を呼び出す権限が必要です。

CREATE OR REPLACE VIEWが既存のビューに対して使用されると、ビューを定義するSELECTルールのみが変更されます。所有者、権限、SELECT以外のルールなど他のビューの属性はそのまま変更されません。置き換えるためにはビューの所有者（所有ロールのメンバである場合も含む）でなければなりません。

更新可能ビュー

簡単なビューは自動更新可能になります。システムは、ビューに対するINSERT、UPDATE、DELETE文を通常のテーブルの場合と同じ方法で使用できるようにします。以下の条件のすべてを満たす場合に、ビューは自動更新可能になります。

- ビューのFROMリストには正確に1つだけの項目を持たなければならず、それはテーブルまたは他の更新可能ビューでなければなりません。
- ビューの定義の最上位レベルにおいてWITH、DISTINCT、GROUP BY、HAVING、LIMIT、OFFSETを含めてはなりません。
- ビューの定義の最上位レベルにおいて集合操作（UNION、INTERSECT、EXCEPT）を含めてはなりません。
- ビューの選択リストに、集約関数、ウィンドウ関数、集合を返す関数を含めてはなりません。

自動更新可能ビューでは、更新可能な列と更新不可能な列を混在させることができます。基になるリレーションの更新可能な列を単純に参照する列は更新可能です。そうでなければ列は更新不可能で、INSERTあるいはUPDATE文でその列に値を設定しようとしたらエラーが発生します。

ビューが自動更新可能であれば、システムはビューに対するINSERT、UPDATEまたはDELETE文を基となるベースリレーションへの対応する文に変換します。ON CONFLICT UPDATE句を持つINSERT文は完全にサポートされます。

自動更新可能ビューがWHERE条件を持つ場合、ベースリレーションのどの行をビューに対するUPDATE、DELETE文により変更可能かをその条件が制限します。しかしUPDATEによる行の変更の結果WHEREを満たさなくなり、その結果、ビューからは参照することができなくなることがあります。同様にINSERTコマンドはWHERE条件を満たさず、そのためビューを通して参照することができない行をベースリレーションに挿入する可能性があります（ON CONFLICT UPDATEはビューを通して見えない既存の行に同様に影響を及ぼすかもしれません）。CHECK OPTIONはINSERTやUPDATEがビューで見ることができない行を作るのを防ぐために使うことができます。

自動更新可能ビューがsecurity_barrier属性を持つ場合、ビューのすべてのWHERE条件(およびLEAKPROOFの演算子を使ったすべての条件)が、必ず、ビューのユーザが追加した条件より前に評価されます。詳細は40.5を参照してください。このため、最終的には(ユーザのWHERE条件を満たさないために)戻されない行もロックされてしまう場合があることに注意してください。EXPLAINを使って、リレーションのレベルでどの条件が使われ(その結果、行をロックしない)、どの条件が使われないかを調べることができます。

これらの条件をすべて満たさないより複雑なビューはデフォルトで読み取り専用です。システムはビューに対する挿入、更新、削除を許可しません。ビューに対するINSTEAD OFトリガを作成することで、更新可能ビューの効果を得ることができます。このトリガはビューに対する挿入試行などを他のテーブルに対する適切な操作に変換するものでなければなりません。詳細についてはCREATE TRIGGERを参照してください。他にもルールを作成する（CREATE RULE参照）ことでも実現できますが、実際にはトリガの方が理解しやすく正しく使用するのが容易です。

ビューに対する挿入、更新、削除を行うユーザは、ビューに対して対応する挿入、更新、削除権限を持たなければなりません。さらにビューの所有者は基となるベースリレーションに対する適切な権限を持たなければなりません。しかし、更新を行うユーザは基となるベースリレーションに対する権限をまったく必要としません（40.5参照）。

例

全てのコメディ映画（Comedy films）からなるビューを作成します。

```
CREATE VIEW comedies AS
    SELECT *
    FROM films
    WHERE kind = 'Comedy';
```

これはビューを作成した時点でfilmテーブル内にある列を持つビューを作成します。ビューを作成するために*が使用されていますが、その後にテーブルに追加された列はビューには含まれません。

LOCAL CHECK OPTIONを使ってビューを作成します。

```
CREATE VIEW universal_comedies AS
```

```
    SELECT *
    FROM comedies
    WHERE classification = 'U'
    WITH LOCAL CHECK OPTION;
```

これはcomediesビューに基づくビューを作成し、kind = 'Comedy'かつclassification = 'U'である映画だけを表示します。このビューでの行のINSERTやUPDATEは、classification = 'U'でなければ拒絶されますが、映画のkindは検査されません。

CASCADED CHECK OPTIONでビューを作成します。

```
CREATE VIEW pg_comedies AS
    SELECT *
    FROM comedies
    WHERE classification = 'PG'
    WITH CASCADED CHECK OPTION;
```

これは新しい行についてkindとclassificationの両方を検査するビューを作成します。

更新可能な列と更新不可能な列が混在するビューを作成します。

```
CREATE VIEW comedies AS
    SELECT f.*,
           country_code_to_name(f.country_code) AS country,
           (SELECT avg(r.rating)
            FROM user_ratings r
            WHERE r.film_id = f.id) AS avg_rating
    FROM films f
    WHERE f.kind = 'Comedy';
```

このビューはINSERT、UPDATE、DELETEをサポートします。filmsテーブルからのすべての列は更新可能ですが、計算される列countryとavg_ratingは更新できません。

1から100までの数からなる再帰的ビューを作成します。

```
CREATE RECURSIVE VIEW public.nums_1_100 (n) AS
    VALUES (1)
UNION ALL
    SELECT n+1 FROM nums_1_100 WHERE n < 100;
```

上記のCREATEにおいて再帰的ビューの名前はスキーマ修飾されていますが、その内側の自己参照はスキーマ修飾されていないことに注意してください。これは、暗黙的に作成されるCTEの名前はスキーマ修飾できないからです。

互換性

CREATE OR REPLACE VIEWはPostgreSQLの言語拡張です。一時ビューという概念も言語拡張です。同様にWITH (...)句も拡張です。

関連項目

ALTER VIEW, DROP VIEW, CREATE MATERIALIZED VIEW

DEALLOCATE

DEALLOCATE — プリペアド文の割り当てを解除する

概要

```
DEALLOCATE [ PREPARE ] { name | ALL }
```

説明

DEALLOCATEを使用して、過去にプリペアドSQL文の割り当てを解除します。プリペアド文を明示的に割り当て解除しなかった場合、セッションが終了した時に割り当てが解除されます。

プリペアド文に関する詳細についてはPREPAREを参照してください。

パラメータ

PREPARE

このキーワードは無視されます。

name

割り当てを解除する、プリペアド文の名前です。

ALL

プリペアド文の割り当てをすべて解除します

互換性

DEALLOCATE文は標準SQLにもありますが、埋め込みSQLでの使用のみに用途が限定されています。

関連項目

EXECUTE, PREPARE

DECLARE

DECLARE — カーソルを定義する

概要

```
DECLARE name [ BINARY ] [ INSENSITIVE ] [ [ NO ] SCROLL ]
    CURSOR [ { WITH | WITHOUT } HOLD ] FOR query
```

説明

DECLAREを使うと、カーソルが使用できるようになります。これは、巨大な問い合わせの結果から一度に少数の行を取り出す機能です。カーソルを作成した後、FETCHを使用して行を取り出します。

> **注記**
>
> このマニュアルページではSQLコマンドレベルでのカーソルの使用方法について説明します。PL/pgSQL内でカーソルを使用するつもりであれば、規則が異なりますので、42.7を参照してください。

パラメータ

name

作成されるカーソルの名前です。

BINARY

カーソルによるデータの取得が、テキスト形式ではなくバイナリ形式になります。

INSENSITIVE

カーソルから取り出されたデータが、カーソルを作成した後に行われた背後にあるテーブルの更新の影響を受けないことを示します。これはPostgreSQLのデフォルトの動作ですので、このキーワードを使用しても効果はなく、このキーワードは標準SQLとの互換性を保持するために存在しています。

SCROLL
NO SCROLL

SCROLLは、そのカーソルから通常の順序通りでない方法で（例えば後方から）行を取り出し可能であることを指定します。問い合わせの実行計画が複雑になると、SCROLLの指定によって問い合わせの実行時間が増大する可能性があります。NO SCROLLは、そのカーソルから順序通りでない方法では行を取り出せないことを指定します。デフォルトでは、いくつかの場合でスクロール可能です。これはSCROLLの指定と同じではありません。詳細はNotesを参照してください。

WITH HOLD
WITHOUT HOLD

WITH HOLDは、カーソルを生成したトランザクションが正常にコミット処理を行った後も、そのカーソルの使用を続けられることを指定します。WITHOUT HOLDは、カーソルを生成したトランザクションの外部では、そのカーソルを使用できないことを指定します。WITH HOLDもWITHOUT HOLDも指定されない場合、WITHOUT HOLDがデフォルトとなります。

query

カーソルによって返される行を提供するSELECTまたはVALUESコマンドです。

BINARY、INSENSITIVE、SCROLLキーワードは任意の順番で指定することができます。

注釈

通常のカーソルは、SELECTの出力と同じテキスト形式でデータを返します。BINARYは、カーソルがバイナリ形式でデータを返すことを示します。これによりサーバ、クライアントの両方で変換に関する作業を省くことができますが、プラットフォームに依存するバイナリデータ書式を扱うためのプログラマの作業が大きくなります。例えば、問い合わせが整数の列から値として1を返す場合、デフォルトのカーソルからは1という文字列を取得することになりますが、バイナリ形式のカーソルからは、内部表現を使った4バイトの値を(ビッグエンディアンのバイト順で)取得することになります。

バイナリ形式のカーソルは注意して使わなければなりません。psqlなどの多くのアプリケーションは、データはテキスト形式で返されることを期待しており、バイナリ形式のカーソルを扱うことができません。

> **注記**
>
> クライアントアプリケーションが「拡張問い合わせ」プロトコルを使用してFETCHコマンドを発行する場合、テキスト形式とバイナリ形式のどちらでデータを受け取るのかは、バインドプロトコルメッセージで指定します。この選択は、カーソル定義での指定を上書きします。全てのカーソルをテキスト形式/バイナリ形式のどちらでも扱うことができる拡張問い合わせプロトコルでは、バイナリカーソルという概念は旧式なものです。

WITH HOLDが指定されなければ、このコマンドで生成されるカーソルは現在のトランザクションの中でのみ使用することができます。したがって、WITH HOLDのないDECLAREはトランザクションブロックの外側では意味がありません。その場合、カーソルはこの文が完了するまでのみ有効です。そのため、PostgreSQLはトランザクションブロックの外部でこうしたコマンドが使用された場合エラーを報告します。トランザクションブロックを定義するには、BEGINとCOMMIT(またはROLLBACK)を使用してください。

WITH HOLDが指定され、カーソルを作成したトランザクションのコミットに成功した場合、同一セッション内のその後のトランザクションからそのカーソルにアクセスすることができます（ただし、トランザクションがアボートされた場合、そのカーソルは削除されます）。WITH HOLD付きで作成されたカーソルは、そのカーソルに対して明示的なCLOSEが発行された場合やセッションが終了した時に閉じられます。現在の実装では、保持されたカーソルを使って表される行は、その後のトランザクションでも利用できるように、一時ファイルかメモリ領域にコピーされます。

問い合わせがFOR UPDATEまたはFOR SHAREを含む場合、WITH HOLDを指定することはできません。

カーソルから逆方向にデータを取り出す時には、SCROLLオプションを指定するべきです。これは標準SQLでは必須となっています。しかし、以前のバージョンとの互換性を保持するために、PostgreSQLでは、カーソルの問い合わせ計画が単純であり、そのサポートに余計なオーバーヘッドが必要ない場合、SCROLLなしでも逆方向にデータを取り出すことができます。しかし、SCROLLを付けなくても逆方向にデータが取り出せることを利用してアプリケーションを開発するのはお勧めしません。NO SCROLLを指定した場合は、どのような場合でも逆方向に取り出すことはできません。

また、問い合わせがFOR UPDATEまたはFOR SHAREを含む場合は、逆方向の取り出しは許されません。このためこの場合はSCROLLを指定することはできません。

注意

スクロール可能なWITH HOLDカーソルが揮発関数(37.7参照)を含む場合、想定しない結果をもたらす可能性があります。これまで取り出した行を再度取り出した時、関数は再実行される可能性があり、この場合おそらく初回と異なる結果をもたらします。こうした問題の回避方法の1つとして、カーソルをWITH HOLDと宣言し、そこから何か行を読み取る前にトランザクションをコミットすることがあります。これにより強制的にカーソルの出力全体が一時領域に具現化され、揮発関数は各行に対して1度しか実行されなくなります。

カーソルの問い合わせがFOR UPDATEまたはFOR SHAREを含む場合、このオプションを持つ通常のSELECTコマンドと同様、返される行は取り出した時点でロックされます。さらに、返される行はもっとも最新のバージョンになります。したがって、このオプションは、標準SQLで「センシティブカーソル」と呼ばれるものと同じ機能を提供します。(INSENSITIVEをFOR UPDATEまたはFOR SHAREといっしょに指定するとエラーになります。)

注意

カーソルをUPDATE ... WHERE CURRENT OFまたはDELETE ... WHERE CURRENT OFで使用するつもりならば、FOR UPDATEの使用を通常勧めます。FOR UPDATEを使用することで、取り出してから更新されるまでの間に他のセッションが行を変更することを防止します。FOR UPDATEがなければ、カーソル作成後に行が変更された場合に後に行うWHERE CURRENT OFコマンドは効果がなくなります。

FOR UPDATEを使用する他の理由は、「簡単に更新可能」にするためにカーソル問い合わせが標準SQLに合わない場合(具体的にはカーソルは1つのテーブルのみを参照しなければならず、また、グループ化やORDER BYを使用してはならない)、これがないと後に実行されるWHERE CURRENT OFが失敗するかもしれないことです。計画選択の詳細によっては、簡単に更新可能でないカーソルは動作するかもしれませんし、動作しないかもしれません。このため最悪の場合、アプリケーションは試験時に動作するが、運用時に失敗するかもしれません。FOR UPDATEが指定されていれば、カーソルは更新可能であることが保証されています。

FOR UPDATEをWHERE CURRENT OFといっしょに使用しない大きな理由は、カーソルをスクロール可能にする必要がある、または後の更新の影響を受けないようにする(つまり古いデータを表示し続けるようにする)必要がある場合のためです。これが必要ならば、上記の警告に十分注意してください。

標準SQLでは、組み込みSQLにおけるカーソルのみが規定されています。PostgreSQLサーバはカーソル用のOPEN文を実装していません。カーソルは宣言された時に開いたものとみなされています。しかし、PostgreSQL用の埋め込みSQLプリプロセッサであるECPGでは、DECLAREとOPEN文などを含め、標準SQLのカーソル規定をサポートしています。

pg_cursorsシステムビューを問い合わせることで、利用可能なすべてのカーソルを確認することができます。

例

カーソルを宣言します。

```
DECLARE liahona CURSOR FOR SELECT * FROM films;
```

カーソル使用の他の例についてはFETCHを参照してください。

互換性

標準SQLでは、デフォルトでカーソルが背後にあるデータの同時実行更新に影響を受けるかどうかは実装依存であると述べています。PostgreSQLのカーソルはデフォルトでは影響を受けず、FOR UPDATEを指定することで影響を受けることができます。他の製品では異なる動作をするかもしれません。

標準SQLでは、カーソルを埋め込みSQL内とモジュール内でのみ使用できます。PostgreSQLでは、対話式にカーソルを使うことができます。

バイナリカーソルはPostgreSQLの拡張です。

関連項目

CLOSE, FETCH, MOVE

DELETE

DELETE — テーブルから行を削除する

概要

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
DELETE FROM [ ONLY ] table_name [ * ] [ [ AS ] alias ]
    [ USING from_item [, ...] ]
    [ WHERE condition | WHERE CURRENT OF cursor_name ]
    [ RETURNING * | output_expression [ [ AS ] output_name ] [, ...] ]
```

説明

DELETEは、指定したテーブルからWHERE句を満たす行を削除します。WHERE句がない場合、指定したテーブルの全ての行を削除することになります。この結果、そのテーブルは存在するが中身が空のテーブルになります。

> **ヒント**
>
> TRUNCATEは、より高速に、テーブルから全ての行を削除する仕組みを提供します。

データベース内のほかのテーブルに含まれる情報を用いてテーブル内の行を削除する方法には、副問い合わせとUSING句で追加テーブルを指定する方法の2つがあります。どちらの技法が適切かはその状況によります。

RETURNING句を指定すると、DELETEは実際に削除された各行に基づいて計算された値を返すようになります。そのテーブルの列、USINGで指定された他のテーブルの列、あるいは、その両方を使用した式を計算することができます。RETURNINGリストの構文はSELECTの出力リストと同一です。

削除を実行するには、そのテーブルのDELETE権限が必要です。また、USING句内のテーブルに対するSELECT権限、および、conditionで使用する値を読み取るために、その値が含まれるテーブルに対するSELECT権限も必要です。

パラメータ

with_query

WITH句によりDELETE問い合わせ内で名前で参照可能な1つ以上の副問い合わせを指定することができます。詳しくは7.8とSELECTを参照してください。

table_name

行を削除するテーブルの名前です（スキーマ修飾名も可）。テーブル名の前にONLYが指定された場合、そのテーブルでのみマッチする行が削除されます。ONLYが指定されていない場合、そのテーブルおよび

(もしあれば)そのテーブルを継承する全てのテーブルから一致する行が削除されます。オプションで、テーブル名の後に*を指定することで、明示的に継承するテーブルも含まれることを示すことができます。

alias

対象テーブルの別名です。別名が与えられた場合、実際のテーブル名は完全に隠蔽されます。たとえば、DELETE FROM foo AS fとあるとき、このDELETE文の残りの部分ではこのテーブルをfooではなくfとして参照しなければなりません。

from_item

WHERE条件内に他のテーブルの列を記述できるようにするための、テーブル式です。これは、SELECT文のFROM句と同じ文法を使います。例えば、テーブル名の別名が指定できます。自己結合を行う場合を除き、from_itemに対象のテーブルを繰り返してはいけません(自己結合を行う場合は、from_item内で対象のテーブルとその別名を指定しておく必要があります)。

condition

boolean型の値を返す式です。この式がtrueを返す行のみが削除されます。

cursor_name

WHERE CURRENT OF条件で使用されるカーソルの名前です。削除対象の行は、そのカーソルからもっとも最近に取り出される行です。カーソルは、DELETEの対象テーブルに対するグループ化のない問い合わせでなければなりません。WHERE CURRENT OFを論理条件といっしょに指定することはできません。WHERE CURRENT OF付きのカーソルの使用に関する情報についてはDECLAREを参照してください。

output_expression

各行を削除した後にDELETEによって計算され、返される式です。この式には、table_nameで指名したテーブルまたはUSINGで指定したテーブルの任意の列名を使用することができます。すべての列を返す場合は*と記載してください。

output_name

返される列で使用される名前です。

出力

正常に終了した場合、DELETEコマンドは以下の形式のコマンドタグを返します。

```
DELETE count
```

countは削除した行数です。この数は、BEFORE DELETEトリガによって削除が抑止された場合、conditionに合致した行より少なくなる可能性があることに注意してください。countが0の場合、conditionを満たす行が存在しなかったことを示します(これはエラーとはみなされません)。

DELETEコマンドがRETURNING句を持つ場合、その結果は、RETURNINGリストで定義した列と値を持ち、そのコマンドで削除された行全体に対して計算を行うSELECT文の結果と似たものになるでしょう。

注釈

PostgreSQLでは、USING句で他のテーブルを指定することで、WHERE条件内で他のテーブルを参照することができます。例えば、指定したプロデューサが製作した全ての映画を削除する時は、次のようなコマンドを実行します。

```
DELETE FROM films USING producers
  WHERE producer_id = producers.id AND producers.name = 'foo';
```

ここでは、filmsとproducersとを結合して、films行に削除用の印を付けるという作業を行っています。この構文は標準に従ったものではありません。より標準的な方法は以下の通りです。

```
DELETE FROM films
  WHERE producer_id IN (SELECT id FROM producers WHERE name = 'foo');
```

副問い合わせ形式より結合形式の方が書き易い、あるいは、実行が速くなることがあります。

例

ミュージカル以外の全ての映画を削除します。

```
DELETE FROM films WHERE kind <> 'Musical';
```

filmsテーブルを空にします。

```
DELETE FROM films;
```

完了した作業（statusがDONE）を削除し、削除された行のすべての詳細を返します。

```
DELETE FROM tasks WHERE status = 'DONE' RETURNING *;
```

tasksにおいてc_tasksカーソルが現在位置している行を削除します。

```
DELETE FROM tasks WHERE CURRENT OF c_tasks;
```

互換性

このコマンドは標準SQLに準拠しています。ただし、USING句とRETURNING句はPostgreSQLの拡張です。DELETEでWITHが使用可能であることも同様に拡張です。

関連項目

TRUNCATE

DISCARD

DISCARD — セッションの状態を破棄する

概要

```
DISCARD { ALL | PLANS | SEQUENCES | TEMPORARY | TEMP }
```

説明

DISCARDはデータベースセッションに関連した内部リソースを解放します。このコマンドはセッションの状態を部分的あるいは完全にリセットするのに役に立ちます。様々な種類のリソースを解放するためにいくつかのサブコマンドがあります。DISCARD ALLは他のすべてを包含し、さらにまた追加の状態もリセットします。

パラメータ

PLANS

キャッシュされた問い合わせ計画をすべて解放します。これにより、関連するプリペアド文が次に使われたとき、強制的に計画がやり直されます。

SEQUENCES

キャッシュされたシーケンスに関連する状態をすべて破棄します。これには、currval()/lastval()の情報、および事前に割り当てられたシーケンスの値で、まだnextval()によって返されていないものを含みます。(事前に割り当てられたシーケンスの値についてはCREATE SEQUENCEを参照してください。)

TEMPORARYまたはTEMP

現在のセッションで作成された一時テーブルをすべて削除します。

ALL

現在のセッションに関連付いた一時的なリソースを解放し、セッションを初期状態に戻します。現時点でこれは、以下に示す一連の文を実行することと同じ効果があります。

```
CLOSE ALL;
SET SESSION AUTHORIZATION DEFAULT;
RESET ALL;
DEALLOCATE ALL;
UNLISTEN *;
SELECT pg_advisory_unlock_all();
DISCARD PLANS;
DISCARD TEMP;
DISCARD SEQUENCES;
```

注釈

DISCARD ALLをトランザクションブロック内で実行することはできません。

互換性

DISCARDはPostgreSQLの拡張です。

DO

DO — 無名コードブロックを実行します。

概要

```
DO [ LANGUAGE lang_name ] code
```

説明

DOは無名コードブロック、言い換えると、手続き言語内の一時的な無名関数を実行します。

コードブロックはあたかもパラメータを取らずにvoidを返す関数の本体かのように扱われます。これは解析され、一回実行されます。

LANGUAGE句をコードブロックの前または後ろにつけることができます。

パラメータ

code

実行される手続き言語のコードです。これは、CREATE FUNCTIONの場合と同様、文字列リテラルとして指定しなければなりません。ドル記号による引用符付けの使用を勧めます。

lang_name

コードの作成に使用する手続き言語の名前です。省略時のデフォルトはplpgsqlです。

注釈

使用される手続き言語は、CREATE EXTENSIONを使用して現在のデータベースにインストール済みでなければなりません。plpgsqlはデフォルトでインストールされますが、他の言語はインストールされません。

ユーザは手続き言語に対するUSAGE権限を持たなければなりません。また、言語が信用できない場合はスーパーユーザでなければなりません。これは、その言語における関数作成に必要な権限と同じです。

DOがトランザクションブロック内で実行された場合、プロシージャコードはトランザクション制御文を実行できません。DOが自身のトランザクション内で実行された場合にのみ、トランザクション制御文は認められます。

例

スキーマpublic内のすべてのビューに対するすべての権限をロールwebuserに付与します。

```
DO $$DECLARE r record;
BEGIN
    FOR r IN SELECT table_schema, table_name FROM information_schema.tables
             WHERE table_type = 'VIEW' AND table_schema = 'public'
    LOOP
        EXECUTE 'GRANT ALL ON ' || quote_ident(r.table_schema) || '.' || quote_ident(r.table_name)
 || ' TO webuser';
    END LOOP;
END$$;
```

互換性

標準SQLにはDO文はありません。

関連項目

CREATE LANGUAGE

DROP ACCESS METHOD

DROP ACCESS METHOD — アクセスメソッドを削除する

概要

```
DROP ACCESS METHOD [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

説明

DROP ACCESS METHODは既存のアクセスメソッドを削除します。スーパーユーザのみがアクセスメソッドを削除できます。

パラメータ

IF EXISTS

アクセスメソッドが存在しない時にエラーを発生させません。この場合、注意が発行されます。

name

既存のアクセスメソッドの名前です。

CASCADE

アクセスメソッドに依存するオブジェクト（演算子クラス、演算子族、インデックスなど）を自動的に削除し、さらに、それらのオブジェクトに依存するすべてのオブジェクトも削除します（5.14参照）。

RESTRICT

アクセスメソッドに依存するオブジェクトが1つでもあれば、削除を拒絶します。これがデフォルトです。

例

アクセスメソッドheptreeを削除するには次のようにします。

```
DROP ACCESS METHOD heptree;
```

互換性

DROP ACCESS METHODはPostgreSQLの拡張です。

関連項目

CREATE ACCESS METHOD

DROP AGGREGATE

DROP AGGREGATE — 集約関数を削除する

概要

```
DROP AGGREGATE [ IF EXISTS ] name ( aggregate_signature ) [, ...] [ CASCADE | RESTRICT ]


ここでaggregate_signatureは以下の通りです。

* |
[ argmode ] [ argname ] argtype [ , ... ] |
[ [ argmode ] [ argname ] argtype [ , ... ] ] ORDER BY [ argmode ] [ argname ] argtype [ , ... ]
```

説明

DROP AGGREGATEを実行すると、既存の集約関数定義を削除することができます。このコマンドを実行するには、現在のユーザがその集約関数を所有している必要があります。

パラメータ

IF EXISTS

集約が存在しない場合でもエラーになりません。この場合注意メッセージが発行されます。

name

既存の集約関数の名前です（スキーマ修飾名も可）。

argmode

引数のモードで、INまたはVARIADICです。省略した場合のデフォルトはINです。

argname

引数の名前です。DROP AGGREGATEは実際には引数の名前を無視することに注意してください。これは、集約関数の本体を特定するのに必要になるのは、引数のデータ型だけだからです。

argtype

集約関数の操作対象となる入力データ型です。引数を持たない関数を参照する場合は、引数指定の一覧の場所に*を記述してください。順序集合集約関数を参照する場合は、直接引数と集約引数の指定の間にORDER BYを記述してください。

CASCADE

その集約関数に依存しているオブジェクト（集約関数を利用しているビューなど）を自動的に削除し、さらにそれらのオブジェクトに依存するすべてのオブジェクトも削除します（5.14参照）。

RESTRICT

依存しているオブジェクトがある場合、その集約関数の削除要求を拒否します。こちらがデフォルトです。

注釈

順序集合集約を参照するための代替となる構文については、ALTER AGGREGATEに記述されています。

例

integer型のmyavg集約関数を削除します。

```
DROP AGGREGATE myavg(integer);
```

順序列の任意のリストと直接引数の適合するリストをとる、仮想集合集約関数myrankを削除します。

```
DROP AGGREGATE myrank(VARIADIC "any" ORDER BY VARIADIC "any");
```

複数の集約関数を1つのコマンドで削除します。

```
DROP AGGREGATE myavg(integer), myavg(bigint);
```

互換性

標準SQLには、DROP AGGREGATE文はありません。

関連項目

ALTER AGGREGATE, CREATE AGGREGATE

DROP CAST

DROP CAST — キャストを削除する

概要

```
DROP CAST [ IF EXISTS ] (source_type AS target_type) [ CASCADE | RESTRICT ]
```

説明

DROP CASTは、以前に定義したキャストを削除します。

キャストを削除するには、変換元または変換先のデータ型を所有している必要があります。これらは、キャストを作成するために必要な権限と同じです。

パラメータ

IF EXISTS

キャストが存在しない場合でもエラーになりません。この場合注意メッセージが発行されます。

source_type

キャストする変換元のデータ型の名前です。

target_type

キャストする変換先のデータ型の名前です。

CASCADE
RESTRICT

キャストは依存関係を持たないため、これらのキーワードを指定しても効果はありません。

例

text型からint型へのキャストを削除します。

```
DROP CAST (text AS int);
```

互換性

DROP CASTコマンドは、標準SQLに準拠しています。

関連項目

CREATE CAST

DROP COLLATION

DROP COLLATION — 照合順序を削除する

概要

```
DROP COLLATION [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

説明

DROP COLLATIONは事前に定義された照合順序を削除します。照合順序を削除するためには、その照合順序を所有していなければなりません

パラメータ

IF EXISTS

照合順序が存在しない場合でもエラーになりません。この場合注意メッセージが発行されます。

name

照合順序の名前です。照合順序名はスキーマ修飾可能です。

CASCADE

照合順序に依存するオブジェクトを自動的に削除し、さらにそれらのオブジェクトに依存するすべてのオブジェクトも削除します（5.14参照）。

RESTRICT

依存するオブジェクトが存在する場合、照合順序の削除を拒絶します。これがデフォルトです。

例

germanという名前の照合順序を削除します。

```
DROP COLLATION german;
```

互換性

PostgreSQLの拡張であるIF EXISTSオプションを除き、DROP COLLATIONコマンドは標準SQLに従います。

関連項目

ALTER COLLATION, CREATE COLLATION

DROP CONVERSION

DROP CONVERSION — 符号化方式変換を削除する

概要

```
DROP CONVERSION [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

説明

DROP CONVERSIONを使用すると、以前に定義した符号化方式変換（以下、単に「変換」と記します）を削除できます。変換を削除するには、その変換を所有していなければなりません。

パラメータ

IF EXISTS

変換が存在しない場合でもエラーになりません。この場合注意メッセージが発行されます。

name

変換の名前です。変換名はスキーマ修飾可能です。

CASCADE
RESTRICT

変換は依存関係を持たないため、これらのキーワードを指定しても効果はありません。

例

mynameという名前の変換を削除します。

```
DROP CONVERSION myname;
```

互換性

標準SQLにはDROP CONVERSION文はありません。しかし、PostgreSQLのCREATE CONVERSIONに似たCREATE TRANSLATION文があるように、DROP TRANSLATION文があります。

関連項目

ALTER CONVERSION, CREATE CONVERSION

DROP DATABASE

DROP DATABASE — データベースを削除する

概要

```
DROP DATABASE [ IF EXISTS ] name [ [ WITH ] ( option [, ...] ) ]


ここでoptionは以下の通りです。

    FORCE
```

説明

DROP DATABASEは、データベースを削除します。そのデータベースの項目をカタログから削除し、データを保存していたディレクトリを削除します。データベースの所有者のみがこのコマンドを実行することができます。対象とするデータベースに接続している時は実行できません（このコマンドを実行する時は、postgresや他のデータベースに接続してください）。また、他のユーザが対象とするデータベースに接続している時は、以下に書かれたFORCEオプションを使わない限り、このコマンドは失敗します。

DROP DATABASEは元に戻すことができません。十分注意して使用してください。

パラメータ

IF EXISTS

データベースが存在しない場合でもエラーになりません。この場合注意メッセージが発行されます。

name

削除するデータベースの名前です。

FORCE

対象とするデータベースへの既存の接続をすべて終了することを試みます。対象とするデータベースにプリペアドトランザクション、実行中の論理レプリケーションスロット、サブスクリプションが存在する場合には終了しません。

これは、現在のユーザに他の接続を終了する権限がない場合には失敗します。要求される権限は、9.27.2に書かれているpg_terminate_backendと同じです。これは接続を終了できない場合も失敗します。

注釈

DROP DATABASEはトランザクションブロックの内部では実行できません。

対象とするデータベースに接続している間は、このコマンドを実行することができません。したがって、このコマンドのラッパであるdropdbプログラムを使用する方がより便利かもしれません。

互換性

標準SQLにはDROP DATABASE文はありません。

関連項目

CREATE DATABASE

DROP DOMAIN

DROP DOMAIN —ドメインを削除する

概要

```
DROP DOMAIN [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

説明

DROP DOMAINはドメインを削除します。ドメインを削除できるのは、ドメインの所有者のみです。

パラメータ

IF EXISTS

ドメインが存在しない場合でもエラーになりません。この場合注意メッセージが発行されます。

name

既存のドメインの名前です（スキーマ修飾名も可）。

CASCADE

ドメインに依存するオブジェクト（テーブルの列など）を自動的に削除し、さらにそれらのオブジェクトに依存するすべてのオブジェクトも削除します（5.14参照）。

RESTRICT

依存しているオブジェクトがある場合、そのドメインの削除要求を拒否します。これがデフォルトです。

例

ドメインboxを削除します。

```
DROP DOMAIN box;
```

互換性

このコマンドは、PostgreSQLの拡張であるIF EXISTSオプションを除き、標準SQLに準拠しています。

関連項目

CREATE DOMAIN, ALTER DOMAIN

DROP EVENT TRIGGER

DROP EVENT TRIGGER — イベントトリガを削除する

概要

```
DROP EVENT TRIGGER [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

説明

DROP EVENT TRIGGERは既存のイベントトリガを削除します。このコマンドを実行するためには、現在のユーザがイベントトリガの所有者でなければなりません。

パラメータ

IF EXISTS

イベントトリガが存在しない場合でもエラーを発生しません。この場合は注意が発生します。

name

削除対象のイベントトリガの名前です。

CASCADE

トリガに依存するオブジェクトを自動的に削除し、さらにそれらのオブジェクトに依存するすべてのオブジェクトも削除します（5.14参照）。

RESTRICT

依存するオブジェクトが存在する場合はトリガの削除を取りやめます。これがデフォルトです。

例

トリガsnitchを破棄します。

```
DROP EVENT TRIGGER snitch;
```

互換性

DROP EVENT TRIGGER文は標準SQLにはありません。

関連項目

CREATE EVENT TRIGGER, ALTER EVENT TRIGGER

DROP EXTENSION

DROP EXTENSION — 拡張を削除する

概要

```
DROP EXTENSION [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

説明

DROP EXTENSIONはデータベースから拡張を削除します。拡張を削除すると、その構成オブジェクトも削除されます。

DROP EXTENSIONを使用するためにはその拡張を所有していなければなりません。

パラメータ

IF EXISTS

拡張が存在しない場合でもエラーになりません。この場合注意メッセージが発行されます。

name

インストールされている拡張の名前です。

CASCADE

削除する拡張に依存しているオブジェクトを自動的に削除し、さらにそれらのオブジェクトに依存するすべてのオブジェクトも削除します(5.14参照)。

RESTRICT

(それ自身が所有するメンバオブジェクトと同じDROPコマンドで指定された他の拡張以外に)依存しているオブジェクトがある場合に、拡張の削除を拒否します。こちらがデフォルトです。

例

現在のデータベースからhstore拡張を削除します。

```
DROP EXTENSION hstore;
```

例えば何らかのテーブルがhstore型の列を持つなど、データベース内でhstoreのオブジェクトを使用している場合、このコマンドは失敗します。こうした依存するオブジェクトも含めて強制的に削除するにはCASCADEを付けてください。

互換性

DROP EXTENSIONはPostgreSQLの拡張です。

関連項目

CREATE EXTENSION, ALTER EXTENSION

DROP FOREIGN DATA WRAPPER

DROP FOREIGN DATA WRAPPER — 外部データラッパを削除する

概要

```
DROP FOREIGN DATA WRAPPER [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

説明

DROP FOREIGN DATA WRAPPERは既存の外部データラッパを削除します。このコマンドを実行するためには、現在のユーザは外部データラッパの所有者でなければなりません。

パラメータ

IF EXISTS

外部データラッパが存在しない場合にエラーを発生しません。この場合、注意が発行されます。

name

既存の外部データラッパの名前です。

CASCADE

外部データラッパに依存するオブジェクト（外部テーブルやサーバなど）を自動的に削除し、さらにそれらのオブジェクトに依存するすべてのオブジェクトも削除します（5.14参照）。

RESTRICT

外部データラッパに依存するオブジェクトが存在する場合に削除を取りやめます。これがデフォルトです。

例

外部データラッパdbiを削除します。

```
DROP FOREIGN DATA WRAPPER dbi;
```

互換性

DROP FOREIGN DATA WRAPPERはISO/IEC 9075-9 (SQL/MED)に従います。IF EXISTS句はPostgreSQLの拡張です。

関連項目

CREATE FOREIGN DATA WRAPPER, ALTER FOREIGN DATA WRAPPER

DROP FOREIGN TABLE

DROP FOREIGN TABLE — 外部テーブルを削除する

概要

```
DROP FOREIGN TABLE [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

説明

DROP FOREIGN TABLEは外部テーブルを削除します。外部テーブルの所有者のみが削除することができます。

パラメータ

IF EXISTS

外部テーブルが存在しない場合でもエラーになりません。この場合注意メッセージが発行されます。

name

削除する外部テーブルの名前です（スキーマ修飾名も可）。

CASCADE

削除する外部テーブルに依存しているオブジェクト（ビューなど）を自動的に削除し、さらにそれらのオブジェクトに依存するすべてのオブジェクトも削除します（5.14参照）。

RESTRICT

依存しているオブジェクトがある場合に、外部テーブルの削除を拒否します。こちらがデフォルトです。

例

filmsおよびdistributorsという2つの外部テーブルを破棄します。

```
DROP FOREIGN TABLE films, distributors;
```

互換性

このコマンドは、ISO/IEC 9075-9 (SQL/MED)では1つのコマンドで1つの外部テーブルのみを削除できるという点、および、PostgreSQLの拡張であるIF EXISTSオプションを除き、この標準に従います。

関連項目

ALTER FOREIGN TABLE, CREATE FOREIGN TABLE

DROP FUNCTION

DROP FUNCTION — 関数を削除する

概要

```
DROP FUNCTION [ IF EXISTS ] name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ] [, ...]
    [ CASCADE | RESTRICT ]
```

説明

DROP FUNCTIONは既存の関数定義を削除します。このコマンドを実行できるのは、その関数の所有者のみです。関数の引数の型は必ず指定しなければなりません。異なる引数を持つ同じ名前の関数が複数存在する可能性があるからです。

パラメータ

IF EXISTS

関数が存在しない場合でもエラーになりません。この場合注意メッセージが発行されます。

name

既存の関数の名前です（スキーマ修飾名も可）。引数リストを指定しない場合、名前はスキーマ内で一意でなければなりません。

argmode

引数のモードで、IN、OUT、INOUT、VARIADICのいずれかです。省略された場合のデフォルトはINです。関数の識別を行うには入力引数のみが必要ですので、実際にはDROP FUNCTIONがOUT引数を無視することに注意してください。ですので、IN、INOUT、およびVARIADIC引数を列挙することで十分です。

argname

引数の名前です。関数の識別を行うには引数のデータ型のみが必要ですので、実際にはDROP FUNCTIONは引数の名前を無視することに注意してください。

argtype

もしあれば、その関数の引数のデータ型（スキーマ修飾可能）です。

CASCADE

関数に依存するオブジェクト（演算子やトリガなど）を自動的に削除し、さらにそれらのオブジェクトに依存するすべてのオブジェクトも削除します（5.14参照）。

RESTRICT

依存しているオブジェクトがある場合、その関数の削除を拒否します。これがデフォルトです。

例

次のコマンドは平方根関数を削除します。

```
DROP FUNCTION sqrt(integer);
```

複数の関数を1つのコマンドで削除します。

```
DROP FUNCTION sqrt(integer), sqrt(bigint);
```

関数名がスキーマ内で一意であれば、引数リストを付けなくても参照することができます。

```
DROP FUNCTION update_employee_salaries;
```

これは以下とは異なることに注意してください。

```
DROP FUNCTION update_employee_salaries();
```

後者は引数がゼロ個の関数を参照するのに対し、前者は引数の個数はゼロ個も含め、何個でも良く、ただし名前が一意である必要があります。

互換性

このコマンドは標準SQLに従いしますが、以下はPostgreSQLの拡張です。

- 標準SQLでは1つのコマンドで関数を1つ削除することしかできません。
- IF EXISTSオプション
- 引数のモードと名前を指定できること

関連項目

CREATE FUNCTION, ALTER FUNCTION, DROP PROCEDURE, DROP ROUTINE

DROP GROUP

DROP GROUP — データベースロールを削除する

概要

```
DROP GROUP [ IF EXISTS ] name [, ...]
```

説明

DROP GROUPはDROP ROLEの別名になりました。

互換性

標準SQLにはDROP GROUP文はありません。

関連項目

DROP ROLE

DROP INDEX

DROP INDEX — インデックスを削除する

概要

```
DROP INDEX [ CONCURRENTLY ] [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

説明

DROP INDEXはデータベースシステムから既存のインデックスを削除します。このコマンドを実行するには、 そのインデックスを所有していなければなりません。

パラメータ

CONCURRENTLY

インデックスのテーブルに対して同時に実行される選択、挿入、更新、削除をロックすることなくインデックスを削除します。通常のDROP INDEXではテーブルに対する排他ロックを獲得し、インデックスの削除が完了するまで他のアクセスをブロックします。このオプションを使うと、競合するトランザクションが完了するまでコマンドは待たされます。

このオプションを使用する時に注意すべき、複数の警告があります。指定できるインデックス名は1つだけであり、また、CASCADEオプションはサポートされません。(したがってUNIQUEまたはPRIMARY KEY制約をサポートするインデックスをこの方法で削除することはできません。) また、通常のDROP INDEXはトランザクションブロックの中で行うことができますが、DROP INDEX CONCURRENTLYはできません。最後に、 パーティションテーブルのインデックスをこのオプションで削除することはできません。

一時テーブルに対してはDROP INDEXは常に同時削除ではありません。他のセッションはアクセスできませんし、同時でないインデックス削除の方がより安価だからです。

IF EXISTS

インデックスが存在しない場合でもエラーになりません。この場合注意メッセージが発行されます。

name

削除するインデックスの名前です(スキーマ修飾名も可)。

CASCADE

そのインデックスに依存しているオブジェクトを自動的に削除し、さらにそれらのオブジェクトに依存するすべてのオブジェクトも削除します(5.14参照)。

RESTRICT

依存しているオブジェクトがある場合、そのインデックスの削除を拒否します。こちらがデフォルトです。

例

次のコマンドはインデックスtitle_idxを削除します。

```
DROP INDEX title_idx;
```

互換性

DROP INDEXはPostgreSQLの言語拡張です。標準SQLにはインデックスに関する規定はありません。

関連項目

CREATE INDEX

DROP LANGUAGE

DROP LANGUAGE — 手続き言語を削除する

概要

```
DROP [ PROCEDURAL ] LANGUAGE [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

説明

DROP LANGUAGEは過去に登録された手続き言語の定義を削除します。DROP LANGUAGEを使用するにはスーパーユーザか言語の所有者でなければなりません。

> **注記**
>
> PostgreSQL 9.1からほとんどの手続き言語は「拡張」にまとめられましたので、DROP LANGUAGEではなくDROP EXTENSIONを使用して削除すべきです。

パラメータ

IF EXISTS

言語が存在しない場合でもエラーになりません。この場合注意メッセージが発行されます。

name

既存の手続き言語の名前です。後方互換性を保持するため、この名前は単一引用符で囲むことができます。

CASCADE

その言語に依存するオブジェクト（その言語で記述された関数など）を自動的に削除し、さらにそれらのオブジェクトに依存するすべてのオブジェクトも削除します（5.14参照）。

RESTRICT

依存しているオブジェクトがある場合、その言語の削除を拒否します。こちらがデフォルトです。

例

次のコマンドはplsampleという手続き言語を削除します。

```
DROP LANGUAGE plsample;
```

互換性

標準SQLにはDROP LANGUAGE文はありません。

関連項目

ALTER LANGUAGE, CREATE LANGUAGE

DROP MATERIALIZED VIEW

DROP MATERIALIZED VIEW — マテリアライズドビューを削除する

概要

```
DROP MATERIALIZED VIEW [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

説明

DROP MATERIALIZED VIEWは既存のマテリアライズドビューを削除します。このコマンドを実行するためにはマテリアライズドビューの所有者でなければなりません。

パラメータ

IF EXISTS

マテリアライズドビューが存在しない場合でもエラーを発生しません。この場合注意が発生します。

name

削除対象のマテリアライズドビューの名前（スキーマ修飾可）です。

CASCADE

マテリアライズドビューに依存するオブジェクト（他のマテリアライズドビューや通常のビューなど）を自動的に削除し、さらにそれらのオブジェクトに依存するすべてのオブジェクトも削除します（5.14参照）。

RESTRICT

依存するオブジェクトがある場合にマテリアライズドビューの削除を取りやめます。これがデフォルトです。

例

以下のコマンドはorder_summaryという名前のマテリアライズドビューを削除します。

```
DROP MATERIALIZED VIEW order_summary;
```

互換性

DROP MATERIALIZED VIEWはPostgreSQLの拡張です。

関連項目

CREATE MATERIALIZED VIEW, ALTER MATERIALIZED VIEW, REFRESH MATERIALIZED VIEW

DROP OPERATOR

DROP OPERATOR — 演算子を削除する

概要

```
DROP OPERATOR [ IF EXISTS ] name ( { left_type | NONE } , { right_type | NONE } ) [, ...]
 [ CASCADE | RESTRICT ]
```

説明

DROP OPERATORはデータベースシステムから既存の演算子を削除します。このコマンドを実行するには、その演算子の所有者でなければなりません。

パラメータ

IF EXISTS

演算子が存在しない場合でもエラーになりません。この場合注意メッセージが発行されます。

name

既存の演算子の名前です（スキーマ修飾名も可）。

left_type

演算子の左オペランドのデータ型です。演算子に左オペランドがない場合は、NONEと記述します。

right_type

演算子の右オペランドのデータ型です。演算子に右オペランドがない場合は、NONEと記述します。

CASCADE

演算子に依存するオブジェクト（その演算子を使用するビューなど）を自動的に削除し、さらにそれらのオブジェクトに依存するすべてのオブジェクトも削除します（5.14参照）。

RESTRICT

依存するオブジェクトがある場合、演算子の削除を拒否します。これがデフォルトです。

例

integer型の累乗を求める演算子a^nを削除します。

```
DROP OPERATOR ^ (integer, integer);
```

bit型のビット列の補数を求める左単項演算子~bを削除します。

```
DROP OPERATOR ~ (none, bit);
```

bigint型の階乗を求める右単項演算子x!を削除します。

```
DROP OPERATOR ! (bigint, none);
```

複数の演算子を1つのコマンドで削除します。

```
DROP OPERATOR ~ (none, bit), ! (bigint, none);
```

互換性

標準SQLにはDROP OPERATOR文はありません。

関連項目

CREATE OPERATOR, ALTER OPERATOR

DROP OPERATOR CLASS

DROP OPERATOR CLASS — 演算子クラスを削除する

概要

```
DROP OPERATOR CLASS [ IF EXISTS ] name USING index_method [ CASCADE | RESTRICT ]
```

説明

DROP OPERATOR CLASSは既存の演算子クラスを削除します。このコマンドを実行するには、演算子クラスの所有者でなければなりません。

DROP OPERATOR CLASSはそのクラスで参照される演算子や関数をまったく削除しません。演算子クラスに依存するインデックスがある場合、削除を成功させるためにはCASCADEを指定する必要があります。

パラメータ

IF EXISTS

演算子クラスが存在しない場合でもエラーになりません。この場合注意メッセージが発行されます。

name

既存の演算子クラスの名前です（スキーマ修飾名も可）。

index_method

この演算子クラスを使用するインデックスアクセスメソッドの名前です。

CASCADE

この演算子クラスに依存しているオブジェクト（インデックスなど）を自動的に削除し、さらにそれらのオブジェクトに依存するすべてのオブジェクトも削除します（5.14参照）。

RESTRICT

依存しているオブジェクトがある場合に、その演算子クラスの削除を拒否します。こちらがデフォルトです。

注釈

DROP OPERATOR CLASSは、そのクラスを含む演算子族を削除しません。たとえその演算子族が空になったとしても（特にその演算子族がCREATE OPERATOR CLASSで暗黙的に作成された場合でも）です。空の演算子族は存在しても害はありませんが、整理するためにDROP OPERATOR FAMILYを使用してこの演算子族を削除することができます。あるいは最初にDROP OPERATOR FAMILYを使って下さい。

例

widget_opsという名前のB-tree演算子クラスを削除します。

```
DROP OPERATOR CLASS widget_ops USING btree;
```

演算子クラスを使用するインデックスが存在する場合、このコマンドは実行できません。このようなインデックスを演算子クラスとともに削除するには、CASCADEを指定します。

互換性

標準SQLにはDROP OPERATOR CLASSは存在しません。

関連項目

ALTER OPERATOR CLASS, CREATE OPERATOR CLASS, DROP OPERATOR FAMILY

DROP OPERATOR FAMILY

DROP OPERATOR FAMILY — 演算子族を削除する

概要

```
DROP OPERATOR FAMILY [ IF EXISTS ] name USING index_method [ CASCADE | RESTRICT ]
```

説明

DROP OPERATOR FAMILYは既存の演算子族を削除します。このコマンドを実行するためには、その演算子族の所有者でなければなりません。

DROP OPERATOR FAMILYには、その演算子族に含まれるすべての演算子クラスの削除も含まれています。しかし、演算子族から参照される演算子や関数はまったく削除されません。この演算子族内の演算子クラスに依存するインデックスが存在する場合、削除を完了させるためにはCASCADEを指定しなければなりません。

パラメータ

IF EXISTS

演算子族が存在しない場合にエラーとしません。この場合注意メッセージが表示されます。

name

既存の演算子族の名前（スキーマ修飾可）です。

index_method

演算子族が対象とするインデックスアクセスメソッドの名前です。

CASCADE

演算子族に依存するオブジェクトを自動的に削除し、さらにそれらのオブジェクトに依存するすべてのオブジェクトも削除します（5.14参照）。

RESTRICT

何らかのオブジェクトが演算子族に依存している場合、削除を中止します。これがデフォルトです。

例

B-tree演算子族float_opsを削除します。

```
DROP OPERATOR FAMILY float_ops USING btree;
```

この演算子族内の演算子クラスを使用するインデックスが存在する場合、このコマンドは失敗します。演算子族に関連するこうしたインデックスも削除する場合はCASCADEを付けてください。

互換性

標準SQLにはDROP OPERATOR FAMILY文はありません。

関連項目

ALTER OPERATOR FAMILY, CREATE OPERATOR FAMILY, ALTER OPERATOR CLASS, CREATE OPERATOR CLASS, DROP OPERATOR CLASS

DROP OWNED

DROP OWNED — データベースロールにより所有されるデータベースオブジェクトを削除します。

概要

```
DROP OWNED BY { name | CURRENT_USER | SESSION_USER } [, ...] [ CASCADE | RESTRICT ]
```

説明

DROP OWNEDは、現在のデータベース内にある、指定したロールが所有するオブジェクトをすべて削除します。また、現在のデータベース内にあるオブジェクトや共有オブジェクト（データベース、テーブル空間）に対して指定したロールに与えられた権限も取り消されます。

パラメータ

name

所有するオブジェクトを削除し、その権限が取り消されるロールの名称です。

CASCADE

関連するオブジェクトに依存するオブジェクトを自動的に削除し、さらにそれらのオブジェクトに依存するすべてのオブジェクトも削除します（5.14参照）。

RESTRICT

他のデータベースオブジェクトが関連オブジェクトに依存する場合、ロールにより所有されるオブジェクトの削除を取りやめます。これがデフォルトです。

注釈

DROP OWNEDはよく、複数ロールの削除の前処理として使用されます。DROP OWNEDは現在のデータベース内のオブジェクトにのみ影響しますので、このコマンドを通常、削除予定のロールが所有するオブジェクトを持つデータベース毎に実行する必要があります。

CASCADEオプションを使用すると、このコマンドで再帰的に他のユーザが所有するオブジェクトを処理する可能性があります。

代わりにREASSIGN OWNEDコマンドを使い、1つまたは複数のロールが所有するすべてのデータベースオブジェクトの所有権を再割り当てすることもできます。ただしREASSIGN OWNEDは他のオブジェクトの権限については処理しません。

ロールにより所有されるデータベースおよびテーブル空間は削除されません。

詳しくは21.4を参照してください。

互換性

DROP OWNEDコマンドはPostgreSQLの拡張です。

関連項目

REASSIGN OWNED, DROP ROLE

DROP POLICY

DROP POLICY — テーブルから行単位のセキュリティポリシーを削除する

概要

```
DROP POLICY [ IF EXISTS ] name ON table_name [ CASCADE | RESTRICT ]
```

説明

DROP POLICYはテーブルから指定したポリシーを削除します。テーブルの最後のポリシーが削除され、そのテーブルではまだALTER TABLEによる行単位セキュリティが有効な場合は、デフォルト拒否のポリシーが使われることに注意して下さい。テーブルのポリシーの存在の有無に関わらず、ALTER TABLE ... DISABLE ROW LEVEL SECURITYを使い、テーブルの行単位セキュリティを無効にすることができます。

パラメータ

IF EXISTS

ポリシーが存在しない時にエラーを発生させません。この場合、注意が発行されます。

name

削除するポリシーの名前です。

table_name

ポリシーが適用されているテーブルの名前（スキーマ修飾可）です。

CASCADE
RESTRICT

これらのキーワードには何の効果もありません。ポリシーには依存関係がないからです。

例

my_tableという名前のテーブル上のp1というポリシーを削除するには、次のようにします。

```
DROP POLICY p1 ON my_table;
```

互換性

DROP POLICYはPostgreSQLの拡張です。

関連項目

CREATE POLICY, ALTER POLICY

DROP PROCEDURE

DROP PROCEDURE — プロシージャを削除する

概要

```
DROP PROCEDURE [ IF EXISTS ] name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ] [, ...]
    [ CASCADE | RESTRICT ]
```

説明

DROP PROCEDUREは既存のプロシージャ定義を削除します。このコマンドを実行できるのは、そのプロシージャの所有者のみです。プロシージャの引数の型は必ず指定しなければなりません。異なる引数を持つ同じ名前のプロシージャが複数存在する可能性があるからです。

パラメータ

IF EXISTS

プロシージャが存在しない場合でもエラーになりません。この場合注意(NOTICE)メッセージが発行されます。

name

既存の関数の名前です（スキーマ修飾名も可）。引数リストを指定しない場合、名前はスキーマ内で一意でなければなりません。

argmode

引数モードで、INかVARIADICのいずれかです。省略した場合のデフォルトはINです。

argname

引数の名前です。プロシージャの識別を行うには引数のデータ型のみが必要ですので、実際にはDROP PROCEDUREは引数の名前を無視することに注意してください。

argtype

もしあれば、そのプロシージャの引数のデータ型（スキーマ修飾可能）です。

CASCADE

プロシージャに依存するオブジェクトを自動的に削除し、さらにそれらのオブジェクトに依存するすべてのオブジェクトも削除します（5.14参照）。

RESTRICT

依存しているオブジェクトがある場合、そのプロシージャの削除を拒否します。これがデフォルトです。

例

```
DROP PROCEDURE do_db_maintenance();
```

互換性

このコマンドはSQL標準に準拠しますが、以下のPostgreSQLの拡張があります。

- 標準はコマンドごとに一つのプロシージャしか削除できません。
- IF EXISTSオプション
- 引数モードと引数名を指定できます。

関連項目

CREATE PROCEDURE, ALTER PROCEDURE, DROP FUNCTION, DROP ROUTINE

DROP PUBLICATION

DROP PUBLICATION — パブリケーションを削除する

概要

```
DROP PUBLICATION [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

説明

DROP PUBLICATIONは既存のパブリケーションをデータベースから削除します。

パブリケーションはその所有者またはスーパーユーザによってのみ削除することができます。

パラメータ

IF EXISTS

パブリケーションが存在しない場合でもエラーになりません。この場合、注意メッセージが発行されます。

name

既存のパブリケーションの名前です。

CASCADE
RESTRICT

パブリケーションに依存するものはないので、これらのキーワードは何も効果がありません。

例

パブリケーションを削除します。

```
DROP PUBLICATION mypublication;
```

互換性

DROP PUBLICATIONはPostgreSQLの拡張です。

関連項目

CREATE PUBLICATION, ALTER PUBLICATION

DROP ROLE

DROP ROLE — データベースロールを削除する

概要

```
DROP ROLE [ IF EXISTS ] name [, ...]
```

説明

DROP ROLEは指定したロール（複数可）を削除します。スーパーユーザロールを削除するには、自身もスーパーユーザでなければなりません。スーパーユーザ以外のロールを削除するには、CREATEROLE権限を持たなければなりません。

データベースクラスタのいずれかから参照されている場合、ロールを削除することができません。削除しようとしてもエラーとなります。ロールを削除する前に、そのロールが所有するオブジェクトすべてを削除（またはその所有権を変更）しなければなりません。また、そのロールが他のオブジェクトについて付与された権限も取り消されなければなりません。この目的のためにはREASSIGN OWNEDおよびDROP OWNEDコマンドが有用です。詳しくは21.4を参照して下さい。

しかし、ロール内のロールメンバ資格を削除する必要はありません。DROP ROLEは自動的に他のロール内にある対象ロールのメンバ資格を取り消します。他のロールは削除されることも何らかの影響を受けることもありません。

パラメータ

IF EXISTS

ロールが存在しない場合でもエラーになりません。この場合注意メッセージが発行されます。

name

削除対象のロールの名前です。

注釈

PostgreSQLには、このコマンドと同じ機能を持つプログラムdropuserがあります。（実際には、このプログラムはこのコマンドを呼び出しています。）こちらはコマンドシェルから実行することができます。

例

ロールを削除します。

```
DROP ROLE jonathan;
```

互換性

標準SQLではDROP ROLEを定義していますが、一度に1つのロールしか削除することができません。また、PostgreSQLとは異なる権限が必要であると規定しています。

関連項目

CREATE ROLE, ALTER ROLE, SET ROLE

DROP ROUTINE

DROP ROUTINE — ルーチンを削除する

概要

```
DROP ROUTINE [ IF EXISTS ] name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ] [, ...]
    [ CASCADE | RESTRICT ]
```

説明

DROP ROUTINEは既存のルーチン、すなわち、集約関数や通常の関数、プロシージャを削除します。パラメータや例、さらなる詳細の説明はDROP AGGREGATEやDROP FUNCTION、DROP PROCEDUREを参照してください。

例

integer型に対するルーチンfooを削除します。

```
DROP ROUTINE foo(integer);
```

このコマンドはfooが集約、関数、プロシージャの何れであるかによらず動作します。

互換性

このコマンドはSQL標準に準拠していますが、以下のPostgreSQLの拡張があります。

- 標準ではコマンド毎に一つのルーチンしか削除できません。
- IF EXISTSオプション
- 引数のモードと名前を指定できます。
- 集約関数は拡張です。

関連項目

DROP AGGREGATE, DROP FUNCTION, DROP PROCEDURE, ALTER ROUTINE

CREATE ROUTINEコマンドは無いことに注意してください。

DROP RULE

DROP RULE — 書き換えルールを削除する

概要

```
DROP RULE [ IF EXISTS ] name ON table_name [ CASCADE | RESTRICT ]
```

説明

DROP RULEは書き換えルールを削除します。

パラメータ

IF EXISTS

ルールが存在しない場合でもエラーになりません。この場合注意メッセージが発行されます。

name

削除するルールの名前です。

table_name

そのルールが適用されたテーブルもしくはビューの名前です（スキーマ修飾名も可）。

CASCADE

ルールに依存するオブジェクトを自動的に削除し、さらにそれらのオブジェクトに依存するすべてのオブジェクトも削除します（5.14参照）。

RESTRICT

依存するオブジェクトがある場合、ルールの削除を拒否します。こちらがデフォルトです。

例

newruleという書き換えルールを削除します。

```
DROP RULE newrule ON mytable;
```

互換性

DROP RULEはPostgreSQLの言語拡張で、問い合わせ書き換えシステム全体も言語拡張です。

関連項目

CREATE RULE, ALTER RULE

DROP SCHEMA

DROP SCHEMA — スキーマを削除する

概要

```
DROP SCHEMA [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

説明

DROP SCHEMAはデータベースからスキーマを削除します。

スキーマを削除できるのは、所有者またはスーパーユーザのみです。所有者は、スキーマ内に自分が所有していないオブジェクトが含まれていても、そのスキーマ（およびそこに含まれる全てのオブジェクト）を削除できます。

パラメータ

IF EXISTS

スキーマが存在しない場合でもエラーになりません。この場合注意メッセージが発行されます。

name

スキーマの名前です。

CASCADE

スキーマに含まれるオブジェクト（テーブル、関数など）を自動的に削除し、さらにそれらのオブジェクトに依存するすべてのオブジェクトも削除します（5.14参照）。

RESTRICT

スキーマにオブジェクトが含まれている場合は、スキーマの削除を拒否します。こちらがデフォルトです。

注釈

CASCADEオプションを使用すると、指定されたスキーマ以外にあるオブジェクトを削除することになる可能性があります。

例

データベースからmystuffスキーマ、およびそこに含まれる全てのオブジェクトを削除します。

```
DROP SCHEMA mystuff CASCADE;
```

互換性

標準SQLでは一度のコマンド実行につき1つのスキーマしか削除できないという点を除き、および、PostgreSQLの拡張であるIF EXISTSを除き、DROP SCHEMAは、標準SQLと完全な互換性を持ちます。

関連項目

ALTER SCHEMA, CREATE SCHEMA

DROP SEQUENCE

DROP SEQUENCE — シーケンスを削除する

概要

```
DROP SEQUENCE [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

説明

DROP SEQUENCEはシーケンス番号ジェネレータを削除します。シーケンスの削除はその所有者またはスーパーユーザのみが可能です。

パラメータ

IF EXISTS

シーケンスが存在しない場合でもエラーになりません。この場合、注意メッセージが発行されます。

name

シーケンスの名前です(スキーマ修飾名も可)。

CASCADE

このシーケンスに依存しているオブジェクトを自動的に削除し、さらにそれらのオブジェクトに依存するすべてのオブジェクトも削除します(5.14参照)。

RESTRICT

依存オブジェクトがある場合に、シーケンスの削除を拒否します。こちらがデフォルトです。

例

serialという名前のシーケンスを削除します。

```
DROP SEQUENCE serial;
```

互換性

標準では1コマンドで1つのシーケンスだけを削除できるという点、および、PostgreSQLの拡張であるIF EXISTSオプションを除き、DROP SEQUENCEは標準SQLに準拠します。

関連項目

CREATE SEQUENCE, ALTER SEQUENCE

DROP SERVER

DROP SERVER — 外部サーバの記述子を削除する

概要

```
DROP SERVER [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

説明

DROP SERVERは既存の外部サーバ記述子を削除します。このコマンドを実行するためには、現在のユーザはサーバの所有者でなければなりません。

パラメータ

IF EXISTS

サーバが存在しない場合にエラーを発生しません。この場合、注意が発行されます。

name

既存のサーバの名前です。

CASCADE

サーバに依存するオブジェクト（ユーザマップなど）を自動的に削除し、さらにそれらのオブジェクトに依存するすべてのオブジェクトも削除します（5.14参照）。

RESTRICT

依存するオブジェクトが存在する場合サーバの削除を取りやめます。これがデフォルトです。

例

サーバfooが存在すれば、を削除します。

```
DROP SERVER IF EXISTS foo;
```

互換性

DROP SERVERはISO/IEC 9075-9 (SQL/MED)に従います。IF EXISTS句はPostgreSQLの拡張です。

関連項目

CREATE SERVER, ALTER SERVER

DROP STATISTICS

DROP STATISTICS — 拡張統計を削除する

概要

```
DROP STATISTICS [ IF EXISTS ] name [, ...]
```

説明

DROP STATISTICSはデータベースから統計オブジェクトを削除します。統計オブジェクトの所有者、スキーマの所有者、あるいはスーパーユーザのみが統計オブジェクトを削除できます。

パラメータ

IF EXISTS

統計オブジェクトが存在しない場合でもエラーになりません。この場合、注意メッセージが発行されます。

name

削除する統計オブジェクトの名前（オプションでスキーマ修飾可）です。

例

別のスキーマにある2つの統計オブジェクトを削除し、それが存在しなくてもエラーにならないようにします。

```
DROP STATISTICS IF EXISTS
    accounting.users_uid_creation,
    public.grants_user_role;
```

互換性

標準SQLにはDROP STATISTICSコマンドはありません。

関連項目

ALTER STATISTICS, CREATE STATISTICS

DROP SUBSCRIPTION

DROP SUBSCRIPTION — サブスクリプションを削除する

概要

```
DROP SUBSCRIPTION [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

説明

DROP SUBSCRIPTIONはデータベースクラスタからサブスクリプションを削除します。

スーパーユーザのみがサブスクリプションを削除できます。

サブスクリプションがレプリケーションスロットに紐付けられている場合、DROP SUBSCRIPTIONをトランザクションブロックの内側で実行することはできません。(スロットの設定を解除するにはALTER SUBSCRIPTIONを使うことができます。)

パラメータ

name

削除対象のサブスクリプションの名前です。

CASCADE
RESTRICT

サブスクリプションに依存するものはないので、これらのキーワードは何も効果がありません。

注釈

リモートホストのレプリケーションスロットに紐付けられているサブスクリプション(これが通常の状態です)を削除するとき、DROP SUBSCRIPTIONはその操作の一部として、リモートホストに接続し、レプリケーションスロットを削除しようとします。リモートホスト上でサブスクリプションに割り当てられたリソースを解放するために、これが必要となります。リモートホストに到達できない、あるいはリモートのレプリケーションスロットが削除できない、存在しない、存在したことがない、という理由で削除に失敗した場合、DROP SUBSCRIPTIONコマンドは失敗します。この状況において先へ進むためには、ALTER SUBSCRIPTION ... SET (slot_name = NONE)を実行してサブスクリプションとレプリケーションスロットの紐付けを解除してください。その後ならDROP SUBSCRIPTIONはリモートホスト上で何のアクションも起こそうとしません。リモートのレプリケーションスロットがそれでも存在する場合、それを手作業で削除すべきであることに注意してください。そうしなければ、WALを保存し続け、最終的にはディスクを一杯にしてしまうかもしれません。30.2.1も参照してください。

サブスクリプションがレプリケーションスロットと紐付けられている場合、DROP SUBSCRIPTIONをトランザクションブロックの内側で実行することはできません。

例

サブスクリプションを削除します。

```
DROP SUBSCRIPTION mysub;
```

互換性

DROP SUBSCRIPTIONはPostgreSQLの拡張です。

関連項目

CREATE SUBSCRIPTION, ALTER SUBSCRIPTION

DROP TABLE

DROP TABLE — テーブルを削除する

概要

```
DROP TABLE [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

説明

DROP TABLEはデータベースからテーブルを削除します。テーブル所有者、スキーマ所有者、スーパーユーザのみがテーブルを削除することができます。テーブルを削除するのではなく、テーブルの行を空にするには、DELETEまたはTRUNCATEを使用してください。

DROP TABLEは、削除対象のテーブルについて存在するインデックス、ルール、トリガ、制約を全て削除します。しかし、ビューや他のテーブルの外部キー制約によって参照されているテーブルを削除するにはCASCADEを指定する必要があります（CASCADEを指定すると、テーブルに依存するビューは完全に削除されます。外部キー制約によって参照されている場合は、外部キー制約のみが削除され、その外部キーを持つテーブルそのものは削除されません）。

パラメータ

IF EXISTS

テーブルが存在しない場合でもエラーになりません。この場合、注意メッセージが発行されます。

name

削除するテーブルの名前です（スキーマ修飾名も可）。

CASCADE

削除するテーブルに依存しているオブジェクト（ビューなど）を自動的に削除し、さらにそれらのオブジェクトに依存するすべてのオブジェクトも削除します（5.14参照）。

RESTRICT

依存しているオブジェクトがある場合に、テーブルの削除を拒否します。こちらがデフォルトです。

例

2つのテーブル、filmsとdistributorsを削除します。

```
DROP TABLE films, distributors;
```

互換性

標準では1コマンドで1テーブルのみを削除できるという点、および、PostgreSQLの拡張である`IF EXISTS`オプションを除き、このコマンドは標準SQLに従います。

関連項目

ALTER TABLE, CREATE TABLE

DROP TABLESPACE

DROP TABLESPACE — テーブル空間を削除する

概要

```
DROP TABLESPACE [ IF EXISTS ] name
```

説明

DROP TABLESPACEはシステムからテーブル空間を削除します。

テーブル空間を削除できるのは、その所有者もしくはスーパーユーザのみです。テーブル空間を削除する前に、全てのデータベースオブジェクトが空になっていなければなりません。現在のデータベース内のオブジェクトが使用していなかったとしても、他のデータベース内のオブジェクトがそのテーブル空間上にあることがあります。また、活動中のセッションのいずれかのtemp_tablespacesのリストにそのテーブル空間が含まれている場合、一時ファイルがそのテーブル空間に存在するためにDROPが失敗する可能性があります。

パラメータ

IF EXISTS

テーブル空間が存在しない場合でもエラーになりません。この場合注意メッセージが発行されます。

name

テーブル空間の名前です。

注釈

トランザクションブロック内でDROP TABLESPACEを実行することはできません。

例

テーブル空間mystuffをシステムから削除します。

```
DROP TABLESPACE mystuff;
```

互換性

DROP TABLESPACEはPostgreSQLの拡張です。

関連項目

CREATE TABLESPACE, ALTER TABLESPACE

DROP TEXT SEARCH CONFIGURATION

DROP TEXT SEARCH CONFIGURATION — テキスト検索設定を削除する

概要

```
DROP TEXT SEARCH CONFIGURATION [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

説明

DROP TEXT SEARCH CONFIGURATIONは既存のテキスト検索設定を削除します。このコマンドを実行するためには、その設定の所有者でなければなりません。

パラメータ

IF EXISTS

テキスト検索設定が存在しない場合でもエラーとしません。この場合は注意が発行されます。

name

既存のテキスト検索設定の名称（スキーマ修飾可）です。

CASCADE

テキスト検索設定に依存するオブジェクトを自動的に削除し、さらにそれらのオブジェクトに依存するすべてのオブジェクトも削除します（5.14参照）。

RESTRICT

依存するオブジェクトが存在する場合、テキスト検索設定の削除を中止します。これがデフォルトです。

例

テキスト検索設定my_englishを削除します。

```
DROP TEXT SEARCH CONFIGURATION my_english;
```

to_tsvector呼び出し内でこの設定を参照するインデックスが存在する場合、このコマンドは成功しません。こうしたインデックスをテキスト検索設定と一緒に削除するためにはCASCADEを付けてください。

互換性

標準SQLにはDROP TEXT SEARCH CONFIGURATION文はありません。

関連項目

ALTER TEXT SEARCH CONFIGURATION, CREATE TEXT SEARCH CONFIGURATION

DROP TEXT SEARCH DICTIONARY

DROP TEXT SEARCH DICTIONARY — テキスト検索辞書を削除する

概要

```
DROP TEXT SEARCH DICTIONARY [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

説明

DROP TEXT SEARCH DICTIONARYは既存のテキスト検索辞書を削除します。このコマンドを実行するためには、その辞書の所有者でなければなりません。

パラメータ

IF EXISTS

テキスト検索辞書が存在しない場合でもエラーとしません。この場合は注意が発行されます。

name

既存のテキスト検索辞書の名称（スキーマ修飾可）です。

CASCADE

テキスト検索辞書に依存するオブジェクトを自動的に削除し、さらにそれらのオブジェクトに依存するすべてのオブジェクトも削除します（5.14参照）。

RESTRICT

依存するオブジェクトが存在する場合、テキスト検索辞書の削除を中止します。これがデフォルトです。

例

テキスト検索辞書englishを削除します。

```
DROP TEXT SEARCH DICTIONARY english;
```

この辞書を使用するテキスト検索設定が存在する場合、このコマンドは成功しません。こうした設定を辞書と一緒に削除するためにはCASCADEを付けてください。

互換性

標準SQLにはDROP TEXT SEARCH DICTIONARY文はありません。

関連項目

ALTER TEXT SEARCH DICTIONARY, CREATE TEXT SEARCH DICTIONARY

DROP TEXT SEARCH PARSER

DROP TEXT SEARCH PARSER — テキスト検索パーサを削除する

概要

```
DROP TEXT SEARCH PARSER [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

説明

DROP TEXT SEARCH PARSERは既存のテキスト検索パーサを削除します。このコマンドを実行するためには、スーパーユーザでなければなりません。

パラメータ

IF EXISTS

テキスト検索パーサが存在しない場合でもエラーとしません。この場合は注意が発行されます。

name

既存のテキスト検索パーサの名称（スキーマ修飾可）です。

CASCADE

テキスト検索パーサに依存するオブジェクトを自動的に削除し、さらにそれらのオブジェクトに依存するすべてのオブジェクトも削除します（5.14参照）。

RESTRICT

依存するオブジェクトが存在する場合、テキスト検索パーサの削除を中止します。これがデフォルトです。

例

テキスト検索パーサmy_parserを削除します。

```
DROP TEXT SEARCH PARSER my_parser;
```

このパーサを使用するテキスト検索設定が存在する場合、このコマンドは成功しません。こうした設定をパーサと一緒に削除するためにはCASCADEを付けてください。

互換性

標準SQLにはDROP TEXT SEARCH PARSER文はありません。

関連項目

ALTER TEXT SEARCH PARSER, CREATE TEXT SEARCH PARSER

DROP TEXT SEARCH TEMPLATE

DROP TEXT SEARCH TEMPLATE — テキスト検索テンプレートを削除する

概要

```
DROP TEXT SEARCH TEMPLATE [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

説明

DROP TEXT SEARCH TEMPLATEは既存のテキスト検索テンプレートを削除します。このコマンドを実行するためには、スーパーユーザでなければなりません。

パラメータ

IF EXISTS

テキスト検索テンプレートが存在しない場合でもエラーとしません。この場合は注意が発行されます。

name

既存のテキスト検索テンプレートの名称（スキーマ修飾可）です。

CASCADE

テキスト検索テンプレートに依存するオブジェクトを自動的に削除し、さらにそれらのオブジェクトに依存するすべてのオブジェクトも削除します（5.14参照）。

RESTRICT

依存するオブジェクトが存在する場合、テキスト検索テンプレートの削除を中止します。これがデフォルトです。

例

テキスト検索テンプレートthesaurusを削除します。

```
DROP TEXT SEARCH TEMPLATE thesaurus;
```

このテンプレートを使用するテキスト検索辞書が存在する場合、このコマンドは成功しません。こうした辞書をテンプレートと一緒に削除するためにはCASCADEを付けてください。

互換性

標準SQLにはDROP TEXT SEARCH TEMPLATE文はありません。

関連項目

ALTER TEXT SEARCH TEMPLATE, CREATE TEXT SEARCH TEMPLATE

DROP TRANSFORM

DROP TRANSFORM — 変換を削除する

概要

```
DROP TRANSFORM [ IF EXISTS ] FOR type_name LANGUAGE lang_name [ CASCADE | RESTRICT ]
```

説明

DROP TRANSFORMは以前に定義された変換を削除します。

変換を削除するには、型と言語を所有していなければなりません。これらは変換を作成するのに必要とされるのと同じ権限です。

パラメータ

IF EXISTS

変換が存在しない場合にエラーを発生させません。この場合、注意が発行されます。

type_name

変換のデータ型の名前です。

lang_name

変換の言語の名前です。

CASCADE

変換に依存するオブジェクトを自動的に削除し、さらにそれらのオブジェクトに依存するすべてのオブジェクトも削除します（5.14参照）。

RESTRICT

変換に依存するオブジェクトがある場合は、変換を削除しません。これがデフォルトです。

例

hstore型で言語plpythonuの変換を削除するには次のようにします。

```
DROP TRANSFORM FOR hstore LANGUAGE plpythonu;
```

互換性

この構文のDROP TRANSFORMはPostgreSQLの拡張です。詳しくはCREATE TRANSFORMを参照してください。

関連項目

CREATE TRANSFORM

DROP TRIGGER

DROP TRIGGER ―トリガを削除する

概要

```
DROP TRIGGER [ IF EXISTS ] name ON table_name [ CASCADE | RESTRICT ]
```

説明

DROP TRIGGERは既存のトリガ定義を削除します。このコマンドを実行できるのは、トリガが定義されたテーブルの所有者のみです。

パラメータ

IF EXISTS

トリガが存在しない場合でもエラーになりません。この場合注意メッセージが発行されます。

name

削除するトリガの名前です。

table_name

トリガが定義されたテーブルの名前です（スキーマ修飾名も可）。

CASCADE

このトリガに依存しているオブジェクトを自動的に削除し、さらにそれらのオブジェクトに依存するすべてのオブジェクトも削除します（5.14参照）。

RESTRICT

依存しているオブジェクトがある場合に、トリガの削除を拒否します。こちらがデフォルトです。

例

テーブルfilmsにあるトリガif_dist_existsを削除します。

```
DROP TRIGGER if_dist_exists ON films;
```

互換性

PostgreSQLのDROP TRIGGER文には標準SQLとの互換性がありません。標準SQLでは、トリガ名がテーブルに局所化されていないので、DROP TRIGGER nameというコマンドが使われています。

関連項目

CREATE TRIGGER

DROP TYPE

DROP TYPE — データ型を削除する

概要

```
DROP TYPE [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

説明

DROP TYPEはユーザ定義のデータ型を削除します。データ型を削除できるのは、その所有者のみです。

パラメータ

IF EXISTS

型が存在しない場合でもエラーになりません。この場合、注意メッセージが発行されます。

name

削除するデータ型の名前です（スキーマ修飾名も可）。

CASCADE

削除するデータ型に依存するオブジェクト（テーブルの列、関数、演算子など）を自動的に削除し、さらにそれらのオブジェクトに依存するすべてのオブジェクトも削除します（5.14参照）。

RESTRICT

依存しているオブジェクトがある場合に、データ型の削除を拒否します。こちらがデフォルトです。

例

boxデータ型を削除します。

```
DROP TYPE box;
```

互換性

このコマンドは、PostgreSQLの拡張であるIF EXISTSオプションを除き、標準SQL内の対応するコマンドと似ています。しかし、PostgreSQLのCREATE TYPEコマンドの多くとデータ型拡張機構は標準SQLとは異なる点に注意してください。

関連項目

ALTER TYPE, CREATE TYPE

DROP USER

DROP USER — データベースロールを削除する

概要

```
DROP USER [ IF EXISTS ] name [, ...]
```

説明

DROP USERはDROP ROLEの単なる別の綴りです。

互換性

DROP USER文はPostgreSQLの拡張です。標準SQLでは、ユーザの定義は実装に任されています。

関連項目

DROP ROLE

DROP USER MAPPING

DROP USER MAPPING — 外部サーバ用のユーザマップを削除します。

概要

```
DROP USER MAPPING [ IF EXISTS ] FOR { user_name | USER | CURRENT_USER | PUBLIC }
 SERVER server_name
```

説明

DROP USER MAPPINGは既存のユーザマップを外部サーバから削除します。

外部サーバの所有者は任意のユーザに対するそのサーバ向けのユーザマップを削除することができます。また、サーバ上でUSAGE権限がユーザに付与されている場合、ユーザは自身の持つユーザ名に対応するユーザマップを削除することができます。

パラメータ

IF EXISTS

ユーザマップが存在しない場合にエラーを発生しません。この場合、注意が発行されます。

user_name

対応付けされるユーザ名です。CURRENT_USERとUSERは現在のユーザの名前に対応します。PUBLICは、システム上の現在および将来のユーザ名すべてに対応させるために使用します。

server_name

ユーザマップのサーバ名です。

例

存在する場合、サーバfooからユーザマップbobを削除します。

```
DROP USER MAPPING IF EXISTS FOR bob SERVER foo;
```

互換性

DROP USER MAPPINGはISO/IEC 9075-9 (SQL/MED)に従います。IF EXISTS句はPostgreSQLの拡張です。

関連項目

CREATE USER MAPPING, ALTER USER MAPPING

DROP VIEW

DROP VIEW — ビューを削除する

概要

```
DROP VIEW [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

説明

DROP VIEWは既存のビューを削除します。このコマンドを実行できるのは、ビューの所有者のみです。

パラメータ

IF EXISTS

ビューが存在しなかったとしてもエラーになりません。この場合注意メッセージが発行されます。

name

削除するビューの名前です（スキーマ修飾名も可）。

CASCADE

削除するビューに依存しているオブジェクト（他のビューなど）を自動的に削除し、さらにそれらのオブジェクトに依存するすべてのオブジェクトも削除します（5.14参照）。

RESTRICT

依存するオブジェクトがある場合は、ビューの削除を拒否します。こちらがデフォルトです。

例

次のコマンドはkindsという名前のビューを削除します。

```
DROP VIEW kinds;
```

互換性

標準では1コマンドで1つのビューのみを削除できるという点を除き、およびPostgreSQLの拡張であるIF EXISTSオプションを除き、このコマンドは標準SQLに従っています。

関連項目

ALTER VIEW, CREATE VIEW

END

END — 現在のトランザクションをコミットする

概要

```
END [ WORK | TRANSACTION ] [ AND [ NO ] CHAIN ]
```

説明

ENDは現在のトランザクションをコミットします。これにより、そのトランザクションでなされた全ての変更は、 他に対して可視状態となります。また、クラッシュが発生したとしても永続性が保証されます。このコマンドは、PostgreSQLの拡張で、COMMITと同等です。

パラメータ

WORK
TRANSACTION

省略可能なキーワードです。何も効果がありません。

AND CHAIN

AND CHAINが指定されていれば、新しいトランザクションは、直前に終了したものと同じトランザクションの特性(SET TRANSACTIONを参照してください)で即時に開始されます。そうでなければ、新しいトランザクションは開始されません。

注釈

トランザクションのアボートにはROLLBACKを使用してください。

トランザクションの外側でENDを発行しても問題はありませんが、警告メッセージが表示されます。

例

現在のトランザクションをコミットし、全ての変更を永続化します。

```
END;
```

互換性

ENDはPostgreSQLの拡張です。このコマンドの機能は、標準SQLで規定されたCOMMITと同じです。

関連項目

BEGIN, COMMIT, ROLLBACK

EXECUTE

EXECUTE — プリペアド文を実行する

概要

```
EXECUTE name [ ( parameter [, ...] ) ]
```

説明

EXECUTEは、事前に作成されたプリペアド文を実行する際に使用します。プリペアド文はセッション中にしか存在できないため、事前に同一セッション中のPREPARE文によって作成されたものでなければなりません。

文を作成したPREPARE文にパラメータが指定されている場合は、これに適合するパラメータの集合がEXECUTEに渡される必要があります。そうしないと、エラーになります。(関数とは異なり)プリペアド文は、パラメータのデータ型や個数によってオーバーロードされることはありません。プリペアド文の名前は、1つのデータベースセッション内で一意でなければなりません。

プリペアド文の作成方法と使用方法についての詳細はPREPAREを参照してください。

パラメータ

`name`

実行するプリペアド文の名前を指定します。

`parameter`

プリペアド文に対するパラメータの実際の値を指定します。これは、プリペアド文が生成された時に決定される、そのパラメータのデータ型と互換性のある値を返す式である必要があります。

出力

EXECUTEで返されるコマンドタグは、EXECUTEではなく、プリペアド文のコマンドタグとなります。

例

例はPREPAREの説明のExamplesにあります。

互換性

標準SQLにはEXECUTE文が含まれていますが、これは埋め込みSQLでのみ使用できます。また、このバージョンのEXECUTE文では、多少異なる構文が使用されています。

関連項目

DEALLOCATE, PREPARE

EXPLAIN

EXPLAIN — 問い合わせ文の実行計画を表示する

概要

```
EXPLAIN [ ( option [, ...] ) ] statement
EXPLAIN [ ANALYZE ] [ VERBOSE ] statement


ここでoptionは以下のいずれかを取ることができます。

    ANALYZE [ boolean ]
    VERBOSE [ boolean ]
    COSTS [ boolean ]
    SETTINGS [ boolean ]
    BUFFERS [ boolean ]
    WAL [ boolean ]
    TIMING [ boolean ]
    SUMMARY [ boolean ]
    FORMAT { TEXT | XML | JSON | YAML }
```

説明

与えられた文に対して、PostgreSQLプランナが生成する実行計画を表示します。実行計画は、問い合わせ文が参照するテーブル（複数の場合もある）をスキャンする方法（単純なシーケンシャルスキャン、インデックススキャンなど）、複数のテーブルを参照する場合に、各テーブルから取り出した行を結合するために使用する結合アルゴリズムを示すものです。

表示内容の中でも、最も重要なのは、文の実行にかかるコストの見積もりです。これは、プランナが文の実行にかかる時間（任意の、しかし慣習的にはディスクページ抽出を意味するコスト単位で計測）を推測したものです。具体的には、2つの数が表示されます。1つは最初の行が返されるまでのスタートアップコスト、もう1つはすべての行が返されるまでの合計コストです。ほとんどの問い合わせにとって問題となるのは合計コストの方ですが、EXISTS副問い合わせなどのコンテキストでは、プランナは合計コストが最も短くなるプランよりも、スタートアップコストが最も短くなるプランを選びます（エクゼキュータは行を1つ取得した後に停止するからです）。また、LIMIT句を使って問い合わせが返す行数を制限する場合、プランナは実際にはどの計画が一番低コストになるかを概算するため、全体を処理した場合のコストの間で適切な補間を行います。

ANALYZEオプションを付けると、計画を作るだけでなく、文が実際に実行されます。この場合は、各計画ノードで費された総経過時間（ミリ秒単位）と実際に返された全行数など、実際の実行時の統計情報が追加表示されます。プランナの推測と実際の値が近いかどうかを確認するために、このオプションは有用です。

> **重要**
>
> ANALYZEを使用した場合は、文が実際に実行されることを忘れないでください。EXPLAINはSELECTが返す出力をまったく表示しませんが、文に伴う副作用は通常通り発生します。INSERT、UPDATE、

DELETE、CREATE TABLE AS、EXECUTE文に対して、データに影響を与えないようにEXPLAIN ANALYZEを実行したい場合は、以下の方法を使用してください。

```
BEGIN;
EXPLAIN ANALYZE ...;
ROLLBACK;
```

ANALYZEおよびVERBOSEオプションのみが、この順序でのみ、オプションリストを括弧で括ることなく、指定可能です。PostgreSQL 9.0より前までは、括弧がない構文のみがサポートされていました。すべての新しいオプションは括弧付き構文のみでサポートされることを想定しています。

パラメータ

ANALYZE

コマンドを実行し、実際の実行時間やその他の統計情報を表示します。このパラメータのデフォルトはFALSEです。

VERBOSE

計画についての追加情報を出力します。具体的には、計画ツリー、スキーマ修飾テーブル、関数名内の各ノードに対して出力列リストを含めます。常に範囲テーブルの別名を付けて式内の変数を命名し、また常に統計情報が表示される各トリガの名前を出力します。このパラメータのデフォルトはFALSEです。

COSTS

各計画ノードの推定起動コストと総コスト、さらに推定行数と各行の推定幅に関する情報を含めます。このパラメータのデフォルトはTRUEです。

SETTINGS

設定パラメータに関する情報を含めます。特に、組み込みのデフォルト値と異なる値で問い合わせ計画に影響するオプションを含めます。このパラメータのデフォルトはFALSEです。

BUFFERS

バッファの使用状況に関する情報を含めます。具体的には、共有ブロックのヒット数、読み取り数、ダーティブロック数、書き出し数、ローカルブロックのヒット数、ダーティブロック数、読み取り数、書き出し数、一時ブロックの読み取り数、書き出し数、そして、track_io_timingが有効にされていればデータファイルブロックの読み取り、書き出しに掛かった時間(ミリ秒単位)が含められます。ヒットとは、必要な時にキャッシュ内にそのブロックが見つかったため読み取りが避けられたことを意味します。共有ブロックには、通常のテーブルとインデックスからのデータが含まれます。ローカルブロックには、一時テーブルとそのインデックスからのデータが含まれます。一時ブロックには、ソートやハッシュ、マテリアライズ計画ノードなどで使用される短期間有効なデータが含まれます。ダーティブロック数は、これまでは変更がなかったがその問い合わせによって変更されたブロックの数を示します。書き出しブロック数は、問い合わせ処理の間にバックエンドにより、ダーティ状態だったブロックの内キャッシュから追い出されたブロックの数を示します。上位レベルのノードで表示されるブロック数には、その子ノードすべて

で使用されるブロックが含まれます。テキスト形式では、非ゼロの値のみが出力されます。デフォルトはFALSEです。

WAL

WALレコード生成に関する情報を含めます。具体的には、レコード数、ページ全体のイメージ(fpi)の数、生成されたWALのバイト量が含められます。テキスト形式では、非ゼロの値のみが出力されます。このパラメータはANALYZEパラメータも有効である場合にのみ使用できます。デフォルトはFALSEです。

TIMING

実際のスタートアップ時間とノードで費やされた時間が追加表示されます。一部のシステムでは、システムクロックを何度も読み取るオーバーヘッドのため問い合わせがかなり低速になる可能性があります。このため、実際の時間ではなく実際の行数のみが必要であるのであれば、このパラメータはFALSEに設定する方が有益でしょう。文全体の実行時間は、このオプションによってノードレベルの時間計測が無効であった場合であっても、常に計測されます。このパラメータはANALYZEパラメータも有効である場合にのみ使用することができます。デフォルトはTRUEです。

SUMMARY

要約情報(例えば、時間の情報の合計)を問い合わせ計画の後に出力します。要約情報はANALYZEが使われるときはデフォルトで含まれ、それ以外の場合はデフォルトでは含まれませんが、このオプションを使えば有効にできます。EXPLAIN EXECUTEの計画時間には、計画をキャッシュから取得するのに要する時間、および必要なら再計画するのに要する時間も含まれます。

FORMAT

出力形式を指定します。TEXT、XML、JSON、YAMLを指定可能です。TEXT以外の出力にはTEXT出力と同じ情報が含まれていますが、プログラムによる解析がより容易になります。このパラメータのデフォルトはTEXTです。

boolean

選択したオプションを有効とするか無効とするかを指定します。オプションを有効にするためにはTRUE、ON、1のいずれかを書きます。無効にするにはFALSE、OFF、0のいずれかを書きます。boolean値は省略可能です。省略時はTRUEとみなされます。

statement

実行計画の表示対象となる、SELECT、INSERT、UPDATE、DELETE、VALUES、EXECUTE、DECLARE、CREATE TABLE AS、CREATE MATERIALIZED VIEW ASのいずれかの文です。

出力

コマンドの結果は、statementに対して選択された計画をテキストで説明します。オプションで、実行時の統計情報で注釈が付けられます。14.1では出力される情報について説明します。

注釈

PostgreSQL問い合わせプランナが十分な情報を使って問合せを最適化できるようにするには、問い合わせ内で使用されるすべてのテーブルに関するpg_statisticのデータを最新状態にしなければなりません。通

常自動バキュームデーモンにより自動的に処理されます。しかし最近その内容が大きく変更されたテーブルでは、自動バキュームがその変更に追いつくまで待つのではなく、手作業によるANALYZEを実行する必要があるかもしれません。

実行計画内の各ノードの実行時コストを測定するために、現在のEXPLAIN ANALYZE実装は、問い合わせ実行に対し、情報収集のためのオーバーヘッドを加えます。この結果、ある問い合わせについてのEXPLAIN ANALYZE実行が、普通に問い合わせを実行した場合より非常に時間がかかることがあります。このオーバーヘッドの量は問い合わせの性質と使用するプラットフォームに依存します。実行の間非常に短い時間を必要とする計画ノードに関して、時刻を得るためのシステムコールの操作が相対的に低速なプラットフォーム上で最悪な場合が発生します。

例

integer列を1つ持ち、10000行が存在するテーブルに対して、単純な問い合わせを行った場合の問い合わせ計画を表示します。

```
EXPLAIN SELECT * FROM foo;

                      QUERY PLAN
-----------------------------------------------------------
 Seq Scan on foo  (cost=0.00..155.00 rows=10000 width=4)
(1 row)
```

以下は同じ問い合わせをJSON出力形式で出力したものです。

```
EXPLAIN (FORMAT JSON) SELECT * FROM foo;
           QUERY PLAN
--------------------------------
 [                             +
   {                           +
     "Plan": {                 +
       "Node Type": "Seq Scan",+
       "Relation Name": "foo", +
       "Alias": "foo",         +
       "Startup Cost": 0.00,   +
       "Total Cost": 155.00,   +
       "Plan Rows": 10000,     +
       "Plan Width": 4         +
     }                         +
   }                           +
 ]
(1 row)
```

インデックスが存在し、問い合わせのWHERE条件でインデックスを利用できる場合、EXPLAINは異なる計画を表示する可能性があります。

```
EXPLAIN SELECT * FROM foo WHERE i = 4;

                          QUERY PLAN
-------------------------------------------------------------
 Index Scan using fi on foo  (cost=0.00..5.98 rows=1 width=4)
   Index Cond: (i = 4)
(2 rows)
```

以下は同じ問い合わせをYAML形式で表したものです。

```
EXPLAIN (FORMAT YAML) SELECT * FROM foo WHERE i='4';
          QUERY PLAN
------------------------------
 - Plan:                      +
     Node Type: "Index Scan"  +
     Scan Direction: "Forward"+
     Index Name: "fi"         +
     Relation Name: "foo"     +
     Alias: "foo"             +
     Startup Cost: 0.00       +
     Total Cost: 5.98         +
     Plan Rows: 1             +
     Plan Width: 4            +
     Index Cond: "(i = 4)"
(1 row)
```

読者への演習としてXML形式については記載しません。

以下は同じ計画ですが、コスト推定値を出力しません。

```
EXPLAIN (COSTS FALSE) SELECT * FROM foo WHERE i = 4;

        QUERY PLAN
---------------------------
 Index Scan using fi on foo
   Index Cond: (i = 4)
(2 rows)
```

次に、集約関数を使用した問い合わせに対する問い合わせ計画の例を示します。

```
EXPLAIN SELECT sum(i) FROM foo WHERE i < 10;

                             QUERY PLAN
---------------------------------------------------------------------
 Aggregate  (cost=23.93..23.93 rows=1 width=4)
   ->  Index Scan using fi on foo  (cost=0.00..23.92 rows=6 width=4)
```

```
         Index Cond: (i < 10)
(3 rows)
```

以下は、EXPLAIN EXECUTEによってプリペアド文に対する実行計画を表示する例です。

```
PREPARE query(int, int) AS SELECT sum(bar) FROM test
    WHERE id > $1 AND id < $2
    GROUP BY foo;

EXPLAIN ANALYZE EXECUTE query(100, 200);

                                                       QUERY PLAN

-------------------------------------------------------------------
-----------------------------------------------------
 HashAggregate  (cost=9.54..9.54 rows=1 width=8) (actual time=0.156..0.161 rows=11 loops=1)
   Group Key: foo
   ->  Index Scan using test_pkey on test  (cost=0.29..9.29 rows=50 width=8) (actual
 time=0.039..0.091 rows=99 loops=1)
         Index Cond: ((id > $1) AND (id < $2))
 Planning time: 0.197 ms
 Execution time: 0.225 ms
(6 rows)
```

もちろん、ここで示した具体的な数値は対象とするテーブルの実際の中身によって変わります。また、この数値や選択された問い合わせ戦略は、プランナの改良のため、PostgreSQLのリリース間で異なる可能性がありますので注意してください。さらに、ANALYZEコマンドは、データの統計情報を推定する際にランダムなサンプリングを行うため、実際のテーブル内の分布が変わっていなくても、新たにANALYZEを実行すると推定コストが変わることがあります。

互換性

標準SQLではEXPLAIN文は定義されていません。

関連項目

ANALYZE

FETCH

FETCH — カーソルを使用して問い合わせから行を取り出す

概要

```
FETCH [ direction [ FROM | IN ] ] cursor_name


ここでdirectionは空にするか、
      次のいずれかを指定します。

    NEXT
    PRIOR
    FIRST
    LAST
    ABSOLUTE count
    RELATIVE count
    count
    ALL
    FORWARD
    FORWARD count
    FORWARD ALL
    BACKWARD
    BACKWARD count
    BACKWARD ALL
```

説明

FETCHは事前に作成されたカーソルを使用して行を取り出します。

カーソルはそれぞれ位置情報を持っており、FETCHはこれを使用します。カーソルの位置は、問い合わせの結果の先頭行の前、結果内の任意の特定の行、結果の最終行の後のいずれにもなります。カーソルの生成時は、カーソル位置は先頭行の前にあります。行を取り出した後は、カーソル位置は最後に取り出した行にあります。FETCHが利用可能な行の終わりを過ぎると、カーソル位置は最終行の後にあります（後方抽出の場合は先頭行の前になります）。FETCH ALLもしくはFETCH BACKWARD ALLでは、カーソルの位置は常に最終行の後か、先頭行の前になります。

NEXT、PRIOR、FIRST、LAST、ABSOLUTE、RELATIVE構文では、カーソルを適切に移動した後、行を1つ取り出します。行が存在しない場合、空の結果が返され、カーソルは先頭行の前か最終行の後に適切に位置づけられます。

FORWARDおよびBACKWARDを使用した構文では、指定数の行を前方もしくは後方方向に取り出します。この時、最後に取り出した行がカーソル位置となります（countが利用可能な行数を超えた場合は、全行の後/前になります）。

RELATIVE 0、FORWARD 0、およびBACKWARD 0は全て、カーソルを移動することなく現在の行を取り出します。つまり、一番最後に取り出した行を再度取り出すことになります。カーソルが先頭行の前や最終行の後になければ、これらのコマンドは成功します。先頭行の前や最終行の後にあれば、行は返されません。

> **注記**
>
> このマニュアルページではSQLコマンドレベルでのカーソルの使用方法について説明しています。PL/pgSQL内でカーソルを使用する場合は、規則が異なりますので、42.7.3を参照してください。

パラメータ

direction

directionは、取り出す方向と取り出す行数を定義します。以下のいずれかを指定できます。

NEXT

次の行を取り出します。これは、directionが省略された時のデフォルトです。

PRIOR

1つ前の行を取り出します。

FIRST

問い合わせの先頭行を取り出します（ABSOLUTE 1と同じです）。

LAST

問い合わせの最終行を取り出します（ABSOLUTE -1と同じです）。

ABSOLUTE count

問い合わせのcount番目の行を取り出します。countが負ならば、終わりからabs(count)番目の行を取り出します。countが範囲外の場合、カーソル位置は先頭行の前か最終行の後になります。特に、ABSOLUTE 0と指定すると、先頭行の前になります。

RELATIVE count

カーソルの現在位置からcount番目の行を取り出します。countが負の場合、現在位置よりabs(count)行分前の行を取り出します。RELATIVE 0と指定すると、現在の行があれば、その行を再度取り出します。

count

次のcount行を取り出します（FORWARD countと同じです）。

ALL

残っている行を全て取り出します（FORWARD ALLと同じです）。

FORWARD

1つ次の行を取り出します（NEXTと同じです）。

FORWARD count

次のcount行分の行を取り出します。FORWARD 0と指定すると、現在の行を再度取り出します。

FORWARD ALL

残っている行を全て取り出します。

BACKWARD

1つ前の行を取り出します（PRIORと同じです）。

BACKWARD count

前のcount行分の行を（逆方向に走査して）取り出します。BACKWARD 0と指定すると、現在の行を再度取り出します。

BACKWARD ALL

現在位置より前の行を（逆方向に走査して）全て取り出します。

count

countは、整数定数で、符号を付けることができ、取り出す位置や行数を決定します。
FORWARDとBACKWARDにおいて、countに負の値を指定するのは、FORWARDとBACKWARDの意味を入れ替えるのと同等です。

cursor_name

開いているカーソルの名前を指定します。

出力

正常に終了すると、FETCHコマンドは以下の形式のコマンドタグを返します。

```
FETCH count
```

countは取り出した行数です（0の可能性もあります）。psqlでは取り出した行数を別途表示するため、このコマンドタグは実際には表示されないことに注意してください。

注釈

FETCHコマンドとして、FETCH NEXTもしくは正のcountのFETCH FORWARD以外を使用する場合、カーソルをSCROLLオプション付きで宣言しなければなりません。単純な問い合わせでは、PostgreSQLでは、カーソル

がSCROLL付きで宣言されていなくても後方向の取り出しを行うことができることがありますが、この動作に依存すべきではありません。カーソルがNO SCROLL付きで宣言された場合は、後方向の取り出しを行うことができません。

ABSOLUTEによる取り出しは、相対的な指定による指定行への移動に比べて高速ではありません。内部的な実装では、必ず中間の行を全て経由しているからです。絶対指定で負の値を指定した場合、速度はさらに悪化します。まず、最終行を見つけるために最後まで問い合わせを読み取って、その後に最終行から後方に移動するためです。ただし、(FETCH ABSOLUTE 0を使用して)問い合わせの先頭へ戻るのは高速です。

DECLAREを使用してカーソルを定義します。データを取り出さずにカーソル位置を変更する場合はMOVEを使用してください。

例

次の例では、カーソルを使用してテーブル内を走査しています。

```
BEGIN WORK;


-- カーソルを設定します。
DECLARE liahona SCROLL CURSOR FOR SELECT * FROM films;


-- カーソルliahonaから最初の5行を取り出します。
FETCH FORWARD 5 FROM liahona;

 code  |          title          | did | date_prod  |   kind   |  len
-------+-------------------------+-----+------------+----------+-------
 BL101 | The Third Man           | 101 | 1949-12-23 | Drama    | 01:44
 BL102 | The African Queen       | 101 | 1951-08-11 | Romantic | 01:43
 JL201 | Une Femme est une Femme | 102 | 1961-03-12 | Romantic | 01:25
 P_301 | Vertigo                 | 103 | 1958-11-14 | Action   | 02:08
 P_302 | Becket                  | 103 | 1964-02-03 | Drama    | 02:28


-- 1つ前の行を取り出します。
FETCH PRIOR FROM liahona;

 code  |  title  | did | date_prod  |  kind  |  len
-------+---------+-----+------------+--------+-------
 P_301 | Vertigo | 103 | 1958-11-14 | Action | 02:08


-- カーソルを閉じ、トランザクションを終了します。
CLOSE liahona;
COMMIT WORK;
```

互換性

標準SQLでは、埋め込みSQLにおけるFETCHのみが定義されています。上記で説明した各種のFETCHは、ホスト変数への代入ではなく、SELECTの結果であるかのようにデータを返します。この点を除き、FETCHは完全に標準SQLと上位互換性を持ちます。

FORWARDとBACKWARDを持つFETCHの形式や、暗黙的なFORWARDを持つFETCH countとFETCH ALLはPostgreSQLの拡張です。

標準SQLでは、カーソル名の前に付けられるのはFROMのみです。INを使用するオプション、または、どちらも省略することはPostgreSQLの拡張です。

関連項目

CLOSE, DECLARE, MOVE

GRANT

GRANT — アクセス権限を定義する

概要

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }
    [, ...] | ALL [ PRIVILEGES ] }
    ON { [ TABLE ] table_name [, ...]
         | ALL TABLES IN SCHEMA schema_name [, ...] }
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { { SELECT | INSERT | UPDATE | REFERENCES } ( column_name [, ...] )
    [, ...] | ALL [ PRIVILEGES ] ( column_name [, ...] ) }
    ON [ TABLE ] table_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { { USAGE | SELECT | UPDATE }
    [, ...] | ALL [ PRIVILEGES ] }
    ON { SEQUENCE sequence_name [, ...]
         | ALL SEQUENCES IN SCHEMA schema_name [, ...] }
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { { CREATE | CONNECT | TEMPORARY | TEMP } [, ...] | ALL [ PRIVILEGES ] }
    ON DATABASE database_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON DOMAIN domain_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON FOREIGN DATA WRAPPER fdw_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON FOREIGN SERVER server_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { EXECUTE | ALL [ PRIVILEGES ] }
    ON { { FUNCTION | PROCEDURE | ROUTINE } routine_name [ ( [ [ argmode ] [ arg_name ] arg_type
 [, ...] ] ) ] [, ...]
         | ALL { FUNCTIONS | PROCEDURES | ROUTINES } IN SCHEMA schema_name [, ...] }
    TO role_specification [, ...] [ WITH GRANT OPTION ]
```

```
GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON LANGUAGE lang_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { { SELECT | UPDATE } [, ...] | ALL [ PRIVILEGES ] }
    ON LARGE OBJECT loid [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { { CREATE | USAGE } [, ...] | ALL [ PRIVILEGES ] }
    ON SCHEMA schema_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { CREATE | ALL [ PRIVILEGES ] }
    ON TABLESPACE tablespace_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON TYPE type_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT role_name [, ...] TO role_specification [, ...]
    [ WITH ADMIN OPTION ]
    [ GRANTED BY role_specification ]


ここでrole_specificationは以下の通りです。

    [ GROUP ] role_name
  | PUBLIC
  | CURRENT_USER
  | SESSION_USER
```

説明

GRANTには基本的に2つの種類があります。1つはデータベースオブジェクト（テーブル、列、ビュー、外部テーブル、シーケンス、データベース、外部データラッパ、外部サーバ、関数、プロシージャ、手続き言語、スキーマ、テーブル空間）に対する権限の付与、もう1つはロール内のメンバ資格の付与です。これらの種類は多くの点で似ていますが、説明は別々に行わなければならない程違いがあります。

データベースオブジェクトに対するGRANT

この種類のGRANTコマンドはデータベースオブジェクトの特定の権限を1つ以上のロールに付与します。既に権限が他のロールに付与されている場合でも、追加として付与されます。

PUBLICキーワードは、今後作成されるロールを含む、全てのロールへの許可を示します。PUBLICは、全てのロールを常に含む、暗黙的に定義されたグループと考えることができます。個々のロールは全て、ロールに直接許可された権限、ロールが現在属しているロールに許可された権限、そして、PUBLICに許可された権限を合わせた権限を持っています。

WITH GRANT OPTIONが指定されると、権限の受領者は、その後、他にその権限を与えることができます。グラントオプションがない場合、受領者はこれを行うことができません。グラントオプションはPUBLICには与えることができません。

所有者（通常はオブジェクトを作成したユーザ）はデフォルトで全ての権限を保持しているため、オブジェクトの所有者に権限を許可する必要はありません（ただし、オブジェクトの作成者が、安全性のために自らの権限を取り消すことは可能です）。

オブジェクトを削除する権限や何らかの方法でオブジェクトの定義を変更する権限は、付与可能な権限として扱われません。これらの権限は、所有者固有のものであり、許可したり取り消したりすることはできません。（しかし、オブジェクトを所有するロール内のメンバ関係を付与したり取り消すことで、同等な効果を得ることができます。後で説明します。）所有者は、オブジェクトに対する全てのグラントオプションも暗黙的に保持しています。

設定可能な権限は以下のものです。

SELECT
INSERT
UPDATE
DELETE
TRUNCATE
REFERENCES
TRIGGER
CREATE
CONNECT
TEMPORARY
EXECUTE
USAGE

権限の特定の種類です。5.7で定義されています。

TEMP

TEMPORARYの別の綴り方です。

ALL PRIVILEGES

対象の型に対して利用可能な全ての権限を一度に付与します。PRIVILEGESキーワードはPostgreSQLでは省略可能ですが、厳密なSQLでは必須です。

FUNCTION構文は通常の関数、集約関数、ウィンドウ関数に対して有効ですが、プロシージャには有効ではありません。プロシージャにはPROCEDUREを使ってください。あるいは、関数、集約関数、プロシージャを参照するのに、その正確な種類に関係なくROUTINEを使ってください。

1つまたは複数のスキーマ内で同じ型のオブジェクトすべてに対する権限を付与するオプションもあります。この機能は現在、テーブル、シーケンス、関数、プロシージャだけをサポートしています。ALL TABLESは、特

定の対象のGRANTコマンドと同様に、ビューや外部テーブルにも影響します。ALL FUNCTIONSは、集約関数やウィンドウ関数にも影響しますが、プロシージャには影響しません。ここでも、特定の対象のGRANTコマンドと同様です。プロシージャを含めるにはALL ROUTINESを使ってください。

ロールに対するGRANT

この種類のGRANTコマンドは、ロール内のメンバ資格を1つ以上の他のロールに付与します。これによりロールに付与された権限が各メンバに与えられるので、ロール内のメンバ資格は重要です。

WITH ADMIN OPTIONが指定された場合、メンバはロール内のメンバ資格を他に付与することができるようになります。また同様にロール内のメンバ資格を取り消すこともできるようになります。アドミンオプションがないと、一般ユーザは他への権限の付与や取り消しを行うことができません。ロールはそれ自体についてのWITH ADMIN OPTIONを保持しているとはみなされませんが、データベースセッションのユーザがロールにマッチする場合について、ロール内のメンバ資格を付与あるいは取り消しを行うことができます。データベーススーパーユーザはすべてのロール内のメンバ資格を誰にでも付与したり、取り消したりすることができます。CREATEROLE権限を持つロールは、スーパーユーザロール以外の任意のロール内のメンバ資格の付与、取り上げが可能です。

GRANTED BYが指定された場合、付与は指定されたロールにより行なわれたと記録されます。コマンドを実行しているのと同じロールの名前を指定する場合を除き、データベーススーパーユーザだけがこのオプションを利用できます。

権限の場合と異なり、ロール内のメンバ資格をPUBLICに付与することはできません。また、このコマンド構文では、role_specificationで無意味なGROUPという単語を受け付けないことに注意してください。

注釈

アクセス権限を取り消すには、REVOKEコマンドが使用されます。

PostgreSQL 8.1から、ユーザとグループという概念は、ロールと呼ばれる1種類の実体に統合されました。そのため、付与者がユーザかグループかどうかを識別するためにGROUPキーワードを使用する必要はなくなりました。このコマンドではまだGROUPを使うことはできますが、何の意味もありません。

ユーザは特定の列あるいはテーブル全体に対する権限を持つ場合にSELECT、INSERTなどを実行することができます。テーブルレベルの権限を付与してからある列に対する権限を取り消しても、望むことは実現できません。テーブルレベルの権限は列レベルの操作による影響を受けないからです。

オブジェクトの所有者でもなく、そのオブジェクトに何の権限も持たないユーザが、そのオブジェクトの権限をGRANTしようとしても、コマンドの実行は直ちに失敗します。何らかの権限を持っている限り、コマンドの実行は進行しますが、与えることのできる権限は、そのユーザがグラントオプションを持つ権限のみです。グラントオプションを持っていない場合、GRANT ALL PRIVILEGES構文は警告メッセージを発します。一方、その他の構文では、コマンドで名前を指定した権限に関するグラントオプションを持っていない場合に警告メッセージを発します（原理上、ここまでの説明はオブジェクトの所有者に対しても当てはまりますが、所有者は常に全てのグラントオプションを保持しているものとして扱われるため、こうした状態は決して起こりません）。

データベーススーパーユーザは、オブジェクトに関する権限設定に関係なく、全てのオブジェクトにアクセスできることには注意しなければなりません。スーパーユーザが持つ権限は、Unixシステムにおけるroot権限

に似ています。rootと同様、どうしても必要という場合以外は、スーパーユーザとして操作を行わないのが賢明です。

スーパーユーザがGRANTやREVOKEの発行を選択した場合、それらのコマンドは対象とするオブジェクトの所有者が発行したかのように実行されます。特に、こうしたコマンドで与えられる権限は、オブジェクトの所有者によって与えられたものとして表されます。(ロールのメンバ資格では、メンバ資格は含まれるロール自身が与えたものとして表されます。)

GRANTおよびREVOKEは、対象のオブジェクトの所有者以外のロールによって実行することもできますが、オブジェクトを所有するロールのメンバであるか、そのオブジェクトに対しWITH GRANT OPTION権限を持つロールのメンバでなければなりません。この場合、その権限は、そのオブジェクトの実際の所有者ロールまたはWITH GRANT OPTION権限を持つロールによって付与されたものとして記録されます。例えば、テーブルt1がロールg1によって所有され、ロールu1がロールg1のメンバであるとします。この場合、u1はt1に関する権限をu2に付与できます。しかし、これらの権限はg1によって直接付与されたものとして現れます。後でロールg1の他のメンバがこの権限を取り消すことができます。

GRANTを実行したロールが、ロールの持つ複数メンバ資格の経路を通して間接的に必要な権限を持つ場合、どのロールが権限を付与したロールとして記録されるかについては指定されません。こうした場合、SET ROLEを使用して、GRANTを行わせたい特定のロールになることを推奨します。

テーブルへの権限付与によって、SERIAL列によって関連付けされたシーケンスを含め、そのテーブルで使用されるシーケンスへの権限の拡張は自動的に行われません。シーケンスへの権限は別途設定しなければなりません。

特定の権限の種類に関するより詳しい情報や、対象の権限を調査する方法は5.7を参照してください。

例

テーブルfilmsにデータを追加する権限を全てのユーザに与えます。

```
GRANT INSERT ON films TO PUBLIC;
```

ビューkindsにおける利用可能な全ての権限を、ユーザmanuelに与えます。

```
GRANT ALL PRIVILEGES ON kinds TO manuel;
```

上のコマンドをスーパーユーザやkindsの所有者が実行した場合は、全ての権限が付与されますが、他のユーザが実行した場合は、そのユーザがグラントオプションを持つ権限のみが付与されることに注意してください。

ロールadmins内のメンバ資格をユーザjoeに与えます。

```
GRANT admins TO joe;
```

互換性

標準SQLでは、ALL PRIVILEGES内のPRIVILEGESキーワードは必須です。標準SQLでは、1つのコマンドで複数のオブジェクトに権限を設定することはサポートしていません。

PostgreSQLでは、オブジェクトの所有者は、自身が持つ権限を取り消すことができます。例えば、テーブル所有者は自身のINSERT、UPDATE、DELETE、TRUNCATE権限を取り消すことで、自分にとってそのテーブルが読み取り専用になるよう変更することができます。これは、標準SQLでは不可能です。PostgreSQLでは、所有者の権限を、所有者自身により与えられたものとして扱っているため、同様に所有者自身で権限を取り消すことができるようになっています。標準SQLでは、所有者の権限は仮想的なエンティティ「_SYSTEM」によって与えられたものとして扱っています。所有者は「_SYSTEM」ではないため、その権限を取り消すことができません。

標準SQLにしたがうと、グラントオプションはPUBLICに対して与えることができます。PostgreSQLではグラントオプションはロールに対して与えることのみをサポートしています。

標準SQLではGRANTED BYオプションはGRANTのすべての構文で使えます。PostgreSQLでは、ロールのメンバ資格を与える場合、それもスーパーユーザが簡単ではない方法で使う場合のみをサポートしています。

標準SQLでは、文字セット、照合順序、翻訳といったその他の種類のオブジェクトに対して、USAGE権限を付与することができます。

標準SQLでは、シーケンスはUSAGE権限のみを持ちます。これはPostgreSQLにおけるnextval関数と等価なNEXT VALUE FOR式の使用を制御するものです。シーケンスに関するSELECT権限とUPDATE権限はPostgreSQLの拡張です。シーケンスに関するUSAGE権限がcurrval関数にも適用される点もPostgreSQLの拡張です(この関数自体が拡張です)。

データベース、テーブル空間、スキーマ、言語についての権限はPostgreSQLの拡張です。

関連項目

REVOKE, ALTER DEFAULT PRIVILEGES

IMPORT FOREIGN SCHEMA

IMPORT FOREIGN SCHEMA — 外部サーバからテーブル定義をインポートする

概要

```
IMPORT FOREIGN SCHEMA remote_schema
    [ { LIMIT TO | EXCEPT } ( table_name [, ...] ) ]
    FROM SERVER server_name
    INTO local_schema
    [ OPTIONS ( option 'value' [, ... ] ) ]
```

説明

IMPORT FOREIGN SCHEMAは外部サーバ上に存在するテーブルを表す外部テーブルを作成します。新しい外部テーブルは、コマンドを実行するユーザに所有され、リモートのテーブルにマッチする正しい列定義とオプションで作成されます。

デフォルトでは、外部サーバ上の特定のスキーマ内に存在するすべてのテーブルとビューがインポートされます。オプションで、インポートするテーブルを指定した部分集合に制限したり、特定のテーブルを除外することができます。新しい外部テーブルは、すべてターゲットとなるスキーマ内に作成され、そのスキーマは既存である必要があります。

IMPORT FOREIGN SCHEMAを使用するには、外部サーバのUSAGE権限、およびターゲットとなるスキーマのCREATE権限が必要です。

パラメータ

remote_schema

インポート元となるリモートのスキーマです。リモートのスキーマの具体的な意味は、使用する外部データラッパに依存します。

LIMIT TO (table_name [, ...])

指定したテーブル名の1つにマッチする外部テーブルだけをインポートします。外部スキーマ内に存在する他のテーブルは無視されます。

EXCEPT (table_name [, ...])

指定した外部テーブルをインポートから除外します。ここに列挙したものを除き、外部スキーマ内に存在するすべてのテーブルがインポートされます。

server_name

インポート元となる外部サーバです。

local_schema

インポートされた外部テーブルが作成されるスキーマです。

OPTIONS (option 'value' [, ...])

インポート時に使用されるオプションです。使用できるオプションの名前と値は、各外部データラッパに依存します。

例

サーバfilm_server上のリモートのスキーマforeign_filmsからテーブルの定義をインポートし、ローカルのスキーマfilms内に外部テーブルを作成します。

```
IMPORT FOREIGN SCHEMA foreign_films
    FROM SERVER film_server INTO films;
```

上と同様ですが、2つのテーブルactorsとdirectorsだけを(それらが存在するなら)インポートします。

```
IMPORT FOREIGN SCHEMA foreign_films LIMIT TO (actors, directors)
    FROM SERVER film_server INTO films;
```

互換性

IMPORT FOREIGN SCHEMAコマンドは、OPTIONS句がPostgreSQLの拡張であるという点を除き、標準SQLに準拠しています。

関連項目

CREATE FOREIGN TABLE, CREATE SERVER

INSERT

INSERT — テーブルに新しい行を作成する

概要

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
INSERT INTO table_name [ AS alias ] [ ( column_name [, ...] ) ]
    [ OVERRIDING { SYSTEM | USER } VALUE ]
    { DEFAULT VALUES | VALUES ( { expression | DEFAULT } [, ...] ) [, ...] | query }
    [ ON CONFLICT [ conflict_target ] conflict_action ]
    [ RETURNING * | output_expression [ [ AS ] output_name ] [, ...] ]


ここでconflict_targetは以下のいずれかです。

    ( { index_column_name | ( index_expression ) } [ COLLATE collation ] [ opclass ] [, ...] )
 [ WHERE index_predicate ]
    ON CONSTRAINT constraint_name


またconflict_actionは以下のいずれかです。

    DO NOTHING
    DO UPDATE SET { column_name = { expression | DEFAULT } |
                    ( column_name [, ...] ) = [ ROW ] ( { expression | DEFAULT } [, ...] ) |
                    ( column_name [, ...] ) = ( sub-SELECT )
                  } [, ...]
              [ WHERE condition ]
```

説明

INSERTはテーブルに新しい行を挿入します。値式を使用して行(複数可)を挿入すること、および、問い合わせの結果を使って0行以上の行を挿入することができます。

対象の列名はどのような順番でも指定できます。列名リストが指定されなかった場合は、テーブル内の全ての列を宣言時の順番に並べたものがデフォルトとして使われます。また、VALUES句やqueryでN列のみが与えられた場合は、先頭のN列の名前が指定されたものとみなされます。VALUES句やqueryで提供される値は、明示的または暗黙的な列リストと左から右への順で関連付けられます。

明示的または暗黙的な列リストにない各列にはデフォルト値(デフォルト値が宣言されていればその値、未宣言ならばNULL)が挿入されます。

各列の式が正しいデータ型でない場合は、自動的に型の変換が行われます。

ON CONFLICTは一意制約または排他制約について、違反のエラーを発生させるのに代わる動作を指定するのに使うことができます。(以下のON CONFLICT Clauseを参照してください。)

RETURNING句を指定すると、INSERTは実際に挿入された(あるいはON CONFLICT DO UPDATE句によって更新された)各行に基づいて計算された値を返すようになります。これは、通番のシーケンス番号など、デフォルトで与えられた値を取り出す時に主に便利です。しかし、そのテーブルの列を使用した任意の式を指定することができます。RETURNINGリストの構文はSELECTの出力リストと同一です。挿入または更新に成功した行だけが返されます。例えば、行がロックされていて、ON CONFLICT DO UPDATE ... WHERE句の conditionが満たされなかったために更新されなかった行は返されません。

テーブルに行を追加するには、そのテーブルに対してINSERT権限を持っている必要があります。ON CONFLICT DO UPDATEがある場合は、テーブルのUPDATE権限も必要です。

列リストを指定する場合は、列挙された列に対するINSERT権限のみが必要です。同様に、ON CONFLICT DO UPDATEが指定されている場合、更新対象として列挙されている列についてのみ、UPDATE権限が必要です。しかし、ON CONFLICT DO UPDATEはまた、その式あるいはconditionで読み取られるすべての列についてのSELECT権限も必要です。

RETURNING句を使用するには、RETURNINGで使用するすべての列に対するSELECT権限が必要です。queryを使用して問い合わせ結果を元に行を挿入する場合は当然ながら、その問い合わせ内で使われる全てのテーブルまたは列に対してSELECT権限を持っている必要があります。

パラメータ

挿入

この節では新しい行を挿入するときにのみ使われるパラメータについて説明します。ON CONFLICT句においてのみ使われるパラメータについては、別に説明します。

with_query

WITH句により、INSERT問い合わせ内で名前により参照することができる1つ以上の副問い合わせを指定することができます。詳しくは7.8とSELECTを参照してください。

query(SELECT文)でもまた、WITH句を含めることができます。こうした場合、with_queryの集合との両方をquery内で参照することができます。しかし、第二の問い合わせがより近くにネストされているため優先します。

table_name

既存のテーブルの名前です(スキーマ修飾名も可)。

alias

table_nameの代替名です。aliasを指定すると、テーブルの実際の名前が完全に隠されます。これは、excludedという名前のテーブルをON CONFLICT DO UPDATEが対象にしている場合、これを指定しなければ、それが挿入で処理される行を表現する特別なテーブルの名前とみなされるため、特に有用となります。

column_name

table_nameで指名されたテーブル内の列名です。必要なら列名を副フィールドの名前や配列の添え字で修飾することができます。(複合型の列の一部のフィールドのみを挿入すると他のフィールドはNULLになります。) ON CONFLICT DO UPDATEで列を参照する場合、対象列の指定にテーブル名を含めてはいけません。例えば、INSERT INTO table_name ... ON CONFLICT DO UPDATE SET table_name.col = 1は無効です(これはUPDATEの一般的な動作に従います)。

OVERRIDING SYSTEM VALUE

この句が指定されると、IDENTITY列について指定された値がシーケンスが生成したデフォルト値に優先します。

GENERATED ALWAYSと定義されているIDENTITY列に対しては、OVERRIDING SYSTEM VALUEやOVERRIDING USER VALUEを指定せずに(DEFAULT以外の)明示的な値を挿入するのはエラーです。(GENERATED BY DEFAULTと定義されているIDENTITY列に対しては、OVERRIDING SYSTEM VALUEが通常の振る舞いであり、指定したとしても何もしませんが、PostgreSQLは拡張として許容します。)

OVERRIDING USER VALUE

この句が指定されると、IDENTITY列について指定された値はすべて無視されて、シーケンスが生成したデフォルト値が適用されます。

この句は例えばテーブル間で値をコピーする時に有用です。INSERT INTO tbl2 OVERRIDING USER VALUE SELECT * FROM tbl1とすると、tbl1の列でtbl2のIDENTITY列でないものがすべてコピーされる一方、tbl2のIDENTITY列の値は、tbl2に紐付けられたシーケンスによって生成されます。

DEFAULT VALUES

各列に対してDEFAULTが明示的に指定されたかのように、すべての列にそれぞれのデフォルト値が設定されます。(OVERRIDINGはこの構文では使用できません。)

expression

対応する列に代入する式または値を指定します。

DEFAULT

対応する列にデフォルト値を設定します。IDENTITY列には関連付けられた列により新しく生成された値が書き込まれます。生成列に対して、これを指定することは許されていますが、単に生成式から列を計算するという普通の振る舞いを指定するだけです。

query

挿入する行を提供する問い合わせ(SELECT文)を指定します。構文の説明についてはSELECT文を参照してください。

output_expression

各行が挿入または更新された後、INSERTにより計算され、返される式です。この式にはtable_nameで指名されたテーブルの任意の列名を使用することができます。挿入または更新された行のすべての列を返す場合は*と記載してください。

output_name

返される列で使用される名前です。

ON CONFLICT句

オプションのON CONFLICT句では、一意制約や排他制約の違反について、エラーを発生させる代替となる動作を指定します。挿入しようとされた各行について、挿入の処理が進められるか、あるいは、conflict_targetにより指定された競合制約またはインデックスに違反した場合の代替のconflict_actionが実行されるか、のいずれかです。ON CONFLICT DO NOTHINGは代替の動作として、単に行の挿入をしなくなるだけです。ON CONFLICT DO UPDATEは代替の動作として、挿入されようとしていた行と競合する既存の行を更新します。

conflict_targetは一意インデックスの推定を実行することができます。推定を実行するとき、それは1つ以上のindex_column_name列、またはindex_expression式、あるいはその両方、およびオプションでindex_predicateから構成されます。table_nameの一意インデックスでconflict_targetで指定された列と式を(順序は関係なく)正確に含むものは、すべて競合解決インデックスとして推定されます(選ばれます)。index_predicateが指定されている場合は、推定のさらなる条件として、それは競合解決インデックスを満たさなければなりません。これは、部分インデックスでない一意インデックス(述語のない一意インデックス)は、それが他のすべての条件を満たすのであれば推定される(従ってON CONFLICTで使用される)ことを意味することに注意して下さい。推定に失敗した時は、エラーが発生します。

ON CONFLICT DO UPDATEはINSERTまたはUPDATEの原子的な結果を保証します。無関係のエラーが発生しなければ、多数の同時実行がある状況においてさえも、それら2つの結果のうちの1つになります。これは*UPSERT*、つまり「UPDATE or INSERT」としても知られています。

conflict_target

ON CONFLICTが競合解決インデックスを選ぶことで代替の動作をするときの競合を指定します。一意インデックスの推定を実行するか、あるいは制約を明示的に指定するかのいずれかです。ON CONFLICT DO NOTHINGではconflict_targetを指定するのはオプションです。省略すると、利用可能なすべての制約(および一意インデックス)との競合が処理されます。ON CONFLICT DO UPDATEではconflict_targetを指定しなければなりません。

conflict_action

conflict_actionではON CONFLICTの代替の動作を指定します。これはDO NOTHINGあるいはDO UPDATE句のいずれかをとることができ、後者では競合が発生した場合に実行されるUPDATEの動作の正確な詳細を記述します。ON CONFLICT DO UPDATEのSET句とWHEREは既存の行にテーブルの名前(または別名)を使ってアクセスでき、また挿入されようとしていた行には、特別なexcludedテーブルを使ってアクセスできます。excludedの列を読み取るときには、対象テーブルの対応する列のSELECT権限が必要です。

すべての行レベルのBEFORE INSERTトリガーの結果がexcludedの値に反映されることに注意して下さい。これらの結果として、行が挿入から除外されることになったかもしれないからです。

index_column_name

table_nameの列の名前です。競合解決インデックスを推定するのに使われます。CREATE INDEXの形式に従います。index_column_nameのSELECTが必要です。

index_expression

index_column_nameと似ていますが、インデックスの定義に現れるtable_nameの列の式(単純な列ではない)の推定に使われます。CREATE INDEXの形式に従います。index_expressionに現れるすべての列のSELECT権限が必要です。

collation

これを指定すると、推定時に、対応するindex_column_nameあるいはindex_expressionをマッチさせるときに、特定の照合順序を指定することになります。普通は照合順序は制約違反が発生するかどうかに関係しないので、通常は省略されます。CREATE INDEXの形式に従います。

opclass

これを指定すると、推定時に、対応するindex_column_nameあるいはindex_expressionをマッチさせるときに、特定の演算子クラスを指定することになります。等価の意味は、いずれにせよ、型の演算子クラスをまたがって同等であることが多いですし、また定義された一意インデックスは等価を適切に定義していると信頼すれば十分なので、通常はこれは省略されます。CREATE INDEXの形式に従います。

index_predicate

部分一意インデックスの推定を可能にします。述語を満たすすべてのインデックス(実際に部分インデックスである必要はありません)は推定可能になります。CREATE INDEXの形式に従います。index_predicateに現れるすべての列についてSELECT権限が必要です。

constraint_name

競合解決の制約を制約やインデックスの推定によるのではなく、明示的に名前で指定します。

condition

boolean型の値を返す式です。この式がtrueを返す行のみが更新されます。ただし、ON CONFLICT DO UPDATEの動作が行われるときは、すべての行がロックされます。conditionは最後に評価される、競合が更新対象候補として特定された後であることに注意して下さい。

排他制約はON CONFLICT DO UPDATEの競合解決としてはサポートされないことに注意して下さい。すべての場合について、NOT DEFERRABLEである制約と一意インデックスのみが競合解決としてサポートされます。

ON CONFLICT DO UPDATE句のあるINSERTは「決定論的な」文です。これは、そのコマンドが既存のどの行に対しても、2回以上影響を与えることが許されない、ということを意味します。これに反する状況が発生した時は、カーディナリティ違反のエラーが発生します。挿入されようとする行は、競合解決インデックスあるいは制約により制限される属性の観点で、複製されてはなりません。

パーティションテーブルに適用されたINSERTのON CONFLICT DO UPDATE句に対しては、その行を新しいパーティションに移動する必要のあるような競合する行のパーティションキーを更新することは現在サポートされていないことに注意してください。

ヒント

ON CONFLICT ON CONSTRAINT constraint_nameを使って制約を直接指定するより、一意インデックスの推定を使う方が望ましいことが多いです。背景にあるインデックスが、他のほぼ同等のインデッ

クスと重なり合う形で置換されるとき、推定は正しく動作し続けます。例えば、置換されるインデックスを削除する前にCREATE UNIQUE INDEX ... CONCURRENTLYを使う場合です。

出力

正常に終了すると、INSERTは以下のようなコマンドタグを返します。

```
INSERT oid count
```

countは挿入または更新された行数です。oidは常に0です(countが正確に1であり、対象のテーブルがWITH OIDSと宣言されていた場合、挿入された行にOIDが、そうでなければ0が割り当てられていましたが、WITH OIDSでテーブルを作成することは今はもうサポートされていません)。

INSERTコマンドがRETURNING句を持つ場合、その結果は、RETURNINGリストで定義した列と値を持ち、そのコマンドで挿入または更新された行全体に対して計算を行うSELECT文の結果と似たものになるでしょう。

注釈

指定したテーブルがパーティションテーブルの場合、各行は適切なパーティションに回され、そちらに挿入されます。指定したテーブルがパーティションの場合、挿入行にパーティションの制約に違反するものがあれば、エラーが発生します。

例

filmsテーブルに1行を挿入します。

```
INSERT INTO films VALUES
    ('UA502', 'Bananas', 105, '1971-07-13', 'Comedy', '82 minutes');
```

次の例では、len列を省略しています。したがって、ここにはデフォルト値が入ります。

```
INSERT INTO films (code, title, did, date_prod, kind)
    VALUES ('T_601', 'Yojimbo', 106, '1961-06-16', 'Drama');
```

次の例では、日付列に対して値を指定する代わりにDEFAULTを使用します。

```
INSERT INTO films VALUES
    ('UA502', 'Bananas', 105, DEFAULT, 'Comedy', '82 minutes');
INSERT INTO films (code, title, did, date_prod, kind)
    VALUES ('T_601', 'Yojimbo', 106, DEFAULT, 'Drama');
```

全てデフォルト値からなる行を挿入します。

```
INSERT INTO films DEFAULT VALUES;
```

複数行のVALUES構文を使用して複数行を挿入します。

```
INSERT INTO films (code, title, did, date_prod, kind) VALUES
    ('B6717', 'Tampopo', 110, '1985-02-10', 'Comedy'),
    ('HG120', 'The Dinner Game', 140, DEFAULT, 'Comedy');
```

次の例では、filmsテーブルと同じ列レイアウトを持つtmp_filmsテーブルからfilmsテーブルへいくつか行を挿入します。

```
INSERT INTO films SELECT * FROM tmp_films WHERE date_prod < '2004-05-07';
```

次の例では、配列型の列に挿入します。

```

-- 三目並べ用の3×3マスのゲーム盤を作成します。
INSERT INTO tictactoe (game, board[1:3][1:3])
    VALUES (1, '{{" "," "," "},{" "," "," "},{" "," "," "}}');

--上の例の添え字は本当は必要ありません。
INSERT INTO tictactoe (game, board)
    VALUES (2, '{{X," "," "},{" ",O," "},{" ",X," "}}');
```

distributorsテーブルに一行を挿入し、そのDEFAULT句により生成されたシーケンス番号を返します。

```
INSERT INTO distributors (did, dname) VALUES (DEFAULT, 'XYZ Widgets')
   RETURNING did;
```

Acme社の顧客を担当する営業担当者の売り上げ数を増やし、ログテーブルに更新行全体と更新時刻を記録します。

```
WITH upd AS (
  UPDATE employees SET sales_count = sales_count + 1 WHERE id =
    (SELECT sales_person FROM accounts WHERE name = 'Acme Corporation')
    RETURNING *
)
INSERT INTO employees_log SELECT *, current_timestamp FROM upd;
```

新しい販売店(distributors)を適切に挿入または更新します。did列に現れる値を制限する一意インデックスが定義されているものとします。元々挿入されようとしていた値を参照するために、特別なexcludedテーブルが使用されていることに注意して下さい。

```
INSERT INTO distributors (did, dname)
    VALUES (5, 'Gizmo Transglobal'), (6, 'Associated Computing, Inc')
    ON CONFLICT (did) DO UPDATE SET dname = EXCLUDED.dname;
```

販売店を挿入するか、あるいは挿入しようとした行について既存の除外行（before insertの行トリガを実行した後で制約列にマッチした行）がある場合は何もしません。例ではdid列に現れる値を制限する一意インデックスがあるものとしています。

```
INSERT INTO distributors (did, dname) VALUES (7, 'Redline GmbH')
    ON CONFLICT (did) DO NOTHING;
```

新しい販売店を適切に挿入または更新します。例ではdid列に現れる値を制限する一意インデックスがあるものとしています。実際に更新される行を制限するためにWHERE句が使われています（ただし、更新されない既存の行もすべてロックされます）。

```

-- 特定の郵便番号については既存の販売店を更新しません
INSERT INTO distributors AS d (did, dname) VALUES (8, 'Anvil Distribution')
    ON CONFLICT (did) DO UPDATE
    SET dname = EXCLUDED.dname || ' (formerly ' || d.dname || ')'
    WHERE d.zipcode <> '21201';


-- 文中で制約を直接指定します（DO NOTHINGの動作をする競合解決のため
-- 関連するインデックスを指定します）
INSERT INTO distributors (did, dname) VALUES (9, 'Antwerp Design')
    ON CONFLICT ON CONSTRAINT distributors_pkey DO NOTHING;
```

可能であれば新しい販売店を挿入しますが、できないときはDO NOTHINGとします。この例では、is_activeというブーリアン列がtrueである行という条件で、did列に一意インデックスが定義されているものとしています。

```

-- この文は"WHERE is_active"という述語を使って、部分インデックスを
-- 推定できますが、単に"did"上の通常の一意制約を使うこともできます
INSERT INTO distributors (did, dname) VALUES (10, 'Conrad International')
    ON CONFLICT (did) WHERE is_active DO NOTHING;
```

互換性

INSERTは標準SQLに準拠します。ただし、RETURNING句、INSERTでWITHが可能であること、ON CONFLICTで代替の動作を指定できることはPostgreSQLの拡張です。また、標準SQLでは、列名リストが省略された時に、VALUES句またはqueryで一部の列のみを指定することはできません。

標準SQLでは、必ず値を生成するIDENTITY列が存在する場合にのみOVERRIDING SYSTEM VALUEを指定できるとしています。PostgreSQLではこの句はどのような場合でも指定でき、それが適用できないときには無視します。

query句の制限については、SELECTにて記述されています。

LISTEN

LISTEN — 通知を監視する

概要

```
LISTEN channel
```

説明

LISTENは現在のセッションを、通知チャネルchannelのリスナとして登録します。現在のセッションが既に指定した通知チャネルのリスナとして登録されている場合は、何も起こりません。

このセッションまたは同一データベースに接続している別のセッションによってNOTIFY channelが実行されると、現在その通知チャネルを監視している全てのセッションに対して通知されます。次に、各セッションは接続中のクライアントアプリケーションにこれを通知します。

UNLISTENコマンドを使って、セッションに登録された指定通知チャネルを解除できます。また、セッションの監視登録はそのセッションが終了した時点で自動的に削除されます。

クライアントアプリケーションが通知イベントを検出する方法は、使用しているPostgreSQLアプリケーションプログラミングインタフェースに依存します。libpqライブラリを使用するアプリケーションでは、LISTENを通常のSQLコマンドとして発行し、その後、PQnotifiesルーチンを定期的に呼び出して通知イベントが受信されたかどうかを調べる必要があります。libpgtcl等の他のインタフェースには、通知イベントを扱うより高レベルな方法が用意されています。実際、libpgtclを使ったアプリケーションの場合、プログラマがLISTENやUNLISTENを直接発行する必要すらありません。詳細については、使用中のインタフェースのドキュメントを参照してください。

パラメータ

channel

通知チャネルの名前です（任意の識別子）。

注釈

LISTENはトランザクションのコミット時に有効になります。LISTENまたはUNLISTENがトランザクション内で実行され、それがロールバックされた場合、監視している通知チャネルの集合は変更されません。

LISTENを実行したトランザクションでは二相コミットの準備を行うことはできません。

監視するセッションを最初に設定する時に、競合状態があります。同時にコミット中の複数のトランザクションが通知イベントを送った場合、新しく監視を始めたセッションはそのうちのどれをまさに受信するでしょうか。答は、トランザクションのコミット段階のある瞬間の後にコミットされたすべてのイベントを受信する、で

す。しかし、これは問い合わせにおいてトランザクションが気づくデータベースの状態よりもわずかに後です。ここからLISTENを使う場合の以下のような規則が導かれます。まずそのコマンドを実行する(そしてコミットする!)、それから新しいトランザクションでアプリケーションのロジックの必要に応じてデータベースの状態を検査する、それから通知に基づいてデータベースの状態に対するその後の変更を確認する。最初に受信した通知のいくつかはデータベースの最初の検査ですでに確認した更新を参照しているかもしれませんが、これは普通は無害です。

NOTIFYには、LISTENおよびNOTIFYについてのより広範な説明があります。

例

psqlから、監視/通知処理の設定と実行を行います。

```
LISTEN virtual;
NOTIFY virtual;
Asynchronous notification "virtual" received from server process with PID 8448.
```

互換性

標準SQLにLISTENはありません。

関連項目

NOTIFY, UNLISTEN

LOAD

LOAD — 共有ライブラリファイルの読み込みを行う

概要

```
LOAD 'filename'
```

説明

LOADコマンドは、共有ライブラリファイルをPostgreSQLサーバのアドレス空間にロードします。そのファイルが既にロード済みなら、このコマンドは何も行いません。C関数を含む共有ライブラリファイルは、その中の1つの関数が呼び出された時に常に、自動的にロードされます。このため通常、明示的なLOADは、関数群を提供するのではなく「フック」を通してサーバの動作を変更するライブラリをロードするためだけに必要となります。

ライブラリファイルの名前は通常は単なるファイル名だけで指定され、それが（dynamic_library_pathで設定される）サーバのライブラリサーチパス内で検索されます。あるいは、フルパス名で指定することもできます。いずれの場合も、プラットフォームでの共有ライブラリファイル名の標準的な拡張子は省略できます。この点についての詳細な情報は37.10.1を参照してください。

非特権ユーザは$libdir/plugins/にあるライブラリファイルのみをLOADさせることができます。つまり、指定したfilenameはこの文字列から始まらなければなりません。（このディレクトリ以下に確実に「安全な」ライブラリのみをインストールすることはデータベース管理者の責任です。）

互換性

LOADはPostgreSQLの拡張です。

関連項目

CREATE FUNCTION

LOCK

LOCK — テーブルをロックする

概要

```
LOCK [ TABLE ] [ ONLY ] name [ * ] [, ...] [ IN lockmode MODE ] [ NOWAIT ]


ここでlockmodeには以下のいずれかが入ります。

    ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE EXCLUSIVE
    | SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE
```

説明

LOCK TABLEはテーブルレベルのロックを取得します。必要であれば競合するロックが解除されるまで待機します。NOWAITが指定された場合は、対象のロックを取得できるまで待機せず、すぐにロックが取得できなければ、このコマンドを中止し、エラーを出力します。ロックは、一度取得されると現行のトランザクションが完了するまで保持されます（UNLOCK TABLEといったコマンドはありません。ロックが解除されるのは常にトランザクションの終了時です）。

ビューがロックされると、ビューを定義する問い合わせに現れるテーブルもすべて同じロックモードで再帰的にロックされます。

テーブルを参照するコマンドのために自動的にロックを取得する場合、PostgreSQLは使用可能な一番弱いロックモードを常に使用します。LOCK TABLEはより制限の強いロックが必要な場合のために用意されています。例えば、分離レベルREAD COMMITTEDでトランザクションを実行するアプリケーションで、トランザクションの間中、テーブルのデータを確実に安定させる必要がある場合を考えます。この場合、問い合わせ実行前にテーブル全体にSHAREロックモードを使用します。これにより、データが同時に変更されるのを防ぎ、それ以降のテーブルの読み取りは、コミット済みの安定したデータが見えるようになります。なぜならSHAREロックモードは書き込み側が取得するROW EXCLUSIVEロックと競合するので、LOCK TABLE name IN SHARE MODE文は、ROW EXCLUSIVEを保持しているトランザクションがコミットまたはロールバックされるのを待つからです。したがって、一度ロックを取得してしまえば、コミットされていない状態の書き込みは存在しないことになります。さらに、ロックを解除するまで他のアプリケーションは書き込みを開始することができません。

REPEATABLE READまたはSERIALIZABLE分離レベルで実行しているトランザクションで同様の効果を得るには、全てのSELECT文とデータを更新する文を実行する前にLOCK TABLE文を実行する必要があります。REPEATABLE READまたはSERIALIZABLEトランザクション側から参照するデータの状態は、最初にSELECT文またはデータ更新用文が開始された時点で固定されます。後からトランザクション内でLOCK TABLEを実行した場合も同時書き込みを防ぐことはできますが、トランザクションの読み込み対象データの値がコミットされた最新の値であることは保証されません。

このようなトランザクションでテーブルのデータを変更する場合は、SHAREモードではなくSHARE ROW EXCLUSIVEロックモードを使用する必要があります。これによって、この種のトランザクションが同時に複数実行されることがなくなります。SHARE ROW EXCLUSIVEを使用しないと、デッドロックが発生する可能性があります。2つのトランザクションの両方が、SHAREモードを取得していながら、実際の更新に必要なROW EXCLUSIVEモードを取得できない状態になる可能性があるためです（トランザクション自身が所有しているロック間は競合しないので、トランザクションはSHAREモードを保持している間もROW EXCLUSIVEを獲得することができます。しかし、他のトランザクションがSHAREモードを保持している時にはROW EXCLUSIVEを獲得することはできません）。デッドロックを回避するには、全てのトランザクションが、必ず同一オブジェクトに対して同一の順番でロックを取得するようにしてください。また、1つのオブジェクトに対して複数のロックモードを呼び出す場合、トランザクションは常に最も制限の強いモードを最初に取得するべきです。

ロックモードとロック取得方針についてのより詳細については13.3を参照してください。

パラメータ

name

ロックする既存のテーブルの名前です（スキーマ修飾名も可）。テーブル名の前にONLYが指定された場合、そのテーブルのみをロックします。ONLYが指定されない場合、そのテーブルとすべての子テーブル（もしあれば）をロックします。オプションで、テーブル名の後に*を指定することで、明示的に継承するテーブルも含まれることを示すことができます。

LOCK a, b;というコマンドはLOCK TABLE a; LOCK TABLE b;と同じです。テーブルは1つひとつLOCKで指定された順番でロックされます。

lockmode

ロックモードには、取得するロックと競合するロックを指定します。ロックモードについては、13.3で説明します。

ロックモードを指定しない場合、最も制限が強いACCESS EXCLUSIVEが使用されます。

NOWAIT

LOCK TABLEが競合するロックの解放まで待機しないことを指定します。指定したロックがすぐに取得できない場合、トランザクションはアボートされます。

注釈

LOCK TABLE ... IN ACCESS SHARE MODEには、対象テーブルのSELECT権限が必要です。LOCK TABLE ... IN ROW EXCLUSIVE MODEには、対象テーブルのINSERT、UPDATE、DELETEまたはTRUNCATE権限が必要です。他の形式のLOCKには、テーブルレベルのUPDATE、DELETEあるいはTRUNCATE権限を持たなければなりません。

ビューに対するロックを実行するユーザはビューに対して対応する権限を持っていなければなりません。さらに、ビューの所有者は被参照テーブルに対して関連する権限を持っていなければなりませんが、ロックを実行するユーザは被参照テーブルに対する権限は必要ありません。

LOCK TABLEはトランザクションブロックの外側では意味がありません。文が完了するまでしかロックは保持されません。したがってPostgreSQLはLOCKがトランザクションブロックの外側で使用された場合にエラーを報

告します。トランザクションブロックを定義するためにはBEGINおよびCOMMIT（またはROLLBACK）を使用してください。

LOCKが扱うのはテーブルレベルのロックのみです。そのため、モード名にROWが含まれるのは適切ではありません。これらのモード名は、普通は、ロックされたテーブル内で行レベルのロックを取得する意図と解釈されるでしょう。また、ROW EXCLUSIVEモードは共有可能なテーブルロックです。LOCK TABLEに関しては、全てのロックモードが同じ意味を持っており、違うのは、どのモードがどのモードと競合するかという規則だけであることに注意して下さい。実際の行レベルでのロックを獲得する方法については、SELECTの文書の13.3.2とThe Locking Clauseを参照してください。

例

外部キーテーブルへの挿入を行う際に、主キーテーブルへのSHAREロックを獲得します。

```
BEGIN WORK;
LOCK TABLE films IN SHARE MODE;
SELECT id FROM films
    WHERE name = 'Star Wars: Episode I - The Phantom Menace';

-- レコードがなければROLLBACKしてください。
INSERT INTO films_user_comments VALUES
    (_id_, 'GREAT! I was waiting for it for so long!');
COMMIT WORK;
```

削除操作を行う際に主キーテーブルのSHARE ROW EXCLUSIVEロックを取得します。

```
BEGIN WORK;
LOCK TABLE films IN SHARE ROW EXCLUSIVE MODE;
DELETE FROM films_user_comments WHERE id IN
    (SELECT id FROM films WHERE rating < 5);
DELETE FROM films WHERE rating < 5;
COMMIT WORK;
```

互換性

標準SQLにはLOCK TABLEはありません。その代わりにトランザクションの同時性レベルを指定するSET TRANSACTIONが使用されます。PostgreSQLはこのコマンドもサポートしています。詳細はSET TRANSACTIONを参照してください。

ACCESS SHARE、ACCESS EXCLUSIVE、SHARE UPDATE EXCLUSIVEロックモードを除き、PostgreSQLのロックモードとLOCK TABLE構文はOracleのものと互換性があります。

MOVE

MOVE — カーソルの位置を決める

概要

```
MOVE [ direction [ FROM | IN ] ] cursor_name


ここでdirectionは空または以下のいずれかを取ることができます。

    NEXT
    PRIOR
    FIRST
    LAST
    ABSOLUTE count
    RELATIVE count
    count
    ALL
    FORWARD
    FORWARD count
    FORWARD ALL
    BACKWARD
    BACKWARD count
    BACKWARD ALL
```

説明

MOVEはデータを取り出すことなくカーソルの位置を変更します。MOVEはFETCHコマンドとまったく同じように動作しますが、カーソルの位置を変えるだけで行を返しません。

MOVEコマンドのパラメータはFETCHコマンドと同一です。構文と使用方法についての詳細はFETCHを参照してください。

出力

正常に終了すると、MOVEは以下の形式のコマンドタグを返します。

```
MOVE count
```

countは同じパラメータを与えたFETCHコマンドが返すはずの行数です（この値は0の場合もあります）。

例

```
BEGIN WORK;
DECLARE liahona CURSOR FOR SELECT * FROM films;


-- 最初の5行を飛ばします。
MOVE FORWARD 5 IN liahona;
MOVE 5


-- liahonaカーソル内の6行目を抽出します。
FETCH 1 FROM liahona;
 code  | title  | did | date_prod  |  kind  |  len
-------+--------+-----+------------+--------+-------
 P_303 | 48 Hrs | 103 | 1982-10-22 | Action | 01:37
(1 row)


-- カーソルliahonaを閉じ、トランザクションを終了します。
CLOSE liahona;
COMMIT WORK;
```

互換性

標準SQLにはMOVE文はありません。

関連項目

CLOSE, DECLARE, FETCH

NOTIFY

NOTIFY — 通知を生成する

概要

```
NOTIFY channel [ , payload ]
```

説明

NOTIFYコマンドは、現在のデータベース内で事前に指定チャネル名についてLISTEN channelコマンドを実行したクライアントアプリケーションに「ペイロード」文字列(省略可能)を持つ通知イベントを送ります。通知はすべてのユーザから可視です。

NOTIFYは同一のPostgreSQLデータベースにアクセスするプロセスの集合に対する単純なプロセス間通信の仕組みを提供します。通知の際にペイロード文字列を送信することができます。また、データベース内のテーブルを使用して通知者から(1つまたは複数の)リスナに追加的なデータを渡すことにより、構造化されたデータを渡す高度な仕組みを構築することができます。

通知イベントとしてクライアントに渡される情報には、通知チャネル名と通知を行うセッションのサーバプロセスのPID、ペイロード文字列(指定されていなければ空文字列)が含まれます。

各データベースにおいて使用される通知チャネル名とその意味についての定義は、データベース設計者に任されています。通知チャネル名には、データベース内のテーブル名と同じものを使用するのが一般的です。通知イベントは本質的に「このテーブルを変更しました。変更された箇所を確認してください」ということを意味するものです。しかし、NOTIFYコマンドとLISTENコマンドでは、そのような関連付けは強制されていません。例えば、データベース設計者は、1つのテーブルに対する異なる種類の変更を通知するために、複数の異なる通知チャネル名を使用することができます。他の方法としてペイロード文字列を使用して各種様々な状況に対応させることもできます。

特定のテーブルが変更されたことを通知するためにNOTIFYを使用する場合、NOTIFYをテーブル更新時に発行される文トリガー内に配置すると便利です。こうすると、通知はテーブルが変更された時に自動的に行われるので、アプリケーションプログラマが通知の実行を忘れるといった事故を防ぐことができます。

NOTIFYとSQLトランザクションの間には、いくつかの重要な相互作用があります。まず、NOTIFYがトランザクション内部で実行された場合、通知イベントはトランザクションがコミットされない限り配送されません。トランザクションがアボートされた場合、NOTIFYだけでなく、そのトランザクション内で行われたコマンドが全て無効化されるので、これは妥当といえます。しかし、通知イベントが即座に配送されることを期待していた場合、当惑するかもしれません。次に、監視中のセッションがトランザクション処理中に通知シグナルを受け取った場合、そのトランザクションが(コミットもしくはアボートされて)完了するまで、通知イベントは接続しているクライアントに配送されません。この理由も同じです。トランザクションに通知が配送された後にそのトランザクションがアボートされた場合、何とかして通知を取り消したくなりますが、サーバはいったんクライアントに送信した通知を「取り戻す」ことはできません。したがって、通知イベントはトランザクションとトランザクションの合間にのみ配送されます。結論として、NOTIFYを使用してシグナルの実時間処理をするアプリケーションではトランザクションを短くしておかなければなりません。

同じチャネル名が、同一トランザクションから同じペイロード文字列で複数回通知される場合、1つの通知インスタンスのみをリスナに伝えます。一方、異なるペイロード文字列を持つ通知は常に別の通知として伝えられます。同様に別のトランザクションからの通知が1つの通知にまとめられることは決してありません。重複する通知インスタンスを後で削除する場合は例外ですが、NOTIFYは同一トランザクションからの通知は送信された順番に配送されることを保証します。また異なるトランザクションからのメッセージがトランザクションのコミット順で配送されることも保証します。

NOTIFYを実行するクライアント自身が、その通知の通知チャネルを監視していることはよくあります。この場合、同じ通知名を監視する他のセッションに対するのと同じように通知イベントが戻ってきます。アプリケーションのロジックにもよりますが、これは無駄な作業になることがあります。例えば、そのセッションが書き出したばかりのデータベースに対する更新を調べるためにテーブルの再読み込みを行う場合などが考えられます。通知元セッションのサーバプロセスのPID(通知イベントメッセージ内にあります)と、自分自身のPID(libpqで得られます)が同じかどうか調べることで、こういった余計な作業を回避できます。PIDが同じであれば、その通知イベントは自分自身から跳ね返ってきたものであり、無視することができます。

パラメータ

channel

シグナルとして送られる通知チャネル名です(任意の識別子)。

payload

通知と一緒に通信される「ペイロード」文字列です。これは単純な文字列リテラルとして指定されなければなりません。デフォルトの設定では、8000バイト未満でなければなりません。(バイナリデータまたは大規模な情報を渡さなければならないのであれば、データベーステーブル内に格納しレコードのキーを送信することが最善です。)

注釈

送信済みだがすべての監視セッションでは処理されていない通知を保持するためのキューが存在します。このキューがいっぱいになると、NOTIFYを呼び出すトランザクションのコミットに失敗します。キューはかなり大きなもの(標準のインストレーションで8ギガバイト)であり、ほとんどすべての環境で十分な大きさであるはずです。しかしセッションがNOTIFYを実行した後に長期間のトランザクションに入った場合、キューからクリーンアップできなくなります。キューの半分までたまると、ログファイル内にクリーンアップを妨げているセッションを指し示す警告が現れるようになります。この場合、クリーンアップ処理が進むように、確実にそのセッションでその現在のトランザクションを完了させるようにしなければなりません

関数pg_notification_queue_usageは現在、保留中の通知によって占められているキューの割合を返します。詳細な情報については9.26を参照してください。

NOTIFYを実行したトランザクションでは二相コミットを準備することはできません。

pg_notify

通知を送信するために関数pg_notify(text,text)を使用することもできます。この関数は第1引数としてチャネル名、第2引数としてペイロードを取ります。不定のチャネル名、ペイロードで作業しなければならない場合は、NOTIFYコマンドよりこの関数を使用する方がかなり簡単です。

例

psqlから監視/通知処理の設定と実行を行います。

```
LISTEN virtual;
NOTIFY virtual;
Asynchronous notification "virtual" received from server process with PID 8448.
NOTIFY virtual, 'This is the payload';
Asynchronous notification "virtual" with payload "This is the payload" received from server
 process with PID 8448.

LISTEN foo;
SELECT pg_notify('fo' || 'o', 'pay' || 'load');
Asynchronous notification "foo" with payload "payload" received from server process with PID
 14728.
```

互換性

標準SQLにはNOTIFYはありません。

関連項目

LISTEN, UNLISTEN

PREPARE

PREPARE — 実行する文を準備する

概要

```
PREPARE name [ ( data_type [, ...] ) ] AS statement
```

説明

PREPAREはプリペアド文を作成します。プリペアド文は、性能を最適化するために利用可能なサーバ側オブジェクトです。PREPARE文を実行すると、指定された問い合わせの構文解析、書き換えが行われます。その後、EXECUTE文が発行された際に、プリペアド文は実行計画が作成され、実行されます。この作業の分割により構文解析作業が繰り返されることを防止でき、さらに、特定のパラメータ値に合わせた実行計画を提供することができます。

プリペアド文はパラメータ、すなわち文が実行される時に代入される値を取ることができます。プリペアド文を作成する時には$1や$2などを使用して、位置によりパラメータを参照してください。対応するパラメータのデータ型のリストをオプションで指定することもできます。パラメータのデータ型の指定がない、または、unknownと宣言されている場合、型はパラメータが最初に参照される文脈より(可能ならば)推測されます。文の実行時には、EXECUTE文内にこれらのパラメータの実際の値を指定します。詳細はEXECUTEを参照してください。

プリペアド文は現在のデータベースセッションの期間中にのみ保持されます。セッションが終了すると、プリペアド文は破棄されます。そのため、再び利用する場合は、再作成する必要があります。また、これは、1つのプリペアド文を同時実行中の複数のデータベースクライアントから使用することはできないことを意味します。ただし、各クライアントが個別にプリペアド文を作成することはできます。プリペアド文を手作業で削除するには、DEALLOCATEコマンドを使用します。

プリペアド文は潜在的には、単一のセッションで同類の問い合わせを多数実行する場合に、パフォーマンスにおける最大の利益がえられます。パフォーマンスの違いは、文の書き換えや実行計画が複雑なほど顕著になるでしょう。例えば、問い合わせに多数のテーブルの結合が含まれている場合や、いくつものルールを適用しなければならない場合などが考えられます。書き換えおよび実行計画が比較的単純で、実行コストが高い文の場合は、プリペアド文の効果はそれほど現れないでしょう。

パラメータ

name

個々のプリペアド文に与えられる任意の名前です。この名前は、1つのセッション内で一意でなければいけません。プリペアド文の実行および削除の時に、この名前が使用されます。

data_type

プリペアド文に対するパラメータのデータ型です。特定のパラメータのデータ型の指定がない、または、unknownと指定された場合、パラメータが最初に参照される文脈から推測されます。プリペアド文自体の中でこのパラメータを参照する時は、$1、$2などを使用します。

statement

任意のSELECT、INSERT、UPDATE、DELETE、VALUES文です。

注釈

プリペアド文は、汎用的な計画または独自の計画のいずれかで実行することができます。汎用的な計画は全実行に渡って同じであるのに対して、独自の計画はその呼出しで与えられたパラメータ値を使った特別な実行のために生成されます。汎用的な計画の使用は計画のオーバーヘッドを回避しますが、プランナがパラメータ値の知識を使えるので、独自の計画の方がずっと効率よく実行される場合があります。(もちろん、プリペアド文にパラメータがなければ、これは無意味で、汎用的な計画が常に使われます。)

デフォルト(すなわち、plan_cache_modeがautoに設定されている場合)では、パラメータのあるプリペアド文に対して、汎用的な計画を使うか独自の計画を使うかを、サーバは自動的に選択します。これに対する現在の規則は、最初の5回が独自の計画で実行され、その計画の推定コストの平均が計算される、というものです。それから汎用的な計画が作成され、その推定コストが独自の計画のコストの平均と比較されます。そのコストが独自の計画のコストの平均よりもそれほど高くなければ、再計画を繰り返すことが望ましいと考えて、その後の実行は汎用的な計画を使います。

plan_cache_modeをforce_generic_planまたはforce_custom_planに設定して、サーバにそれぞれ汎用的な計画または独自の計画を使うように強制することで、この発見的手法を置き換えることができます。汎用的な計画の実際のコストが独自の計画のものよりもずっと多い場合でも、汎用的な計画を選べるようになりますので、汎用的な計画のコスト推定が何らかの理由でひどく外れる場合に、この設定は主として有用です。

プリペアド文に対してPostgreSQLが使用する問い合わせ計画を検証するためには、EXPLAIN、例えば

```
EXPLAIN EXECUTE stmt_name(parameter_values);
```

を使用してください。汎用的な計画が使用される場合には、$nというパラメータ記号が含まれ、独自の計画が使用される場合は提供されたパラメータの値で置換されます。

問い合わせの実行計画や問い合わせの最適化のためにPostgreSQLが収集する統計に関する詳細は、ANALYZEのドキュメントを参照してください。

プリペアド文の主要な利点は、文の解析処理と計画作成処理の繰り返しを防止することですが、PostgreSQLでは、以前にそのプリペアド文を使用してから、文の中で使用されているデータベースオブジェクトが定義(DDL)の変更を受けた時は常に再解析処理と計画再作成処理を強制します。また、一度使用してからsearch_pathの値が変わった場合も、文は新しいsearch_pathを使用して再解析されます。(後者の振る舞いはPostgreSQL 9.3の時に追加されました。) これらの規則により、プリペアド文の使用は意味的に同じ問い合わせを繰り返し再投入することとほぼ同じになりますが、特に最善の計画が使用している間に変わらずに残る場合、オブジェクトの変更がない場合の性能という利点があります。意味的な等価性が完全ではない場合の例は、文が未修飾名によってテーブルを参照し、その後同じ名前のテーブルが新た

にsearch_path内で前に現れるスキーマ内に作成された場合、文の中で使用されるオブジェクトには変更がありませんので、自動再解析は行われません。しかし他の何らかの変更により強制的に再解析された場合、その後の使用では新しいテーブルが参照されるようになります。

pg_prepared_statementsシステムビューを問い合わせることによりセッションで利用可能なプリペアド文をすべて確認することができます。

例

INSERT文に対してプリペアド文を作成し、実行します。

```
PREPARE fooplan (int, text, bool, numeric) AS
    INSERT INTO foo VALUES($1, $2, $3, $4);
EXECUTE fooplan(1, 'Hunter Valley', 't', 200.00);
```

SELECT文に対してプリペアド文を作成し、実行します。

```
PREPARE usrrptplan (int) AS
    SELECT * FROM users u, logs l WHERE u.usrid=$1 AND u.usrid=l.usrid
    AND l.date = $2;
EXECUTE usrrptplan(1, current_date);
```

この例では第2パラメータのデータ型が指定されていません。このため$2が使用される文脈からデータ型が推測されます。

互換性

標準SQLにはPREPARE文が含まれていますが、埋め込みSQLでの使用に限られています。また、標準SQLのPREPARE文では多少異なる構文が使用されます。

関連項目

DEALLOCATE, EXECUTE

PREPARE TRANSACTION

PREPARE TRANSACTION — 二相コミット用に現在のトランザクションを準備する

概要

```
PREPARE TRANSACTION transaction_id
```

説明

PREPARE TRANSACTIONは、二相コミット用に現在のトランザクションを準備します。このコマンドの後、トランザクションは現在のセッションと関連しなくなります。トランザクションの状態は完全にディスク上に保存され、コミット要求前にデータベースがクラッシュしてしまったとしても、ほぼ確実に正常にコミットできるようになります。

準備された後、そのトランザクションをCOMMIT PREPAREDによりコミット、あるいはROLLBACK PREPAREDによりロールバックすることができます。元のトランザクションを実行したセッションだけではなく、任意のトランザクションからこれらのコマンドを発行することができます。

コマンドを発行したセッションから見ると、PREPARE TRANSACTIONはROLLBACKコマンドと似たような動作をします。実行した後、実行中の現在のトランザクションはなくなり、準備したトランザクションの効果は不可視になります。(そのトランザクションがコミットされた場合に効果が可視になります。)

何らかの原因でPREPARE TRANSACTIONコマンドが失敗した場合、ROLLBACKが行われます。つまり、現在のトランザクションが取り消されます。

パラメータ

transaction_id

後でCOMMIT PREPAREDやROLLBACK PREPAREDでトランザクションを識別するための任意の識別子です。この識別子は文字列リテラルでなければなりません。また、200バイト未満でなければなりません。また、その時点で準備されたトランザクションとして使用されている他の識別子と同じものは使用できません。

注釈

PREPARE TRANSACTIONはアプリケーションや対話式のセッションでの使用を目的としていません。この目的は、外部トランザクションマネージャにより、複数のデータベースやその他のトランザクションを持つリソースを跨るグローバルなトランザクションを原子的に実現できるようにすることです。トランザクションマネージャを作成しているのでなければ、おそらくPREPARE TRANSACTIONを使用するべきではありません。

このコマンドはトランザクションブロック内で使用しなければなりません。トランザクションブロックを始めるには、BEGINを使用してください。

現時点では、一時テーブルもしくはセッションの一時的な名前空間を含む操作を行ったトランザクション、カーソルをWITH HOLDで作成したトランザクション、LISTEN、UNLISTENまたはNOTIFYを実行したトランザクションをPREPAREさせることはできません。準備したトランザクションで便利に使用するには、これらの機能は現在のセッションにあまりに強く結びついているためです。

トランザクションで何らかの実行時パラメータが（LOCALオプションなしの）SETで設定されている場合、その影響はPREPARE TRANSACTIONの後も残ります。また、その後のCOMMIT PREPAREDやROLLBACK PREPAREDの影響を受けません。したがって、この意味では、PREPARE TRANSACTION はROLLBACKよりCOMMITと似た動きであるといえます。

その時点で利用できるすべての準備されたトランザクションはpg_prepared_xactsシステムビューで列挙されます。

> **注意**
>
> トランザクションを長期間準備された状態のままとすることは勧められません。これは格納領域を回収するVACUUM機能を妨害し、極端な場合では、トランザクションの周回（24.1.5参照）を回避するためにデータベースを停止させてしまいます。またトランザクションが保持しているすべてのロックを保持し続けていることにも注意してください。この機能の想定している使用方法は、外部トランザクションマネージャが他のデータベースがコミットの準備をしたと検証した後すぐに、準備されたトランザクションは通常コミットまたはロールバックされることです。
>
> 準備されたトランザクションを追跡し、それを即座に終了できるように外部トランザクションマネージャを設定していない場合、max_prepared_transactionsをゼロに設定して準備されたトランザクション機能を無効にしておくことが最善です。こうすれば事故により準備されたトランザクションが作成され、それが忘れられて問題を引き起こすことを防止できます。

例

二相コミット用に現在のトランザクションを準備します。トランザクション識別子としてfoobarを使用します。

```
PREPARE TRANSACTION 'foobar';
```

互換性

PREPARE TRANSACTIONはPostgreSQLの拡張です。これは外部のトランザクション管理システムによる利用を意図したものです。トランザクション管理システムの一部（X/Open XAなど）は標準化されていますが、こうしたシステムのSQL側は標準化されていません。

関連項目

COMMIT PREPARED, ROLLBACK PREPARED

REASSIGN OWNED

REASSIGN OWNED — あるデータベースロールにより所有されたデータベースオブジェクトの所有権を変更する

概要

```
REASSIGN OWNED BY { old_role | CURRENT_USER | SESSION_USER } [, ...]
               TO { new_role | CURRENT_USER | SESSION_USER }
```

説明

REASSIGN OWNEDは、old_rolesのいずれかが所有するデータベースオブジェクトの所有権をnew_roleに変更するようシステムに指示します。

パラメータ

old_role

ロール名です。このロールが所有する、現在のデータベースのすべてのオブジェクトの所有権、および共有オブジェクトの中のすべて(データベースやテーブル空間)をnew_roleに割り当てます。

new_role

対象とするオブジェクトの新しい所有者となるロール名です。

注釈

REASSIGN OWNEDは、1つ以上のロールの削除準備によく使用されます。REASSIGN OWNEDは他のデータベース内のオブジェクトには影響を与えませんので、通常は、削除対象のロールにより所有されるオブジェクトを有するデータベース毎にこのコマンドを実行する必要があります。

REASSIGN OWNEDは元のロールと対象のロール上にメンバ資格が必要です。

代わりにDROP OWNEDコマンドを使用して、1つ以上のロールにより所有されるデータベースオブジェクトすべてを単に削除することができます。

REASSIGN OWNEDコマンドは、old_rolesにより所有されていないオブジェクトにおいてold_rolesに与えられた権限には影響しません。同様に、ALTER DEFAULT PRIVILEGESで作成されたデフォルトの権限には影響しません。こうした権限を取り消すには、DROP OWNEDを使用してください。

詳しくは21.4を参照してください。

互換性

REASSIGN OWNEDコマンドはPostgreSQLの拡張です。

関連項目

DROP OWNED, DROP ROLE, ALTER DATABASE

REFRESH MATERIALIZED VIEW

REFRESH MATERIALIZED VIEW — マテリアライズドビューの内容を置換する

概要

```
REFRESH MATERIALIZED VIEW [ CONCURRENTLY ] name
    [ WITH [ NO ] DATA ]
```

説明

REFRESH MATERIALIZED VIEWはマテリアライズドビューの内容を完全に置き換えます。このコマンドを実行するには、マテリアライズドビューの所有者でなければなりません。古い内容は破棄されます。WITH DATAが指定されている場合(またはデフォルトでは)、新しいデータを提供するために裏付け問い合わせが実行され、マテリアライズドビューはスキャン可能状態になります。WITH NO DATAが指定されている場合、新しいデータは生成されず、マテリアライズドビューはスキャン不可状態になります。

CONCURRENTLYとWITH NO DATAを同時に指定することはできません。

パラメータ

CONCURRENTLY

そのマテリアライズドビューに対して同時に実行されるSELECTをロックすることなく、マテリアライズドビューをREFRESHします。このオプションを使わない場合、多くの行に影響を与えるREFRESHはリソースをあまり使わず、早く終わる代わりに、そのマテリアライズドビューから読み込もうとしている他の接続をブロックするかもしれません。影響を与える行が少ない場合は、このオプションは速いかもしれません。

このオプションは、マテリアライズドビューに、列名だけを使い、すべての行を含むUNIQUEインデックスが少なくとも1つある場合にのみ使えます。つまり、それは式のインデックスであったり、WHERE句を含んでいてはいけません。

このオプションは、マテリアライズドビューがスキャン不可状態のときは使うことができません。

このオプションを使う場合でも、1つのマテリアライズドビューに対して同時に実行できるREFRESHは一つだけです。

name

更新するマテリアライズドビューの名前(スキーマ修飾可)です。

注釈

将来のCLUSTER操作用のデフォルトインデックスは保持されますが、この属性に基づいた順序でREFRESH MATERIALIZED VIEWは行を生成しません。生成時にデータを順序付けしたければ、裏付け問い合わせの中でORDER BYを使用しなければなりません。

例

以下のコマンドは、マテリアライズドビューの定義からの問い合わせを用いてorder_summaryというマテリアライズドビューの内容を置き換え、スキャン可能状態とします。

```
REFRESH MATERIALIZED VIEW order_summary;
```

以下のコマンドはマテリアライズドビューannual_statistics_basisに関連する格納領域を解放し、スキャン不可状態とします。

```
REFRESH MATERIALIZED VIEW annual_statistics_basis WITH NO DATA;
```

互換性

REFRESH MATERIALIZED VIEWはPostgreSQLの拡張です。

関連項目

CREATE MATERIALIZED VIEW, ALTER MATERIALIZED VIEW, DROP MATERIALIZED VIEW

REINDEX

REINDEX — インデックスを再構築する

概要

```
REINDEX [ ( option [, ...] ) ] { INDEX | TABLE | SCHEMA | DATABASE | SYSTEM }
 [ CONCURRENTLY ] name


ここでoptionは以下の一つです。

    VERBOSE
```

説明

REINDEXは、インデックスのテーブルに保存されたデータを使用してインデックスを再構築し、古いインデックスのコピーと置き換えます。以下にREINDEXが使用される状況を示します。

- インデックスが破損してしまい、有効なデータがなくなった場合です。理論的には決して起こらないはずですが、実際には、ソフトウェアのバグやハードウェアの障害によりインデックスが破損することがあります。REINDEXはこの修復手段を提供します。
- インデックスが「膨張状態」、つまり、多くの空、もしくは、ほとんど空のページを持つ状態になっている場合です。この状況は、PostgreSQLのB-treeインデックスが特定の普通でないパターンでアクセスされた場合に起こり得ます。REINDEXを使って、使用されないページを取り除いた新しいインデックス作成すると、インデックスの領域消費量を減少することができます。詳細は24.2を参照してください。
- インデックスの格納パラメータ（フィルファクタなど）を変更し、この変更を確実に有効にしたい場合です。
- CONCURRENTLYオプションをつけたインデックス作成が失敗すると、このインデックスは「無効」として残されます。こうしたインデックスは使用されませんが、REINDEXを使用して再作成するのが便利かもしれません。REINDEX INDEXだけが無効なインデックスでの同時構築を実行できることに注意してください。

パラメータ

INDEX

指定したインデックスを再作成します。

TABLE

指定したテーブルの全インデックスを再作成します。テーブルに2次的な「TOAST」テーブルがある場合、それについてもインデックスを再作成します。

SCHEMA

指定したスキーマのすべてのインデックスを再作成します。このスキーマのテーブルが二次的な「TOAST」テーブルを持っている場合は、そのインデックスも再作成されます。共有システムカタログのインデックスも処理されます。この構文のREINDEXはトランザクションブロックの内側で実行することはできません。

DATABASE

現在のデータベースのすべてのインデックスを再作成します。共有システムカタログのインデックスも処理されます。この構文のREINDEXをトランザクションブロック内で実行することはできません。

SYSTEM

現在のデータベースのシステムカタログに対するすべてのインデックスを再作成します。共有システムカタログのインデックスも含みます。ユーザテーブルのインデックスは処理されません。この構文のREINDEXをトランザクションブロック内で実行することはできません。

name

インデックスを再作成するインデックス、テーブル、データベースの名前です。インデックスとテーブルはスキーマ修飾可能です。現状では、REINDEX DATABASEとREINDEX SYSTEMは現在のデータベースのインデックスのみを再作成することができます。そのため、このパラメータは現在のデータベース名と一致する必要があります。

CONCURRENTLY

このオプションが使われると、PostgreSQLは、そのテーブルで同時実行される挿入、更新、削除を妨げるようなロックを取得せずにインデックスを再構築します。一方、標準のインデックス再構築は終了するまでテーブルの書き込みをロックします(読み込みはロックしません)。このオプションを使用する場合に注意すべき点がいくつかあります—下記のRebuilding Indexes Concurrentlyを参照してください。

一時テーブルに対してはREINDEXは常に同時再作成ではありません。他のセッションはアクセスできませんし、同時でないインデックス再作成の方がより安価だからです。

VERBOSE

各インデックスが再作成されるときに、進捗レポートを表示します。

注釈

ユーザテーブル上の特定のインデックスに破損の疑いがある場合、REINDEX INDEXを使ってそのインデックスを再構築することもできますし、REINDEX TABLEを使ってそのテーブルのすべてのインデックスを再構築することもできます。

システムテーブルのインデックスの破損を復旧する場合の手順はより複雑になります。この場合、システムによって破損の可能性があるインデックス自体が使用されないようにすることが重要です（実際は、このようなケースでは、破損したインデックスに依存していたため、サーバプロセスが起動時に強制終了してしまう可能性があります）。安全に復旧させるには、システムカタログ検索時のインデックスの使用を禁止する-Pオプションを使用してサーバを起動しなければなりません。

考えられる方法の1つは次の方法です。まず、サーバを停止して、コマンドラインから-Pオプションを指定してシングルユーザ状態のPostgreSQLサーバを起動します。そして、再構成する範囲に応じて、REINDEX DATABASE、REINDEX SYSTEM、REINDEX TABLE、または、REINDEX INDEXコマンドを発行します。範囲が不明な場合は、REINDEX SYSTEMを使用して、そのデータベースの全てのシステムインデックスを再構成してください。その後、シングルユーザ状態のサーバセッションを停止して、通常のサーバを再起動します。シングルユーザ状態のサーバインタフェースの操作方法についての詳細は、postgresマニュアルページを参照してください。

その他、コマンドラインで-Pを指定して通常のサーバセッションを起動することもできます。具体的な方法は、クライアントによって異なります。しかし、libpqベースのクライアントであれば、クライアントを起動する前に環境変数PGOPTIONSを-Pに設定すれば実現できます。この方法では他のクライアントを締め出す必要はありませんが、修復が終わるまで破損したデータベースへの他のユーザの接続を防止する方が良いことに注意してください。

REINDEXは、インデックスの中身を1から作り直すという点では、インデックスを削除してから再作成する処理と似ています。しかし、ロックに関しては異なります。REINDEXはインデックスの元となるテーブルの書き込みをロックしますが、読み込みはロックしません。また、処理中のインデックスに対する排他ロックを取得するので、そのインデックスを使用する読み込みはブロックされます。一方、DROP INDEXは瞬間的に元となるテーブルの排他ロックを取得するので、書き込みも読み込みもブロックされます。その後に行うCREATE INDEXでは書き込みのみをロックし、読み込みはロックしません。インデックスは存在しないので、インデックスを使用する読み込みは発生しません。したがって、読み込みがブロックされることはありませんが、コストが高いシーケンシャルスキャンの使用を強制されることになります。

単一インデックスまたは単一テーブルのインデックス再作成を行うには、そのインデックスまたはテーブルの所有者でなければなりません。スキーマまたはデータベースに対するインデックス再作成を行うには、そのスキーマまたはデータベースの所有者でなければなりません。したがって、非スーパーユーザが他のユーザが所有するテーブルのインデックスを再作成できることに特に注意してください。しかし、特別な例外として、REINDEX DATABASE、REINDEX SCHEMA、REINDEX SYSTEMが非スーパーユーザにより発行された時には、 そのユーザがカタログを所有している場合(そのようなことは通常はありません)を除いて、共有カタログのインデックスは飛ばされます。もちろん、スーパーユーザは常にすべてのインデックス再作成を行うことができます。

パーティションテーブルやパーティションインデックスのインデックス再作成はサポートされていません。その代わり、個々のパーティションで別々にインデックスを再作成できます。

インデックスを同時に再構築

インデックスの再構築は、通常のデータベース操作を妨げることがあります。通常、PostgreSQLはインデックスが再構築されるテーブルへの書き込みをロックし、一度のテーブル走査で全インデックスの構築を実行します。他のトランザクションはテーブルを読み込めますが、そのテーブルで行を挿入、更新、削除しようとするとインデックスの再構築が終わるまでブロックされます。実行中の運用状態のデータベースシステムの場合、これは重大な影響を与えるかもしれません。非常に大規模なテーブルに対するインデックス作成は何時間もかかることがあり得ます。また小規模なテーブルであっても、インデックス再構築により、運用状態のシステムとしては受け入れられないほど長い時間、書き込みロックがかかる可能性があります。

PostgreSQLは最小限の書き込みロックでのインデックス再構築をサポートしています。REINDEXにCONCURRENTLYオプションをつけることでこの方式が行われます。このオプションを使うと、

PostgreSQLは再構築が必要な各インデックスに関してテーブルを2回走査しなければなりません。さらに、潜在的にそのインデックスを使用する可能性がある、実行中のすべてのトランザクションが終わるまで待機しなければなりません。したがって、この方式は通常のインデックス再構築よりも総作業時間がかかり、また、インデックスを修正する可能性のある終了していないトランザクションが待つ必要がありますので、完了するまでの時間が非常に長くなります。しかし、インデックス再構築中に通常の操作を続けることができますので、この方式は運用環境でのインデックス再構築に有用です。もちろん、インデックス再構築によりCPUやメモリ、入出力に余分に負荷がかかりますので、他の操作が低速になる可能性があります。

同時実行再インデックスは以下のような段階で行なわれます。各段階は分離したトランザクション内で実行されます。複数のインデックスを再構築する場合、次の段階に移る前にすべてのインデックスに対して各段階が繰り返されます。

1. カタログpg_indexに新しく一時的なインデックス定義が追加されます。この定義は古いインデックスを置き換えるのに使われます。処理中は、再インデックスされるインデックスと関連するテーブルに対して、セッションレベルでのSHARE UPDATE EXCLUSIVEロックを取得します。スキーマが修正されないようにするためです。

2. インデックス構築の第1段階は新しいインデックスそれぞれに対して行なわれます。インデックスが一度構築されれば、挿入の準備ができたということで、そのフラグpg_index.indisreadyは「true」に切り替わります。構築を実行したトランザクションが終わった後で、他のセッションから見えるようになります。この過程は各インデックスに対して分離したトランザクションで行なわれます。

3. 次に、第1段階実行中に追加されたタプルを追加する第2段階が行なわれます。この過程は各インデックスに対して分離したトランザクションで行なわれます。

4. インデックスを参照する制約は、すべて新しいインデックス定義を参照するよう変更され、インデックスの名前が変更されます。この時点で、pg_index.indisvalidは新しいインデックスに対しては「true」に切り替えられ、古いものに対しては「false」に切り替えられます。そして、古いインデックスを参照するセッションをすべて無効にするためキャッシュの無効化が行なわれます。

5. 古いインデックスを参照している可能性のある実行中の問い合わせが完了するのをまってから、新しいタプルが挿入されないように古いインデックスはpg_index.indisreadyが「false」に切り替えられます。

6. 古いインデックスが削除されます。インデックスやテーブルに対するSHARE UPDATE EXCLUSIVEセッションロックは解放されます。

インデックスの再構築中に、一意性インデックスでの一意性違反などの問題が発生したら、REINDEXコマンドは失敗しますが、既に存在しているものに加えて「無効な」新しいインデックスを残します。このインデックスは不完全な可能性がありますので、問い合わせの目的では無視されます。しかし、更新のオーバーヘッドは消費し続けるでしょう。psql \dコマンドはそのようなインデックスをINVALIDと報告します。

```
postgres=# \d tab
       Table "public.tab"
 Column |  Type   | Modifiers
--------+---------+-----------
 col    | integer |
Indexes:
    "idx" btree (col)
    "idx_ccnew" btree (col) INVALID
```

INVALIDと印づけられたインデックスに接尾辞ccnewがついている場合、それは同時実行操作中に作られた一時的なインデックスに対応します。お勧めの回復法はDROP INDEXを使ってそれを削除して、再度REINDEX CONCURRENTLYを試みることです。無効なインデックスにその代わりに接尾辞ccoldがついている場合、それは削除できなかった元のインデックスに対応します。正式な再構築は成功していますので、お勧めの回復法は単に前記のインデックスを削除することです。

通常のインデックス構築は、同じテーブルでの他の通常のインデックス構築を許しますが、同時実行インデックス構築は1つだけが一度に1つのテーブルでできます。どちらの場合でも、その間のそのテーブルでの他の種類のスキーマ修正は認められていません。もう一つの違いは、通常のREINDEX TABLEやREINDEX INDEXコマンドはトランザクションブロックの内側で実行できますが、REINDEX CONCURRENTLYはできないことです。

システムカタログは同時実行で再インデックスできませんので、REINDEX SYSTEMはCONCURRENTLYをサポートしません。

さらに、排他制約に対するインデックスは同時実行で再インデックスできません。このコマンドでそのようなインデックスの名前が直接指定されたら、エラーが起きます。排他制約インデックスのあるテーブルやテータベースが同時実行で再インデックスされる場合、そのインデックスは飛ばされます。(そのようなインデックスをCONCURRENTLYオプションなしで再インデックスすることは可能です。)

例

単一のインデックスを再構築します。

```
REINDEX INDEX my_index;
```

テーブルmy_table上のすべてのインデックスを再構築します。

```
REINDEX TABLE my_table;
```

システムインデックスが有効かどうかを確認することなく、あるデータベース内の全てのインデックスを再構築します。

```
$ export PGOPTIONS="-P"
$ psql broken_db
...
broken_db=> REINDEX DATABASE broken_db;
broken_db=> \q
```

再インデックスの進行中に、関連するリレーションの読み書きをブロックすることなく、テーブルに対するインデックスを再構築します。

```
REINDEX TABLE CONCURRENTLY my_broken_table;
```

互換性

標準SQLにはREINDEXはありません。

関連項目

CREATE INDEX, DROP INDEX, reindexdb

RELEASE SAVEPOINT

RELEASE SAVEPOINT — 設定済みのセーブポイントを破棄する

概要

```
RELEASE [ SAVEPOINT ] savepoint_name
```

説明

RELEASE SAVEPOINTは、現在のトランザクションで事前に設定されていたセーブポイントを破棄します。

セーブポイントを破棄すると、ロールバックするポイントとして使用できなくなります。他にユーザの目に付くような動作はありません。このコマンドは、セーブポイントの設定後に実行されたコマンドの効果を取り消すわけではありません（これを行う方法はROLLBACK TO SAVEPOINTを参照してください）。不要になったセーブポイントを破棄することにより、システムがトランザクションの終了前に多少のリソースを回収することができます。

また、RELEASE SAVEPOINTが実行されると、指定したセーブポイントの後に設定されたセーブポイントは全て破棄されます。

パラメータ

savepoint_name

破棄するセーブポイントの名前です。

注釈

設定されていないセーブポイント名を指定するとエラーになります。

トランザクションがアボート状態の時には、セーブポイントを解放することはできません。

同じ名前のセーブポイントが複数存在する場合、最後に設定されたセーブポイントが解放されます。

例

セーブポイントを設定し、その後、破棄します。

```
BEGIN;
    INSERT INTO table1 VALUES (3);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (4);
```

```
    RELEASE SAVEPOINT my_savepoint;
COMMIT;
```

上記のトランザクションでは、3と4の両方が挿入されます。

互換性

このコマンドは標準SQLに準拠しています。SQL:2003標準では、SAVEPOINTは必須であると規定されています。PostgreSQLではSAVEPOINTキーワードを省略することができます。

関連項目

BEGIN, COMMIT, ROLLBACK, ROLLBACK TO SAVEPOINT, SAVEPOINT

RESET

RESET — 実行時パラメータの値をデフォルト値に戻す

概要

```
RESET configuration_parameter
RESET ALL
```

説明

RESETは実行時パラメータをデフォルト値に戻します。RESETは下記に対する代替の記述方法です。

```
SET configuration_parameter TO DEFAULT
```

詳細はSETを参照してください。

デフォルト値とは、現行セッション内でSETコマンドが発行されなかった場合に変数が保持していた値として定義されます。デフォルト値は、コンパイル時に指定したデフォルト、設定ファイル、コマンドラインオプション、データベースごと、ユーザごとのデフォルト設定などが元になります。これは「セッション起動時にそのパラメータが取る値」という定義と若干異なります。なぜなら、例えば設定ファイルを元にした値である場合、現在の設定ファイルによって指定される値に再設定されるからです。詳細は第19章を参照してください。

RESETのトランザクションでの振舞いはSETと同じです。この効果は、トランザクションのロールバックによって取り消されます。

パラメータ

configuration_parameter

設定可能な実行時パラメータの名前です。利用できるパラメータについては第19章で説明します。また、SETマニュアルページを参照してください。

ALL

設定可能な全ての実行時パラメータをデフォルト値に戻します。

例

timezone設定変数をデフォルト値に設定します。

```
RESET timezone;
```

互換性

RESETはPostgreSQLの拡張です。

関連項目

SET, SHOW

REVOKE

REVOKE — アクセス権限を取り消す

概要

```
REVOKE [ GRANT OPTION FOR ]
    { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }
    [, ...] | ALL [ PRIVILEGES ] }
    ON { [ TABLE ] table_name [, ...]
         | ALL TABLES IN SCHEMA schema_name [, ...] }
    FROM role_specification [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { SELECT | INSERT | UPDATE | REFERENCES } ( column_name [, ...] )
    [, ...] | ALL [ PRIVILEGES ] ( column_name [, ...] ) }
    ON [ TABLE ] table_name [, ...]
    FROM role_specification [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { USAGE | SELECT | UPDATE }
    [, ...] | ALL [ PRIVILEGES ] }
    ON { SEQUENCE sequence_name [, ...]
         | ALL SEQUENCES IN SCHEMA schema_name [, ...] }
    FROM role_specification [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { CREATE | CONNECT | TEMPORARY | TEMP } [, ...] | ALL [ PRIVILEGES ] }
    ON DATABASE database_name [, ...]
    FROM role_specification [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON DOMAIN domain_name [, ...]
    FROM role_specification [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON FOREIGN DATA WRAPPER fdw_name [, ...]
    FROM role_specification [, ...]
```

```
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON FOREIGN SERVER server_name [, ...]
    FROM role_specification [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { EXECUTE | ALL [ PRIVILEGES ] }
    ON { { FUNCTION | PROCEDURE | ROUTINE } function_name [ ( [ [ argmode ] [ arg_name ] arg_type
 [, ...] ] ) ] [, ...]
         | ALL { FUNCTIONS | PROCEDURES | ROUTINES } IN SCHEMA schema_name [, ...] }
    FROM role_specification [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON LANGUAGE lang_name [, ...]
    FROM role_specification [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { SELECT | UPDATE } [, ...] | ALL [ PRIVILEGES ] }
    ON LARGE OBJECT loid [, ...]
    FROM role_specification [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { CREATE | USAGE } [, ...] | ALL [ PRIVILEGES ] }
    ON SCHEMA schema_name [, ...]
    FROM role_specification [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { CREATE | ALL [ PRIVILEGES ] }
    ON TABLESPACE tablespace_name [, ...]
    FROM role_specification [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON TYPE type_name [, ...]
    FROM role_specification [, ...]
    [ CASCADE | RESTRICT ]
```

```
REVOKE [ ADMIN OPTION FOR ]
    role_name [, ...] FROM role_specification [, ...]
    [ GRANTED BY role_specification ]
    [ CASCADE | RESTRICT ]


ここでrole_specificationは以下の通りです。

    [ GROUP ] role_name
  | PUBLIC
  | CURRENT_USER
  | SESSION_USER
```

説明

REVOKEコマンドは、1つ以上のロールに対して、以前に与えた権限を取り消します。PUBLICキーワードは暗黙的に定義された全ロールからなるグループです。

権限の種類の意味についてはGRANTコマンドの説明を参照してください。

全てのロールは、そのロールに直接許可された権限、現在属しているロールに許可された権限、PUBLICに許可された権限という3つの権限を合わせた権限を持っていることに注意してください。したがって、例えば、PUBLICからSELECT権限を取り消すことは、必ずしも全てのロールがそのオブジェクトに対するSELECT権限を失うことを意味しません。権限が直接許可されているロール、あるいは、別ロール経由で許可されているロールは、SELECT権限を持ち続けます。同様にユーザからSELECTを取り消しても、PUBLICまたはメンバとして属する他のロールがSELECT権限を持つ場合、SELECTの使用を拒否できません。

GRANT OPTION FORが指定された場合、権限自体ではなく、その権限のグラントオプションのみが取り消されます。指定されていなければ、権限とグラントオプションの両方が取り消されます。

グラントオプション付きの権限を保持しているユーザが、その権限を他ユーザに与えていた場合、与えられたユーザが保持する権限は依存権限と呼ばれます。権限を与えたユーザ自身の権限やグラントオプションが取り消され、その権限に依存権限が存在する場合、CASCADEが指定されていると依存権限も取り消されます。指定されていなければ、権限の取り消しが失敗します。この再帰的な権限の取り消しは、ユーザ権限の連鎖を通じて与えられた権限の中でも、REVOKEを実行されたユーザから追跡可能な範囲にのみ影響します。したがって、依存権限を持つユーザが他のユーザからも同じ権限を与えられている場合は、REVOKEが実行された後もその権限を保持している可能性があります。

テーブルの権限を取り消す場合、対応する列の権限（もしあれば）も自動的に、そのテーブルの各列から取り消されます。一方、ロールがテーブルに対する権限を持つ場合、個々の列から同じ権限を取り消しても影響ありません。

ロールのメンバ資格を取り消す場合、同様に振舞いますが、GRANT OPTIONはADMIN OPTIONと呼ばれます。このコマンド構文ではGRANTED BYオプションも受け付けますが、そのオプション今のところ(その名前の存在確認を除いて)無視されます。また、このコマンド構文では、role_specificationで無意味なGROUPという単語を受け付けないことに注意してください。

注釈

取り消すことができるのは、ユーザが直接付与した権限のみです。例えば、もし、ユーザAがグラントオプションを付けてユーザBに権限を与え、その後、ユーザBがユーザCにその権限を与えたとすると、ユーザAはユーザCの権限を直接取り消すことはできません。その代わり、ユーザAがユーザBのグラントオプションをCASCADEオプション付きで取り消すことで、ユーザCに与えられた権限を取り消すことができます。別の状況を考えてみます。AとBの両方が同じ権限をCに与えた場合、AはAの与えた権限を取り消すことはできますが、Bの与えた権限を取り消すことはできません。したがって、Cは実質的にその権限を持ち続けることになります。

オブジェクトの所有者以外がそのオブジェクト上の権限に対してREVOKEを実行した場合、ユーザがオブジェクトに対して何の権限も持っていなければ、即座にコマンドが失敗します。何らかの権限があればコマンド処理が続行されますが、取り消すことができるのはそのユーザがグラントオプションを持つ権限のみです。REVOKE ALL PRIVILEGES構文をまったく権限を持たない状態で実行すると、警告が出力されます。他の構文の場合は、そのコマンドで指定した権限に対するグラントオプションを持たない状態で実行すると、警告が出力されます（原理上、上記の説明はオブジェクト所有者にも適用されますが、所有者は常に全てのグラントオプションを保持しているので、こうした問題が発生することはありません）。

スーパーユーザがGRANTやREVOKEコマンドを発行した場合、そのコマンドは、対象のオブジェクトの所有者によって発行されたものであるかのように動作します。根本的には全ての権限はオブジェクトの所有者から渡されるものなので（ただし、グラントオプションの連鎖により間接的に渡される場合もありますが）、スーパーユーザは、全ての権限を取り消すことができます。ただし、この場合は上述のCASCADEを使用する必要があります。

REVOKEは、対象のオブジェクトの所有者以外のロールによって実行することもできますが、オブジェクトを所有するロールのメンバであるか、そのオブジェクトに対しWITH GRANT OPTION権限を持つロールのメンバでなければなりません。この場合、そのオブジェクトの実際の所有者ロールまたはWITH GRANT OPTION権限を持つロールによって発行されたかのように、このコマンドは実行されます。例えば、t1テーブルがg1ロールによって所有され、u1がg1ロールのメンバであるとします。この場合、u1はg1で付与されたものと記録されている権限を取り消すことができます。これには、u1が付与した権限とg1ロールの他のメンバによって付与された権限が含まれます。

REVOKEを実行したロールが、ロールの持つ複数メンバ資格の経路を通して間接的に必要な権限を持つ場合、このコマンドがどのロールで実行されたかについては指定されません。こうした場合、SET ROLEを使用して、REVOKEを行わせたい特定のロールになることを推奨します。こうしないと、意図しない権限を取り消すことになったり、取り消し自体が失敗することになったりします。

特定の権限のより詳細な情報やオブジェクトの権限を調べる方法については5.7を参照してください。

例

テーブルfilmsに対するpublicに与えた挿入権限を取り消します。

```
REVOKE INSERT ON films FROM PUBLIC;
```

ビューkindsから、ユーザmanuelに与えた全ての権限を取り消します。

```
REVOKE ALL PRIVILEGES ON kinds FROM manuel;
```

これは実際には「自分が与えた全ての権限を取り消す」ことを意味します。

ユーザjoeからロールadmins内のメンバ資格を取り消します。

```
REVOKE admins FROM joe;
```

互換性

GRANTコマンドの互換性についての注釈はREVOKEにも同様に当てはまります。標準では、キーワードRESTRICTかCASCADEのどちらかが必須です。しかし、PostgreSQLではデフォルトでRESTRICTとみなされます。

関連項目

GRANT, ALTER DEFAULT PRIVILEGES

ROLLBACK

ROLLBACK — 現在のトランザクションをアボートする

概要

```
ROLLBACK [ WORK | TRANSACTION ] [ AND [ NO ] CHAIN ]
```

説明

ROLLBACKは現在のトランザクションをロールバックし、そのトランザクションで行われた全ての更新を廃棄させます。

パラメータ

WORK
TRANSACTION

省略可能なキーワードです。効果は何もありません。

AND CHAIN

AND CHAINが指定されていれば、新しいトランザクションは、直前に終了したものと同じトランザクションの特性(SET TRANSACTIONを参照してください)で即時に開始されます。そうでなければ、新しいトランザクションは開始されません。

注釈

トランザクションを正常に終了させるにはCOMMITを使用してください。

トランザクションブロックの外部でROLLBACKを発行すると警告が発生しますが、それ以外は何の効果もありません。トランザクションブロックの外部でROLLBACK AND CHAINを発行するとエラーになります。

例

全ての変更をアボートします。

```
ROLLBACK;
```

互換性

コマンドROLLBACKは標準SQLに準拠しています。ROLLBACK TRANSACTIONの構文はPostgreSQLでの拡張です。

関連項目

BEGIN, COMMIT, ROLLBACK TO SAVEPOINT

ROLLBACK PREPARED

ROLLBACK PREPARED — 二相コミット用に事前に準備されたトランザクションを取り消す

概要

```
ROLLBACK PREPARED transaction_id
```

説明

ROLLBACK PREPAREDは、準備された状態のトランザクションをロールバックします。

パラメータ

transaction_id

ロールバックさせるトランザクションのトランザクション識別子です。

注釈

準備されたトランザクションをロールバックするには、トランザクションを元々実行したユーザかスーパーユーザでなければなりません。しかし、トランザクションを実行したのと同じセッション内で実行する必要はありません。

このコマンドはトランザクションブロック内では実行できません。準備されたトランザクションは即座にロールバックされます。

現在利用できるすべての準備されたトランザクションはpg_prepared_xactsシステムビュー内に列挙されています。

例

トランザクション識別子foobarで識別されるトランザクションをロールバックします。

```
ROLLBACK PREPARED 'foobar';
```

互換性

ROLLBACK PREPAREDはPostgreSQLの拡張です。これは外部のトランザクション管理システムによる利用を意図したものです。ただし外部のトランザクション管理システムの中には標準化されたもの(X/Open XAなど)もありますが、こうしたシステムでもSQL側は標準化されていません。

関連項目

PREPARE TRANSACTION, COMMIT PREPARED

ROLLBACK TO SAVEPOINT

ROLLBACK TO SAVEPOINT — セーブポイントまでロールバックする

概要

```
ROLLBACK [ WORK | TRANSACTION ] TO [ SAVEPOINT ] savepoint_name
```

説明

セーブポイントの設定後に実行されたコマンドを全てロールバックします。セーブポイントは有効なまま残るので、必要に応じて、その後再度ロールバックすることができます。

ROLLBACK TO SAVEPOINTは、指定したセーブポイントより後に設定した全てのセーブポイントを暗黙的に破棄します。

パラメータ

savepoint_name

ロールバック先のセーブポイントです。

注釈

セーブポイントの設定後に実行されたコマンドの結果を維持したままセーブポイントを破棄するには、RELEASE SAVEPOINTを使用してください。

設定されていないセーブポイントの名前を指定するとエラーになります。

カーソルはセーブポイントという観点から見るとトランザクションの外にあるかのように振舞います。セーブポイントの内部で開かれたカーソルは全て、そのセーブポイントがロールバックした時に閉ざされます。セーブポイントの前に開かれたカーソルに対しセーブポイント内でFETCHまたはMOVEコマンドを実行した場合、その後、ロールバックされたとしても、カーソルの位置はFETCHの結果、移動した位置から変わりません（つまりFETCHによる位置の移動はロールバックされません）。また、カーソルのクローズはロールバックしても取り消すことはできません。しかしカーソルの問い合わせにより発生するその他の副作用（問い合わせにより呼出される揮発性関数の影響など）は、セーブポイント内で実行され、それがロールバックされた場合に、ロールバックされます。カーソルの実行によってトランザクションのアボートが引き起こされた場合、そのカーソルは実行不可能状態に遷移します。この場合、トランザクションはROLLBACK TO SAVEPOINTを使用して戻すことができますが、そのカーソルは使用することができません。

例

my_savepointの設定後に実行されたコマンドの効果を取り消します。

```
ROLLBACK TO SAVEPOINT my_savepoint;
```

セーブポイントへのロールバックは、カーソル位置に影響を与えません。

```
BEGIN;

DECLARE foo CURSOR FOR SELECT 1 UNION SELECT 2;

SAVEPOINT foo;

FETCH 1 FROM foo;
 ?column?
----------
        1

ROLLBACK TO SAVEPOINT foo;

FETCH 1 FROM foo;
 ?column?
----------
        2

COMMIT;
```

互換性

標準SQLでは、SAVEPOINTキーワードは必須です。しかし、PostgreSQLとOracleでは省略することができます。SQLで使用できるのは、WORKのみです。TRANSACTIONは使用できず、ROLLBACKの後の意味のない言葉として扱われます。また、SQLではAND [NO] CHAIN句(省略可能)がありますが、これはPostgreSQLでは現在サポートされていません。その他については、このコマンドは標準SQLと互換性を持ちます。

関連項目

BEGIN, COMMIT, RELEASE SAVEPOINT, ROLLBACK, SAVEPOINT

SAVEPOINT

SAVEPOINT — 現在のトランザクション内に新規にセーブポイントを定義する

概要

```
SAVEPOINT savepoint_name
```

説明

SAVEPOINTは、現在のトランザクション内に新しいセーブポイントを設定します。

セーブポイントとはトランザクション内に付ける特別な印です。セーブポイントを設定しておくと、それ以降に実行されたコマンドを全てロールバックし、トランザクションを設定時の状態に戻すことができます。

パラメータ

`savepoint_name`

新しいセーブポイントに付与する名前です。

注釈

セーブポイントまでロールバックするにはROLLBACK TO SAVEPOINTを使用してください。セーブポイント後に行われたコマンドの効果を保持したままセーブポイントを破棄するには、RELEASE SAVEPOINTを使用してください。

セーブポイントはトランザクションブロックの内側のみに設定することができます。1つのトランザクションの中には、複数のセーブポイントを設定することができます。

例

セーブポイントを設定し、その後に実行した全てのコマンドの効果を取り消します。

```
BEGIN;
    INSERT INTO table1 VALUES (1);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (2);
    ROLLBACK TO SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (3);
COMMIT;
```

上記のトランザクションでは、1と3は挿入されますが、2は挿入されません。

セーブポイントを設定し、その後に破棄します。

```
BEGIN;
    INSERT INTO table1 VALUES (3);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (4);
    RELEASE SAVEPOINT my_savepoint;
COMMIT;
```

上記のトランザクションでは、3と4の両方が挿入されます。

互換性

SQLでは、同じ名前のセーブポイントが設定された時は、自動的に古い方のセーブポイントを破棄することになっています。PostgreSQLでは、古いセーブポイントも保持されますが、ロールバックや解放時には新しい方のセーブポイントが使用されます（RELEASE SAVEPOINTを用いて新しいセーブポイントが解放されると、再びROLLBACK TO SAVEPOINTやRELEASE SAVEPOINTから古いセーブポイントが使用できるようになります）。この点以外は、SAVEPOINTは完全にSQLに従っています。

関連項目

BEGIN, COMMIT, RELEASE SAVEPOINT, ROLLBACK, ROLLBACK TO SAVEPOINT

SECURITY LABEL

SECURITY LABEL — オブジェクトに適用するセキュリティラベルを定義または変更する

概要

```
SECURITY LABEL [ FOR provider ] ON
{
  TABLE object_name |
  COLUMN table_name.column_name |
  AGGREGATE aggregate_name ( aggregate_signature ) |
  DATABASE object_name |
  DOMAIN object_name |
  EVENT TRIGGER object_name |
  FOREIGN TABLE object_name
  FUNCTION function_name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ] |
  LARGE OBJECT large_object_oid |
  MATERIALIZED VIEW object_name |
  [ PROCEDURAL ] LANGUAGE object_name |
  PROCEDURE procedure_name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ] |
  PUBLICATION object_name |
  ROLE object_name |
  ROUTINE routine_name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ] |
  SCHEMA object_name |
  SEQUENCE object_name |
  SUBSCRIPTION object_name |
  TABLESPACE object_name |
  TYPE object_name |
  VIEW object_name
} IS 'label'


ここでaggregate_signatureは以下の通りです。

* |
[ argmode ] [ argname ] argtype [ , ... ] |
[ [ argmode ] [ argname ] argtype [ , ... ] ] ORDER BY [ argmode ] [ argname ] argtype [ , ... ]
```

説明

SECURITY LABELはセキュリティラベルをデータベースオブジェクトに適用します。ラベルプロバイダごとに1つの、任意の数のセキュリティラベルを指定したデータベースオブジェクトに関連付けることができます。ラベルプロバイダは、register_label_provider関数を使用して自身を登録する、ロード可能なモジュールです。

> **注記**
>
> register_label_providerはSQL関数ではありません。バックエンドにロードされたCコードからのみ呼び出すことができます。

ラベルプロバイダは、指定されたラベルが有効かどうか、および指定されたオブジェクトにラベルを割り当てることが許されているかどうかを決定します。また、ラベルプロバイダは指定されたラベルの意味の決定権を持ちます。PostgreSQLは、ラベルプロバイダがセキュリティラベルを解釈するかしないか、どのように解釈するかに関して制限を持ちません。単にこれらを格納するための機構を提供するだけです。実際には、この機能はSELinuxなどのラベルベースの強制アクセス制御（MAC）システムと統合できるようにすることを意図したものです。こうしたシステムでは、すべてのアクセス制御の決定は、ユーザとグループなどの伝統的な任意アクセス制御（DAC）という考えではなく、オブジェクトラベルに基づいて行われます。

パラメータ

object_name
table_name.column_name
aggregate_name
function_name
procedure_name
routine_name

ラベル付けされるオブジェクトの名前です。テーブル、集約、ドメイン、外部テーブル、関数、プロシージャ、ルーチン、シーケンス、型、ビューの名前はスキーマ修飾可能です。

provider

このラベルが関連するプロバイダの名前です。指定されたプロバイダはロードされていなければならず、かつ、提供されるラベル付け操作と一致しなければなりません。プロバイダが1つだけロードされていた場合、プロバイダの名前を省略して簡略化することができます。

argmode

関数、プロシージャ、または集約の引数のモードです。IN、OUT、INOUT、VARIADICのいずれかです。省略された場合のデフォルトはINです。関数の識別を決定するためには入力引数のみが必要ですので、実際にはSECURITY LABELはOUTをまったく考慮しないことに注意してください。このためIN、INOUT、VARIADICのリストで十分です。

argname

関数、プロシージャ、または集約の引数の名前です。関数の識別を決定するためには引数のデータ型のみが必要ですので、実際にはSECURITY LABEL ON FUNCTIONは引数名をまったく考慮しないことに注意してください。

argtype

関数、プロシージャ、または集約の引数のデータ型です。

large_object_oid

ラージオブジェクトのOIDです。

PROCEDURAL

これは意味がない単語です。

label

文字列リテラルで記述された新しいセキュリティラベルです。セキュリティラベルを削除するためにはNULLと記述します。

例

以下の例はテーブルのセキュリティラベルを変更する方法を示します。

```
SECURITY LABEL FOR selinux ON TABLE mytable IS 'system_u:object_r:sepgsql_table_t:s0';
```

互換性

標準SQLにはSECURITY LABELコマンドはありません。

関連項目

sepgsql, src/test/modules/dummy_seclabel

SELECT

SELECT, TABLE, WITH — テーブルもしくはビューから行を検索する

概要

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    [ * | expression [ [ AS ] output_name ] [, ...] ]
    [ FROM from_item [, ...] ]
    [ WHERE condition ]
    [ GROUP BY grouping_element [, ...] ]
    [ HAVING condition ]
    [ WINDOW window_name AS ( window_definition ) [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ] select ]
    [ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST | LAST } ] [, ...] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start [ ROW | ROWS ] ]
    [ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } { ONLY | WITH TIES } ]
    [ FOR { UPDATE | NO KEY UPDATE | SHARE | KEY SHARE } [ OF table_name [, ...] ] [ NOWAIT |
 SKIP LOCKED ] [...] ]


ここでfrom_itemは以下のいずれかです。

    [ ONLY ] table_name [ * ] [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
                [ TABLESAMPLE sampling_method ( argument [, ...] ) [ REPEATABLE ( seed ) ] ]
    [ LATERAL ] ( select ) [ AS ] alias [ ( column_alias [, ...] ) ]
    with_query_name [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
    [ LATERAL ] function_name ( [ argument [, ...] ] )
                [ WITH ORDINALITY ] [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
    [ LATERAL ] function_name ( [ argument [, ...] ] ) [ AS ] alias ( column_definition [, ...] )
    [ LATERAL ] function_name ( [ argument [, ...] ] ) AS ( column_definition [, ...] )
    [ LATERAL ] ROWS FROM( function_name ( [ argument [, ...] ] ) [ AS ( column_definition
 [, ...] ) ] [, ...] )
                [ WITH ORDINALITY ] [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
    from_item [ NATURAL ] join_type from_item [ ON join_condition | USING ( join_column
 [, ...] ) ]


またgrouping_elementは以下のいずれかです。

    ( )
```

```
    expression
    ( expression [, ...] )
    ROLLUP ( { expression | ( expression [, ...] ) } [, ...] )
    CUBE ( { expression | ( expression [, ...] ) } [, ...] )
    GROUPING SETS ( grouping_element [, ...] )


またwith_queryは以下の通りです。

    with_query_name [ ( column_name [, ...] ) ] AS [ [ NOT ] MATERIALIZED ] ( select | values
 | insert | update | delete )

TABLE [ ONLY ] table_name [ * ]
```

説明

SELECTは0個以上のテーブルから行を返します。SELECTの一般的な処理は以下の通りです。

1. WITHリスト内のすべての問い合わせが計算されます。これらは実質的には、FROMリスト内から参照可能な一時テーブルとして提供されます。NOT MATERIALIZEDが指定された場合を除き、FROM内で2回以上参照されるWITH問い合わせは一度のみ計算されます。(後述のWITH Clauseを参照してください。)

2. FROMリストにある全要素が計算されます (FROMリストの要素は実テーブルか仮想テーブルのいずれかです)。FROMリストに複数の要素が指定された場合、それらはクロス結合されます (後述のFROM Clauseを参照してください)。

3. WHERE句が指定された場合、条件を満たさない行は全て出力から取り除かれます (後述のWHERE Clauseを参照してください)。

4. GROUP BY句が指定された場合、および集約関数の呼び出しがある場合は、1つまたは複数の値が条件に合う行ごとにグループに組み合わせて出力され、また集約関数の結果が計算されます。HAVING句が指定された場合、指定した条件を満たさないグループは取り除かれます (後述のGROUP BY ClauseとHAVING Clauseを参照してください)。

5. 実際には、選択された各行または行グループに対して、SELECTの出力式を使用して計算した結果の行が出力されます (後述のSELECT Listを参照してください)。

6. SELECT DISTINCTは結果から重複行を取り除きます。SELECT DISTINCT ONは指定した全ての式に一致する行を取り除きます。SELECT ALLでは、重複行も含め、全ての候補行を返します(これがデフォルトです。詳しくは、後述のDISTINCT Clauseを参照してください)。

7. UNION、INTERSECT、EXCEPT演算子を使用すると、複数のSELECT文の出力を1つの結果集合にまとめることができます。UNION演算子は、両方の結果集合に存在する行と、片方の結果集合に存在する行を全て返します。INTERSECT演算子は、両方の結果集合に存在する行を返します。EXCEPT演算子は、最初の結果集合にあり、2番目の結果集合にない行を返します。ALLが指定されない限り、いずれの場合も、重複する行は取り除かれます。無意味なDISTINCTという単語を付けて、明示的に重複行を除去することを指定することができます。SELECT自体はALLがデフォルトですが、この場合はDISTINCTがデフォルトの動作

であることに注意してください。(後述のUNION Clause、INTERSECT Clause、EXCEPT Clauseを参照してください。)

8. ORDER BY句が指定された場合、返される行は指定した順番でソートされます。ORDER BYが指定されない場合は、システムが計算過程で見つけた順番で行が返されます (後述のORDER BY Clauseを参照してください)。

9. LIMIT(またはFETCH FIRST)あるいはOFFSET句が指定された場合、SELECT文は結果行の一部分のみを返します (詳しくは、後述のLIMIT Clauseを参照してください)。

10. FOR UPDATE、FOR NO KEY UPDATE、FOR SHAREまたはFOR KEY SHARE句を指定すると、SELECT文は引き続き行われる更新に備えて選択行をロックします (詳しくは、後述のThe Locking Clauseを参照してください)。

SELECTコマンド内で使われる列それぞれに対するSELECT権限が必要です。FOR NO KEY UPDATE、FOR UPDATE、FOR SHAREまたはFOR KEY SHAREを使用するためには、さらに、(選択された各テーブルで少なくとも1列に対する)UPDATE権限が必要です。

パラメータ

WITH句

WITH句により主問い合わせ内で名前により参照可能な、1つ以上の副問い合わせを指定することができます。副問い合わせは実質的に主問い合わせの間の一時的なテーブルかビューのように動作します。各副問い合わせはSELECT、TABLE、VALUES、INSERT、UPDATE、DELETEにすることができます。WITH内でデータ変更文(INSERT、UPDATE、DELETE)を記述する場合は、RETURNING句を含めるのが普通です。主問い合わせで読み取られる一時テーブルを形成するのは、RETURNINGの出力であり、文が変更する背後のテーブルではありません。RETURNINGを省いても文は実行されますが、出力を生成しませんので、主問い合わせでテーブルとして参照することができません。

(スキーマ修飾がない)名前を各WITH問い合わせで指定しなければなりません。列名のリストをオプションで指定することもできます。これを省略すると、列名は副問い合わせから推定されます。

RECURSIVEが指定されると、SELECT副問い合わせは自身で名前により参照することができます。こうした副問い合わせは以下のような形式でなければなりません。

```
non_recursive_term UNION [ ALL | DISTINCT ] recursive_term
```

ここで再帰的な自己参照はUNIONの右辺に現れなければなりません。問い合わせ当たり1つの再帰的な自己参照のみが許されます。再帰的なデータ変更文はサポートされていませんが、データ変更文で再帰的なSELECTの結果を使用することができます。例は7.8を参照してください。

RECURSIVEには他にも、WITH問い合わせが順序通りでなくても構わないという効果があります。つまり、問い合わせはリストの後にある別のものを参照することができます。(しかし巡回する参照や相互的な参照は実装されていません。) RECURSIVEがないと、WITH問い合わせは主問い合わせが共通するWITH問い合わせのうち、WITHリストの前方にあるもののみを参照することができます。

WITH句に複数の問い合わせがある場合、RECURSIVEはWITHの直後に一度だけ書くべきです。再帰や前方参照を使わない問い合わせには効果はないですが、WITH句内の問い合わせすべてに適用されます。

主問い合わせとWITH問い合わせは（理論的には）同時に実行されます。このことは、WITH中のデータ更新文の効果は、RETURNING出力の読み込みを行ったことによるものを除き、問い合わせ中の他の部分から見えないことを意味します。2つのそうしたデータ更新文が同じ行を更新しようとした時の結果は不定です。

WITH問い合わせの重要な特性は、これらを主問い合わせが複数回参照していたとしても、主問い合わせの実行当たり通常一度のみ評価される点です。特にデータ変更文は、主問い合わせがその出力のすべてまたは一部を読み取るかに関係なく、本当に一度のみ実行されることが保証されています。

しかし、WITH問い合わせにNOT MATERIALIZEDと印を付けることにより、この保証を取り除くことができます。その場合、WITH問い合わせは、主問い合わせのFROM句中の単純な副SELECTであるかのように可能な限り主問い合わせ中に畳み込むことができます。この結果、主問い合わせがWITH問い合わせを複数回参照している場合には複数回の計算が行われます。しかし、そこで使用される問い合わせがWITH問い合わせ全体の出力のうちの数行しか必要としないなら、NOT MATERIALIZEDは問い合わせを連携して最適化することができるので、全体のコストの節約ができます。再帰問い合わせあるいは副作用のある問い合わせ（すなわち揮発性の関数を含まない単純SELECTではないもの）にNOT MATERIALIZEDを適用しても無視されます。

デフォルトでは、主問い合わせ中のFROM句で正確に一度だけ使われているなら、副作用のないWITH問い合わせは主問い合わせに畳み込まれます。これにより意味論的に不可視の二つの問い合わせレベルが共同して最適化されることを可能にします。しかし、WITH問い合わせにMATERIALIZEDと印を付けることにより、そうした畳込みを防ぐことができます。たとえば、プランナが悪いプランを選択するのを防ぐために最適化障壁としてWITH問い合わせを使っている場合にこれは有用です。PostgreSQLバージョン12よりも前ではそうした畳込みは決して行われていなかったので、古いバージョン用に書かれた問い合わせはWITHが最適化障壁として働くことに依存しているかもしれません。

追加情報については7.8を参照してください。

FROM句

FROM句にはSELECTの対象となるソーステーブルを1つ以上指定します。複数のソースが指定された場合、結果は全てのソースの直積（クロス結合）となります。しかし、通常は（WHEREを介して）制約条件を付けて、直積のごく一部を返すように結果行を限定します。

FROM句には以下の要素を指定できます。

`table_name`

> 既存のテーブルもしくはビューの名前です（スキーマ修飾名も可）。テーブル名の前にONLYが指定された場合、そのテーブルのみがスキャンされます。ONLYが指定されない場合、テーブルと（もしあれば）それを継承する全てのテーブルがスキャンされます。省略することもできますが、テーブル名の後に*を指定することで、明示的に継承するテーブルも含まれることを示すことができます。

`alias`

> 別名を含むFROM項目の代替名です。別名は、指定を簡潔にするため、もしくは、自己結合（同じテーブルを複数回スキャンする結合）の曖昧さをなくすために使われます。別名が指定されている場合は、そ

の別名によって実際のテーブル名または関数名が完全に隠されます。例えば、FROM foo AS fと指定されている場合、SELECT文の以降の部分ではこのFROM項目をfooではなくfとして参照する必要があります。テーブルの別名があれば、そのテーブルの複数の列の名前を置き換える列の別名リストを記述することができます。

TABLESAMPLE sampling_method (argument [, ...]) [REPEATABLE (seed)]

table_nameの後のTABLESAMPLE句は、そのテーブルの行の部分集合を取り出すときに、指定したsampling_methodを使うべきであることを示唆します。このサンプリングはWHEREなど他のすべてのフィルタの適用に先立って行われます。PostgreSQLの標準ディストリビューションには、BERNOULLIとSYSTEMの2つのサンプリングメソッドが含まれています。他のサンプリングメソッドも拡張(extension)によりデータベースにインストールすることができます。

サンプリングメソッドBERNOULLIとSYSTEMはいずれも1つだけargumentを取り、これはテーブルからサンプリングする割合で0から100までのパーセントで表現されます。この引数はreal型の値を取る任意の式にできます。(他のサンプリングメソッドは、複数の、あるいは異なる引数を受け取るかもしれません。) これら2つの方法はいずれも、テーブルのうち指定された割合に近い行数を含む、ランダムに選択されたサンプルテーブルを返します。BERNOULLIでは、テーブル全体を走査し、個々の行を別々に、指定された確率に従って、選択あるいは無視します。SYSTEMではブロックレベルのサンプリングを行います。各ブロックは指定された確率で選択され、選択されたブロック内のすべての行が返されます。サンプリングに小さな割合が指定された場合、SYSTEMはBERNOULLIよりもかなり高速ですが、クラスタリング効果により、BERNOULLIに比べてランダムでないサンプルを返すかもしれません。

オプションのREPEATABLE句では、サンプリングメソッドで乱数を生成するためのseedの数あるいは式を指定します。シード値はNULL以外の任意の浮動点小数値とすることができます。シードとargumentの値が同じ2つの問い合わせは、その間にテーブルに変更がなければ、同じサンプルテーブルを返します。しかし、シードの値が異なれば、通常は異なるサンプルが生成されます。REPEATABLEが指定されていなければ、システムが生成したシードに基づいて、問い合わせ毎に新しくランダムなサンプルが生成されます。一部のアドオンのサンプリングメソッドではREPEATABLEが利用できず、使用の度に常に新しいサンプルを生成することに注意してください。

select

FROM句では、副SELECTを使うことができます。SELECTコマンドの実行中、副SELECTの出力は一時テーブルであるかのように動作します。副SELECTは括弧で囲まれなければなりません。また、必ず別名を与えなければなりません。VALUESコマンドをここで使用することもできます。

with_query_name

WITH問い合わせは、問い合わせの名前があたかもテーブル名であるかのように、名前を記述することで参照されます。(実際にはWITH問い合わせは主問い合わせの対象とするテーブルと同じ名前の実テーブルを隠蔽します。必要ならばテーブル名をスキーマ修飾することで同じ名前の実テーブルを参照することができます。) テーブルと同様の方法で別名を提供することができます。

function_name

FROM句では、関数呼び出しを使用することができます (これは特に関数が結果セットを返す場合に有用ですが、任意の関数を使用することもできます)。SELECTコマンドの実行中は、この関数の結果は一時テーブルであるかのように動作します。関数呼び出しにWITH ORDINALITY句を追加した時は、すべての関数の出力列の後に各行の番号の列が追加されます。

テーブルに対するのと同じように、別名を使用することができます。別名が記述されていれば、列の別名リストを記述して、関数の複合型の戻り値の1つ以上の、ORDINALITYがある場合はそれが追加する列を含め、属性に対する代替名を提供することもできます。

複数の関数呼び出しをROWS FROM(...)で括ることにより、1つのFROM句の項目にまとめることができます。このような項目の出力は各関数の最初の行を結合した項目、次いで各関数の2番目の行、といった具合になります。一部の関数が他の関数より少ない行数を出力した場合は、存在しないデータについてNULL値が代用され、戻される行数はいつでも最大の行数を返した関数と同じになります。

関数がrecordデータ型を返すと定義されている場合は、別名すなわちASキーワードと、それに続く（column_name data_type [, ...]）という形式の列定義リストが必要です。列定義リストは、関数によって返される実際の列の数およびデータ型に一致していなければなりません。

ROWS FROM(...)の構文を使う時、関数の1つが列定義のリストを必要としている場合は、ROWS FROM(...)内の関数呼び出しの後に列定義のリストを置くのが望ましいです。関数が1つだけで、WITH ORDINALITY句がない場合に限り、列定義のリストをROWS FROM(...)の後に置くことができます。

ORDINALITYを列定義のリストと一緒に使うには、ROWS FROM(...)構文を使い、列定義のリストをROWS FROM(...)の内側に置かなければなりません。

join_type

以下のいずれかです。

- [INNER] JOIN
- LEFT [OUTER] JOIN
- RIGHT [OUTER] JOIN
- FULL [OUTER] JOIN
- CROSS JOIN

INNERおよびOUTER結合型では、結合条件、すなわち、NATURAL, ON join_condition、USING (join_column [, ...])のいずれか1つのみを指定する必要があります。それぞれの意味は後述します。CROSS JOINでは、これらの句を記述しなくても構いません。

JOIN句は、2つのFROM項目を結び付けます。便宜上「テーブル」と呼びますが、実際には任意の種類のFROM項目とすることができます。入れ子の順番を決めるために、必要ならば括弧を使用してください。括弧がないと、JOINは左から右へ入れ子にします。どのような場合でもJOINは、カンマで分けられたFROM項目よりも強い結び付きを持ちます。

CROSS JOINとINNER JOINは直積を1つ生成します。これは、FROMの最上位で2つのテーブルを結合した結果と同一です。しかし、（指定すれば）結合条件によって制限をかけることができます。CROSS JOINはINNER JOIN ON (TRUE)と等価であり、条件によって削除される行はありません。これらの結合型は記述上の便宜のためだけに用意されています。なぜなら、通常のFROMとWHEREでできないことは何もしないからです。

LEFT OUTER JOINは、条件に合う直積の全ての行（つまり、その結合条件を満たす全ての組み合わせ）に加え、左側テーブルの中で、右側テーブルには結合条件を満たす行が存在しなかった行のコピーも返します。この左側テーブルの行を結合結果のテーブルの幅に拡張するために、右側テーブルが入る列には

NULL値が挿入されます。マッチする行を決める時は、JOIN句自身の条件のみが考慮されることに注意してください。外部結合条件は後で適用されます。

逆に、RIGHT OUTER JOINは、全ての結合行と、左側テーブルに当てはまるものがなかった右側の行(左側はNULLで拡張されています)の1行ずつを返します。左右のテーブルを入れ替えればLEFT OUTER JOINに変換できるので、RIGHT OUTER JOINは記述上の便宜を図るため用意されているに過ぎません。

FULL OUTER JOINは、全ての結合行に加え、一致しなかった左側の行(右側はNULLで拡張)、一致しなかった右側の行(左側はNULLで拡張)を全て返します。

ON join_condition

join_conditionは、結合においてどの行が一致するかを指定する、boolean型の値を返す式です(WHERE句に類似しています)。

USING (join_column [, ...])

USING (a, b, ...)という形式の句はON left_table.a = right_table.a AND left_table.b = right_table.b ...の省略形です。またUSINGは等価な列の両方ではなく片方のみが結合の出力に含まれることを意味します。

NATURAL

NATURALは、2つのテーブル内の同じ名前を持つ列を全て指定したUSINGリストの省略形です。共通の列名がない場合、NATURALはON TRUEと同等になります。

LATERAL

LATERALキーワードを副SELECTのFROM項目の前に付けることができます。これにより、副SELECTがFROMリストの中で前に現れるFROM項目の列を参照することができます。(LATERALがないと、副SELECTそれぞれが個別に評価され、他のFROM項目とのクロス参照を行うことができません。)

LATERALを関数を呼び出すFROMの前に付けることもできます。しかしこの場合、無意味な単語になります。関数式はどのような場合でもより前のFROM項目を参照することができるからです。

LATERAL項目はFROMの最上位レベルやJOINツリー内に記述することができます。後者の場合、JOINの右辺にあれば、左辺にある任意の項目を参照することができます。

FROM項目がLATERALクロス参照を含む場合、評価は次のように行われます。クロス参照される列を提供するFROM項目の各行、または、その列を提供する複数のFROM項目の行集合に対して、LATERAL項目は列の行または行集合を使用して評価されます。結果となる行は、計算された行と通常通り結合されます。これが各行または列ソーステーブルからの行集合に対して繰り返されます。

列ソーステーブルはLATERAL項目とINNERまたはLEFT結合されていなければなりません。さもないと、LATERAL項目において各行集合を計算するための行集合が完全に定義することができません。したがってX RIGHT JOIN LATERAL Yという式は構文としては有効ですが、実際にはYではXを参照することができません。

WHERE句

WHERE句の一般的な構文は以下の通りです(この句は省略可能です)。

```
WHERE condition
```

conditionは、評価の結果としてboolean型を返す任意の式です。この条件を満たさない行は全て出力から取り除かれます。全ての変数に実際の行の値を代入して、式が真を返す場合、その行は条件を満たすとみなされます。

GROUP BY句

GROUP BY句の一般的な構文は以下の通りです(この句は省略可能です)。

```
GROUP BY grouping_element [, ...]
```

GROUP BYは、グループ化のために与えられた式を評価し、結果が同じ値になった行を1つの行にまとめる機能を持ちます。grouping_elementの内側で使われるexpressionには、入力列の名前、出力列(SELECTリスト項目)の名前/序数、あるいは入力列の値から計算される任意の式を取ることができます。判断がつかない時は、GROUP BYの名前は出力列名ではなく入力列名として解釈されます。

グループ化の要素としてGROUPING SETS、ROLLUP、CUBEのいずれかが指定されている場合、GROUP BY句は全体でいくつかの独立したグループ化セットを定義します。この効果は、個々のグループ化セットをGROUP BY句で定義する副問い合わせをUNION ALLするのと同等です。グループ化セットの処理の詳細については、7.2.4を参照してください。

集約関数が使用された場合、各グループ内の全ての行を対象に計算が行われ、グループごとに別々の値が生成されます(集約関数が使われていてGROUP BYがない場合、その問い合わせは選択された全ての行からなる1つのグループを持つものとして扱われます)。集約関数の入力となる行の集合は、集約関数の呼び出しにFILTER句を付けることで、さらに絞り込むことができます。詳しくは4.2.7を参照してください。FILTER句があると、その条件に適合する行だけが集約関数の入力行に取り込まれます。

GROUP BYが存在する場合、あるいは集約関数が存在する場合、集約関数内部以外で、グループ化されていない列を参照する、あるいはグループ化されていない列がグループ化された列に関数依存するSELECTリストの式は無効になります。こうしないとグループ化されていない列について返される値は複数の値になってしまう可能性があるからです。グループ化された列(またはその部分集合)がグループ化されていない列を含むテーブルの主キーである場合、関数従属性が存在します。

すべての集約関数は、HAVING句やSELECTリストのどの「スカラー」式よりも先に評価されることに注意してください。これは例えば、CASE式を集約関数の評価をスキップするために使うことはできない、ということを意味します。4.2.14を参照してください。

現在は、FOR NO KEY UPDATE、FOR UPDATE、FOR SHARE、FOR KEY SHAREをGROUP BYと合わせて使うことはできません。

HAVING句

HAVING句の一般的な構文は以下の通りです(この句は省略可能です)。

```
HAVING condition
```

conditionはWHERE句で指定するものと同じです。

HAVINGは、グループ化された行の中で、条件を満たさない行を取り除く機能を持ちます。HAVINGとWHEREは次の点が異なります。WHEREが、GROUP BYの適用前に個々の行に対してフィルタを掛けるのに対し、HAVINGは、GROUP BYの適用後に生成されたグループ化された行に対してフィルタをかけます。condition内で使用する列は、集約関数内で使用される場合とグループ化されない列がグループ化される列に関数依存する場合を除き、グループ化された列を一意に参照するものでなければなりません。

HAVING句があると、GROUP BY句がなかったとしても問い合わせはグループ化された問い合わせになります。GROUP BY句を持たない問い合わせが集約関数を含む場合と同様です。選択された行はすべて、1つのグループを形成するものとみなされます。また、SELECTリストとHAVING句では、集約関数が出力するテーブル列しか参照することができません。こうした問い合わせでは、HAVINGが真の場合には単一の行を、真以外の場合は0行を出力します。

現在は、FOR NO KEY UPDATE、FOR UPDATE、FOR SHARE、FOR KEY SHAREをHAVINGと合わせて使うことはできません。

WINDOW句

WINDOW句(省略可能)の一般的な構文は以下の通りです。

```
WINDOW window_name AS ( window_definition ) [, ...]
```

ここでwindow_nameは、OVER句やこの後のウィンドウ定義で参照することができる名前です。また、window_definitionは以下の通りです。

```
[ existing_window_name ]
[ PARTITION BY expression [, ...] ]
[ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST | LAST } ] [, ...] ]
[ frame_clause ]
```

existing_window_nameを指定する場合、それはWINDOWリスト内のそれより前にある項目を参照しなければなりません。新しいウィンドウはそのPARTITION BY句をその項目からコピーします。ORDER BY句があった場合も同様です。この場合、新しいウィンドウでは独自のPARTITION BY句を指定することはできません。また、コピーされたウィンドウがORDER BYを持たない場合のみORDER BYを指定することができます。新しいウィンドウは常に独自のフレーム句を使用します。コピーされたウィンドウはフレーム句を指定してはなりません。

PARTITION BYリストの要素はGROUP BY句の要素とほとんど同じように解釈されます。ただし、こちらは常に単純な式であり、出力列の名前や番号ではないことが異なります。他にも違いがあり、これらの式は、通常のGROUP BY句では許されない、集約関数を含めることができるという点です。グループ化および集約処理の後にウィンドウ処理が動作するため、これらでは許されています。

同様に、ORDER BYリストの要素は文レベルのORDER BY句の要素とほとんど同じように解釈されます。ただし、この式は常に単純な式であり、出力列の名前や番号ではないことが異なります。

frame_clauseを指定すると、(すべてではありませんが)フレームに依存するウィンドウ関数用のウィンドウフレームを定義できます。ウィンドウフレームは、問い合わせの各行(現在の行と呼ばれます)に関連する行の集合です。frame_clauseは以下のいずれかを取ることができます。

```
{ RANGE | ROWS | GROUPS } frame_start [ frame_exclusion ]
{ RANGE | ROWS | GROUPS } BETWEEN frame_start AND frame_end [ frame_exclusion ]
```

ここでframe_startとframe_endは以下のいずれかを取ることができます。

```
UNBOUNDED PRECEDING
offset PRECEDING
CURRENT ROW
offset FOLLOWING
UNBOUNDED FOLLOWING
```

また、frame_exclusionには以下のいずれかを取ることができます。

```
EXCLUDE CURRENT ROW
EXCLUDE GROUP
EXCLUDE TIES
EXCLUDE NO OTHERS
```

frame_endが省略された場合、デフォルトでCURRENT ROWとなります。frame_startはUNBOUNDED FOLLOWINGとすることができない、frame_endはUNBOUNDED PRECEDINGとすることができない、また、frame_startとframe_endのオプションの上記リストでframe_endの選択をframe_startの選択よりも手前に現れるものにはできない、という制限があります。例えばRANGE BETWEEN CURRENT ROW AND offset PRECEDINGは許されません。

デフォルトのフレーム化オプションはRANGE UNBOUNDED PRECEDINGです。これはRANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROWと同じで、パーティションの先頭から現在の行の最後のピア（ウィンドウのORDER BY句が現在行と同等とみなす行、ORDER BYが無ければ全ての行がピア）までのすべての行をフレームとします。一般的に、RANGEやROWS、GROUPSのモードにかかわらず、UNBOUNDED PRECEDINGはフレームがパーティションの先頭行から開始することを意味し、同様にUNBOUNDED FOLLOWINGはフレームがパーティションの最終行で終了することを意味します。ROWSモードではCURRENT ROWはフレームが現在の行で開始または終了することを意味しますが、RANGEあるいはGROUPSモードではフレームがORDER BY順序における現在行の最初または最後のピアで開始または終了することを意味します。offset PRECEDINGおよびoffset FOLLOWINGオプションの意味はフレームのモードによって異なります。ROWSモードでは、offsetはフレームが現在行の何行前または何行後に開始または終了するかを示す整数です。GROUPSモードでは、offsetはフレームが現在行のピアグループからピアグループ何個、前または後で開始または終了するかを示す整数です。ここでピアグループとはウィンドウのORDER BY句において等価の行のグループです。RANGEモードでは、offsetオプションを使うには、ウィンドウ定義に一つだけORDER BY列があることが必要です。それで、整列する列の値がoffsetを超えないだけ、現在行の整列する列の値より小さい（PRECEDINGに対して）、あるいは、より大きい（FOLLOWINGに対して）行がフレームに含まれます。この場合、offset式のデータ型は整列する列のデータ型によって決まります。数値の整列する列に対するoffsetは一般的に整列する列と同じ型ですが、日付時刻の整列する列に対してはintervalになります。これら全ての場合で、offsetの値は非NULLかつ非負でなければなりません。また、offsetが単純な定数である必要はありませんが、変数や集約関数、ウィンドウ関数を含めることはできません。

frame_exclusionオプションは現在行の周辺の行を、フレーム開始とフレーム終了のオプションにより含まれるものであっても、フレームから除外することができます。EXCLUDE CURRENT ROWはフレームから現在行を除外します。EXCLUDE GROUPはフレームから現在行とその整列ピアを除外します。EXCLUDE TIESは現在行自身

を除いた現在行のピアをフレームから除外します。EXCLUDE NO OTHERSは単に、現在行もそのピアも除外しないというデフォルトの振る舞いを明示的に指定します。

ORDER BY順序によりその行を一意に順序付けできない場合、ROWSモードが予期できない結果をもたらす可能性があることに注意して下さい。RANGEおよびGROUPSモードは、ORDER BY順序におけるピアとなる行が同等に扱われる、すなわち、与えられたピアグループの全行がフレームに入るか除外されるように設計されています。

WINDOW句の目的は、問い合わせのSELECTリストまたはORDER BY句に記載されるウィンドウ関数の動作を規定することです。これらの関数はそのOVER句において名前でWINDOW句の項目を参照することができます。しかしWINDOW句の項目は他で参照される必要はありません。問い合わせ内で使用されなかったものは、単に無視されます。ウィンドウ関数呼び出しはOVER句でウィンドウ定義を直接規定することができますので、WINDOW句を全く使わずにウィンドウ関数を使用することができます。しかしWINDOW句は、同じウィンドウ定義が複数のウィンドウ関数で必要とされる場合に入力量を省くことができます。

現在は、FOR NO KEY UPDATE、FOR UPDATE、FOR SHARE、FOR KEY SHAREをWINDOWと合わせて使うことはできません。

ウィンドウ関数に関する詳細については3.5、4.2.8、7.2.5を参照してください。

SELECTリスト

SELECTリスト（SELECTキーワードとFROMキーワードの間にあるもの）は、SELECT文の出力行を形成する式を指定するものです。この式では、FROM句で処理後の列を参照することができます（通常は実際に参照します）。

テーブルの場合と同様に、SELECTの出力列はすべて名前を持ちます。簡単なSELECTでは、この名前は列に表示用のラベルを付けるために使用されるだけです。しかしSELECTが大規模な問い合わせの副問い合わせである場合、大規模な問い合わせ側で副問い合わせで生成された仮想のテーブルの列名としてこの名前が参照されます。出力列として使用するための名前を指定するためには、列式の後にAS output_nameと記述してください。（希望する列名がPostgreSQLのキーワード（付録Cを参照）に一致しない場合にのみASを省略することができます。将来あり得るキーワードの追加に備えるために、常にASを記述する、あるいは、出力名を二重引用符で括ることを推奨します。）列名を指定しない場合、名前はPostgreSQLにより自動的に付けられます。列式が単純な列参照であれば、つけられる名前はその列の名前と同じものです。より複雑な場合では、関数名または型名が使用されるかもしれません。さもなければ?column?のように生成される名前になるかもしれません。

ORDER BY句とGROUP BY句内で列の値を参照する時も、出力列名を使用できます。しかし、WHEREやHAVING句では使用できません。これらでは式を書かなければなりません。

リストには、選択された行の全ての列を表す省略形として、式ではなく*と書くことができます。また、そのテーブルに由来する列のみを表す省略形として、table_name.*と書くこともできます。このような場合、ASにより新しい名前を指定することはできません。出力列名はテーブルの列名と同一になります。

標準SQLによれば、出力リスト内の式は、DISTINCT、ORDER BY、LIMITを適用する前に計算することになっています。DISTINCTを使う場合は、これは明らかに必要です。なぜなら、そうしなければどの値がDISTINCTであるかわからないからです。しかし、多くの場合、ORDER BYやLIMITの後で出力式を計算する方が便利です。特に出力式が揮発性(volatile)あるいは高価な式を含んでいる場合はそうです。この動作により、関数の評価順序はより直感的になり、出力に現れない行については評価されなくなります。PostgreSQLでは、式

がDISTINCT、ORDER BY、GROUP BYの中で参照されていない限り、ソートと制限(limit)の後にそれらの式を実際に評価します。(この反例として、SELECT f(x) FROM tab ORDER BY 1 では明らかにf(x)をソートの前に評価しなければなりません。) 集合を返す関数を含む出力式は、ソートの後、制限の前に実際の評価が行われ、これによりLIMITが集合を返す関数の出力を制限することになります。

> **注記**
>
> PostgreSQLのバージョン9.6より前では、出力式がソートや制限に対して評価されるタイミングについて何の保証もしていませんでした。それは選択された問い合わせの計画の形式に依存します。

DISTINCT句

SELECT DISTINCTが指定されると、重複する行は全て結果セットから削除されます (重複するグループの中で1行が保持されます)。SELECT ALLはこの反対で、全ての行が保持されます。デフォルトはこちらです。

SELECT DISTINCT ON (expression [, ...])は指定した式が等しいと評価した各行集合の中で、最初の行のみを保持します。DISTINCT ON式は、ORDER BY(上述)と同じ規則で扱われます。各集合の「最初の行」は、ORDER BYを使用して目的の行が確実に最初に現れるようにしない限り予測することはできないことに注意してください。例えば、次の例は各地点の最新の気象情報を取り出します。

```
SELECT DISTINCT ON (location) location, time, report
    FROM weather_reports
    ORDER BY location, time DESC;
```

しかしORDER BYを使用して各地点を時間によって降順にソートしなければ、各地点について得られる情報がいつのものかはわかりません。

DISTINCT ONに指定する式はORDER BYの最も左側の式と一致しなければなりません。ORDER BY句は、通常、各DISTINCT ONグループの中での行の優先順位を決定する追加的な式を含みます。

現在は、FOR NO KEY UPDATE、FOR UPDATE、FOR SHARE、FOR KEY SHAREをDISTINCTと合わせて使うことはできません。

UNION句

UNION句の一般的な構文は以下の通りです。

```
select_statement UNION [ ALL | DISTINCT ] select_statement
```

select_statementには、ORDER BY、LIMIT、FOR NO KEY UPDATE、FOR UPDATE、FOR SHARE、FOR KEY SHARE句を持たない任意のSELECT文が入ります (ORDER BYとLIMITは、括弧で囲めば副式として付与することができます。括弧がない場合、これらの句は右側に置かれた入力式ではなく、UNIONの結果に対して適用されてしまいます)。

UNION演算子は、2つのSELECT文が返す行の和集合を作成します。この和集合には、2つのSELECT文の結果集合のいずれか（または両方）に存在する行が全て含まれています。UNIONの直接のオペランドとなる2つのSELECT文が返す列数は、同じでなければなりません。また、対応する列のデータ型には互換性が存在する必要があります。

ALLオプションが指定されていない限り、UNIONの結果には重複行は含まれません。ALLを指定するとこのような重複除去が行われません（したがって、通常UNION ALLはUNIONよりかなり高速です。できればALLを使用してください）。重複行を除去するデフォルトの動作を明示的に指定するためにDISTINCTを記述することができます。

1つのSELECT文に複数のUNION演算子がある場合、括弧がない限り、それらは左から右に評価されます。

現時点では、UNIONの結果やUNIONに対する入力に、FOR NO KEY UPDATE、FOR UPDATE、FOR SHARE、FOR KEY SHAREを指定することはできません。

INTERSECT句

INTERSECT句の一般的な構文は以下の通りです。

```
select_statement INTERSECT [ ALL | DISTINCT ] select_statement
```

select_statementには、ORDER BY、LIMIT、FOR NO KEY UPDATE、FOR UPDATE、FOR SHARE、FOR KEY SHARE句を持たない、任意のSELECT文が入ります。

INTERSECTは、2つのSELECT文が返す行の積集合を計算します。この積集合に含まれるのは、2つのSELECT文の結果集合の両方に存在する行です。

ALLオプションを指定しない限り、INTERSECTの結果に重複行は含まれません。ALLが指定された場合、左側テーブルにm個、右側テーブルにn個の重複がある行は、結果集合ではmin(m,n)個出現します。重複行を除去するデフォルトの動作を明示的に指定するためにDISTINCTを記述することができます。

1つのSELECT文に複数のINTERSECT演算子がある場合、括弧がない限り、それらは左から右に評価されます。INTERSECTはUNIONよりも強い結び付きを持ちます。つまり、A UNION B INTERSECT CはA UNION (B INTERSECT C)と解釈されます。

現時点では、INTERSECTの結果やINTERSECTに対する入力に、FOR NO KEY UPDATE、FOR UPDATE、FOR SHAREまたはFOR KEY SHAREを指定することはできません。

EXCEPT句

EXCEPT句の一般的な構文は以下の通りです。

```
select_statement EXCEPT [ ALL | DISTINCT ] select_statement
```

select_statementには、ORDER BY、LIMIT、FOR NO KEY UPDATE、FOR UPDATE、FOR SHARE、FOR KEY SHARE句を持たない、任意のSELECT文が入ります。

EXCEPTは、左側のSELECT文の結果には存在し、右側のSELECT文の結果には存在しない行の集合を生成します。

ALLオプションが指定されていない限り、EXCEPTの結果には重複行は含まれません。ALLがある場合、左側テーブルにm個、右側テーブルにn個の重複がある行は、結果集合ではmax(m-n,0)個出現します。重複行を除去するデフォルトの動作を明示的に指定するためにDISTINCTを記述することができます。

1つのSELECT文に複数のEXCEPT演算子がある場合、括弧がない限り、それらは左から右に評価されます。EXCEPTの結び付きの強さはUNIONと同じです。

現時点では、EXCEPTの結果やEXCEPTに対する入力に、FOR NO KEY UPDATE、FOR UPDATE、FOR SHAREまたはFOR KEY SHAREを指定することはできません。

ORDER BY句

ORDER BY句の一般的な構文は以下の通りです(この句は省略可能です)。

```
ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST | LAST } ] [, ...]
```

ORDER BY句を使うと、結果行を指定した式(複数可)に従ってソートすることができます。最も左側の式を使って比較した結果、2つの行が等しいと判断された場合は、1つ右側の式を使って比較します。その結果も等しければ、さらに次の式に進みます。指定した全ての式で等しいと判断された場合は、実装に依存した順番で返されます。

expressionには、出力列(SELECTリスト項目)の名前または序数、あるいは入力列値から形成される任意の式を取ることができます。

序数は、出力列の位置(左から右に割り当てられます)を示します。これを使うと、一意な名前を持たない列の順序を定義することができます。AS句を使用すれば出力列に名前を割り当てることができるので、これはどうしても必要な機能というわけではありません。

また、ORDER BY句には、SELECT出力リストに出現しない列を含む、任意の式を使用できます。したがって、以下の文は有効です。

```
SELECT name FROM distributors ORDER BY code;
```

ただし、UNION、INTERSECT、EXCEPTの結果にORDER BYを適用する場合は、式は使用できず、出力列の名前か序数のみを指定できるという制限があります。

ORDER BYの式として出力列名と入力列名の両方に一致する単なる名前が与えられた場合、ORDER BYはそれを出力列名として扱います。これは、同じ状況におけるGROUP BYの選択とは反対です。この不整合は、標準SQLとの互換性を保持するために発生しています。

ORDER BY中の任意の式の後に、キーワードASC(昇順)、DESC(降順)を付加することができます(省略可能)。指定がなければ、デフォルトでASCがあるものとして扱われます。その他、順序を指定する演算子名をUSING句に指定する方法もあります。順序指定演算子は何らかのB-Tree演算子族の小なりまたは大なり演算子でなければなりません。通常、ASCはUSING <と、DESCはUSING >と同じです (ただし、ユーザ定義データ型の作成時には、デフォルトのソート順を定義することができます。また、異なる名前の演算子と対応付けすることもできます)。

NULLS LASTが指定されると、NULL値はすべての非NULL値の後にソートされます。NULLS FIRSTが指定されると、NULL値はすべての非NULL値の前にソートされます。どちらも指定されない場合のデフォルト動作は、明示的あるいは暗黙的なASCの場合はNULLS LAST、DESCが指定された場合はNULLS FIRSTです。(したがって、デフォルトでは、NULLが非NULLよりも大きい値であるかのように動作します。) USINGが指定されると、デフォルトのNULLの順序は、演算子が小なり演算子か大なり演算子によって変わります。

順序付けオプションは直前の演算子にのみ適用されます。たとえば、ORDER BY x, y DESCはORDER BY x DESC, y DESCと同一の意味ではありません。

文字型データでは、格納する列に適用された照合順序に従ってソートされます。これは必要に応じてexpression内にCOLLATE句を含めることで上書きできます。例えばORDER BY mycolumn COLLATE "en_US"です。より詳細については4.2.10および23.2を参照してください。

LIMIT句

LIMIT句は2つの独立した副句から構成されます。

```
LIMIT { count | ALL }
OFFSET start
```

パラメータcountには返される行の最大数を、一方、startには行を返し始める前に飛ばす行数を指定します。両方とも指定された場合、start行分が飛ばされ、そこから数えてcount行が返されます。

count式がNULLと評価された場合、LIMIT ALLとして、つまり制限無しとして扱われます。startがNULLと評価された場合、OFFSET 0と同様に扱われます。

SQL:2008では同じ結果を実現する異なる構文が導入されました。PostgreSQLでもサポートしています。以下の構文です。

```
OFFSET start { ROW | ROWS }
FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } { ONLY | WITH TIES }
```

この構文において、startまたはcountの値は標準SQLでは、リテラル定数、パラメータもしくは変数名を要求します。PostgreSQLの拡張では他の表現が許容されていますが、曖昧さを防ぐために通常は括弧で囲まれる必要があるでしょう。countをFETCH句で省略した場合、そのデフォルトは1です。WITH TIESオプションは、結果の集合でORDER BY句に従って最後の場所で同点になる追加の行を返すのに使われます。この場合ORDER BYは必須です。ROWおよびROWS、そしてFIRSTおよびNEXTは意味がない単語で、この句に影響を与えることはありません。SQL標準ではOFFSET句は、FETCH句と同時に使用する場合、これより前に存在しなければなりません。しかしPostgreSQLは厳密ではなく、どちらが先でも許されます。

LIMITを使う時は、結果行を一意な順番に強制するORDER BY句を使うとよいでしょう。そうしないと、問い合わせ結果のどの部分が返されるのかがわかりません。10〜20行目までを出力するとしても、どの順番で並べた時の10〜20行目なのでしょうか。ORDER BYを指定しない限り、行が返される順番は不明です。

問い合わせプランナは問い合わせ計画を作成する時にLIMITを考慮するので、LIMITとOFFSETの指定によって異なった計画を得ることになるでしょう。計画が異なれば、異なる順番で行が返ります。したがって、LIMIT/OFFSET値の変更によって異なる結果行を選択しようとすると、ORDER BYで順序を並び替えない限り、

矛盾した結果を返すことになります。これはバグではありません。「SQLは、ORDER BYで順序を制御されない限り、問い合わせ結果が返す順序を約束しない」という事実の当然の帰結なのです。

厳密的に部分集合の選択を強制するORDER BYがなければ、同じLIMIT問い合わせを繰り返し実行してもテーブル行から異なる部分集合が取り出される可能性すらあります。繰り返しますが、これは不具合ではありません。こうした場合に確定した結果は単に保証されていないのです。

ロック処理句

FOR UPDATE、FOR NO KEY UPDATE、FOR SHAREおよびFOR KEY SHAREはロック処理句です。これらはテーブルから行を入手する時にどのようにSELECTがその行をロックするかに影響します。

ロック処理句の一般的な構文は以下の通りです。

```
FOR lock_strength [ OF table_name [, ...] ] [ NOWAIT | SKIP LOCKED ]
```

ここでlock_strengthは以下のいずれかを取ることができます。

```
UPDATE
NO KEY UPDATE
SHARE
KEY SHARE
```

それぞれの行レベルロックモードについての詳しい説明は13.3.2を参照してください。

他のトランザクションのコミットを待機することなく操作を進めるには、NOWAITあるいはSKIP LOCKEDオプションを使用してください。NOWAITでは、選択行のロックを即座に獲得できない時、文は待機せずに、エラーを報告します。SKIP LOCKEDでは、即座にロックできない行はすべてスキップされます。行のロックをスキップすると、一貫性のないデータが見えることになるので、一般的な目的の作業のためには適しませんが、複数の消費者がキューのようなテーブルにアクセスするときのロック競合の回避などに利用できます。NOWAITおよびSKIP LOCKEDは行レベルロックにのみに適用される点に注意してください。つまり、必要なROW SHAREテーブルレベルロックは通常通りの方法（第13章を参照）で獲得されます。もし、テーブルレベルのロックを待機せずに獲得しなければならないのであれば、最初にLOCKのNOWAITオプションを使用してください。

ロック処理句内に特定のテーブルが指定されている場合は、そのテーブルの行のみがロックされます。SELECT内の他のテーブルは通常通りに読み込まれます。テーブルリストを持たないロック処理句は、その文で使用されるすべてのテーブルに影響を与えます。ロック処理句がビューまたは副問い合わせで使用された場合、そのビューや副問い合わせで使用されるすべてのテーブルに影響を与えます。しかしこれらの句は主問い合わせで参照されるWITH問い合わせには適用されません。WITH問い合わせ内での行ロックを行いたい場合は、WITH問い合わせ内でロック処理句を指定してください。

異なるロック方式を異なるテーブルに指定する必要があれば、複数のロック処理句を記述することができます。複数のロック処理句で同一のテーブルを記述した（または暗黙的に影響が与えられた）場合、最も強いものだけが指定されたかのように処理されます。同様に、あるテーブルに影響を与える句のいずれかでNOWAITが指定された場合、そのテーブルはNOWAITとして処理されます。それ以外の場合、あるテーブルに影響を与える句のいずれかでSKIP LOCKEDが指定されていれば、そのテーブルはSKIP LOCKEDとして処理されます。

ロック処理句は、返される行がテーブルのどの行に対応するのかが明確に識別できない場合には使用することができません。例えば、集約には使用できません。

ロック処理句がSELECT問い合わせの最上位レベルに存在する場合、ロック対象行は問い合わせが返す行に正確に一致します。結合問い合わせ内の場合、ロック対象行は返される結合行に関連する行となります。さらに、スナップショットを更新した後に問い合わせ条件を満たさなくなった場合は返されなくなりますが、問い合わせのスナップショット時点で問い合わせ条件を満たす行もロックされます。LIMITが使用された場合、 制限を満たす行が返されるとロック処理は止まります。(しかし、OFFSETにより飛ばされた行はロックされることに注意してください。) 同様に、ロック処理句がカーソル問い合わせで使用された場合、カーソルにより実際に取り込んだ行または通り過ぎた行のみがロックされます。

ロック処理句が副SELECTに存在する場合、ロック対象行は副問い合わせの外側の問い合わせに返される行となります。外側の問い合わせからの条件が副問い合わせ実行の最適化に使用される可能性がありますので、これには副問い合わせ自体の検査が提示する行より少なくなるかもしれません。例えば、

```
SELECT * FROM (SELECT * FROM mytable FOR UPDATE) ss WHERE col1 = 5;
```

は、副問い合わせ内では文字として条件が記載されていなくても、col1 = 5を持つ行のみがロックされます。

以前のリリースでは、セーブポイント以降に更新されるロックの保持は失敗しました。例えば以下のコードです。

```
BEGIN;
SELECT * FROM mytable WHERE key = 1 FOR UPDATE;
SAVEPOINT s;
UPDATE mytable SET ... WHERE key = 1;
ROLLBACK TO s;
```

ROLLBACK TO後のFOR UPDATEロックの保持に失敗します。これはリリース9.3で修正されました。

注意

ORDER BY句とロック処理句を使用した、READ COMMITTEDトランザクション分離レベルで実行するSELECTコマンドでは、順序通りにならない行を返す可能性があります。ORDER BYが最初に適用されるためです。このコマンドは結果をソートしますが、その後、1行または複数の行のロック獲得がブロックされる可能性があります。このSELECTのブロックが解除された時点で、順序付け対象の列値の一部が変更されているかもしれません。これによりこうした行が(元の列値という観点では順序通りではありますが、)順序通りに現れません。必要に応じて、これは以下のように副問い合わせ内にFOR UPDATE/SHARE句を記述することで、回避することができます。

```
SELECT * FROM (SELECT * FROM mytable FOR UPDATE) ss ORDER BY column1;
```

最上位レベルにおけるFOR UPDATEは実際に返される行のみをロックするのに対して、これは結果としてmytableのすべての行をロックすることに注意してください。これは、特にORDER BYがLIMITやその他の制限と組み合わせている場合、性能上大きな違いを生み出す可能性があります。このため、この技法は、順序付け対象の列に対する同時実行の更新が想定され、かつ、厳密にソートされた結果が要求される場合にのみ推奨されます。

> REPEATABLE READまたはSERIALIZABLEトランザクション分離レベルでは、('40001'というSQLSTATEを持つ)シリアライゼーション失敗が発生します。このためこれらの分離レベルでは順序通りでない行を受け取る可能性はありません。

TABLEコマンド

```
TABLE name
```

というコマンドは以下と同じです。

```
SELECT * FROM name
```

これは、最上位のコマンドとして、あるいは複雑な問い合わせの一部として、入力を省略する構文の一種としても使用することができます。WITH、UNION、INTERSECT、EXCEPT、ORDER BY、LIMIT、OFFSET、FETCH、FORのロック句だけをTABLEと一緒に使うことができます。WHERE句およびいかなる形式の集約も使うことはできません。

例

filmsテーブルをdistributorsテーブルと結合します。

```
SELECT f.title, f.did, d.name, f.date_prod, f.kind
    FROM distributors d, films f
    WHERE f.did = d.did

       title       | did |     name     | date_prod  |   kind
-------------------+-----+--------------+------------+----------
 The Third Man     | 101 | British Lion | 1949-12-23 | Drama
 The African Queen | 101 | British Lion | 1951-08-11 | Romantic
 ...
```

全ての映画のlen列を合計しkind列によって結果をグループ化します。

```
SELECT kind, sum(len) AS total FROM films GROUP BY kind;

   kind   | total
----------+-------
 Action   | 07:34
 Comedy   | 02:58
 Drama    | 14:28
 Musical  | 06:42
 Romantic | 04:38
```

全ての映画のlen列を合計しkind列によって結果をグループ化し、合計が5時間より少ないグループの合計を表示します。

```
SELECT kind, sum(len) AS total
    FROM films
    GROUP BY kind
    HAVING sum(len) < interval '5 hours';

   kind   | total
----------+-------
 Comedy   | 02:58
 Romantic | 04:38
```

次に、結果を2番目の列（name）の内容に基づいてソートする方法を2つ例示します。

```
SELECT * FROM distributors ORDER BY name;
SELECT * FROM distributors ORDER BY 2;

 did |       name
-----+------------------
 109 | 20th Century Fox
 110 | Bavaria Atelier
 101 | British Lion
 107 | Columbia
 102 | Jean Luc Godard
 113 | Luso films
 104 | Mosfilm
 103 | Paramount
 106 | Toho
 105 | United Artists
 111 | Walt Disney
 112 | Warner Bros.
 108 | Westward
```

次の例は、distributorsテーブルとactorsテーブルの和集合を取得する方法を示しています。さらに、両方のテーブルで結果をWという文字で始まる行のみに限定しています。重複しない行のみが必要なので、ALLキーワードは省略されています。

```
distributors:               actors:
 did |     name              id |     name
-----+--------------       ----+----------------
 108 | Westward               1 | Woody Allen
 111 | Walt Disney            2 | Warren Beatty
 112 | Warner Bros.           3 | Walter Matthau
 ...                          ...

SELECT distributors.name
    FROM distributors
    WHERE distributors.name LIKE 'W%'
```

```
UNION
SELECT actors.name
    FROM actors
    WHERE actors.name LIKE 'W%';

      name
---------------
 Walt Disney
 Walter Matthau
 Warner Bros.
 Warren Beatty
 Westward
 Woody Allen
```

次に、FROM句内での関数の使用方法について、列定義リストがある場合とない場合の両方の例を示します。

```
CREATE FUNCTION distributors(int) RETURNS SETOF distributors AS $$
    SELECT * FROM distributors WHERE did = $1;
$$ LANGUAGE SQL;

SELECT * FROM distributors(111);
 did |    name
-----+-------------
 111 | Walt Disney

CREATE FUNCTION distributors_2(int) RETURNS SETOF record AS $$
    SELECT * FROM distributors WHERE did = $1;
$$ LANGUAGE SQL;

SELECT * FROM distributors_2(111) AS (f1 int, f2 text);
 f1  |     f2
-----+-------------
 111 | Walt Disney
```

以下は序数列が追加された関数の例です。

```
SELECT * FROM unnest(ARRAY['a','b','c','d','e','f']) WITH ORDINALITY;
 unnest | ordinality
--------+----------
 a      |        1
 b      |        2
 c      |        3
 d      |        4
 e      |        5
 f      |        6
(6 rows)
```

以下の例では簡単なWITH句の使用方法を示します。

```
WITH t AS (
    SELECT random() as x FROM generate_series(1, 3)
  )
SELECT * FROM t
UNION ALL
SELECT * FROM t

         x
--------------------
  0.534150459803641
  0.520092216785997
 0.0735620250925422
  0.534150459803641
  0.520092216785997
 0.0735620250925422
```

WITH問い合わせが一度だけ評価されることに注意してください。このため3つのランダムな値の同じ集合2組を得ることになります。

以下の例ではWITH RECURSIVEを使用して、直接の部下しか表示しないテーブルから、従業員Maryの（直接または間接的な）部下とその間接度を見つけ出します。

```
WITH RECURSIVE employee_recursive(distance, employee_name, manager_name) AS (
    SELECT 1, employee_name, manager_name
    FROM employee
    WHERE manager_name = 'Mary'
  UNION ALL
    SELECT er.distance + 1, e.employee_name, e.manager_name
    FROM employee_recursive er, employee e
    WHERE er.employee_name = e.manager_name
  )
SELECT distance, employee_name FROM employee_recursive;
```

初期条件、続いてUNION、さらに問い合わせの再帰部分という再帰問い合わせの典型的な構文に注意してください。問い合わせの再帰部分は最終的にはタプルを返さないことを確実にしてください。さもないと問い合わせは無限にループします。（より多くの例については7.8を参照してください。）

以下の例では、manufacturersテーブルの各行に対して集合を返すget_product_names()関数を適用するためにLATERALを使用します。

```
SELECT m.name AS mname, pname
FROM manufacturers m, LATERAL get_product_names(m.id) pname;
```

これは内部結合ですので、現時点で製品をまったく持たないメーカは結果に現れません。こうしたメーカの名前も結果に含めたければ以下のようにします。

```
SELECT m.name AS mname, pname
FROM manufacturers m LEFT JOIN LATERAL get_product_names(m.id) pname ON true;
```

互換性

当然ながら、SELECT文は標準SQLと互換性があります。しかし、拡張機能や実現されていない機能もいくつかあります。

FROM句の省略

PostgreSQLでは、FROM句を省略することができます。これによって、以下のように単純な式を計算させることができます。

```
SELECT 2+2;

 ?column?
----------
        4
```

他のSQLデータベースでは、このようなSELECTを行うためにはダミーの1行テーブルを使わなければならないものもあります。

FROM句の指定がない場合、問い合わせではデータベーステーブルを参照することができません。例えば、以下の問い合わせは無効です。

```
SELECT distributors.* WHERE distributors.name = 'Westward';
```

PostgreSQLリリース8.1より前まででは、こうした形の問い合わせを受け付け、問い合わせで参照する各テーブルに対する暗黙的な項目を問い合わせのFROM句に追加していました。これは許されなくなりました。

空のSELECTリスト

SELECTの後の出力式のリストは空でも良く、このとき列数がゼロの結果テーブルが生成されます。これは標準SQLでは有効な構文ではありませんが、PostgreSQLは列数がゼロのテーブルを許すので、それと整合性を保つために許しています。しかし、DISTINCTを使う時は、空のリストを使うことはできません。

ASキーワードの省略

標準SQLでは、キーワードAS(省略可能)は、新しい列名が有効な列名（つまり予約済みのどのキーワードとも異なるもの）である場合は常に、出力列名の前から省くことができます。PostgreSQLには多少より強い制限があります。新しい列名が予約済みか否かに関わらず何らかのキーワードに一致する場合はASが必要です。推奨する実践方法は、今後のキーワードの追加と競合する可能性に備え、ASを使用する、または出力列名を二重引用符で括ることです。

FROM項目において標準およびPostgreSQLでは、未予約のキーワードである別名の前のASを省略することができます。しかし、構文があいまいになるため、出力名では実践的ではありません。

ONLYと継承関係

標準SQLでは、SELECT * FROM ONLY (tab1), ONLY (tab2) WHERE ...のように、ONLYを記述する時にテーブル名の前後を括弧でくくることを要求します。PostgreSQLではこの括弧を省略可能であるとみなしています。

PostgreSQLでは最後に*を付けることで 明示的に子テーブルを含めるというONLYではない動作を指定することができます。標準ではこれを許していません。

（これらの点はONLYオプションをサポートするすべてのSQLコマンドで同様に適用されます。）

TABLESAMPLE句の制限

現在のところ、TABLESAMPLE句は通常のテーブルとマテリアライズドビューでのみ受け付けられます。SQL標準では、FROM句の任意の要素について適用可能であるべきとされています。

FROM内の関数呼び出し

PostgreSQLでは、FROMリストのメンバとして直接関数呼び出しを記述することができます。標準SQLではこうした関数呼び出しを副SELECT内に囲む必要があります。つまりFROM func(...) aliasはおおよそFROM LATERAL (SELECT func(...)) aliasと同じです。暗黙的にLATERALであるとみなされることに注意してください。標準ではFROM内のUNNEST()項目にはLATERAL構文を必要とするためです。PostgreSQLではUNNEST()を他の集合を返す関数と同じものとして扱います。

GROUP BYとORDER BYにおける利用可能な名前空間

標準SQL-92では、ORDER BY句で使用できるのは、出力列名か序数のみであり、GROUP BY句で使用できるのは、入力列名からなる式のみです。PostgreSQLは、これらの句で両方が指定できるように拡張されています（ただし、不明瞭さがある場合は標準の解釈が使用されます）。さらに、PostgreSQLではどちらの句にも任意の式を指定できます。式で使われる名前は、常に出力列名ではなく入力列の名前とみなされることに注意してください。

SQL:1999以降では、SQL-92と完全には上位互換でない、多少異なる定義が採用されています。しかし、ほとんどの場合、PostgreSQLはSQL:1999と同じ方法でORDER BYやGROUP BYを解釈します。

関数従属性

テーブルの主キーがGROUP BYリストに含まれる場合に限り、PostgreSQLは（GROUP BYで列を省くことができる）関数従属性を認識します。標準SQLでは、認識しなければならない追加の条件を規定しています。

LIMITおよびOFFSET

LIMITおよびOFFSET句はPostgreSQL独自の構文ですが、MySQLでも使用されています。LIMIT Clauseで説明したように、標準SQL:2008にて同じ機能のOFFSET ... FETCH {FIRST|NEXT} ...が導入されました。この構文はIBM DB2でも使用されています。(Oracle用に開発されたアプリケーションでは、これらの句の機能を実装するために自動生成されるrownum列を含めるという回避策を使用することが多いですが、PostgreSQLでは利用できません。)

FOR NO KEY UPDATE、FOR UPDATE、FOR SHARE、FOR KEY SHARE

FOR UPDATEは標準SQLに存在しますが、標準では、DECLARE CURSORのオプションとしてしか許されていません。PostgreSQLでは、副SELECTなど任意のSELECTで許されます。これは拡張です。FOR NO KEY UPDATE、FOR SHARE、FOR KEY SHAREの亜種、およびNOWAITとSKIP LOCKEDオプションは標準にはありません。

WITH内のデータ変更文

PostgreSQLではWITH問い合わせとしてINSERT、UPDATEおよびDELETEを使用することができます。これは標準SQLにはありません。

非標準句

DISTINCT ON (...)は標準SQLの拡張です。

ROWS FROM(...)は標準SQLの拡張です。

WITHのMATERIALIZEDとNOT MATERIALIZEDオプションはSQL標準の拡張です。

SELECT INTO

SELECT INTO — 問い合わせの結果からの新しいテーブルを定義する

概要

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    * | expression [ [ AS ] output_name ] [, ...]
    INTO [ TEMPORARY | TEMP | UNLOGGED ] [ TABLE ] new_table
    [ FROM from_item [, ...] ]
    [ WHERE condition ]
    [ GROUP BY expression [, ...] ]
    [ HAVING condition ]
    [ WINDOW window_name AS ( window_definition ) [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ] select ]
    [ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST | LAST } ] [, ...] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start [ ROW | ROWS ] ]
    [ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY ]
    [ FOR { UPDATE | SHARE } [ OF table_name [, ...] ] [ NOWAIT ] [...] ]
```

説明

SELECT INTOは新しいテーブルを作成し、そこに問い合わせによって計算したデータを格納します。このデータは通常のSELECTのようにはクライアントに返されません。新しいテーブルの列はSELECTの出力列に関連するデータ型と名前を持ちます。

パラメータ

TEMPORARYまたはTEMP

このオプションが指定された場合、テーブルは一時テーブルとして作成されます。詳細はCREATE TABLEを参照してください。

UNLOGGED

指定された場合、テーブルはログをとらないテーブルとして作成されます。詳細はCREATE TABLEを参照してください。

new_table

作成するテーブルの名前です（スキーマ修飾名も可）。

その他のパラメータについては、SELECTで詳細に説明されています。

注釈

CREATE TABLE ASは機能的にはSELECT INTOと同等です。ECPGやPL/pgSQLではINTO句の解釈が異なるため、SELECT INTOという形式は使用できません。そのため、CREATE TABLE AS構文を使用することをお勧めします。さらに、CREATE TABLE ASは、SELECT INTOの機能に加え、さらに多くの機能を提供します。

CREATE TABLE ASとは対照的に、SELECT INTOではUSING methodでのテーブルアクセスメソッドやTABLESPACE tablespace_nameでのテーブルのテーブル空間のような属性を指定できません。必要ならCREATE TABLE ASを使ってください。そのため、新しいテーブルにはデフォルトテーブルアクセスメソッドが選ばれます。より詳細な情報はdefault_table_access_methodを参照してください。

例

テーブルfilmsの最近の項目のみから構成される、新しいテーブルfilms_recentを作成します。

```
SELECT * INTO films_recent FROM films WHERE date_prod >= '2002-01-01';
```

互換性

標準SQLでは、SELECT INTOは新しいテーブルの作成ではなく、選択した値をホストプログラムのスカラ変数とするために使われます。これは実際、ECPG（第35章を参照）やPL/pgSQL（第42章を参照）で見られる使用方法です。PostgreSQLにおいて、テーブルを作成するSELECT INTOの用法は歴史的なものです。新しいコードでは、テーブルの作成にはCREATE TABLE ASを使うのが最善です。

関連項目

CREATE TABLE AS

SET

SET — 実行時パラメータを変更する

概要

```
SET [ SESSION | LOCAL ] configuration_parameter { TO | = } { value | 'value' | DEFAULT }
SET [ SESSION | LOCAL ] TIME ZONE { timezone | LOCAL | DEFAULT }
```

説明

SETコマンドは実行時設定パラメータを変更します。第19章に列挙されている実行時パラメータの多くは稼働中にSETコマンドで変更できます（ただし、変更するためにスーパーユーザ権限が必要なものがあります。また、サーバあるいはセッションの始動後は変更できないパラメータもあります）。SETは現行セッションで使用される値にのみ影響することに注意してください。

SET(またはSET SESSIONも同じ)が発行された後にトランザクションがアボートされると、トランザクションがロールバックした時点でSETコマンドの効力は失われます。一度トランザクションがコミットされると、別のSETコマンドで上書きされない限り、セッションが終了するまでその効果が持続します。

SET LOCALの効果は、コミットのされたかどうかにかかわらず現在のトランザクションが終了するまでしか持続しません。1つのトランザクション内でSETの後にSET LOCALが続く特殊な例を考えてみましょう。この場合、トランザクションが終了するまではSET LOCAL値が有効ですが、その後(トランザクションがコミットされたとして)SET値が有効になります。

SETもしくはSET LOCALの効果は、このコマンドより以前のセーブポイントまでロールバックした場合は取り消されます。

SET LOCALを同じ変数に対するSETオプション（CREATE FUNCTIONを参照）を持つ関数内で使用する場合、SET LOCALコマンドの効果は関数終了時に消滅します。つまり、関数が呼び出された時に有効だった値にとにかく戻されます。これによりSET LOCALは、呼び出し元の値を保管し元に戻すというSETオプションを使用する利点を持ちつつ、関数内で動的または繰り返し変更されるパラメータ用に使用できます。しかし、通常のSETコマンドでは上位の関数のSETオプションを上書きしてしまい、その効果はロールバックしない限り永続します。

> **注記**
>
> PostgreSQLバージョン8.0から8.2までは、SET LOCALの効果は、より以前のセーブポイントを解放すること、または、PL/pgSQL例外ブロックから正常終了することで取り消されました。直観的ではないようですので、この動作は変更されました。

パラメータ

SESSION

コマンドの有効範囲が現行セッションであることを指定します（SESSIONもLOCALも指定されていない場合は、これがデフォルトです）。

LOCAL

コマンドの有効範囲が現行のトランザクションのみであることを指定します。COMMITまたはROLLBACKの後は、再びセッションレベルの設定が有効になります。これをトランザクションブロックの外側で実行すると、警告が発生しますが、それ以外には何の効果もありません。

configuration_parameter

設定可能な実行時パラメータ名です。利用可能なパラメータは、第19章と以下に示します。

value

パラメータの新しい値です。値として、文字列定数、識別子、数字、あるいはこれらをカンマで区切ったリストを対象のパラメータで適切となるように、指定することができます。DEFAULTと記述することで、パラメータをデフォルト値（つまり、現在のセッションでSETが実行されなかった時に設定される値）に再設定することができます。

第19章に記載された設定パラメータの他に、SETコマンドを使用してのみ調整できるパラメータや特殊な構文を持つパラメータがいくつかあります。

SCHEMA

SET SCHEMA 'value'はSET search_path TO valueの別名です。この構文を使用する場合は1つのスキーマのみを指定することができます。

NAMES

SET NAMES valueは、SET client_encoding TO valueの別名です。

SEED

乱数ジェネレータ（random関数）用の内部シードを設定します。-1から1までの浮動小数点数を値として設定できます。その後、この値には2^{31}-1がかけられます。

シードはsetseed関数を呼び出すことでも設定可能です。

```
SELECT setseed(value);
```

TIME ZONE

SET TIME ZONE valueはSET timezone TO valueの別名です。SET TIME ZONE構文では、時間帯の指定に特殊な構文を使用できます。有効な値の例を以下に示します。

'PST8PDT'

カリフォルニア州バークレイ用の時間帯です。

'Europe/Rome'

イタリア用の時間帯です。

-7

UTCから西に7時間分ずらした時間帯です（PDTと同じです）。正の値はUTCから東方向です。

INTERVAL '-08:00' HOUR TO MINUTE

UTCから西に8時間分ずらした時間帯です（PSTと同じです）。

LOCAL
DEFAULT

時間帯をユーザのローカルな時間帯（サーバのデフォルトのtimezone値）に設定します。

時間帯を数字あるいは時間で指定した時は、内部的にPOSIXの時間帯構文として解釈されます。例えば、SET TIME ZONE -7とした後、SHOW TIME ZONEを実行すると、その結果は<-07>+07となります。

時間帯に関する詳細は8.5.3を参照してください。

注釈

set_config関数は等価な機能を提供します。9.27を参照してください。また、pg_settingsシステムビューを更新することで、SETと同じことを実行することができます。

例

スキーマの検索パスを設定します。

```
SET search_path TO my_schema, public;
```

日付のスタイルを、伝統的なPOSTGRES入力方式に設定し、さらに「day before month(月の前に日)」を使います。

```
SET datestyle TO postgres, dmy;
```

時間帯をカリフォルニア州バークレイに設定します。

```
SET TIME ZONE 'PST8PDT';
```

時間帯をイタリアに設定します。

```
SET TIME ZONE 'Europe/Rome';
```

互換性

SET TIME ZONEは標準SQLで定義された構文を拡張したものです。標準では数値による時間帯オフセットしか使用できないのに対し、PostgreSQLでは、より柔軟に時間帯を指定することができます。SETが持つその他の機能は、全てPostgreSQLの拡張です。

関連項目

RESET, SHOW

SET CONSTRAINTS

SET CONSTRAINTS — 現在のトランザクションの制約検査のタイミングを設定する

概要

```
SET CONSTRAINTS { ALL | name [, ...] } { DEFERRED | IMMEDIATE }
```

説明

SET CONSTRAINTSは、現在のトランザクションにおける制約検査の動作を設定します。IMMEDIATE制約は、1つの文の実行が終わるごとに検査されます。DEFERRED制約は、トランザクションがコミットされるまで検査されません。全ての制約は、IMMEDIATEかDEFERREDのどちらかのモードを持ちます。

制約にはその生成時点で、DEFERRABLE INITIALLY DEFERRED、DEFERRABLE INITIALLY IMMEDIATE、NOT DEFERRABLEの3つのうちのいずれかの性質が与えられます。3番目のNOT DEFERRABLE制約は、常にIMMEDIATEモードであり、SET CONSTRAINTSコマンドの影響を受けません。DEFERRABLE INITIALLY DEFERRED制約とDEFERRABLE INITIALLY IMMEDIATE制約の2つは、トランザクションを指定されたモードで開始しますが、トランザクション内でSET CONSTRAINTSを使用するとその振舞いを変更することができます。

SET CONSTRAINTSに制約名のリストをつけて実行すると、指定された制約（これらは全て遅延可能でなければなりません）のみのモードが変更されます。制約名はそれぞれスキーマ修飾可能です。スキーマ名が指定されていない場合、現在のスキーマ検索パスを使用して、最初に一致する名前を見つけます。SET CONSTRAINTS ALLは遅延可能な全ての制約のモードを変更します。

制約のモードをDEFERREDからIMMEDIATEに変更した場合は、新しい制約モードが遡及的に有効になります。つまり、トランザクションの終了時に検査される予定だった未検査のデータ変更が、SET CONSTRAINTSコマンドの実行中に検査されます。もし、この時に何らかの制約違反があった場合、SET CONSTRAINTSは失敗します（そして、制約モードは変更されません）。したがって、SET CONSTRAINTSを利用すれば、トランザクションの特定の時点で強制的に制約の検査を実行することができます。

現在UNIQUE、PRIMARY KEY、REFERENCES（外部キー）、EXCLUDE制約のみがこの設定の影響を受けます。NOT NULLおよびCHECK制約は、行が挿入または変更された時に（文の終了時ではありません）、常に即座に検査されます。DEFERRABLE宣言されていない一意性制約および排除制約も即座に検査されます。

また、「制約トリガ」として宣言されたトリガの発行もこの設定により制御されます。これらは関連する制約が検査されるはずの時に同時に発行されます。

注釈

PostgreSQLでは、スキーマ内で制約名が一意となることを要求していません（テーブル単位での一意性のみ要求します）ので、指定した制約名に複数が一致する可能性があります。この場合SET CONSTRAINTSは一致するすべてに対して動作します。スキーマ修飾がない名前では、検索パス上のあるスキーマに1つまたは複数の一致があると、パス上のそれより後にあるスキーマは検索されません。

このコマンドが変更するのは、現在のトランザクション内の制約の動作のみです。トランザクションブロックの外部でこのコマンドが実行されても、警告を発するだけで、他には何の効果もありません。

互換性

このコマンドは、標準SQLで定義された動作に準拠しています。ただし、PostgreSQLでは`NOT NULL`および`CHECK`制約に適用できないという制限があります。またPostgreSQLは非遅延一意性制約を、標準が提案する文の終わりにではなく、即座に検査します。

SET ROLE

SET ROLE — 現在のセッションにおける現在のユーザ識別子を設定する

概要

```
SET [ SESSION | LOCAL ] ROLE role_name
SET [ SESSION | LOCAL ] ROLE NONE
RESET ROLE
```

説明

このコマンドは現在のSQLセッションにおける現在のユーザ識別子をrole_nameに設定します。ロール名は識別子あるいは文字列リテラルのどちらを使用しても記述することができます。SET ROLEの後、SQLコマンドに対する権限検査は、指定されたロールで普通にログインした場合と同様に行われます。

指定するrole_nameは、現在のセッションユーザがメンバとして属するロールでなければなりません。（セッションユーザがスーパーユーザであった場合、任意のロールを選択することができます。）

SESSIONおよびLOCAL修飾子は通常のSETコマンドと同様に動作します。

NONEおよびRESET構文は、現在のユーザ識別子を現在のセッションユーザ識別子に戻します。この構文はすべてのユーザが実行することができます。

注釈

このコマンドを使用して、権限を追加することも制限することもできます。セッションユーザのロールがINHERIT属性を持つ場合、自動的にSET ROLEで設定可能なすべてのロールの権限を持ちます。この場合、SET ROLEは実際、セッションユーザに直接割り当てられている権限、セッションユーザが属するロールに割り当てられている権限の内、指定されたロールで使用可能な権限を残し、他をすべて削除します。一方、セッションユーザのロールがNOINHERIT属性を持つ場合、セッションユーザに直接割り当てられた権限をすべて削除し、指定されたロールで利用可能な権限を獲得します。

特に、スーパーユーザが非特権ユーザへのSET ROLEを行うと、スーパーユーザ権限を失うことになります。

SET ROLEの影響はSET SESSION AUTHORIZATIONと似ていますが、行われる権限検査はかなり異なります。また、SET SESSION AUTHORIZATIONは、それ以降に実行するSET ROLEコマンドでどのロールに変更できるかを決定しますが、SET ROLEを使用してロールを変更した場合、それ以降に実行するSET ROLEコマンドで変更可能なロール群は変更されません。

SET ROLEはロールのALTER ROLE設定で指定されたセッション変数を処理しません。これはログイン時のみ適用されます。

SET ROLEをSECURITY DEFINER関数内で使用することはできません。

例

```
SELECT SESSION_USER, CURRENT_USER;

 session_user | current_user
--------------+--------------
 peter        | peter

SET ROLE 'paul';

SELECT SESSION_USER, CURRENT_USER;

 session_user | current_user
--------------+--------------
 peter        | paul
```

互換性

PostgreSQLでは、識別子構文("rolename")を使用できます。しかし、標準SQLではロール名を文字列リテラルとして記述しなければなりません。SQLでは、トランザクション内でこのコマンドを実行することを許可していません。PostgreSQLでは、このように制限する理由がありませんので、この制限はありません。SESSION、LOCAL修飾子、および、RESET構文はPostgreSQLの拡張です。

関連項目

SET SESSION AUTHORIZATION

SET SESSION AUTHORIZATION

SET SESSION AUTHORIZATION — セッションのユーザ識別子、現在のセッションの現在のユーザ識別子を設定する

概要

```
SET [ SESSION | LOCAL ] SESSION AUTHORIZATION user_name
SET [ SESSION | LOCAL ] SESSION AUTHORIZATION DEFAULT
RESET SESSION AUTHORIZATION
```

説明

このコマンドはセッションのユーザ識別子、ならびに、現在のSQLセッションにおける現在のユーザ識別子をuser_nameに設定します。ユーザ名は、識別子、あるいは文字列リテラルとして記述することもできます。このコマンドを使用すると、例えば、一時的に非特権ユーザとなり、その後に特権ユーザに戻るといったことが可能です。

セッションのユーザ識別子はクライアントから渡される(おそらく認証済みの)ユーザ名で初期化されます。現在のユーザ識別子は通常セッションのユーザ識別子と同一ですが、SECURITY DEFINER関数や類似の機能によって一時的に変更される可能性があります。SET ROLEでこれを変更することもできます。現在のユーザ識別子は権限の検査に影響を与えます。

セッションのユーザ識別子は、最初のセッションユーザ(認証されたユーザ)がスーパーユーザ権限を持っている場合にのみ変更できます。スーパーユーザ権限を持っていない場合、認証されたユーザ名を指定した場合のみ、このコマンドは受け入れられます。

SESSION修飾子およびLOCAL修飾子は、通常のSETコマンドの場合と同じように機能します。

DEFAULT構文およびRESET構文は、セッションと現在のユーザ識別子を元の認証ユーザに戻します。これらの構文は全てのユーザが実行できます。

注釈

SET SESSION AUTHORIZATIONをSECURITY DEFINER関数内で使用することはできません。

例

```
SELECT SESSION_USER, CURRENT_USER;

 session_user | current_user
--------------+--------------
 peter        | peter
```

```
SET SESSION AUTHORIZATION 'paul';

SELECT SESSION_USER, CURRENT_USER;

 session_user | current_user
--------------+--------------
 paul         | paul
```

互換性

標準SQLでは、user_nameリテラルの指定方法としてその他の表現を使用できます。しかし、この違いは実用上は重要ではありません。PostgreSQLでは識別子構文("username")を使用できますが、SQLでは使用できません。SQLではこのコマンドをトランザクション中に実行することができませんが、PostgreSQLでは、禁止する理由が見当たらないため、この制限を付けていません。SESSION修飾子およびLOCAL修飾子とRESET構文は、PostgreSQLの拡張です。

標準SQLでは、このコマンドを実行するために必要な権限は、実装に依存するとされています。

関連項目

SET ROLE

SET TRANSACTION

SET TRANSACTION — 現在のトランザクションの特性を設定する

概要

```
SET TRANSACTION transaction_mode [, ...]
SET TRANSACTION SNAPSHOT snapshot_id
SET SESSION CHARACTERISTICS AS TRANSACTION transaction_mode [, ...]


ここでtransaction_modeは以下のいずれかです。

    ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED }
    READ WRITE | READ ONLY
    [ NOT ] DEFERRABLE
```

説明

SET TRANSACTIONは現在のトランザクションの特性を設定します。これはそれより後のトランザクションには影響を及ぼしません。SET SESSION CHARACTERISTICSは、セッションにおけるそれ以後のトランザクションのデフォルトのトランザクション特性を設定します。個々のトランザクションについてSET TRANSACTIONによりデフォルト特性を上書きすることができます。

利用可能なトランザクション特性はトランザクションの分離レベル、トランザクションのアクセスモード（読み書きモードもしくは読み取りのみモード）、遅延モードです。さらに、セッションのデフォルトとしてではなく、現在のトランザクションのみに対してスナップショットを選択することができます。

トランザクションの分離レベルは、並行して実行中の他のトランザクションが存在する場合、そのトランザクションが見ることができるデータを決定するものです。

READ COMMITTED

1つひとつの文から見ることができるのは、その文が開始される前にコミットされた行のみです。これがデフォルトです。

REPEATABLE READ

現在のトランザクションにおける全ての文は、トランザクションで最初の問い合わせ文またはデータを変更する文が実行される前にコミットされた行だけを見ることができます。

SERIALIZABLE

現在のトランザクションにおける全ての文は、トランザクションで最初の問い合わせ文またはデータを変更する文が実行される前にコミットされた行だけを見ることができます。同時実行のシリアラ

イザブルトランザクションの中で読み取りと書き込みのパターンによって、これらのトランザクションの実行を直列に(同時に一度)行うことができない状況になる場合、その内1つのトランザクションはserialization_failureというエラーでロールバックされます。

標準SQLでは、READ UNCOMMITTEDというもう1つのレベルを定義しています。PostgreSQLではREAD UNCOMMITTEDはREAD COMMITTEDとして扱われます。

トランザクション分離レベルは、そのトランザクションにおける最初の問い合わせ文やデータ更新文(SELECT、INSERT、DELETE、UPDATE、FETCH、COPY)が実行された後では変更することができません。トランザクションの分離や同時実行制御についての詳細情報は第13章を参照してください。

トランザクションのアクセスモードは、そのトランザクションが読み書き可能か読み取りのみかを決定します。デフォルトは読み書き可能です。読み取りのみのトランザクションでは、以下のSQLコマンドの実行が制限されます。書き込み対象のテーブルが一時テーブルでない場合、INSERT、UPDATE、DELETE、COPY FROMなどのSQLコマンドを実行できません。すべてのCREATE、ALTER、DROP系のSQLコマンド、COMMENT、GRANT、REVOKE、TRUNCATEは、実行できません。さらに、上述のコマンドが含まれるEXPLAIN ANALYZEとEXECUTEコマンドも実行できません。この方法ではディスクへの書き込みをすべて防ぐわけではないので、読み取り専用の高レベルの概念です。

DEFERRABLEトランザクション属性は、トランザクションがSERIALIZABLEかつREAD ONLYである場合のみ効果があります。あるトランザクションでこれら3つの属性がすべて選択されている場合、最初にスナップショットを獲得する時にブロックされる可能性があります。その後、そのトランザクションをSERIALIZABLEトランザクションの通常のオーバーヘッドを伴わず、またシリアライズ処理の失敗を引き起こす恐れやシリアライズ処理の失敗によりキャンセルされる恐れもなく実行することができます。これは時間がかかるレポート処理やバックアップによく適しています。

SET TRANSACTION SNAPSHOTコマンドにより、既存のトランザクションと同じスナップショットを持つ新しいトランザクションを実行することができます。既存のトランザクションはpg_export_snapshot関数(9.27.5参照)を使用してそのスナップショットを公開していなければなりません。この関数はスナップショット識別子を返します。どのスナップショットを取り込むかを指定するために、この識別子をSET TRANSACTION SNAPSHOTに渡さなければなりません。このコマンドでは、この識別子を例えば'000003A1-1'のようにリテラル文字列として記述しなければなりません。SET TRANSACTION SNAPSHOTはトランザクションの開始時、つまり、トランザクションの最初の問い合わせまたはデータ変更文(SELECT、INSERT、DELETE、UPDATE、FETCH、COPY)の前でのみ実行できます。さらに、そのトランザクションを前もってSERIALIZABLEまたはREPEATABLE READ分離レベルに設定していなければなりません。(さもないと、READ COMMITTEDではコマンドそれぞれに対して新しいスナップショットを取りますので、このスナップショットは即座に破棄されます。) 取り込むトランザクションがSERIALIZABLE分離レベルを使用している場合、スナップショットを公開したトランザクションもこの分離レベルを使用しなければなりません。また、読み取り専用ではないシリアライザブルトランザクションは、読み取り専用トランザクションから公開されたスナップショットを取り込むことができません。

注釈

SET TRANSACTIONを、その前にSTART TRANSACTIONやBEGINを発行することなく実行した場合、警告が発生しますが、それ以外は何の効果もありません。

BEGINあるいはSTART TRANSACTIONで目的のtransaction_modesを指定すれば、SET TRANSACTIONを使わずに済ませることができます。しかしSET TRANSACTION SNAPSHOTに対応するオプションはありません。

セッションのデフォルトのトランザクションモードは、設定パラメータdefault_transaction_isolation、default_transaction_read_only、default_transaction_deferrableで設定することができます（実際、SET SESSION CHARACTERISTICSはこれらの変数をSETで設定することと同等の冗長な記述に過ぎません。）。したがって、トランザクションモードのデフォルトは設定ファイルやALTER DATABASEなどで設定可能です。詳細は第19章を参照してください。

例

既存のトランザクションと同じスナップショットを持つトランザクションを新しく開始するためには、まず既存のトランザクションからスナップショットを公開します。以下の例に示すように、これはスナップショット識別子を返します。

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT pg_export_snapshot();
 pg_export_snapshot
---------------------
 00000003-0000001B-1
(1 row)
```

そして、新規に開始したトランザクションの先頭のSET TRANSACTION SNAPSHOTでこのスナップショット識別子を渡します。

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SET TRANSACTION SNAPSHOT '00000003-0000001B-1';
```

互換性

このコマンドは標準SQLで定義されています。DEFERRABLEトランザクションモードとSET TRANSACTION SNAPSHOT構文は例外であり、PostgreSQLの拡張です。

標準SQLではデフォルトのトランザクション分離レベルはSERIALIZABLEです。PostgreSQLでは、通常、READ COMMITTEDがデフォルトですが、これは上述の通り変更可能です。

標準SQLでは、もう1つ、診断領域の大きさというトランザクション特性があり、このコマンドで設定可能です。この概念は組み込みSQL固有のものなので、PostgreSQLサーバには実装されていません。

標準SQLでは、連続するtransaction_modesの間にはカンマが必要です。歴史的な理由よりPostgreSQLではカンマを省略することができます。

SHOW

SHOW — 実行時パラメータの値を表示する

概要

```
SHOW name
SHOW ALL
```

説明

SHOWは、実行時パラメータの現在の設定を表示します。これらの変数は、SET文、postgresql.conf設定ファイルの編集、環境変数PGOPTIONS(libpqの使用時、あるいはlibpqを使用したアプリケーションの使用時)、または、postgresサーバの始動時のコマンドラインフラグで設定することができます。詳細は第19章を参照してください。

パラメータ

name

実行時パラメータの名前です。利用可能なパラメータは第19章とSETマニュアルページに記載されています。さらに、表示のみ可能で、変更できない次のようなパラメータがあります。

SERVER_VERSION

サーバのバージョン番号を示します。

SERVER_ENCODING

サーバ側の文字セット符号化方式を表示します。現時点では、符号化方式はデータベース作成時に決定されるため、このパラメータは表示のみ可能で、変更することができません。

LC_COLLATE

照合(テキストの順序付け)についてのデータベースのロケール設定を示します。現時点では、この設定はデータベース作成時に決定されるため、このパラメータは表示のみ可能で、変更することができません。

LC_CTYPE

文字分類についてのデータベースのロケール設定を表示します。現時点では、この設定はデータベース作成時に決定されるため、このパラメータは表示のみ可能で、変更することができません。

IS_SUPERUSER

現在のロールがスーパーユーザ権限を持つ場合は真になります。

ALL

全ての設定パラメータの値とその説明を表示します。

注釈

関数current_settingは同等の出力を生成します。9.27を参照してください。また、pg_settingsシステムビューは同じ情報を生成します。

例

パラメータDateStyleの現在の設定を表示します。

```
SHOW DateStyle;
 DateStyle
-----------
 ISO, MDY
(1 row)
```

パラメータgeqoの現在の設定を表示します。

```
SHOW geqo;
 geqo
------
 on
(1 row)
```

全設定を表示します。

```
SHOW ALL;
            name         | setting |                description

-------------------------+---------+-----------------------------------------------
 allow_system_table_mods | off     | Allows modifications of the structure of ...
    .
    .
    .
 xmloption               | content | Sets whether XML data in implicit parsing ...
 zero_damaged_pages      | off     | Continues processing past damaged page headers.
(196 rows)
```

互換性

SHOWコマンドはPostgreSQLの拡張です。

関連項目

SET, RESET

START TRANSACTION

START TRANSACTION —トランザクションブロックを開始する

概要

```
START TRANSACTION [ transaction_mode [, ...] ]


transaction_modeには以下のいずれかが入ります。

    ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED }
    READ WRITE | READ ONLY
    [ NOT ] DEFERRABLE
```

説明

このコマンドは新しいトランザクションブロックを開始します。分離レベルや読み取り/書き込みモード、遅延モードを指定すると、SET TRANSACTIONが実行された時のように、新しいトランザクションはそれらの特性を持ちます。このコマンドの機能は、BEGINコマンドと同じです。

パラメータ

この文のパラメータの意味についてはSET TRANSACTIONを参照してください。

互換性

標準SQLでは、トランザクションブロック開始時のSTART TRANSACTIONコマンドの発行は必須ではありません。任意のSQLコマンドが暗黙的にブロックを開始するからです。PostgreSQLでは、START TRANSACTION（もしくはBEGIN）が実行されていない状態で発行されたコマンドは、その直後に、暗黙的なCOMMITが発行されたかのように動作します。これは「自動コミット」と呼ばれます。他のリレーショナルデータベースシステムの中にも、簡便性のために自動コミット機能を提供しているものもあります。

DEFERRABLE transaction_modeはPostgreSQLの言語拡張です。

標準SQLでは、連続するtransaction_modesの間にはカンマが必須です。しかし、PostgreSQLでは歴史的な理由によりカンマを省略することができます。

SET TRANSACTIONの互換性の節も参照してください。

関連項目

BEGIN, COMMIT, ROLLBACK, SAVEPOINT, SET TRANSACTION

TRUNCATE

TRUNCATE — 1テーブルまたはテーブル群を空にする

概要

```
TRUNCATE [ TABLE ] [ ONLY ] name [ * ] [, ... ]
    [ RESTART IDENTITY | CONTINUE IDENTITY ] [ CASCADE | RESTRICT ]
```

説明

TRUNCATEはテーブル群から全ての行を素早く削除します。各テーブルに対して条件指定のないDELETEコマンドの実行と同じ効果を持ちますが、実際にテーブルを走査しない分、このコマンドの方が高速です。さらに、 その後にVACUUM操作を行うことなく、このコマンドはディスク領域を即座に回収します。このコマンドは、大きなテーブルを対象とする場合に最も有用です。

パラメータ

name

空にするテーブルの名前です（スキーマ修飾名も可）。テーブル名の前にONLYが指定されている場合、 そのテーブルのみを空にします。ONLYが指定されていない場合、そのテーブルとそのすべての子テーブル（もしあれば）を空にします。オプションで、テーブル名の後に*を指定することで、明示的に継承するテーブルも含まれることを示すことができます。

RESTART IDENTITY

消去されるテーブルの列により所有されるシーケンスを自動的に再開始させます。

CONTINUE IDENTITY

シーケンスの値を変更しません。これがデフォルトです。

CASCADE

指定されたテーブル、または、CASCADEにより削除対象テーブルとされたテーブルを参照する外部キーを持つテーブルすべてを自動的に空にします。

RESTRICT

外部キーにより対象のテーブルを参照するテーブルのいずれかがこのコマンドで指定されていない場合、操作を拒否します。これがデフォルトです。

注釈

テーブルを空にするためにはそのテーブルにTRUNCATE権限を持たなければなりません。

TRUNCATEは操作対象の各テーブルに対するACCESS EXCLUSIVEロックを獲得します。これは、この他のそのテーブルに対する同時操作をすべてブロックします。RESTART IDENTITYが指定された場合、初期化対象のシーケンスがあると、それは同様に排他ロックされます。テーブルへの同時アクセスが必要ならば、代わりにDELETEコマンドを使用しなければなりません。

そのテーブルが他のテーブルから外部キーで参照されている場合、その同じコマンドでそれらのテーブルをすべて空にするように指定していない限り、TRUNCATEを使用することはできません。このような場合に有効性を検査するならばテーブルスキャンが必要になりますが、テーブルスキャンを行うのであれば、このコマンドの利点がなくなるからです。CASCADEオプションを使用して、自動的にすべての依存テーブルを含めることができます。しかし、意図しないデータ損失の可能性がありますので、このオプションを使用する時には十分に注意してください。空にするテーブルがパーティションの場合、兄弟のパーティションには手をつけませんが、参照しているテーブルすべてとそのパーティションすべてに対しては、区別することなくカスケードが起こります。

TRUNCATEは、テーブルにON DELETEトリガがあっても、それを発行しません。しかし、ON TRUNCATEトリガを発行します。テーブルのいずれかにON TRUNCATEトリガが定義されている場合、何らかの消去が行われる前にすべてのBEFORE TRUNCATEトリガが発行されます。また、最後の消去がなされ、シーケンスが初期化された後すべてのAFTER TRUNCATEトリガが発行されます。トリガは処理されるテーブルの順番（コマンドに列挙されたものが先、その後にカスケードのために追加されたもの）に発行されます。

TRUNCATEはMVCC的に安全ではありません。同時実行中のトランザクションが、削除の前に取得したスナップショットを使っている場合、削除の後、テーブルはそのトランザクションからは空に見えます。（詳しくは13.5を参照してください。）

テーブル内のデータという観点では、TRUNCATEはトランザクション的に安全です。前後のトランザクションがコミットされなければ消去は安全にロールバックされます。

RESTART IDENTITYが指定された場合、暗黙的にALTER SEQUENCE RESTART操作がトランザクション的に行われます。つまりそれを囲むトランザクションがコミットされなければ、ロールバックされます。トランザクションがロールバックされる前に、初期化したシーケンスに対してさらにシーケンス操作を行う場合には注意してください。シーケンスに対するこれらの操作の影響はロールバックされますが、currval()への影響はロールバックされません。つまりトランザクションの後、currval()は、シーケンス自体と値とが一貫性のない状態になっていたとしても、失敗したトランザクションの内側で得た最後のシーケンス値を継続して反映します。これは、失敗したトランザクションの後のcurrval()の通常の動作と同じです。

現在のところ、TRUNCATEは外部テーブルに対してはサポートされません。このため、指定のテーブルの子孫に外部テーブルがあった場合、コマンドは失敗します。

例

bigtableテーブルおよびfattableテーブルを空にします。

```
TRUNCATE bigtable, fattable;
```

以下も同じですが、ここでは関連するシーケンスジェネレータをすべてリセットします。

```
TRUNCATE bigtable, fattable RESTART IDENTITY;
```

othertableテーブル、および、外部キー制約によりothertableを参照するすべてのテーブルを空にします。

```
TRUNCATE othertable CASCADE;
```

互換性

標準SQL:2008には、TRUNCATE TABLE tablenameという構文のTRUNCATEコマンドが含まれます。CONTINUE IDENTITY/RESTART IDENTITY句も標準に記載され、関連してはいるのですが、若干異なります。標準では、このコマンドの同時実行に関する動作の一部は実装に依存するものとされています。このため、上記注釈を検討し、必要に応じて他の実装と比べなければなりません。

関連項目

DELETE

UNLISTEN

UNLISTEN — 通知の監視を停止する

概要

```
UNLISTEN { channel | * }
```

説明

UNLISTENを使うと、既存のNOTIFYイベントの登録を削除することができます。UNLISTENは、現在のPostgreSQLセッションにある、nameという名前の通知チャネルのリスナ登録を取り消します。ワイルドカード*を指定すると、現在のセッションにある全てのリスナ登録が取り消されます。

NOTIFYには、LISTENとNOTIFYについてのより広範な説明があります。

パラメータ

channel

通知チャネルの名称です（任意の識別子）。

*

このセッションにおける、全ての監視登録をクリアします。

注釈

監視を行っていない通知チャネルに対してもこのコマンドは実行できます。警告やエラーは表示されません。

セッション終了時に、自動的にUNLISTEN *が実行されます。

UNLISTENを実行したトランザクションは二相コミット用を準備することはできません。

例

登録を行います。

```
LISTEN virtual;
NOTIFY virtual;
Asynchronous notification "virtual" received from server process with PID 8448.
```

UNLISTENが実行されると、その後のNOTIFYメッセージは無視されます。

```
UNLISTEN virtual;
NOTIFY virtual;

-- NOTIFYイベントを受け取りません。
```

互換性

標準SQLにはUNLISTENコマンドはありません。

関連項目

LISTEN, NOTIFY

UPDATE

UPDATE — テーブルの行を更新する

概要

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
UPDATE [ ONLY ] table_name [ * ] [ [ AS ] alias ]
    SET { column_name = { expression | DEFAULT } |
          ( column_name [, ...] ) = [ ROW ] ( { expression | DEFAULT } [, ...] ) |
          ( column_name [, ...] ) = ( sub-SELECT )
        } [, ...]
    [ FROM from_item [, ...] ]
    [ WHERE condition | WHERE CURRENT OF cursor_name ]
    [ RETURNING * | output_expression [ [ AS ] output_name ] [, ...] ]
```

説明

UPDATEは、条件を満たす全ての行の指定した列の値を変更します。SET句には、変更する列のみを指定する必要があります。SET句にて明示的に指定されなかった列の値は変更されません。

データベース内の他のテーブルの情報を使用してテーブルを変更するには、2つの方法があります。1つは副問い合わせを使用する方法、もう1つはFROM句で追加のテーブルを指定する方法です。どちらの方法が適切であるかは状況次第です。

RETURNING句を指定すると、UPDATEは実際に更新された各行に基づいて計算された値を返すようになります。そのテーブルの列およびFROMで指定された他のテーブルの列を使用した式を計算することができます。テーブル列の新しい(更新された後の)値が使用されます。RETURNINGリストの構文はSELECTの出力リストと同一です。

更新を行うためには、そのテーブルまたは少なくとも更新対象の列についてUPDATE権限を持たなければなりません。またexpressionsやconditionで値を読み込む列に対するSELECT権限も必要になります。

パラメータ

with_query

WITH句によりUPDATE問い合わせ内で名前で参照可能な1つ以上の副問い合わせを指定することができます。7.8とSELECTを参照してください。

table_name

更新対象のテーブルの名前です(スキーマ修飾名でも可)。テーブルの前にONLYを指定すると、指名されたテーブルでのみマッチする行が更新されます。ONLYを指定しないと、指名したテーブルから継承さ

れたすべてのテーブルでもマッチする行が同時に更新されます。オプションで、テーブル名の後に*を指定して、明示的に子テーブルが含まれることを示すこともできます。

alias

対象テーブルの代替名です。別名が指定されると、テーブルの実際の名前は完全に隠蔽されます。たとえば、UPDATE foo AS fでは、UPDATE文の残りの部分ではfooではなくfとしてこのテーブルを参照しなければなりません。

column_name

table_nameで指名されたテーブル内の列名です。必要に応じて、列名を副フィールド名や配列の指示子で修飾することも可能です。対象列の指定にはテーブル名を含めないでください。たとえば、UPDATE table_name SET table_name.col = 1は無効です。

expression

列に代入する式です。この式では、テーブル内の対象列やその他の列の変更前の値を使用することができます。

DEFAULT

列にデフォルト値を設定します(デフォルト式が割り当てられていない場合はNULLになります)。IDENTITY列には関連するシーケンスにより生成された新しい値が設定されます。生成列に対して、これを指定することは許されていますが、単に生成式から列を計算するという普通の振る舞いを指定するだけです。

sub-SELECT

その前の括弧内の列リストに列挙されているのと同じ数の出力列を生成するSELECT副問い合わせです。副問い合わせは実行時に最大でも1行しか生成してはいけません。1行だけ生成されたときは、各列の値が対象の列に代入されます。1行も生成されなかったときは、対象の列にNULL値が代入されます。副問い合わせは、更新対象のテーブルの現在行の古い値を参照することができます。

from_item

WHERE条件や更新用の式において、他のテーブルの列を指定するために使用するテーブル式です。これはSELECT文のFROM句と同じ文法を使います。例えば、テーブル名の別名が指定できます。自己結合を行う場合を除き、from_itemに更新対象のテーブルを繰り返してはいけません(自己結合を行う場合は、from_item内で更新対象のテーブルとその別名を指定しておく必要があります)。

condition

boolean型の値を返す式です。この式がtrueを返す行のみが更新されます。

cursor_name

WHERE CURRENT OF条件で使用されるカーソルの名前です。更新対象の行は、そのカーソルからもっとも最近に取り出された行です。カーソルはUPDATEの対象テーブルに対するグループ化のない問い合わせでなければなりません。WHERE CURRENT OFを論理条件といっしょに指定することはできません。WHERE CURRENT OF付きのカーソル使用に関する情報についてはDECLAREを参照してください。

output_expression

各行を更新した後に計算され、UPDATEによって返される式です。この式には、table_nameまたはFROMで指定したテーブル(複数可)の任意の列名を使用することができます。すべての列を返す場合は*と記載してください。

output_name

返される列で使用される名前です。

出力

正常に処理が終わると、UPDATEコマンドは以下の形式のコマンドタグを返します。

```
UPDATE count
```

countは、合致したが変更されなかった行を含む、更新された行数を意味します。BEFORE UPDATEトリガにより更新が抑制された場合に、conditionに合致した行数より少なくなる可能性があることに注意してください。countが0の場合はconditionに一致する行がなかったことを意味します (これはエラーとはみなされません)。

UPDATEコマンドがRETURNING句を持つ場合、その結果は、RETURNINGリストで定義した列と値を持ち、そのコマンドで更新された行全体に対して計算を行うSELECT文の結果と似たものになるでしょう。

注釈

FROM句が存在する場合、基本的に、対象テーブルとfrom_itemリストで指定されたテーブルが結合され、この結合の出力行が対象テーブルの更新操作の結果となります。FROM句を使用する場合、更新対象テーブルの1行に対して、結合結果が複数行にならないように注意してください。言い換えると、対象テーブルの個々の行は、他テーブルの複数の行と結合すべきではありません。結合結果が複数行になった場合、対象行の更新には結合結果のいずれか1行のみが使用されますが、どの行が使用されるかは簡単には予測できません。

このような不定性の問題があるため、他テーブルの参照は副問い合わせ内のみに留めておいた方がより安全です(ただし、結合よりも可読性や実行速度は低下します)。

パーティションテーブルの場合、行を更新することによって含んでいるパーティションのパーティション制約を満たさなくなることがありえます。その場合、この行がそのパーティション制約を満たす他のパーティションがパーティションツリー内にあれば、行はそのパーティションに移されます。もし、そのようなパーティションがなければ、エラーが発生します。舞台裏では、行の移動は実際はDELETEとINSERT操作です。

移される行に対して同時に実行されるUPDATEやDELETEのために直列化の失敗エラーになる可能性があります。セッション1がパーティションキーに対してUPDATEを実行中であるとしましょう。一方、同時に実行しているセッション2に対してこの行は可視であり、セッション2はこの行に対してUPDATEまたはDELETE操作をするとしましょう。その場合、セッション2のUPDATEまたはDELETEは、行の移動を検出し、直列化の失敗エラー(常にSQLSTATE値が'40001'で返る)を発生させます。これが起きた場合には、アプリケーションはトランザクションを再試行すると良いでしょう。テーブルがパーティション化されていない、または、行の移動がない通常の

場合には、セッション2は新しく更新された行を特定し、この新しい行のバージョンに対してUPDATE/DELETEを実行します。

(外部データラッパがタプルルーティングをサポートしていれば)行をローカルパーティションから外部テーブルパーティションへ移動できますが、外部テーブルパーティションから別のパーティションに移動できないことに注意してください。

例

filmsテーブルのkind列にあるDramaという単語をDramaticに変更します。

```
UPDATE films SET kind = 'Dramatic' WHERE kind = 'Drama';
```

weatherテーブルの特定の行に対し、気温に関する項目を調整し、降水量をデフォルト値に戻します。

```
UPDATE weather SET temp_lo = temp_lo+1, temp_hi = temp_lo+15, prcp = DEFAULT
  WHERE city = 'San Francisco' AND date = '2003-07-03';
```

同じ操作を行い、更新された項目を返します。

```
UPDATE weather SET temp_lo = temp_lo+1, temp_hi = temp_lo+15, prcp = DEFAULT
  WHERE city = 'San Francisco' AND date = '2003-07-03'
  RETURNING temp_lo, temp_hi, prcp;
```

もう一つの方法である列リスト構文を使用して同じ更新を行います。

```
UPDATE weather SET (temp_lo, temp_hi, prcp) = (temp_lo+1, temp_lo+15, DEFAULT)
  WHERE city = 'San Francisco' AND date = '2003-07-03';
```

FROM句の構文を使用して、Acme Corporationを顧客とするセールスマンのセールスカウントを1増加させます。

```
UPDATE employees SET sales_count = sales_count + 1 FROM accounts
  WHERE accounts.name = 'Acme Corporation'
  AND employees.id = accounts.sales_person;
```

WHERE句で副問い合わせを使用して、同じ操作を行います。

```
UPDATE employees SET sales_count = sales_count + 1 WHERE id =
  (SELECT sales_person FROM accounts WHERE name = 'Acme Corporation');
```

accountsテーブルのコンタクト先の氏名を、現在アサインされているセールスマンと一致するよう更新します。

```
UPDATE accounts SET (contact_first_name, contact_last_name) =
    (SELECT first_name, last_name FROM salesmen
```

```
    WHERE salesmen.id = accounts.sales_id);
```

同じような結果は結合を使っても得ることができます。

```
UPDATE accounts SET contact_first_name = first_name,
                    contact_last_name = last_name
  FROM salesmen WHERE salesmen.id = accounts.sales_id;
```

ただし、salesmen.idが一意キーでない場合、2番目の問い合わせは予期しない結果をもたらすかもしれません。一方で、最初の問い合わせは、複数のidがマッチしたときはエラーを発生することが保証されます。また、あるaccounts.sales_idエントリにマッチするレコードがない場合、最初の問い合わせは対応する名前フィールドをNULLに設定しますが、2番目の問い合わせは、その行を全く更新しません。

summaryテーブルの統計情報を現在のデータに合うように更新します。

```
UPDATE summary s SET (sum_x, sum_y, avg_x, avg_y) =
    (SELECT sum(x), sum(y), avg(x), avg(y) FROM data d
     WHERE d.group_id = s.group_id);
```

新しい商品とその在庫数を挿入します。既にその商品が存在している場合は、代わりに既存商品の在庫数を更新します。トランザクション全体が失敗することがないようにこの操作を行うには、セーブポイントを使用してください。

```
BEGIN;

-- 何かしらの他の操作を行います。
SAVEPOINT sp1;
INSERT INTO wines VALUES('Chateau Lafite 2003', '24');

-- 上記のコマンドが一意キー違反により失敗したとします。
-- この場合、
      次のコマンドを実行します。
ROLLBACK TO sp1;
UPDATE wines SET stock = stock + 24 WHERE winename = 'Chateau Lafite 2003';

-- 他の操作を続けた後、最後に次を実行します。
COMMIT;
```

filmsテーブルにおいて、c_filmsカーソルが現在位置している行のkind列を変更します。

```
UPDATE films SET kind = 'Dramatic' WHERE CURRENT OF c_films;
```

互換性

このコマンドは標準SQLに準拠しています。ただしFROM句およびRETURNING句はPostgreSQLの拡張です。UPDATEでWITHが使用可能であることも同様に拡張です。

他のデータベースシステムには、FROMオプション内で、対象テーブルが再度指定されることを前提として動作するものもあります。これはPostgreSQLにおけるFROMの解釈方法とは異なります。この拡張機能を使用するアプリケーションを移植する時は注意してください。

標準に従うと、括弧内の対象列名の部分リストに対する入力値は、正しい数の列を生成する任意の行値による式です。PostgreSQLでは入力値として、行コンストラクタあるいはsub-SELECTしか許していません。行コンストラクタを使う場合、個々の列の更新値をDEFAULTとして指定することができますが、sub-SELECTの内部ではできません。

VACUUM

VACUUM — データベースの不要領域の回収とデータベースの解析（オプション）を行う

概要

```
VACUUM [ ( option [, ...] ) ] [ table_and_columns [, ...] ]
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] [ ANALYZE ] [ table_and_columns [, ...] ]


ここでoptionは以下の一つであり、


    FULL [ boolean ]
    FREEZE [ boolean ]
    VERBOSE [ boolean ]
    ANALYZE [ boolean ]
    DISABLE_PAGE_SKIPPING [ boolean ]
    SKIP_LOCKED [ boolean ]
    INDEX_CLEANUP [ boolean ]
    TRUNCATE [ boolean ]
    PARALLEL integer


table_and_columnsは以下の通りです。

    table_name [ ( column_name [, ...] ) ]
```

説明

VACUUM は、無効タプルが使用する領域を回収します。PostgreSQLの通常動作では、削除されたタプルや更新によって不要となったタプルは、テーブルから物理的には削除されません。これらのタプルはVACUUMが完了するまで存在し続けます。そのため、特に更新頻度が多いテーブルでは、VACUUMを定期的に実行する必要があります。

table_and_columnsリストを指定しない場合、VACUUMは現在のユーザがバキュームできる権限を持つ、現在のデータベース内の全てのテーブルとマテリアライズドビューを処理します。リストを指定した場合、VACUUMは指定したテーブルのみを処理します。

VACUUM ANALYZEは、指定したテーブルの1つひとつに対し、VACUUMを行った後、ANALYZEを行います。このコマンドの組合わせは、日常的な管理スクリプトで使うと便利です。処理の詳細に関しては、ANALYZEを参照してください。

(FULLが指定されていない)通常のVACUUMは、単に領域を回収し、そこを再利用可能な状態に変更します。この形式のコマンドでは排他的ロックを取得しないため、テーブルへの通常の読み書き操作と並行して実行することができます。しかし余った領域はオペレーティングシステムには(ほとんどの場合)返されません。同じテーブル内で再利用できるように保持されるだけです。また、インデックスを処理するために複数のCPUを活用することもできます。この機能は、並列バキュームとして知られています。この機能を無効にするには、PARALLELオプションでパラレルワーカーの数をゼロに指定します。VACUUM FULLでは、テーブルの内容全体を新しいディスクファイルに領域を余すことなく書き換えるため、オペレーティングシステムに未使用の領域を返すことができます。この形式では、実行速度がかなり低速になります。また、処理中のテーブルに対する排他的ロックが必要になります。

オプションリストが括弧でくくられていた場合、オプションを任意の順序で記述することができます。括弧がないと、オプションは上で示した通りの順番で指定しなければなりません。括弧付きの構文はPostgreSQL 9.0で追加されました。カッコがない構文は廃止予定です。

パラメータ

FULL

より多くの領域の回収することができる「完全な」バキュームを選択します。ただし、通常よりも処理に時間がかかります。また、テーブルに対する排他ロックが必要です。またこの方式では、テーブルのコピーを新しく書き出し、操作が終わるまで古いコピーが解放されませんので、余分にディスク領域が必要です。通常、大きな容量がテーブルから回収されなければならない場合にのみこれが使用されるべきです。

FREEZE

積極的なタプルの「凍結」を選択します。FREEZE指定は、vacuum_freeze_min_ageおよびvacuum_freeze_table_ageパラメータをゼロとしてVACUUMを実行することと同じです。テーブルが書き換えられる時は、必ず積極的な凍結が行われるので、FULLが指定されているときは、このオプションは冗長です。

VERBOSE

各テーブルについてバキューム処理の詳細な報告を出力します。

ANALYZE

プランナが使用する統計情報を更新し、問い合わせを実行する最も効率的な方法を決定できるようにします。

DISABLE_PAGE_SKIPPING

通常、VACUUMは可視性マップに基いてページをスキップします。すべてのタプルが凍結されていることがわかっているページは、常にスキップできます。また、すべてのタプルがすべてのトランザクションに対して可視であることがわかっているページは、積極的なバキュームを実行している場合を除き、スキップできます。さらに、積極的なバキュームを実行している場合を除き、一部のページは、他のセッションがその使用を終了するのを待つのを避けるため、スキップされます。このオプションは、ページをスキッ

プする動作をすべて無効にします。これは可視性マップの内容が怪しいときにのみ使用されることを意図したもので、それはデータベースの破損を引き起こすようなハードウェアあるいはソフトウェアの障害がある場合にのみ発生します。

SKIP_LOCKED

VACUUMに、リレーションでの作業開始時、衝突するロックが解放されるのを待たないよう指定します。リレーションが待たずにすぐにロックできない場合、そのリレーションは飛ばされます。このオプションを指定しても、リレーションのインデックスを開く時にVACUUMはブロックするかもしれないことに注意してください。さらに加えて、VACUUM ANALYZEは、パーティションやテーブル継承の子、ある種類の外部テーブルからサンプル行を取得する時にブロックするかもしれません。また、VACUUMは通常、指定されたパーティションテーブルの全パーティションを処理しますが、このオプションが指定されると、パーティションテーブルに衝突するロックがある場合VACUUMは全パーティションを飛ばすようになります。

INDEX_CLEANUP

VACUUMに、無効なタプルを指しているインデックスのエントリーの削除を試みるよう指定します。これは普通は望まれる振舞いであり、バキュームされるテーブルに対してvacuum_index_cleanupオプションが偽に設定されていない限りデフォルトです。このオプションを偽に設定するのは、バキュームの実行をできる限り速くする必要がある場合には有用かもしれません。例えば、切迫したトランザクションIDの周回を避ける場合です(24.1.5を参照してください)。しかしながら、インデックスの掃除を定期的に行なわない場合、テーブルが修正されるに従い、インデックスは無効なタプルを蓄積し、テーブル自身もインデックスの掃除が完了するまで削除できない無効な行ポインタを蓄積していきますので、パフォーマンスは悪くなるでしょう。このオプションはインデックスを持たないテーブルには影響せず、FULLオプションが使われていれば無視されます。

TRUNCATE

VACUUMに、テーブルの最後にある空のページを切り詰め、切り詰めたページのディスクスペースをオペレーティングシステムに返すよう指定します。これは普通は望まれる振舞いであり、バキュームされるテーブルに対してvacuum_truncateオプションが偽に設定されていない限りデフォルトです。このオプションを偽に設定するのは、切り詰めが要求されているテーブルのACCESS EXCLUSIVEロックを回避するのに有用かもしれません。このオプションはFULLオプションが使われていれば無視されます。

PARALLEL

integer個のバックグラウンドワーカーを使用して、VACUUMのインデックスバキュームフェーズとインデックスクリーンアップフェーズを並列に実行します(各バキュームフェーズの詳細については、表 27.37を参照してください)。操作の実行に使用されるワーカーの数は、並列バキュームをサポートするリレーションのインデックスの数と同じです。この数はPARALLELオプションで指定されたワーカーの数によって制限され、max_parallel_maintenance_workersパラメータによってさらに制限されます。インデックスは、インデックスのサイズがmin_parallel_index_scan_sizeパラメータよりも大きい場合にのみ、並列バキュームに参加できます。integerで指定されたパラレルワーカー数が実行中に使用されることは保証されないことに注意してください。指定されたワーカー数より少ないワーカーでバキュームが実行されたり、ワーカーがまったくない状態で実行される可能性があります。1つのインデックスに使用出来るワーカーは1つだけです。よって、パラレルワーカーはテーブルに少なくとも2つのインデックスがある場合にのみ起動されます。バキュームのワーカーは、各フェーズの開始前に起動され、フェーズの終了時に終了します。これらの動作は将来のリリースで変更される可能性があります。このオプションはFULLオプションと一緒に使用することはできません。

boolean

選択されたオプションを有効にするか無効にするかを指定します。オプションを有効にするにはTRUE、ONまたは1と書くことができ、無効にするにはFALSE、OFFまたは0と書くことができます。boolean値は省略することもでき、その場合にはTRUEとみなされます。

integer

選択したオプションに渡される負でない整数値を指定します。

table_name

バキューム対象のテーブルまたはマテリアライズドビューの名前です(スキーマ修飾名も可)。指定したテーブルがパーティションテーブルの場合、そのすべてのリーフパーティションがバキュームされます。

column_name

解析の対象とする列名です。デフォルトは全列です。列リストが指定された場合はANALYZEも指定しなければいけません。

出力

VERBOSEが指定された場合、VACUUMは、現在処理中のテーブルを示す進行状況メッセージを表示します。同様に、テーブルについての各種の統計情報も表示されます。

注釈

テーブルをバキュームするためには、通常はテーブルの所有者もしくはスーパーユーザでなければなりません。しかしデータベースの所有者は共有カタログを除くデータベース内の全テーブルをバキュームすることができます。(共有カタログに関する制限は、データベース全体のVACUUMはスーパーユーザのみが実行可能であることを意味します。) VACUUMは、呼び出したユーザがバキュームするための権限を持たないテーブルはすべて飛ばします。

トランザクションブロック内でVACUUMを実行することはできません。

GINインデックスを持つテーブルでは、VACUUM(全構文)は待ち状態のインデックス挿入を主GINインデックス構造内の適切なところに移動させることにより、待ち状態のインデックス挿入をすべて完了させます。66.4.1を参照してください。

不要となった行を削除するため、実運用状態のデータベースに対しては定期的に(少なくとも毎晩)VACUUMを実行することを推奨します。また、テーブルに対して多数の行を追加/削除した後は、そのテーブルにVACUUM ANALYZEを発行することを推奨します。これによりシステムカタログに最近なされた全ての変更が反映されることになり、PostgreSQLの問い合わせプランナが、問い合わせ計画の作成時により良い選択をできるようになります。

FULLオプションを日常的に使用することは推奨しませんが、特殊なケースでは有用となる場合もあります。例えば、テーブル内のほとんど全ての行を削除または更新し、そのテーブルによるディスクの使用量を物理的に縮小させて高速なテーブルスキャンを行いたい場合です。VACUUM FULLはたいていの場合、通常のVACUUMよりもテーブルを縮小します。

PARALLELオプションはバキュームの用途でのみ使用されます。このオプションをANALYZEオプションで指定した場合、ANALYZEには影響しません。

VACUUMによりI/Oトラフィックがかなり増大しますので、実行中の他のセッションの性能が悪化する可能性があります。このため、コストベースのバキューム遅延機能の使用が推奨される場合があります。並列バキュームの場合、各ワーカーはそのワーカーが行った作業に比例してスリープします。詳細は19.4.4を参照してください。

PostgreSQLには、バキューム保守作業を自動化する「autovacuum」機能があります。自動バキューム処理および手作業によるバキューム処理に関する詳細については、24.1を参照してください。

例

onek というテーブル1つだけを掃除し、オプティマイザ用に解析し、バキューム処理の詳細な報告を出力するには、次のようにします。

```
VACUUM (VERBOSE, ANALYZE) onek;
```

互換性

標準SQLにはVACUUM文はありません。

関連項目

vacuumdb, 19.4.4, 24.1.6

VALUES

VALUES — 行セットを計算する

概要

```
VALUES ( expression [, ...] ) [, ...]
    [ ORDER BY sort_expression [ ASC | DESC | USING operator ] [, ...] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start [ ROW | ROWS ] ]
    [ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY ]
```

説明

VALUES は、値の式で指定された行あるいは行の集合を計算します。大きなコマンドの中で「定数テーブル」を作成するために使用することが多いですが、それ単独で使用することも可能です。

複数行を指定した場合は、すべての行の要素数が同じでなければなりません。できあがるテーブル列のデータ型を決定するには、明示的に指定されている型やその列に登場する式から推測できる型を組み合わせて使用します。これは UNION と同じ方式です (10.5 を参照ください)。

大きなコマンドの中において、SELECT が文法上使える場所ならどこでもVALUESを使用することができます。文法上はSELECTと同じ扱いであるため、ORDER BY、LIMIT(、これと等価なFETCH FIRST)そしてOFFSET句をVALUESコマンドで使用することができます。

パラメータ

expression

定数あるいは式です。これを計算した結果が、表 (行セット) の中の指定した場所に挿入されます。VALUES リストを INSERT の最上位レベルで使用する場合は、expression を DEFAULT で置き換えることができます。これは、その列のデフォルト値を挿入することを表します。他の場所で VALUES を使用する場合には、DEFAULT は使用できません。

sort_expression

式あるいは整数の定数で、結果の行をソートする方法を表します。この式は、VALUES の結果の列を column1、column2などのように参照することができます。詳細はSELECT文書のORDER BY Clauseを参照ください。

operator

ソート用の演算子です。詳細はSELECT文書のORDER BY Clauseを参照ください。

count

返す行の最大数です。詳細はSELECT文書のLIMIT Clauseを参照ください。

start

結果を返す際に読み飛ばす行数です。詳細はSELECT文書のLIMIT Clauseを参照ください。

注釈

VALUES で大量の行を扱うことは避けるべきです。メモリ不足や性能の劣化を生じさせる可能性があります。VALUES を INSERT の中で使用する場合は特別です。(列の型は INSERT 先のテーブルからわかるので、VALUES のリストを調べて型を推測する必要がないからです) そのため、他の場面に比べて大きなリストを扱っても実用に耐えます。

例

必要最小限の VALUES コマンドはこのようになります。

```
VALUES (1, 'one'), (2, 'two'), (3, 'three');
```

これは、列が二つで行が三つの表を返します。事実上、これは次と同じことです。

```
SELECT 1 AS column1, 'one' AS column2
UNION ALL
SELECT 2, 'two'
UNION ALL
SELECT 3, 'three';
```

通常は、VALUES は大きな SQL コマンドの内部で使用します。最もよくあるのは、INSERT での使用です。

```
INSERT INTO films (code, title, did, date_prod, kind)
    VALUES ('T_601', 'Yojimbo', 106, '1961-06-16', 'Drama');
```

INSERT 内で使用する場合には、VALUES のリストに DEFAULT を指定することができます。これは、値を具体的に指定するのではなくその列のデフォルトを使用することを表します。

```
INSERT INTO films VALUES
    ('UA502', 'Bananas', 105, DEFAULT, 'Comedy', '82 minutes'),
    ('T_601', 'Yojimbo', 106, DEFAULT, 'Drama', DEFAULT);
```

VALUES は、副SELECTが書ける場所に使用することができます。例えば FROM 句の中などでも使えます。

```
SELECT f.*
  FROM films f, (VALUES('MGM', 'Horror'), ('UA', 'Sci-Fi')) AS t (studio, kind)
  WHERE f.studio = t.studio AND f.kind = t.kind;

UPDATE employees SET salary = salary * v.increase
  FROM (VALUES(1, 200000, 1.2), (2, 400000, 1.4)) AS v (depno, target, increase)
```

```
  WHERE employees.depno = v.depno AND employees.sales >= v.target;
```

VALUES を FROM 句の中で使用する場合には、AS 句が必須となることに注意しましょう。これは SELECT の場合と同様です。AS 句ですべての列の名前を指定する必要はありませんが、指定しておくことをお勧めします (VALUES のデフォルトの列名は、PostgreSQL においては column1、column2 のようになります。しかし、他のデータベースシステムでは異なるかもしれません)。

VALUES を INSERT の中で使用する場合は、値の型が挿入先列のデータ型に自動変換されます。それ以外の場面で使用する際には、正しいデータ型を指定する必要があるかもしれません。値がすべて引用符付きのリテラル定数である場合は、最初の値にだけ型を指定しておけば十分です。

```
SELECT * FROM machines
WHERE ip_address IN (VALUES('192.168.0.1'::inet), ('192.168.0.10'), ('192.168.1.43'));
```

ヒント

単に IN を試したいのなら、上のような VALUES クエリを使用するよりも IN のスカラリスト形式を使用するほうがよいでしょう。スカラリストの方法の方が記述量が減りますし、たいていはより効率的になります。

互換性

VALUESはSQL標準に従います。LIMITおよびOFFSETはPostgreSQLの拡張です。SELECTも参照してください。

関連項目

INSERT, SELECT

PostgreSQLクライアントアプリケーション

ここには、PostgreSQLクライアントアプリケーションとユーティリティについてのリファレンス情報があります。これらのコマンドがすべて汎用的なユーティリティであるという訳ではありません。一部は特定の権限を必要とします。これらアプリケーションの共通機能は、データベースサーバが稼働しているかどうかに依存しない、どのホストでも実行できるという点です。

コマンドラインから指定された場合、ユーザ名とデータベース名の大文字小文字は保持されます。空白文字や特殊文字がある場合は引用符付けが必要かもしれません。テーブル名やその他の識別子では文書化されていない限り大文字小文字は保持されませんので、引用符付けが必要かもしれません。

目次

clusterdb

clusterdb — PostgreSQLデータベースをクラスタ化する

概要

clusterdb [*connection-option*...] [**--verbose** | **-v**] [**--table** | **-t** *table*] ... [*dbname*]

clusterdb [*connection-option*...] [**--verbose** | **-v**] **--all** | **-a**

説明

clusterdbは、PostgreSQLデータベース内のテーブルを再クラスタ化するユーティリティです。既にクラスタ化されているテーブルを検索し、前回と同じインデックスを使用して再度クラスタ化します。一度もクラスタ化されていないテーブルは処理されません。

clusterdbは、SQLコマンドCLUSTERのラッパです。クラスタ化を行うのに、このユーティリティを使用しても、これ以外のサーバへのアクセス方法を使用しても、特別な違いはありません。

オプション

clusterdbでは、下記のコマンドライン引数を指定できます。

-a
--all

全てのデータベースをクラスタ化します。

[**-d**] *dbname*
[**--dbname=**]*dbname*

-a/--allが使用されていない場合、クラスタ化するデータベースの名前を指定します。これが指定されていない場合、データベース名は環境変数PGDATABASEから読み取られます。この変数も設定されていない場合は、接続のために指定されたユーザ名が使用されます。*dbname*は接続文字列でも構いません。その場合、接続文字列パラメータは衝突するコマンドラインオプションよりも優先します。

-e
--echo

clusterdbが生成し、サーバに送るコマンドをエコー表示します。

-q
--quiet

進行メッセージを表示しません。

-t table
--table=table

tableのみをクラスタ化します。複数の-tスイッチを記述することで複数のテーブルをクラスタ化することができます。

-v
--verbose

処理の間、詳細な情報を出力します。

-V
--version

clusterdbのバージョンを表示し、終了します。

-?
--help

clusterdbのコマンドライン引数の使用方法を表示し、終了します。

clusterdbは、さらに、下記のコマンドライン引数を接続パラメータとして受け付けます。

-h host
--host=host

サーバが稼働しているマシンのホスト名を指定します。この値がスラッシュから始まる場合、Unixドメインソケット用のディレクトリとして使用されます。

-p port
--port=port

サーバが接続を監視するTCPポートもしくはUnixドメインソケットファイルの拡張子を指定します。

-U username
--username=username

接続するためのユーザ名です。

-w
--no-password

パスワードの入力を促しません。サーバがパスワード認証を必要とし、かつ、.pgpassファイルなどの他の方法が利用できない場合、接続試行は失敗します。バッチジョブやスクリプトなどパスワードを入力するユーザが存在しない場合にこのオプションは有用かもしれません。

-W
--password

データベースに接続する前に、clusterdbは強制的にパスワード入力を促します。

サーバがパスワード認証を要求する場合clusterdbは自動的にパスワード入力を促しますので、これが重要になることはありません。しかし、clusterdbは、サーバにパスワードが必要かどうかを判断するため

の接続試行を無駄に行います。こうした余計な接続試行を防ぐために-Wの入力が有意となる場合もあります。

--maintenance-db=dbname

-a/--allが使われている場合に、どのデータベースをクラスタ化しなければならないかを見つけ出すために接続するデータベースの名前を指定します。指定されなければpostgresデータベースが使用され、もし存在しなければtemplate1が使用されます。これは接続文字列でも構いません。その場合、接続文字列パラメータは衝突するコマンドラインオプションよりも優先します。また、データベース名自身以外の接続文字列パラメータは、他のデータベースに接続する時に再利用されます。

環境

PGDATABASE
PGHOST
PGPORT
PGUSER

デフォルトの接続パラメータです。

PG_COLOR

診断メッセージで色を使うかどうかを指定します。可能な値はalways、auto、neverです。

また、このユーティリティは、他のほとんどのPostgreSQLユーティリティと同様、libpqでサポートされる環境変数を使用します（33.14を参照してください）。

診断

問題が発生した場合、考えられる原因とエラーメッセージについてはCLUSTERとpsqlを参照してください。データベースサーバは対象ホスト上で稼働していなければなりません。また、libpqフロントエンドライブラリの、あらゆるデフォルトの設定や環境変数が適用されます。

例

データベースtestをクラスタ化します。

```
$ clusterdb test
```

xyzzyというデータベース内のテーブルの1つfooをクラスタ化します。

```
$ clusterdb --table=foo xyzzy
```

関連項目

CLUSTER

createdb

createdb — 新しいPostgreSQLデータベースを作成する

概要

createdb [connection-option...] [option...] [dbname [description]]

説明

createdbは、新しいPostgreSQLデータベースを作成します。

通常、このコマンドを実行したデータベースユーザが、新しいデータベースの所有者になります。ただし、コマンドを実行するユーザが適切な権限を持っている場合、-0オプションを使用して別のユーザを所有者に指定することができます。

createdbはCREATE DATABASEというSQLコマンドのラッパです。したがって、このユーティリティでデータベースを作成しても、これ以外の方法でサーバにアクセスしてデータベースを作成しても何も違いはありません。

オプション

createdbでは、下記のコマンドライン引数を指定できます。

dbname

作成するデータベースの名前を指定します。この名前はクラスタ内の全てのPostgreSQLデータベースの中で一意でなければなりません。デフォルトでは、現在のシステムユーザと同じ名前でデータベースを作成します。

description

新しく作成されるデータベースに関連付けるコメントを指定します。

-D *tablespace*
--tablespace=*tablespace*

データベース用のデフォルトのテーブル空間を指定します。(この名前は二重引用符で囲まれた識別子として処理されます。)

-e
--echo

createdbが生成し、サーバに送信するコマンドをエコー表示します。

-E encoding
--encoding=encoding

このデータベース内で使用する文字符号化方式を指定します。PostgreSQLサーバでサポートされる文字セットについては23.3.1で説明します。

-l locale
--locale=locale

このデータベースで使用されるロケールを指定します。これは--lc-collateと--lc-ctypeの両方を指定することと等価です。

--lc-collate=locale

このデータベースで使用されるLC_COLLATE設定を指定します。

--lc-ctype=locale

このデータベースで使用されるLC_CTYPE設定を指定します。

-O owner
--owner=owner

新しいデータベースの所有者となるデータベースユーザを指定します。(この名前は二重引用符で囲まれた識別子として処理されます。)

-T template
--template=template

このデータベースの構築に使用するテンプレートデータベースを指定します。(この名前は二重引用符で囲まれた識別子として処理されます。)

-V
--version

createdbのバージョンを表示し、終了します。

-?
--help

createdbのコマンドライン引数の使用方法を表示し、終了します。

オプション-D、-l、-E、-O、および-Tは、基盤となるCREATE DATABASEというSQLコマンドのオプションにそれぞれ対応しています。詳細はそちらを参照してください。

またcreatedbは、以下のコマンドライン引数を接続パラメータとして受け付けます。

-h host
--host=host

サーバが稼働しているマシンのホスト名を指定します。この値がスラッシュから始まる場合、Unixドメインソケット用のディレクトリとして使用されます。

-p port
--port=port

サーバが接続を監視するTCPポートもしくはUnixドメインソケットのファイル拡張子を指定します。

-U username
--username=username

接続に使用するユーザ名を指定します。

-w
--no-password

パスワードの入力を促しません。サーバがパスワード認証を必要とし、かつ、.pgpassファイルなどの他の方法が利用できない場合、接続試行は失敗します。バッチジョブやスクリプトなどパスワードを入力するユーザが存在しない場合にこのオプションは有用かもしれません。

-W
--password

データベースに接続する前に、createdbは強制的にパスワード入力を促します。

サーバがパスワード認証を要求する場合createdbは自動的にパスワード入力を促しますので、これが重要になることはありません。しかし、createdbは、サーバにパスワードが必要かどうかを判断するための接続試行を無駄に行います。こうした余計な接続試行を防ぐために-Wの入力が有意となる場合もあります。

--maintenance-db=dbname

新しいデータベースを作成する時の接続先となるデータベースの名前を指定します。指定がなければ、postgresデータベースが使用されます。もし存在しなければ(またはこれが作成しようとしているデータベースの名前であれば)template1が使用されます。これは接続文字列でも構いません。その場合、接続文字列パラメータは衝突するコマンドラインオプションよりも優先します。

環境

PGDATABASE

この値が設定されている場合、コマンドラインで上書きされなければ、設定された値が作成するデータベースの名前になります。

PGHOST
PGPORT
PGUSER

デフォルトの接続パラメータです。コマンドラインでもPGDATABASEでも名前が指定されていない場合は、PGUSERが作成するデータベースの名前にもなります。

PG_COLOR

診断メッセージで色を使うかどうかを指定します。可能な値はalways、auto、neverです。

また、このユーティリティは、他のほとんどのPostgreSQLユーティリティと同様、libpqでサポートされる環境変数を使用します（33.14を参照してください）。

診断

問題が発生した場合、考えられる原因とエラーメッセージの説明についてCREATE DATABASEとpsqlを参照してください。データベースサーバは対象ホストで稼働していなければなりません。また、libpqフロントエンドライブラリで使用される、全てのデフォルトの接続設定と環境変数が適用されることも覚えておいてください。

例

デフォルトのデータベースサーバを使用してdemoデータベースを作成します。

```
$ createdb demo
```

edenホスト上のポート番号5000のサーバを使用し、template0テンプレートデータベースを使用してdemoデータベースを作成する場合の、コマンド行から入力するコマンドと背後で実行されるSQLコマンドを示します。

```
$ createdb -p 5000 -h eden -T template0 -e demo
CREATE DATABASE demo TEMPLATE template0;
```

関連項目

dropdb, CREATE DATABASE

createuser

createuser — 新しいPostgreSQLのユーザアカウントを定義する

概要

createuser [connection-option...] [option...] [username]

説明

createuserは新しいPostgreSQLのユーザ（より正確にいえばロール）を作成します。新しいユーザを作成できるのは、スーパーユーザとCREATEROLE権限を持つユーザのみです。したがって、createuserは、スーパーユーザもしくはCREATEROLE権限を持つユーザとして接続可能なユーザによって実行されなければなりません。

新しいスーパーユーザを作成したいのであれば、スーパーユーザとして接続しなければなりません。CREATEROLE権限だけではいけません。スーパーユーザであるということは、そのデータベースにおけるアクセス権限の検査を素通りできることを意味しています。したがって、スーパーユーザという地位を簡単に与えてはなりません。

createuserはSQLコマンドCREATE ROLEのラッパです。このユーティリティによってユーザを作成しても、これ以外の方法でサーバにアクセスしてユーザを作成しても特に違いはありません。

オプション

createuserでは、下記のコマンドライン引数を指定できます。

username

作成するPostgreSQLユーザの名前を指定します。この名前は、そのPostgreSQLインストレーションに存在するすべてのロールと異なるものでなければなりません。

-c number
--connection-limit=number

新しいユーザの最大接続数を設定します。デフォルトでは無制限です。

-d
--createdb

新しいユーザに対してデータベースの作成を許可します。

-D
--no-createdb

新しいユーザに対してデータベースの作成を禁止します。これはデフォルトです。

-e
--echo

createuserが生成しサーバに送信するコマンドを出力します。

-E
--encrypted

このオプションは廃止されましたが、後方互換性のためにまだ受け付けられます。

-g role
--role=role

このロールが新しいメンバーとして即座に追加されるロールを示します。複数の-gスイッチを記述することで、このロールがメンバーとして追加される複数のロールを指定することができます。

-i
--inherit

新しいロールは自動的にメンバとして属するロールの権限を継承します。これがデフォルトです。

-I
--no-inherit

新しいロールは自動的にメンバとして属するロールの権限を継承しません。

--interactive

ユーザ名がコマンドラインで指定されない場合、ユーザ名の入力を促し、更に -d/-D、-r/-R、-s/-Sオプションがコマンドラインで指定されない場合にはどちらにするか入力を促します。(これはPostgreSQL 9.1までのデフォルトの動作でした。)

-l
--login

新しいユーザに対してログインを許可します。(つまり、このユーザ名をセッション起動時のユーザ識別子として使用することができます。) これがデフォルトです。

-L
--no-login

新しいユーザに対してログインを禁止します。(ログイン権限を持たないロールはデータベース権限管理という面で有意です。)

-P
--pwprompt

このオプションが指定されると、createuserは新しいユーザのパスワードのプロンプトを表示します。もしパスワード認証を使う予定がなければ、これは必要ありません。

-r
--createrole

新しいユーザに対して新しいロールの作成を許可します。(つまり、このユーザはCREATEROLE権限を持つことになります。)

-R
--no-createrole

新しいユーザに対して新しいロールの作成を禁止します。これはデフォルトです。

-s
--superuser

新しいユーザはスーパーユーザになります。

-S
--no-superuser

新しいユーザはスーパーユーザにはなりません。これはデフォルトです。

-V
--version

createuserのバージョンを表示し、終了します。

--replication

新しいユーザはREPLICATION権限を持ちます。この権限についてはCREATE ROLEの文書で詳しく説明します。

--no-replication

新しいユーザはREPLICATION権限を持ちません。この権限についてはCREATE ROLEの文書で詳しく説明します。

-?
--help

createuserのコマンドライン引数の使用方法を表示し、終了します。

createuserは、以下のコマンドライン引数も接続パラメータとして受け付けます。

-h host
--host=host

サーバが稼働しているマシンのホスト名を指定します。この値がスラッシュから始まる場合、Unixドメインソケット用のディレクトリとして使用されます。

-p port
--port=port

サーバが接続を監視するTCPポートもしくはUnixドメインソケットファイルの拡張子を指定します。

-U username
--username=username

接続に使用するユーザ名です（作成するユーザの名前ではありません）。

-w
--no-password

パスワードの入力を促しません。サーバがパスワード認証を必要とし、かつ、.pgpassファイルなどの他の方法が利用できない場合、接続試行は失敗します。バッチジョブやスクリプトなどパスワードを入力するユーザが存在しない場合にこのオプションは有用かもしれません。

-W
--password

createuserは強制的にパスワード入力を促します。(新しいユーザのパスワードではなく、サーバに接続するためのパスワードです)。

サーバがパスワード認証を要求する場合createuserは自動的にパスワード入力を促しますので、これが重要になることはありません。しかし、createuserは、サーバにパスワードが必要かどうかを判断するための接続試行を無駄に行います。こうした余計な接続試行を防ぐために-Wの入力が有意となる場合もあります。

環境

PGHOST
PGPORT
PGUSER

デフォルトの接続パラメータです。

PG_COLOR

診断メッセージで色を使うかどうかを指定します。可能な値はalways、auto、neverです。

また、このユーティリティは、他のほとんどのPostgreSQLユーティリティと同様、libpqでサポートされる環境変数を使用します(33.14を参照してください)。

診断

問題が発生した場合、考えられる原因とエラーメッセージの説明については、CREATE ROLEとpsqlを参照してください。データベースサーバは対象ホストで稼働していなければなりません。また、libpqフロントエンドライブラリで使用される、全ての接続設定と環境変数が適用されることを覚えておいてください。

例

デフォルトデータベースサーバ上にjoeというユーザを作成します。

```
$ createuser joe
```

デフォルトデータベースサーバ上にjoeというユーザを一部の属性入力が促されるように作成します。

```
$ createuser --interactive joe
```

```
Shall the new role be a superuser? (y/n) n
Shall the new role be allowed to create databases? (y/n) n
Shall the new role be allowed to create more new roles? (y/n) n
```

ホストedenのポート番号5000上のサーバを使って上記と同じjoeというユーザを属性を明示的に指定して作成し、背後で実行される問い合わせを表示します。

```
$ createuser -h eden -p 5000 -S -D -R -e joe
CREATE ROLE joe NOSUPERUSER NOCREATEDB NOCREATEROLE INHERIT LOGIN;
```

joeというユーザをスーパーユーザとして作成します。作成時にパスワードを割り当てます。

```
$ createuser -P -s -e joe
Enter password for new role: xyzzy
Enter it again: xyzzy
CREATE ROLE joe PASSWORD 'md5b5f5ba1a423792b526f799ae4eb3d59e' SUPERUSER CREATEDB CREATEROLE
 INHERIT LOGIN;
```

上の例で、実際には入力した新しいパスワードは画面上に表示されませんが、分かりやすくするために記載しています。上記の通りこのパスワードはクライアントに送信される前に暗号化されます。

関連項目

dropuser, CREATE ROLE

dropdb

dropdb — PostgreSQLデータベースを削除する

概要

dropdb [connection-option...] [option...] dbname

説明

dropdbは既存のPostgreSQLデータベースを削除します。このコマンドを実行できるのは、データベースのスーパーユーザまたはデータベースの所有者のみです。

dropdbは、SQLコマンドDROP DATABASEのラッパです。このユーティリティを使用しても、これ以外の方法でサーバにアクセスして削除しても、特に違いはありません。

オプション

dropdbは、下記のコマンドライン引数を受け付けます。

dbname

削除するデータベース名を指定します。

-e
--echo

dropdbが生成し、サーバに送信するコマンドをエコー表示します。

-f
--force

削除の前に対象データベースへの既存の接続をすべて終了することを試みます。このオプションに関する詳細な情報はDROP DATABASEを参照してください。

-i
--interactive

削除を行う前に、確認のためのプロンプトを表示します。

-V
--version

dropdbのバージョンを表示し、終了します。

--if-exists

指定したデータベースが存在しない場合でもエラーとしません。この場合には注意が発生します。

-?
--help

dropdbのコマンドライン引数の使用方法を表示し、終了します。

またdropdbは、以下のコマンドライン引数を接続パラメータとして受け付けます。

-h host
--host=host

サーバが稼働しているマシンのホスト名を指定します。この値がスラッシュから始まる場合、Unixドメインソケット用のディレクトリとして使用されます。

-p port
--port=port

サーバが接続を監視するTCPポートもしくはUnixドメインソケットファイルの拡張子を指定します。

-U username
--username=username

接続するユーザ名を指定します。

-w
--no-password

パスワードの入力を促しません。サーバがパスワード認証を必要とし、かつ、.pgpassファイルなどの他の方法が利用できない場合、接続試行は失敗します。バッチジョブやスクリプトなどパスワードを入力するユーザが存在しない場合にこのオプションは有用かもしれません。

-W
--password

データベースに接続する前に、dropdbは強制的にパスワード入力を促します。

サーバがパスワード認証を要求する場合dropdbは自動的にパスワード入力を促しますので、これが重要になることはありません。しかし、dropdbは、サーバにパスワードが必要かどうかを判断するための接続試行を無駄に行います。こうした余計な接続試行を防ぐために-Wの入力が有意となる場合もあります。

--maintenance-db=dbname

対象データベースを削除するために接続するデータベースの名前を指定します。指定されない場合は、postgresデータベースが使用されます。このデータベースが存在しない場合（またはこのデータベースが削除中である場合）template1が使用されます。これは接続文字列でも構いません。その場合、接続文字列パラメータは衝突するコマンドラインオプションよりも優先します。

環境

PGHOST
PGPORT
PGUSER

デフォルトの接続パラメータです。

PG_COLOR

診断メッセージで色を使うかどうかを指定します。可能な値はalways、auto、neverです。

また、このユーティリティは、他のほとんどのPostgreSQLユーティリティと同様、libpqでサポートされる環境変数を使用します（33.14を参照してください）。

診断

問題が発生した場合、考えられる原因とエラーメッセージについてはDROP DATABASEとpsqlを参照してください。対象ホストでデータベースサーバが稼働していなければなりません。また、libpqのフロントエンドライブラリの、あらゆるデフォルトの設定や環境変数が適用されます。

例

次のコマンドは、デフォルトのデータベースサーバ上のdemoデータベースを削除します。

```
$ dropdb demo
```

次のコマンドはホストedenのポート番号5000で動作しているサーバからデータベースdemoを削除します。その際、削除を確認し、またバックエンドに送られるコマンドを表示します。

```
$ dropdb -p 5000 -h eden -i -e demo
Database "demo" will be permanently deleted.
Are you sure? (y/n) y
DROP DATABASE demo;
```

関連項目

createdb, DROP DATABASE

dropuser

dropuser — PostgreSQLのユーザアカウントを削除する

概要

dropuser [connection-option...] [option...] [username]

説明

dropuserは、既存のPostgreSQLのユーザを削除します。PostgreSQLのユーザを削除することができるのは、スーパーユーザとCREATEROLE権限があるユーザのみです。

dropuserはSQLコマンドDROP ROLEのラッパです。このユーティリティを使用してユーザを削除しても、この方法以外の方法でサーバでアクセスしてユーザを削除しても特に違いはありません。

オプション

dropuserは、下記のコマンドライン引数を受け付けます。

username

削除するPostgreSQLのユーザ名を指定します。コマンドラインで指定されず、かつ-i/--interactiveオプションが使用されている場合は、入力を促すプロンプトが表示されます。

-e
--echo

dropuserが生成し、サーバに送信するコマンドを表示します。

-i
--interactive

実際にユーザを削除する前に確認のプロンプトを表示します。コマンドラインにてユーザ名が指定されなかった場合にユーザ名の入力を促します。

-V
--version

dropuserのバージョンを表示し、終了します。

--if-exists

ユーザが存在しない場合にエラーを発生しません。この場合は注意が発生します。

-?
--help

dropuserのコマンドライン引数の使用方法を表示し、終了します。

dropuserは以下のコマンドライン引数も接続パラメータとして受け付けます。

-h host
--host=host

サーバが稼働しているマシンのホスト名を指定します。この値がスラッシュから始まる場合、Unixドメインソケット用のディレクトリとして使用されます。

-p port
--port=port

サーバが接続を監視するTCPポートもしくはUnixドメインソケットのファイル拡張子を指定します。

-U username
--username=username

接続に使用するユーザ名です（削除するユーザ名ではありません）。

-w
--no-password

パスワードの入力を促しません。サーバがパスワード認証を必要とし、かつ、.pgpassファイルなどの他の方法が利用できない場合、接続試行は失敗します。バッチジョブやスクリプトなどパスワードを入力するユーザが存在しない場合にこのオプションは有用かもしれません。

-W
--password

データベースに接続する前に、dropuserは強制的にパスワード入力を促します。

サーバがパスワード認証を要求する場合dropuserは自動的にパスワード入力を促しますので、これが重要になることはありません。しかし、dropuserは、サーバにパスワードが必要かどうかを判断するための接続試行を無駄に行います。こうした余計な接続試行を防ぐために-Wの入力が有意となる場合もあります。

環境

PGHOST
PGPORT
PGUSER

デフォルトの接続パラメータです。

PG_COLOR

診断メッセージで色を使うかどうかを指定します。可能な値はalways、auto、neverです。

また、このユーティリティは、他のほとんどのPostgreSQLユーティリティと同様、libpqでサポートされる環境変数を使用します（33.14を参照してください）。

診断

問題が発生した場合、考えられる原因とエラーメッセージについてはDROP ROLEとpsqlを参照してください。データベースサーバは対象ホスト上で稼働していなければなりません。また、libpqフロントエンドライブラリで使用される、あらゆるデフォルトの設定や環境変数が適用されます。

例

デフォルトのデータベースサーバから、ユーザjoeを削除します。

```
$ dropuser joe
```

ホストedenでポート5000を使用しているサーバから、ユーザjoeを削除します。このとき、背後で実行されるコマンドの表示と、削除前の確認をします。

```
$ dropuser -p 5000 -h eden -i -e joe
Role "joe" will be permanently removed.
Are you sure? (y/n) y
DROP ROLE joe;
```

関連項目

createuser, DROP ROLE

ecpg

ecpg — 埋め込みSQL用Cプリプロセッサ

概要

ecpg [option...] file...

説明

ecpgは、Cプログラム用の埋め込みSQLプリプロセッサです。SQL呼び出しを特別な関数呼び出しに置き換えることによって、埋め込みSQL文を含むCプログラムを、通常のCコードに変換します。これにより、出力ファイルは、任意のCコンパイラツールを使用して処理することができます。

ecpgは、コマンドラインで指定される各入力ファイルを対応するCの出力ファイルに変換します。入力ファイル名に拡張子がなければ、.pgcを仮定します。出力ファイル名を構成するために、拡張子が.cに置き換えられます。しかし、出力ファイル名は-oオプションによって指定でき、こちらが優先します。

入力ファイルが-だけであれば、ecpgはプログラムを標準入力から読み込み(-oで上書きされていなければ、標準出力へ書き出し)ます。

このマニュアルページでは埋め込みSQL言語については説明しません。第35章を参照してください。

オプション

ecpgは、以下のコマンドライン引数を受け付けます。

-c

SQLコードから有効なCコードを自動的に生成します。現在、このオプションはEXEC SQL TYPEに対して使用できます。

-C mode

互換モードを設定します。modeはINFORMIX、INFORMIX_SE、ORACLEのどれかを取ることができます。

-D symbol

Cプリプロセッサのシンボルを定義します。

-h

ヘッダファイルを処理します。このオプションが指定されると、出力ファイルの拡張子は.cではなく.hになり、デフォルトの入力ファイルの拡張子は.pgcではなく.pghになります。また、-cオプションが強制的に有効になります。

-i

同様にシステムインクルードファイルも解析します。

-I directory

追加のインクルード用パスを指定します。これは、EXEC SQL INCLUDEを使用してインクルードされるファイルを検索する際に使用されます。デフォルトでは順に、.（カレントディレクトリ）、/usr/local/include、コンパイル時に定義されるPostgreSQLのインクルードディレクトリ（デフォルトでは/usr/local/pgsql/include）、/usr/includeです。

-o filename

ecpgが全ての出力をfilenameに書き込むことを指定します。出力をすべて標準出力に送るには-o -と書いてください。

-r option

実行時の動作を選択します。以下のいずれかをoptionとして取ることができます。

no_indicator

指示子を使用せずにNULL値を表す特殊な値を使用します。歴史的にこの方式を使用したデータベースが存在します。

prepare

すべての文を使用する前に準備(プリペア)します。libecpgはプリペアド文のキャッシュを保持し、再実行される場合に文を再利用します。キャッシュが満杯になった場合、libecpgは最も使用されていない文を解放します。

questionmarks

互換性のためにクエスチョンマークをプレースホルダとして許します。これは大昔にデフォルトでした。

-t

トランザクションの自動コミットを有効にします。このモードでは、各SQLコマンドは明示的なトランザクションブロックの内部にない限り、自動的にコミットされます。デフォルトのモードでは、EXEC SQL COMMITが発行された時にのみコマンドがコミットされます。

-v

バージョンやインクルード用パスなどの補足情報を表示します。

--version

ecpgのバージョンを表示し、終了します。

-?
--help

ecpgのコマンドライン引数の使用方法を表示し、終了します。

注釈

前処理されたCコードファイルをコンパイルする際、コンパイラがPostgreSQLのインクルードディレクトリ内にあるECPGヘッダファイルを検索できるようにしなければなりません。そのため、コンパイラの呼び出し時に、-Iオプションを使用しなければならない可能性があります（例：-I/usr/local/pgsql/include）。

SQLが埋め込まれたCプログラムには、リンカオプション-L/usr/local/pgsql/lib -lecpgを使用するなどして、libecpgライブラリをリンクする必要があります。

使用するシステムにおいて上記の2つに対応するディレクトリを調べるには、pg_configを使用します。

例

埋め込みSQLを使用したprog1.pgcというCソースファイルがある場合、次のコマンドを順番に実行すれば、実行可能プログラムを作成することができます。

```
ecpg prog1.pgc
cc -I/usr/local/pgsql/include -c prog1.c
cc -o prog1 prog1.o -L/usr/local/pgsql/lib -lecpg
```

pg_basebackup

pg_basebackup — PostgreSQLクラスタのベースバックアップを取得する

概要

pg_basebackup [option...]

説明

pg_basebackupは、稼動中のPostgreSQLのデータベースクラスタのベースバックアップを取るために使用されます。データベースへの他のクライアントに影響することなく、バックアップが取られます。また、このバックアップはポイントインタイムリカバリ(25.3参照)とログシッピングやストリーミングレプリケーションスタンバイサーバ用の開始点(26.2参照)としても使用できます。

pg_basebackupは、サーバをバックアップモードに入れ、また戻すことを自動的に確実に行ない、データベースクラスタファイルの厳密なコピーを作成します。バックアップは常にデータベースクラスタ全体のバックアップを取ります。個々のデータベースや個々のデータベースオブジェクトをバックアップすることはできません。特定のものを対象としたバックアップに関してはpg_dumpなどの他のツールを使用しなければなりません。

バックアップは、レプリケーションプロトコルを用いる通常のPostgreSQL接続を経由して作成されます。この接続はREPLICATION権限(21.2参照)を持つ、または、スーパーユーザであるユーザIDが確立しなければなりません。さらにpg_hba.confでレプリケーション用の接続が許可されていなければなりません。またサーバでmax_wal_sendersを、バックアップ用に少なくとも1つのwalsenderとWALストリーミング用にもう1つ(使用する場合)を残すように十分大きく設定する必要があります。

同時にpg_basebackupを複数実行できます。しかし性能の観点からは、1つのバックアップのみを取り結果をコピーする方が通常は優れています。

pg_basebackupは、プライマリサーバからだけではなくスタンバイからもベースバックアップを作成できます。スタンバイからバックアップを取得するためには、レプリケーション接続を受け付けられるようにスタンバイを設定してください(つまりmax_wal_sendersとhot_standbyを設定し、pg_hba.confを適切に設定してください)。またプライマリでfull_page_writesを有効にする必要があります。

スタンバイからバックアップを取る場合にはいくつかの制限があることに注意してください。

- バックアップ履歴ファイルはバックアップされるデータベースクラスタ内に作成されません。
- -X noneを使用している場合、バックアップ終了時にバックアップに必要なすべてのWALファイルがアーカイブされているという保証はありません。
- バックアップ中にスタンバイがプライマリに昇格した場合、バックアップは失敗します。
- バックアップに必要なすべてのWALレコードは、必要なだけの完全ページ書き出しを含んでいなければなりません。つまりこれは、プライマリでfull_page_writesを有効にし、archive_commandとしてWALファイルから完全ページ書き出しを取り除くpg_compresslogのようなツールを使用しないことが要求されます。

pg_basebackupがベースバックアップを取るときには必ず、サーバのpg_stat_progress_basebackupはバックアップの進行状況を報告します。詳細は27.4.5を参照してください。

オプション

以下のコマンドラインオプションは出力の場所と書式を制御します。

-D directory
--pgdata=directory

出力を書き出す対象のディレクトリを設定します。pg_basebackupは、存在していなければこのディレクトリ(とその親ディレクトリのうち存在していないものすべて)を作成します。ディレクトリはすでに存在してもかまいませんが、空でなければなりません。

バックアップがtar形式であり、かつ、対象のディレクトリが-(ダッシュ)の場合、tarファイルをstdoutに書き出します。

このオプションは必須です。

-F format
--format=format

出力形式を選択します。formatには以下のいずれかを取ることができます。

p
plain

普通のファイルとして、ソースサーバのデータディレクトリとテーブル空間と同じレイアウトで、出力を書き出します。クラスタがテーブル空間を追加で持たない場合、データベース全体が指定したディレクトリに格納されます。クラスタが追加のテーブル空間を持つ場合は、主データディレクトリは指定したディレクトリ内に格納されますが、他のテーブル空間はすべて、ソースサーバ上と同じ絶対パスに格納されます。

これがデフォルトの書式です。

t
tar

指定したディレクトリ内にtarファイルとして出力を書き出します。主データディレクトリの内容はbase.tarという名前のファイルに書き出され、他のテーブル空間はすべてテーブル空間のOIDに因んだ名前の別のtarファイルに書き出されます。

対象ディレクトリとして-(ダッシュ)という値が指定された場合、tarの内容は標準出力に書き出されます。これは(例えば)gzipへのパイプ処理に適しています。これはクラスタが追加テーブル空間を持たず、WALストリーミングを使用していない場合のみ行うことができます。

-R
--write-recovery-conf

standby.signalを作成し、対象のディレクトリ(tar形式の場合はベースアーカイブファイルの中)にあるpostgresql.auto.confに接続設定を追加します。これにより、バックアップの結果を利用するスタンバイサーバの設定が容易になります。

postgresql.auto.confファイルは接続設定の情報、また、指定があればpg_basebackupが使用しているレプリケーションスロットの情報を記録するので、ストリーミングレプリケーションは後で同じ設定を使用します。

-T olddir=newdir
--tablespace-mapping=olddir=newdir

ディレクトリolddirにあるテーブル空間を、バックアップ中にnewdirに再配置します。これが有効であるためには、olddirが、ソースサーバでのテーブル空間のパス定義と完全に一致している必要があります。(ただし、ソースサーバのolddir内にテーブル空間がなくてもエラーにはなりません。) 一方、newdirは受け取るホストのファイルシステム内のディレクトリです。主対象ディレクトリと同様に、newdirは既に存在している必要はありませんが、存在している場合には空でなければなりません。olddirとnewdirはいずれも絶対パスでなければなりません。パス名に等号(=)を含む必要がある場合には、バックスラッシュをその前に付けてください。このオプションは、複数のテーブル空間に対して複数回指定することができます。

この方法でテーブル空間を再配置すると、メインのデータディレクトリ内のシンボリックリンクは、新しい場所を指すように更新されます。このため、新しいデータディレクトリは、すべてのテーブル空間が更新された場所にあり、新しいサーバインスタンスがすぐに使える状態になっています。

--waldir=waldir

WAL(先行書き込みログ)ファイルを書き込むディレクトリを設定します。デフォルトでは、WALファイルは対象のpg_walサブディレクトリに置かれますが、どこか他の場所に置くためにこのオプションが使えます。waldirは絶対パスでなければなりません。主対象ディレクトリと同様に、waldirは既に存在している必要はありませんが、存在している場合には空でなければなりません。このオプションは、バックアップがplain形式の場合にのみ指定できます。

-X method
--wal-method=method

必要なWAL(先行書き込みログ)ファイルをバックアップに含めます。これはバックアップ中に生成された先行書き込みログをすべて含みます。方法noneを指定しない場合、ログアーカイブを考慮することなく展開した対象ディレクトリ内でpostmasterを起動することができます。つまり出力は完全なスタンドアローンバックアップを作成します。

以下の先行書き込みログを収集するためのmethodがサポートされます。

n
none

先行書き込みログをバックアップに含めません。

f
fetch

先行書き込みログファイルはバックアップの最後に収集されます。したがって、ソースサーバのwal_keep_sizeパラメータを、バックアップの最後までログが削除されない程度に十分大きくする必要があります。要求されるログデータが転送時点で再利用されていた場合、バックアップは失敗し、使用することができません。

tar形式が使われれば、先行書き込みログファイルはbase.tarファイルに含まれます。

s
stream

バックアップを取る時に先行書き込みログデータをストリームします。この方法は第2のサーバ接続を開き、バックアップを実行している間、並行して先行書き込みログのストリーミングを始めます。したがって、レプリケーション接続を、1つではなく2つ使用します。クライアントが先行書き込みログデータに追従している限り、このモードを使用すれば、ソースサーバ上に余分に保管される先行書き込みログは必要ありません。

tar形式が使われれば、先行書き込みログファイルはpg_wal.tarという名の別のファイルに書き込まれます（サーバが10より前のバージョンの場合、pg_xlog.tarというファイル名になります）。

この値がデフォルトです。

-z
--gzip

tarファイル出力のデフォルトの圧縮レベルによるgzip圧縮を有効にします。tarファイルを生成する場合のみ圧縮を利用することができ、すべてのtarファイルの名前に拡張子.gzが自動的に付加されます。

-Z level
--compress=level

tarファイル出力のgzip圧縮を有効にします。また圧縮レベル（0から9まで、0は圧縮なし、9が最高の圧縮レベル）を指定します。tarファイルを生成する場合のみ圧縮を利用することができ、すべてのtarファイルの名前に拡張子.gzが自動的に付加されます。

以下のコマンドラインオプションはバックアップの生成とこのプログラムの起動を制御します。

-c fast|spread
--checkpoint=fast|spread

チェックポイントモードをfast(即座に発行)またはspread(デフォルト)に設定します(25.3.3を参照してください)。

-C
--create-slot

バックアップ開始前に--slotオプションで名づけられたレプリケーションスロットを作成することを指定します。スロットが既に存在する場合、エラーが生じます。

-l label
--label=label

バックアップのラベルを設定します。何も指定がない場合、「pg_basebackup base backup」というデフォルト値が使用されます。

-n
--no-clean

デフォルトでは、pg_basebackupがエラーでアボートするとき、ジョブを完了できないことがわかるより前に作成したすべてのディレクトリ（例えば、対象のディレクトリと先行書き込みログのディレクトリ）を削除します。このオプションはきれいに片付けることを禁止するので、デバッグのために有用です。

テーブル空間のディレクトリはいずれにせよ削除されないことに注意してください。

-N
--no-sync

デフォルトではpg_basebackupは全てのファイルがディスクに安全に書き出されるのを待ちます。このオプションでpg_basebackupは待つことなく返ります。これは高速ですが、その後のオペレーションシステムのクラッシュでベースバックアップの破損が残るかもしれないことを意味します。一般にこのオプションはテスト用に有用なのであって、本番導入の際に使うべきではありません。

-P
--progress

進行状況報告を有効にします。これを有効にすると、バックアップ中におおよその進行状況が報告されます。データベースはバックアップ中に変更があるかもしれませんので、これはおおよそでしかなくちょうど100%では終わらないかもしれません。特に、WALログがバックアップに含まれる場合、データ総量は前もって予測することはできません。このためこの場合、推定対象容量はWALなしの総推定量を過ぎた後増加します。

-r rate
--max-rate=rate

ソースサーバから収集されるデータの最大転送速度です。これは、サーバでのpg_basebackupの影響を制限するのに有用です。値は秒あたりのキロバイト数です。添字Mを使うと秒あたりのメガバイト数を指定できます。添字kを使うこともできますが、効果はありません。有効な値は秒あたり32キロバイトから秒あたり1024メガバイトまでです。

このオプションはデータディレクトリの転送に対しては、常に影響があります。WALファイルの転送については、収集方法がfetchの場合にのみ影響があります。

-S slotname
--slot=slotname

このオプションは-X streamと一緒でのみ使用できます。これはWALストリーミングに指定したレプリケーションスロットを使用させます。レプリケーションスロットを使うストリーミングレプリケーションのスタンバイとしてベースバックアップを使用するつもりであるなら、スタンバイはprimary_slot_nameと同じレプリケーションスロット名を使うべきです。これにより、ベースバックアップ終了と新しいスタンバイでのストリーミングレプリケーション開始の間の時間にプライマリサーバが必要なWALデータを削除しないことが確実になります。

指定されたレプリケーションスロットは、オプション-Cも使われている場合を除き、存在していなければなりません。

このオプションが指定されておらず、サーバが一時レプリケーションスロットに対応している（バージョン10以降）場合、WALストリーミングに対して一時レプリケーションスロットが自動的に使われます。

-v
--verbose

冗長モードを有効にします。開始時および終了段階でいくつか追加の段階が出力されます。また進行状況報告も有効な場合、現在処理中のファイル名も正しく出力されます。

--manifest-checksums=algorithm

バックアップマニフェストに含まれる各ファイル適用されるチェックサムアルゴリズムを指定します。現在利用できるアルゴリズムはNONE、CRC32C、SHA224、SHA256、SHA384、SHA512です。デフォルトはCRC32Cです。

NONEが選択されれば、バックアップマニフェストはチェックサムを含みません。それ以外の場合、バックアップ内の各ファイルの指定したアルゴリズムを使ったチェックサムを含みます。さらに、マニフェストは自身の内容のSHA256チェックサムを常に含みます。SHAアルゴリズムはCRC32Cよりもかなりの CPU集約的なため、そのどれかを1つを選択するとバックアップの完了に掛かる時間が増えるでしょう。

SHAハッシュ関数を使うと、バックアップが変更されていないことを検証したい利用者にとっては暗号学的に安全な各ファイルのダイジェストが提供されますが、一方、CRC32Cアルゴリズムでは計算がずっと速いチェックサムが提供されます。偶発的な変更によるエラーを補足するのには良いですが、悪意のある修正に抵抗力はありません。バックアップにアクセスした敵に対抗するのに有用なように、バックアップマニフェストは、どこか他のところに安全に保管するか、バックアップが取られて以来変更されたことがないのを検証する必要があることに注意してください。

pg_verifybackupを使ってバックアップマニフェストに対するバックアップの完全性を検査できます。

--manifest-force-encode

バックアップマニフェスト内のファイル名をすべて強制的に16進数でエンコードします。このオプションが指定されなければ、UTF8でないファイル名だけが16進数でエンコードされます。このオプションは主に、バックアップマニフェストファイルを読むツールが、この場合を正しく扱うか試験することを意図しています。

--no-estimate-size

ストリームされるバックアップデータの総量をサーバが評価しないようにします。その結果、pg_stat_progress_basebackupビューのbackup_total列は常にNULLになります。

このオプションがなければ、バックアップはまずデータベース全体容量を計算し、その後バックアップに戻り、実際の内容を送信します。これにより、バックアップに要する時間は少し長くなるかもしれません。特に最初のデータが送られるようになるまでの時間がより長くなります。このオプションは、評価時間が長過ぎる場合にそれを避けるのに有用です。

--progressを使う場合には、このオプションは認められません。

--no-manifest

バックアップマニフェストを生成しないようにします。このオプションが指定されなければ、サーバはpg_verifybackupを使って検証できるバックアップマニフェストを生成し送信します。マニフェストは、含まれるかもしれないWALファイルを除いて、バックアップの中にある各ファイルの一覧です。各ファイルの大きさや最終修正時刻、省略可能なチェックサムも保存されます。

--no-slot

バックアップ時に一時レプリケーションスロットを作成しないようにします。

デフォルトでは、ログストリーミングが選択されたものの-Sオプションでスロット名が与えられなかった場合、(ソースサーバがサポートしていれば)一時レプリケーションスロットが作成されます。

このオプションの主な目的は、サーバにレプリケーションスロットの空きが無いときにベースバックアップを取得できるようにすることです。レプリケーションスロットを使うことは、必要とされるWALがバックアップ中のサーバにより削除されることを防止するため、ほとんどの場合に好ましいです。

--no-verify-checksums

ベースバックアップ取得元のサーバでチェックサムの検証が有効になっている場合に、チェックサムの検証を無効化します。

デフォルトでは、チェックサムは検証され、チェックサムエラーは非ゼロの終了ステータスをもたらします。とはいえ、このような場合には、--no-cleanオプションが使われていたかのように、ベースバックアップは削除されません。チェックサム検証の失敗はpg_stat_databaseビューでも報告されます。

以下のオプションはソースサーバへの接続パラメータを制御します。

-d connstr
--dbname=connstr

サーバとの接続のために使用するパラメータを、接続文字列として指定します。衝突するコマンドラインオプションよりも優先します。

このオプションは他のクライアントアプリケーションとの整合性のために--dbnameと呼ばれます。しかし、pg_basebackupはクラスタ内の何らかの特定のデータベースに接続しませんので、接続文字列内のデータベース名は無視されます。

-h host
--host=host

サーバが稼働しているマシンのホスト名を指定します。この値がスラッシュから始まる場合、Unixドメインソケット用のディレクトリとして使用されます。デフォルトは、設定されていれば環境変数PGHOSTから取得されます。設定されていなければ、Unixドメインソケット接続とみなされます。

-p port
--port=port

サーバが接続を監視するTCPポートもしくはローカルUnixドメインソケットファイルの拡張子を指定します。デフォルトは、設定されている場合、環境変数PGPORTの値となります。設定されていなければ、コンパイル時のデフォルト値となります。

-s interval
--status-interval=interval

状態パケットがソースサーバに返送される間隔を秒単位で指定します。より小さな値を指定することで、より正確にサーバからバックアップの進行状況を監視できます。ゼロという値は定期的な状態更新を完全に無効にします。しかし、タイムアウトによる切断を防止するために、サーバにより要求された場合には更新が送信されます。デフォルト値は10秒です。

-U username
--username=username

接続ユーザ名を指定します。

-w
--no-password

パスワードの入力を促しません。サーバがパスワード認証を必要とし、かつ、.pgpassファイルなどの他の方法が利用できない場合、接続試行は失敗します。バッチジョブやスクリプトなどパスワードを入力するユーザが存在しない場合にこのオプションは有用かもしれません。

-W
--password

ソースサーバに接続する前に、pg_basebackupは強制的にパスワード入力を促します。

サーバがパスワード認証を要求する場合pg_basebackupは自動的にパスワード入力を促しますので、これが重要になることはありません。しかし、pg_basebackupは、サーバにパスワードが必要かどうかを判断するための接続試行を無駄に行います。こうした余計な接続試行を防ぐために-Wの入力が有意となる場合もあります。

以下のその他のオプションも使用することができます。

-V
--version

pg_basebackupのバージョンを表示し終了します。

-?
--help

pg_basebackupコマンドライン引数の使用方法を表示し、終了します。

環境

他のほとんどのPostgreSQLユーティリティと同様このユーティリティはlibpqでサポートされる環境変数（33.14参照）を使用します。

環境変数PG_COLORは診断メッセージで色を使うかどうかを指定します。可能な値はalways、auto、neverです。

注意

バックアップの開始時に、ソースサーバ上でチェックポイントを実行する必要があります。(特にオプション --checkpoint=fast を使用していない場合)これには少し時間を要する場合があり、その間 pg_basebackup はアイドル状態であるように見えます。

このバックアップには、設定ファイルやサードパーティによりディレクトリに格納された追加ファイルを含め、データディレクトリとテーブル空間内のすべてのファイルが含まれますが、PostgreSQLによって管理される一部の一時ファイルは含まれません。ただし、テーブル空間に使われるシンボリックリンクが保存されることを除くと、通常のファイルとディレクトリのみがコピーされます。PostgreSQLが認識している一部のディレクトリを指すシンボリックリンクは空のディレクトリとしてコピーされます。その他のシンボリックリンクおよび特殊デバイスファイルはスキップされます。(正確な詳細については52.4を参照してください。)

plain形式では、オプション--tablespace-mappingが使われなければ、テーブル空間はソースサーバ上のと同じパスでバックアップされます。このオプションがないと、サーバと同じホスト上でのplain形式のベースバックアップの実行は動作しません、というのは、バックアップを元のテーブル空間と同じディレクトリに書き込まなければならないからです。

tar形式を使う場合、そのデータを使うPostgreSQLサーバを起動する前に各tarファイルを解凍するのはユーザの責任です。追加のテーブル空間がある場合、それについてのtarファイルは、正しい場所に解凍される必要があります。この場合、テーブル空間へのシンボリックリンクは、base.tarファイルに含まれるtablespace_mapファイルの内容に基づいて、サーバが作成します。

pg_basebackupは同じまたは9.1以降のより古いメジャーバージョンのサーバで動作します。しかしWALストリーミングモード(-X stream)はバージョン9.3およびそれ以降のサーバでのみ動作します。また、現在のバージョンのtar形式(--format=tar)はバージョン9.5およびそれ以降のサーバでのみ動作します。

pg_basebackupは、ソースのクラスタでグループパーミッションが有効になっている場合、データファイルに対するグループパーミッションを維持します。

例

mydbserverで稼動するサーバのベースバックアップを作成し、ローカルディレクトリ/usr/local/pgsql/dataに保管します。

```
$ pg_basebackup -h mydbserver -D /usr/local/pgsql/data
```

各テーブル空間につき圧縮したtarファイルを1つ作成するようにローカルサーバをバックアップし、backupディレクトリに保管します。同時に実行時に進行状況を表示します。

```
$ pg_basebackup -D backup -Ft -z -P
```

単一のテーブル空間を持つローカルデータベースのバックアップを作成し、それをbzip2で圧縮します。

```
$ pg_basebackup -D - -Ft -X fetch | bzip2 > backup.tar.bz2
```

(データベース内に複数のテーブル空間が存在する場合このコマンドは失敗します。)

/opt/tsにあるテーブル空間を./backup/tsに再配置してローカルデータベースのバックアップを作成します。

```
$ pg_basebackup -D backup/data -T /opt/ts=$(pwd)/backup/ts
```

関連項目

pg_dump

pgbench

pgbench — PostgreSQLに対してベンチマーク試験を行う

概要

pgbench **-i** [*option*...] [*dbname*]

pgbench [*option*...] [*dbname*]

説明

pgbenchはPostgreSQL上でベンチマーク試験を行う単純なプログラムです。これは同一のSQLコマンドの並びを何度も実行します。複数の同時実行データベースセッションで実行することもできます。そして、トランザクションの速度（1秒当たりのトランザクション数）の平均を計算します。デフォルトでpgbenchは、1トランザクション当たり5つのSELECT、UPDATE、INSERTコマンドを含むおおよそTPC-Bに基いたシナリオを試験します。しかし、独自のトランザクションスクリプトファイルを作成することで他の試験ケースを簡単に実行することができます。

pgbenchの典型的な出力を以下に示します。

```
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 10
query mode: simple
number of clients: 10
number of threads: 1
number of transactions per client: 1000
number of transactions actually processed: 10000/10000
tps = 85.184871 (including connections establishing)
tps = 85.296346 (excluding connections establishing)
```

最初の6行はいくつかの最重要パラメータの設定を表示しています。次行が完了トランザクション数と予定トランザクション数です（後者は単なるクライアント数とクライアント毎のトランザクション数の積算結果です。）実行が完了する前に失敗しない限りこれは等しくなります。(-Tモードでは、トランザクションの実際の数が表示されます) 最後の2行は、データベースセッションを開始するための時間を含める場合と含めない場合の1秒当たりのトランザクション数を示します。

デフォルトのTPC-Bと似たトランザクション試験では、あらかじめ設定する特定のテーブルが必要です。これらのテーブルを作成し、データを投入するためには、-i(初期化)オプションを付けてpgbenchを呼び出さなければなりません。(独自スクリプトを試験する場合、この手順は必要ありません。しかし代わりに試験に必要な何らかの設定を行わなければならないでしょう。) 初期化は以下のようになります。

```
pgbench -i [ other-options ] dbname
```

ここでdbnameは試験用に前もって作成されたデータベースの名前です。（またデータベースサーバの接続方法を指定するために、-h、-p、-Uが必要になるかもしれません。）

> **注意**
>
> pgbench -iは4つのテーブルpgbench_accounts、pgbench_branches、pgbench_history、pgbench_tellersを作成します。もしあればこうした名前のテーブルは破壊されます。もし同じ名前のテーブルが存在する場合にはよく注意してください。

デフォルトの「倍数」の1では、テーブルは初期状態で以下の行数を含みます。

```
table                   # of rows
---------------------------------
pgbench_branches        1
pgbench_tellers         10
pgbench_accounts        100000
pgbench_history         0
```

-s（倍数）オプションを使用して行数を増加させることができます（また、ほとんどの目的ではおそらく増加させるべきです）。また、-F（フィルファクタ）オプションをここで使用しても構いません。

一度この必要な設定を行った後、以下のように-iを持たないコマンドでベンチマークを行うことができます。

```
pgbench [ options ] dbname
```

ほとんどすべての場合、有用な試験とするためにいくつかのオプションが必要になります。最重要オプションは-c（クライアント数）、-t（トランザクション数）、-T (制限時間)、-f（独自スクリプトファイルの指定）です。以下の全一覧を参照してください。

オプション

以下では、データベース初期化時に使用されるオプション、ベンチマーク実行時に使用されるオプション、どちらの場合でも使われるオプションの3つに分けて説明します。

初期化用のオプション

pgbenchは以下の初期化用のコマンドライン引数を受け付けます。

-i
--initialize

初期化モードを呼び出すために必要です。

-I init_steps
--init-steps=init_steps

標準の初期化ステップの内、選択したものだけを実行します。init_stepsでは、各ステップ毎に1文字を使って、実行する初期化ステップを指定します。各ステップは指定した順で実行されます。デフォルトはdtgvpです。有効なステップは以下の通りです。

d (Drop)

既存のpgbenchのテーブルを全て削除します。

t (create Tables)

標準のpgbenchシナリオで使われるテーブル、すなわちpgbench_accounts、pgbench_branches、pgbench_historyおよびpgbench_tellersを作成します。

gまたはG (Generate data, クライアント側、またはサーバ側)

データを生成し、既存データを置き換えて、標準の各テーブルに読み込みます。

g(クライアント側データ生成)は、データはpgbenchクライアントで生成されてからサーバに送られます。これはCOPYでクライアント/サーバの帯域を大きく使います。gを使うと、pgbench_accountsテーブルのためにデータを生成する間、100,000行毎にメッセージを1つログ出力するようになります。

G(サーバ側データ生成)では、小さな問い合わせだけがpgbenchクライアントから送られ、データは実際にはサーバで生成されます。こちらは帯域を著しく要求することはありませんが、サーバが、より多くの作業をすることになります。Gを使うと、データを生成する間は進捗メッセージをログ出力しなくなります。

デフォルトの初期化動作は、クライアント側のデータ生成(gと同じ)を使います。

v (Vacuum)

標準の各テーブルに対してVACUUMを実行します。

p (create Primary keys)

標準の各テーブルにプライマリキーインデックスを作成します。

f (create Foreign keys)

標準のテーブル間に外部キー制約を作成します。(このステップはデフォルトでは実行されないことに注意してください)

-F fillfactor
--fillfactor=fillfactor

指定したフィルファクタでpgbench_accounts、pgbench_tellers、pgbench_branchesテーブルを作成します。デフォルトは100です。

-n
--no-vacuum

初期化でバキュームを実行しません。(このオプションは-Iで指定されていたとしても初期化ステップvを抑止します。)

-q
--quiet

ログ処理を、5秒に1つの進行メッセージのみを生成する静寂モードに切り替えます。デフォルトのログ処理では、100,000行毎にメッセージを1つ出力し、(特に優れたハードウェアでは)1秒当たりに多くのメッセージを出力します。

-Iの中でGが指定されていれば、この設定は影響しません。

-s scale_factor
--scale=scale_factor

この倍率で生成される行数を積算します。例えば、-s 100は pgbench_accountsテーブルに10,000,000行を生成することを意味します。デフォルトは1です。この倍率が20000以上になると、アカウント識別子の範囲を保持できる程度に大きくなるように、アカウント識別子を保持するために使用される列（aid列）はより大きな整数（bigint）を使用するように切り替わります。

--foreign-keys

標準テーブル間で外部キー制約を作成します。（このオプションは初期化ステップの並びに、もし無かったならfステップを追加します。）

--index-tablespace=index_tablespace

デフォルトのテーブル空間ではなく、指定したテーブル空間の中にインデックスを作成します。

--partition-method=NAME

NAMEメソッドでパーティション化されたpgbench_accountsテーブルを作成します。期待される値はrangeまたはhashです。このオプションは--partitionsが0でない値に設定されていることを要求します。指定されなければ、デフォルトはrangeです。

--partitions=NUM

アカウントの数に比例したほぼ等しい大きさのNUM個のパーティションにパーティション化されたpgbench_accountsテーブルを作成します。デフォルトは0で、パーティション化しないことを意味します。

--tablespace=tablespace

デフォルトのテーブル空間ではなく、指定したテーブル空間の中にテーブルを作成します。

--unlogged-tables

永続テーブルではなくログを取らないテーブルとしてテーブルを作成します。

ベンチマーク用オプション

pgbenchは以下のベンチマーク用コマンドライン引数を受け付けます。

-b scriptname[@weight]
--builtin=scriptname[@weight]

指定の組み込みスクリプトを実行されたスクリプトのリストに追加します。オプションで@の後に整数のweight（重み）をつけることで、そのスクリプトが選ばれる確率を調整することができます。指定しなかった場合は1に設定されます。利用可能な組み込みのスクリプトは、tpcb-like、simple-update、select-onlyです。組み込みの名前の曖昧な接頭辞も受け付けられます。特別な名前listを使うと、組み込みスクリプトのリストを表示して、即座に終了します。

-c clients
--client=clients

模擬するクライアント数、つまり、同時に実行されるデータベースセッション数です。デフォルトは1です。

-C
--connect

各クライアントセッションが一度だけ接続を確立するのではなく、各トランザクションが新しい接続を確立します。これは接続オーバーヘッドを測定する場合に有用です。

-d
--debug

デバッグ用出力を表示します。

-D varname=value
--define=varname=value

独自スクリプト(後述)で使用される変数を定義します。複数の-Dオプションを使用することができます。

-f filename[@weight]
--file=filename[@weight]

filenameから読み取ったトランザクションスクリプトを実行されたスクリプトのリストに追加します。オプションで@の後に整数のweight(重み)をつけることで、そのテストが選ばれる確率を調整することができます。詳細は後で説明します。

-j threads
--jobs=threads

pgbench内のワーカスレッド数です。複数のスレッドを使用することはマルチCPUマシンで有用になります。クライアントは利用可能なスレッドの間でできる限り均等に分散されます。デフォルトは1です。

-l
--log

各トランザクションに関する情報をログファイルに書き出します。後で詳細を説明します。

-L limit
--latency-limit=limit

limitミリ秒以上続くトランザクションが計数され、別途遅いトランザクションとして報告されます。

制限が使用されると(--rate=...)、limitミリ秒以上遅延がスケジュールされたトランザクションは遅延制限を満たす可能性がないため、サーバに送信されることは決してありません。これらのトランザクションは計数され、別途スキップされたとして報告されます。

-M querymode
--protocol=querymode

サーバへ問い合わせを送信するために使用するプロトコルです。

- simple: 簡易問い合わせプロトコルを使用します。
- extended: 拡張問い合わせプロトコルを使用します。
- prepared: プリペアドステートメントを伴う拡張問い合わせプロトコルを使用します。
preparedモードでは、pgbenchは問い合わせの2回目の繰り返しからは構文解析結果を再利用しますので、pgbenchは他のモードよりも速く動作します。

デフォルトは簡易問い合わせプロトコルです。(詳しい情報は第52章を参照してください)

-n
--no-vacuum

試験を実行する前にバキュームを行いません。pgbench_accounts、pgbench_branches、pgbench_history、pgbench_tellers標準テーブルを含まない独自試験シナリオを実行する場合、このオプションは必要です。

-N
--skip-some-updates

組み込みのsimple-update(単純な更新)のスクリプトを実行します。-b simple-updateの短縮形です。

-P sec
--progress=sec

sec秒毎の進捗レポートを表示します。レポートには起動からの経過時間、前回レポート時からのTPS、前回レポート時からのトランザクションの平均待ち時間、標準偏差を含んでいます。(-R)オプションによる制限下では、待ち時間はトランザクションの実開始時間ではなく、予定開始時間で算出されていますので、平均予定遅延時間が含まれています。

-r
--report-latencies

ベンチマーク完了後の各コマンドにおけるステートメント毎の平均レイテンシ(クライアントから見た実行時間)を報告します。詳しくは下を参照してください。

-R rate
--rate=rate

トランザクションを可能な限り高速(デフォルト)で実行するのではなく、指定された目標レートで実行します。レートは1秒あたりのトランザクション数で与えられます。目標レートが実施可能な最大レートを越えている場合、レート制限は結果に影響を与えません。

レートはトランザクションの開始予定タイムラインがポアソン分布に沿う事を目標としています。期待される開始時刻の予定は、前トランザクションの終了時ではなくクライアントの初期起動時に基づいて動かします。このアプローチはトランザクションがオリジナルの終了予定時刻を過ぎた場合でも、後でまた追い付けることを意味します。

制限がアクティブになると、実行終了時に報告されるトランザクション待ち時間は、予定開始時刻から計算されるので、各トランザクションが前トランザクションの終了を待たねばならなかった時間を含んでいます。この待ち時間はスケジュールラグタイムと呼ばれ、平均と最大値も別々に報告されます。実トランザクション開始時刻についてのトランザクション待ち時間、つまりデータベース内でトランザクション

の実行に要した時間は、報告された待ち時間からスケジュールラグタイムを減算することで算出することができます。

--latency-limitが--rateと一緒に指定された場合、トランザクションは、先行するトランザクションが終了した際にすでに遅延制限を超えていて、非常に遅れてしまうことがあり得ます。そのようなトランザクションはサーバに送信さることなくスキップされ、別途カウントされます。

スケジュールラグタイムの高い値は、システムが選択されたクライアント数とスレッド数で、指定されたレートでトランザクションを処理できなかったことを示しています。トランザクションの平均実行時間が各トランザクション間で予定されていた間隔より長い場合、各逐次トランザクションは更に遅くなり、スケジュールラグタイムはテスト実行がより長く増加し続けます。これが起こる場合、指定トランザクションレートを減らす必要があります。

-s scale_factor
--scale=scale_factor

pgbenchの出力で指定した倍率を報告します。これは組み込みの試験では必要ありません。正確な倍率がpgbench_branchesテーブルの行数を数えることで検出されます。しかし、独自ベンチマーク(-fオプション)のみを試験している場合、このオプションを使用しない限り、倍率は1として報告されます。

-S
--select-only

組み込みのselect-only(SELECTのみ)のスクリプトを実行します。-b select-onlyの短縮形です。

-t transactions
--transactions=transactions

各クライアントが実行するトランザクション数です。デフォルトは10です。

-T seconds
--time=seconds

クライアントあたりのトランザクション数を固定で指定するよりも長くテストを実行したい場合、ここに指定した秒数でテストを実行します。-tと-Tは互いに排他的です。

-v
--vacuum-all

試験前に4つの標準テーブルすべてをバキュームします。-nも-vもなければ、pgbenchはpgbench_tellersとpgbench_branchesテーブルをバキュームし、pgbench_history内のデータをすべて消去します。

--aggregate-interval=seconds

集約間隔の長さ(秒単位)です。これは-lと一緒でのみ使用できます。このオプションを付けると、ログには以下で説明するような指定間隔単位の要約が含まれます。

--log-prefix=prefix

--logにより作成されるログファイルのファイル名の先頭につける文字列を設定します。デフォルトはpgbench_logです。

--progress-timestamp

進捗を表示(-Pオプション)しているとき、実行開始以後の経過秒数の代わりにタイムスタンプ(Unixエポック時刻)を使用します。単位は秒で、ドットの後にミリ秒の精度が付きます。これは様々なツールで生成されたログを比較するのに役立つでしょう。

--random-seed=seed

ランダムジェネレータのシードを設定します。各スレッド毎の初期ジェネレータ状態から一連の値を生成する、システム乱数ジェネレータの種となります。seedの値は以下が可能です。time(デフォルト、現在時刻に基づくシード)、rand(強いランダムソースを使用、使用できなければ失敗します)、あるいは符号無し整数値です。ランダムジェネレータはpgbenchスクリプト(random...関数)から明示的に、あるいは暗黙に(例えばオプション--rateがトランザクションのスケジュールに使用します)、実行されます。明示的に設定した場合、シードに使われる値はターミナルにあらわれます。seedに与えることのできる値は何であれ、環境変数PGBENCH_RANDOM_SEEDを通して付与しても良いです。設定したシードがありうる全ての実行に影響を及ぼすようにするためには、本オプションを最初に置くか、環境変数を使ってください。

明示的にシードを設定することは、乱数に関しては、正確にpgbench実行を再現することを可能にします。ランダム状態はスレッド毎に制御されているので、スレッド毎に一つのクライアントであり、外的な依存やデータ依存が無い場合、同一の起動に対して正確に同じpgbench実行することを意味します。統計的観点からは、性能のばらつきを隠したり、例えば前回実行と同じページにヒットすることで不当に性能改善するので、正確な再現実行は悪い考えです。しかしながら、例えばエラーを起こすトリッキーなケースを再実行するなど、デバッグには大きな助けとなるでしょう。賢く使ってください。

--sampling-rate=rate

データをログに書き出す際に使用される、生成されるログの量を減少するためのサンプリング割合です。このオプションが指定された場合、指定された割合のトランザクションがログに残ります。1.0はすべてのトランザクションが、0.05はトランザクションの5%のみがログに残ることを意味します。

ログファイルを処理する際にはこのサンプリング割合を考慮することを忘れないでください。例えば、TPS値を計算する際には、比例した数を掛け合わせなければなりません(例:サンプリング割合が0.01の場合実際のTPSの1/100を得るだけです。)

--show-script=scriptname

組み込みスクリプトscriptnameの実際のコードを標準エラーに出力し、即座に終了します。

共通オプション

pgbenchは接続パラメータとして以下の共通コマンドライン引数も受け付けます。

-h hostname
--host=hostname

データベースサーバのホスト名

-p port
--port=port

データベースサーバのポート番号

-U login
--username=login

接続ユーザ名

-V
--version

pgbenchのバージョンを表示し、終了します。

-?
--help

pgbenchのコマンドライン引数の説明を表示し、終了します。

終了ステータス

実行に成功すればステータス0で終了します。終了ステータス1は、無効なコマンドラインオプションのような静的な問題を示します。データベースエラーやスクリプトでの問題などの実行中のエラーは終了ステータス2になります。後者の場合、pgbenchは部分的な結果を表示します。

環境

PGHOST
PGPORT
PGUSER

デフォルトの接続パラメータです。

このユーティリティは、他のほとんどのPostgreSQLユーティリティと同様、libpqでサポートされる環境変数を使用します（33.14を参照してください）。

環境変数PG_COLORは診断メッセージで色を使うかどうかを指定します。可能な値はalways、auto、neverです。

注釈

pgbenchで実際に実行される「トランザクション」は何か？

pgbenchは指定したリストからランダムに選択したテストスクリプトを実行します。これには-bの組み込みスクリプトと-fのユーザ定義カスタムスクリプトが含まれます。各スクリプトには@の後に指定される相対的な重みを与えることができ、それが選ばれる確率を変更することができます。デフォルトの重みは1です。重みが0のスクリプトは無視されます。

デフォルトの組み込みトランザクションスクリプト（-b tpcb-likeとすることでも実行されます）は、aid、tid、bid、deltaからランダムに選択され、トランザクション毎に7つのコマンドを発行します。このシナリオはTPC-Bベンチマークに示唆を受けたものですが、実際にはTPC-Bではないので、この名前になっています。

1. BEGIN;

2. UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;

3. SELECT abalance FROM pgbench_accounts WHERE aid = :aid;

4. UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;

5. UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;

6. INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES
   (:tid, :bid, :aid, :delta, CURRENT_TIMESTAMP);

7. END;

simple-updateの組み込みを選択した(あるいは-Nを指定した)場合、第4ステップと第5ステップはトランザクションに含まれません。これにより、これらのテーブルに対する更新の競合を避けられますが、テストケースはさらにTPC-Bらしくなくなります。

select-onlyの組み込みを選択した(あるいは-Sを指定した)場合、SELECTのみが発行されます。

独自スクリプト

pgbenchは、ファイルから読み込んだトランザクションスクリプト(-fオプション)でデフォルトのトランザクションスクリプト(上述)を置き換えて独自のベンチマークシナリオを実行する機能をサポートします。この場合、「トランザクション」はスクリプトファイルの1回の実行として数えられます。

スクリプトファイルにはセミコロンで終了するSQLコマンドが1つ以上含まれます。空行および--から始まる行は無視されます。スクリプトファイルの行には、pgbench自身が解釈する「メタコマンド」(後述)も記述することができます。

> **注記**
>
> PostgreSQLの9.6より前では、スクリプトファイル内のSQLコマンドは改行で終了しており、そのため行をまたがって継続することができませんでした。これからは連続するSQLコマンドを区切るためにセミコロンが必要です(ただし、SQLコマンドの後にメタコマンドが続く場合は、セミコロンは必要ありません)。pgbenchの古いバージョンと新しいバージョンの両方で動作するスクリプトを作る必要があるなら、各SQLコマンドを1行で書き、終わりにセミコロンを付けるようにしてください。

スクリプトファイル向けの簡単な変数置換機能があります。変数名は文字(非ラテン文字を含む)、数字およびアンダースコアで構成されていなければなりません。上で説明したように変数を-Dコマンドラインオプションで設定することができます。また、後で説明するようにメタコマンドで設定することもできます。-Dコマンドラインオプションで設定された変数の他に、表 273に記載されているように、自動的に設定される変数がいくつかあります。-D を使ってこれらの変数に設定された値は、自動設定の値より優先されます。一度設定すると、変数の値は、:variablenameと書かれてSQLコマンドに挿入されます。1つ以上のクライアントセッションが実行される場合、セッション毎に独自の変数群を持ちます。pgbenchは1つの文内で255個までの変数の利用をサポートします。

表273 pgbench Automatic Variables

変数	説明
client_id	クライアントセッションを識別する一意の数値(ゼロから始まる)

変数	説明
default_seed	デフォルトでハッシュ関数で使われるシード
random_seed	ランダムジェネレータのシード(-Dで上書きされていないなら)
scale	現在のスケールファクタ

スクリプトファイルのメタコマンドはバックスラッシュ(\)から始まり、通常は行末まで続きますが、バックスラッシュと改行を書くことで、追加の行に続けることができます。メタコマンドへの引数は空白文字で区切られます。以下のメタコマンドがサポートされています。

\gset [prefix] \aset [prefix]

このコマンドは、終了を意味するセミコロン(;)の置き換えで、SQL問い合わせを終えるために使われます。

\gsetコマンドが使われると、それまでのSQL問い合わせは1行を返すものと期待され、その行の列は列名にちなんだ名前の変数に格納されます。prefixが指定されていれば、変数名の前に付きます。

\asetコマンドが使われると、(\;で分けられた)すべての結合したSQL問い合わせは、その列が列名にちなんだ名前の変数に格納されます。prefixが指定されていれば、変数名の前に付きます。問い合わせが行を返さなければ、割り当ては行なわれませんので、これを検出するために変数の存在をテストできます。問い合わせが2行以上返した場合、最後の値が保持されます。

以下の例は、最初の問い合わせからの最終的な口座残高を変数abalanceに入れ、変数p_twoとp_threeを3番目の問い合わせからの整数で埋めます。2番目の問い合わせの結果は捨てられます。最後の2つの結合した問い合わせの結果は、変数fourとfiveに格納されます。

```
UPDATE pgbench_accounts
  SET abalance = abalance + :delta
  WHERE aid = :aid
  RETURNING abalance \gset

-- 2つの問い合わせの組み合わせ
SELECT 1 \;
SELECT 2 AS two, 3 AS three \gset p_
SELECT 4 AS four \; SELECT 5 AS five \aset
```

\if expression
\elif expression
\else
\endif

このコマンド群はpsqlの\if expressionと似た、入れ子にできる条件ブロックを実現します。条件式は\setと同じで、非ゼロ値は真と解釈されます。

\set varname expression

varname変数をexpressionから計算された値に設定します。式(expression)には、NULL定数、真理値定数のTRUEとFALSE、5432のような整数の定数、3.14159のような倍精度実数の定数、変数を参照する

:variablename、通常のSQLの優先度と結合規則での演算子、関数呼び出し、SQLのCASE一般条件式および括弧を含むことができます。

関数と大部分の演算子はNULL入力にNULLを返します。

条件の用途では非ゼロの数値はTRUE、ゼロ数値とNULLはFALSEです。

大きすぎるもしくは小さすぎる整数や倍精度実数の定数は、整数算術演算子(+、-、*、/)と同様にオーバーフローエラーになります。

CASEに最後のELSE句が与えられないとき、デフォルト値はNULLです。

例

```
\set ntellers 10 * :scale
\set aid (1021 * random(1, 100000 * :scale)) % \
           (100000 * :scale) + 1
\set divx CASE WHEN :x <> 0 THEN :y/:x ELSE NULL END
```

\sleep number [us | ms | s]

スクリプトの実行をマイクロ秒(us)、ミリ秒(ms)、秒(s)単位で指定した間待機させます。単位を省略した場合、デフォルトは秒です。numberは整数定数か整数値を持つ変数への:variablename参照のいずれかです。

例

```
\sleep 10 ms
```

\setshell varname command [argument ...]

commandシェルコマンドを指定のargumentで実行した結果をvarname変数に設定します。このコマンドは標準出力を通して整数値を返さなければなりません。

commandおよび各argumentは、テキスト定数または変数を参照する:variablenameとすることができます。コロンから始まるargumentを使用したい場合、argumentの先頭にさらにコロンを付けなければなりません。

例:

```
\setshell variable_to_be_assigned command
 literal_argument :variable ::literal_starting_with_colon
```

\shell command [argument ...]

\setshellと同じですが、コマンドの結果は廃棄されます。

例:

```
\shell command literal_argument :variable ::literal_starting_with_colon
```

組み込み演算子

表 274に載っている算術、ビットごと、比較、論理の演算子はpgbenchに組み込まれていて、\setの式で使用できます。演算子は優先度の低い順に載っています。注意書きがある場合を除いて、2つの数値を取る演算子は、入力の片方が倍精度実数であれば倍精度実数の値を結果とし、そうでなければ整数の結果になります。

表274 pgbenchの演算子

演算子 説明 例
boolean OR boolean → boolean 論理OR 5 or 0 → TRUE
boolean AND boolean → boolean 論理AND 3 and 0 → FALSE
NOT boolean → boolean 論理NOT not false → TRUE
boolean IS [NOT] (NULL|TRUE|FALSE) → boolean ブール値のテスト 1 is null → FALSE
value ISNULL|NOTNULL → boolean NULLであるかのテスト 1 notnull → TRUE
number = number → boolean 等価 5 = 4 → FALSE
number <> number → boolean 不等 5 <> 4 → TRUE
number != number → boolean 不等 5 != 5 → FALSE
number < number → boolean より小さい 5 < 4 → FALSE
number <= number → boolean 以下 5 <= 4 → FALSE
number > number → boolean

演算子 説明 例
より大きい 5 > 4 → TRUE
number >= number → boolean 以上 5 >= 4 → TRUE
integer | integer → integer ビット毎のOR 1 | 2 → 3
integer # integer → integer ビット毎のXOR 1 # 3 → 2
integer & integer → integer ビット毎のAND 1 & 3 → 1
~ integer → integer ビット毎のNOT ~ 1 → -2
integer << integer → integer ビット毎の左シフト 1 << 2 → 4
integer >> integer → integer ビット毎の右シフト 8 >> 2 → 2
number + number → number 加算 5 + 4 → 9
number - number → number 減算 3 - 2.0 → 1.0
number * number → number 乗算 5 * 4 → 20
number / number → number 除算(入力が両方とも整数であれば、結果は0に向けて丸められる) 5 / 3 → 1
integer % integer → integer 剰余(余り) 3 % 2 → 1
- number → number

演算子 説明 例
符号反転 - 2.0 → -2.0

組み込み関数

表 275に示す関数はpgbenchに組み込まれており、\setに現れる式の中で使うことができます。

表275 pgbenchの関数

関数 説明 例
abs (number) → 入力と同じ型 絶対値 abs(-17) → 17
debug (number) → 入力と同じ型 引数をstderrに出力し、引数を返す。 debug(5432.1) → 5432.1
double (number) → double 倍精度実数にキャストする。 double(5432) → 5432.0
exp (number) → double 指数(eの指定した冪) exp(1.0) → 2.718281828459045
greatest (number [, ...]) → double if any argument is double, else integer 引数の中で最大の値を選択する。 greatest(5, 4, 3, 2) → 5
hash (value [, seed]) → integer これはhash_murmur2の別名です。 hash(10, 5432) → -5817877081768721676
hash_fnv1a (value [, seed]) → integer FNV-1aハッシュ[1]を計算する。 hash_fnv1a(10, 5432) → -7793829335365542153
hash_murmur2 (value [, seed]) → integer MurmurHash2ハッシュ[2]を計算する。 hash_murmur2(10, 5432) → -5817877081768721676
int (number) → integer

[1] https://en.wikipedia.org/wiki/Fowler%E2%80%93Noll%E2%80%93Vo_hash_function

[2] https://en.wikipedia.org/wiki/MurmurHash

関数 説明 例
整数にキャストする。 int(5.4 + 3.8) → 9
least (number [, ...]) → double if any argument is double, else integer 引数の中で最小の値を選択する。 least(5, 4, 3, 2.1) → 2.1
ln (number) → double 自然対数 ln(2.718281828459045) → 1.0
mod (integer, integer) → integer 剰余(余り) mod(54, 32) → 22
pi () → double πの近似値 pi() → 3.14159265358979323846
pow (x, y) → double power (x, y) → double xのy乗 pow(2.0, 10) → 1024.0
random (lb, ub) → integer [lb, ub]内の一様分布の整数の乱数を計算する。 random(1, 10) → 1と10の間の整数
random_exponential (lb, ub, parameter) → integer [lb, ub]内の指数分布の整数の乱数を計算する、後述。 random_exponential(1, 10, 3.0) → 1と10の間の整数
random_gaussian (lb, ub, parameter) → integer [lb, ub]内のガウス分布の整数の乱数を計算する、後述。 random_gaussian(1, 10, 2.5) → 1と10の間の整数
random_zipfian (lb, ub, parameter) → integer [lb, ub]内のジップ分布の整数の乱数を計算する、後述。 random_zipfian(1, 10, 1.5) → 1と10の間の整数
sqrt (number) → double 平方根 sqrt(2.0) → 1.414213562

random関数は一様分布を使って値を生成します。つまり、すべての値は指定された範囲内で同じ確率で発生します。random_exponential、random_gaussian、および、random_zipfian関数は追加の倍精度実数のパラメータを必要とし、それによって分布の正確な形が決まります。

- 指数分布では、parameterが分布を制御します。急速に減少する指数分布をparameterで切り捨て、境界範囲内の整数に射影します。正確には、以下の式に従います。

```
f(x) = exp(-parameter * (x - min) / (max - min + 1)) / (1 - exp(-parameter))
```

これにより、minとmaxの間（両端を含む）の間の値iがf(i) - f(i + 1)の確率で生成されます。

直感的には、parameterが大きければ、minに近い値が発生する確率が高くなり、maxに近い値が発生する確率が低くなります。parameterが0に近ければ、発生の分布はより平ら（より一様）になります。大雑把に分布を近似すると、minに近い最頻の1%の範囲の値は、parameter%の割合で発生します。parameterの値は厳密に正でなければなりません。

- ガウス分布では、標準的な正規分布（古典的なベルの形をしたガウス曲線）で、左に-parameter、右に+parameterのところで切り捨てられたものに間隔が射影されます。間隔の中間の値が発生する確率が最も高くなります。正確に言うと、PHI(x)は標準正規分布の累積分布関数、平均値muを(max + min) / 2.0と定義し、さらに

```
f(x) = PHI(2.0 * parameter * (x - mu) / (max - min + 1)) /
    (2.0 * PHI(parameter) - 1)
```

とすると、minとmaxの間（両端を含む）の値iが発生する確率はf(i + 0.5) - f(i - 0.5)になります。直感的には、parameterが大きくなれば、間隔の中間に近い値になる確率が高く、また、minとmaxの境界に近い値になる確率は低くなります。約67%の値は、中間の1.0 / parameterの範囲、つまり平均値から0.5 / parameterの範囲から、また95%は中間の2.0 / parameterの範囲、つまり平均値から1.0 / parameterの範囲に発生します。例えばparameterが4.0なら、67%の値は間隔の中間の4分の1(1.0/4.0)から（つまり3.0 / 8.0から5.0 / 8.0まで）、95%は間隔の中間の半分（2.0 / 4.0）から（2番目と3番目の四分位）から発生します。許される最小のparameter値は2.0です。

- random_zipfianは制限付きのジップ分布を生成します。parameterはどれほど歪んだ分布かを定義します。より大きいparameterほど、より高頻度に区間の始点に近い値が描かれます。範囲が1から始まるとして、kを描く確率とk+1を描く確率の比が((k+1)/k)**parameterという分布になります。例えば、random_zipfian(1, ..., 2.5)は、値1を2の約(2/1)**2.5 = 5.66倍高い頻度で生成し、値2を3の約(3/2)**2.5 = 2.76倍高い頻度で生成し、以下同様に続きます。

  pgbenchの実装は「Non-Uniform Random Variate Generation」Luc Devroye(Springer 1986, p. 550-551)に基づいており、parameter値は[1.001, 1000]の範囲に限定されています。

ハッシュ関数hash、hash_murmur2およびhash_fnv1aは入力値とオプションシードパラメータを受け付けます。シードが与えられなかった場合、:default_seedの値が使われます。これは、コマンドライン-Dオプションで設定されない限りランダムに初期化されたものです。ハッシュ関数はrandom_zipfianやrandom_exponentialなどのランダム関数の分布を散らすのに使用できます。例えば、以下のpgbenchスクリプトは起こりえる現実世界のソーシャルメディアとブログプラットフォームに対する僅かなアカウントしか過大な負荷をかけないという典型的な負荷をシミュレートします。

```
\set r random_zipfian(0, 100000000, 1.07)
\set k abs(hash(:r)) % 1000000
```

一部のケースでは、互いに無関係ないくつかの異なる分布が必要で、これは暗黙のシードパラメータが役立ちます。

```
\set k1 abs(hash(:r, :default_seed + 123)) % 1000000
\set k2 abs(hash(:r, :default_seed + 321)) % 1000000
```

例えば、組み込みのTPC-Bのようなトランザクションの完全な定義を示します。

```
\set aid random(1, 100000 * :scale)
\set bid random(1, 1 * :scale)
\set tid random(1, 10 * :scale)
\set delta random(-5000, 5000)
BEGIN;
UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;
SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;
UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;
INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (:tid, :bid, :aid, :delta,
 CURRENT_TIMESTAMP);
END;
```

このスクリプトにより、トランザクションを繰り返す度に異なる、ランダムに選ばれた行を参照することができます。(この例はまた、各クライアントセッションがなぜ独自の変数を持つことが重要なのかも表しています。これがないと、異なる行を独立して参照することができないのです。)

トランザクション毎のログ処理

(--aggregate-intervalオプションなしで)-lオプションを使用すると、pgbenchは各トランザクションに関する情報をログファイルに書き出します。ログファイルの名前はprefix.nnnで、prefixのデフォルトはpgbench_log、nnnはpgbenchプロセスのPIDです。ファイル名の先頭の文字列は--log-prefixオプションを使って変更することができます。-jオプションが2以上で複数のワーカスレッドがある場合、それぞれが独自のログファイルを持つことになります。最初のワーカは標準的な単一ワーカの場合と同じ名前を持つログファイルを使用します。他のワーカ用の追加のログファイルはprefix.nnn.mmmと命名され、ここでmmmは1から始まる各ワーカの連番です。

ログの書式は以下の通りです。

```
client_id transaction_no time script_no time_epoch time_us [ schedule_lag ]
```

client_idはどのクライアントセッションがそのトランザクションを実行したかを示します。transaction_noはそのセッションで何個のトランザクションが実行されたかを示します。timeはトランザクションの合計消費時間をマイクロ秒単位で示します。script_noはどのスクリプトファイルが使用されたかを識別するものです(-fまたは-bで複数のスクリプトが指定された場合に有用です)。time_epoch/time_usはトランザクション完了時のUnixエポック時間とマイクロ秒単位のオフセットです(小数秒付きのISO 8601タイムスタンプの作成に適します)。schedule_lagフィールドは、マイクロ秒単位のトランザクションの予定開始時刻と実開始時刻の差です。これは--rateオプションを使用した時だけ表示されます。--rateと--latency-limitの両方のオプションを使用した時は、スキップされたトランザクションのtimeがskippedとして表示されます。

単一クライアントでの実行で生成されたログファイルの一部を示します。

```
0 199 2241 0 1175850568 995598
0 200 2465 0 1175850568 998079
0 201 2513 0 1175850569 608
0 202 2038 0 1175850569 2663
```

--rate=100と--latency-limit=5を指定した例を示します。(schedule_lag列が追加されていることに注意)

```
0 81 4621 0 1412881037 912698 3005
0 82 6173 0 1412881037 914578 4304
0 83 skipped 0 1412881037 914578 5217
0 83 skipped 0 1412881037 914578 5099
0 83 4722 0 1412881037 916203 3108
0 84 4142 0 1412881037 918023 2333
0 85 2465 0 1412881037 919759 740
```

この例では、トランザクション82は遅延(6.173ミリ秒)が5ミリ秒を越えており、遅れています。次の2つのトランザクションは、開始する前にすでに遅れてしまっているため、スキップされています。

大量のトランザクションを処理することができるハードウェアで長時間試験を実行する場合、ログファイルは非常に大きくなる可能性があります。--sampling-rateオプションを使用して、トランザクションのランダムなサンプルだけをログに記録することができます。

ログ処理の集約

--aggregate-intervalオプションを付ける場合、以下のようにログの書式が異なります。

```
interval_start num_transactions sum_latency sum_latency_2 min_latency max_latency
 [ sum_lag sum_lag_2 min_lag max_lag [ skipped ] ]
```

interval_startはインターバルの開始時刻(Unixエポック時間)です。num_of_transactionsはインターバル内のトランザクション数です。latency_sumはインターバル内のトランザクションレイテンシの総和です。sum_latency_2はインターバル内のトランザクションレイテンシの2乗の総和です。min_latencyはインターバル内の最小レイテンシです。max_latencyはインターバル内の最大レイテンシです。これに続くフィールドsum_lag、sum_lag_2、min_lag、max_lagは--rateオプションが指定された場合にのみ表示されます。これらは、各トランザクションが直前のトランザクションの終了を待機しなければならなかった時間、つまりトランザクションの予定開始時刻と実際の開始時刻の差に関する統計を提供します。一番最後のフィールドskippedは--latency-limitオプションも使用されたときにのみ表示されます。これは開始時刻が遅くなったためにスキップされたトランザクションの数を数えます。各トランザクションはインターバル内でコミットされた時に数えられます。

いくつか出力例を示します。

```
1345828501 5601 1542744 483552416 61 2573
1345828503 7884 1979812 565806736 60 1479
1345828505 7208 1979422 567277552 59 1391
```

```
1345828507 7685 1980268 569784714 60 1398
1345828509 7073 1979779 573489941 236 1411
```

通常の(集約されていない)ログファイルは、各トランザクションについてどのスクリプトファイルが使用されたかを示しますが、集約されたログにはそれがないことに注意してください。このためスクリプト単位のデータが必要な場合は、自身でデータを集約する必要があります。

ステートメント毎のレイテンシ

-rオプションを付けると、pgbenchは各クライアントにより実行されたトランザクションのステートメント毎の経過時間を収集します。ベンチマークが終了した後、各値の平均値(各ステートメントのレイテンシと呼びます)が報告されます。

標準スクリプトでは、次のような出力になります。

```
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 10
number of threads: 1
number of transactions per client: 1000
number of transactions actually processed: 10000/10000
latency average = 15.844 ms
latency stddev = 2.715 ms
tps = 618.764555 (including connections establishing)
tps = 622.977698 (excluding connections establishing)
statement latencies in milliseconds:
        0.002  \set aid random(1, 100000 * :scale)
        0.005  \set bid random(1, 1 * :scale)
        0.002  \set tid random(1, 10 * :scale)
        0.001  \set delta random(-5000, 5000)
        0.326  BEGIN;
        0.603  UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;
        0.454  SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
        5.528  UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;
        7.335  UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;
        0.371  INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES
 (:tid, :bid, :aid, :delta, CURRENT_TIMESTAMP);
        1.212  END;
```

複数のスクリプトファイルが定義された場合、平均値はそれぞれのスクリプトファイル毎に分けて報告されます。

ステートメント毎のレイテンシを計算するために必要となる、追加のタイミング情報を収集することは、オーバーヘッドが加わることに注意してください。これは平均実行速度を遅くし、計測TPSを小さくするでしょう。

低下量はプラットフォームとハードウェアに依存して著しく変わります。レイテンシの報告を有効にする、有効にしないで平均TPS値を比較することは、タイミング・オーバーヘッドが顕著かどうかを測定するには良い方法です。

優れた実践

まったく無意味な数値を生み出すようにpgbenchを使用することは非常に簡単です。以下に有意な結果を生み出す手助けとなるガイドラインをいくつか示します。

まず第一に、数秒で終わる試験を決して信用しないでください。-tまたは-Tオプションを使って、雑音を取り除くために、少なくとも数分試験にかかるようにしてください。再現可能な数値を得るために数時間必要になる場合もあります。数回試験を繰り返し、数値が再現できるかどうか確認することを勧めます。

デフォルトのTPC-Bのような試験シナリオでは、初期倍率(-s)を試験予定のクライアント数(-c)の最大値と同程度にしなければなりません。pgbench_branchesテーブルには-s行しかありません。また、全トランザクションはその内の1つを更新しようとします。ですので、-c値を-sより大きくすると、他のトランザクションを待機するためにブロックされるトランザクションが多くなることは間違いありません。

デフォルトの試験シナリオはまた、テーブルを初期化してからの経過時間に非常に敏感です。テーブル内の不要行や不要空間の累積により結果が変わります。結果を理解するためには、更新された行数とバキューム時期を把握する必要があります。自動バキュームが有効な場合、性能を測定する上で結果は予測できないほど変わる可能性があります。

pgbenchの制限は、多くのクライアントセッションを試験しようとする際にpgbench自身がボトルネックになる可能性があることです。これは、データベースサーバとは別のマシンでpgbenchを実行することで緩和させることが可能です。しかし、多少のネットワーク遅延が重要です。同一データベースサーバに対し複数のクライアントマシンから複数のpgbenchインスタンスを同時に実行することが有用かもしれません。

セキュリティ

安全なスキーマの利用パターンを適用していないデータベースに信頼できないユーザがアクセス可能な場合、そのデータベースでpgbenchを実行しないでください。pgbenchは修飾していない名前を使っており、またサーチパスを操作していません。

pg_config

pg_config — インストールしたバージョンのPostgreSQLに関する情報を提供する

概要

`pg_config` [option...]

説明

pg_configユーティリティは、現在インストールしているバージョンのPostgreSQLの設定パラメータを表示します。これは、例えばPostgreSQLとのインタフェースを持つソフトウェアパッケージが必要なヘッダファイルやライブラリを容易に検出できるように用意されたものです。

オプション

pg_configを使用するためには、以下のオプションを1つ以上指定します。

`--bindir`

ユーザコマンドの場所を表示します。例えば、psqlプログラムを検索するために使用します。これは通常、pg_configプログラムが存在するディレクトリでもあります。

`--docdir`

文書ファイルの場所を表示します

`--htmldir`

HTML文書ファイルの場所を表示します。

`--includedir`

クライアントインタフェースのCヘッダファイルの場所を表示します。

`--pkgincludedir`

その他のCヘッダファイルの場所を表示します。

`--includedir-server`

サーバプログラム作成用のCヘッダファイルの場所を表示します。

`--libdir`

オブジェクトコードライブラリのディレクトリを表示します。

`--pkglibdir`

動的ローディング可能なモジュールの場所、またはそれをサーバが検索する場所を表示します。(このディレクトリには、アーキテクチャに依存する他のデータファイルも存在する可能性があります。)

`--localedir`

ロケールサポートファイルの場所を表示します（PostgreSQLをロケールサポートなしで構築した場合は空文字列となります）。

`--mandir`

マニュアルページの場所を表示します。

`--sharedir`

アーキテクチャ非依存のサポートファイルの場所を表示します。

`--sysconfdir`

システム全体の設定ファイルの場所を表示します。

`--pgxs`

拡張用Makefileの場所を表示します。

`--configure`

PostgreSQLを構築する時にconfigureスクリプトに与えたオプションを表示します。まったく同じ構築条件でPostgreSQLを再作成する時、あるいは、バイナリパッケージの構築時のオプションを知りたい時に有益です（バイナリパッケージには、ベンダ固有のカスタムパッチが含まれていることが多いので注意してください）。後述の例も参照してください。

`--cc`

PostgreSQLの構築時に使用されたCC変数の値を表示します。使用したCコンパイラが表示されます。

`--cppflags`

PostgreSQLの構築時に使用されたCPPFLAGS変数の値を表示します。事前処理時に必要としたCコンパイラのスイッチが表示されます。（通常は-Iスイッチです。）

`--cflags`

PostgreSQLの構築時に使用されたCFLAGS変数の値を表示します。Cコンパイラスイッチが表示されます。

`--cflags_sl`

PostgreSQLの構築時に使用されたCFLAGS_SL変数の値を表示します。共有ライブラリの構築に使用された追加のCコンパイラスイッチが表示されます。

`--ldflags`

PostgreSQLの構築時に使用されたLDFLAGS変数の値を表示します。リンカスイッチが表示されます。

--ldflags_ex

PostgreSQLの構築時に使用されたLDFLAGS_EX変数の値を表示します。実行ファイルの構築のみに使用されたリンカスイッチが表示されます。

--ldflags_sl

PostgreSQLの構築時に使用されたLDFLAGS_SL変数の値を表示します。共有ライブラリの構築のみに使用されたリンカスイッチが表示されます。

--libs

PostgreSQLの構築時に使用されたLIBS変数の値を表示します。これには通常、PostgreSQLにリンクする外部ライブラリ用の-lスイッチが含まれます。

--version

PostgreSQLのバージョンを表示します。

-?
--help

pg_configコマンドライン引数に関する説明を表示し、終了します。

1つ以上のオプションが与えられた場合、指定したオプションの順番に従って1行に1つずつ情報を表示します。オプションがない場合、すべての利用可能な情報をラベル付きで表示します。

注釈

オプション--docdir、--pkgincludedir、--localedir、--mandir、--sharedir、--sysconfdir、--cc、--cppflags、--cflags、--cflags_sl、--ldflags、--ldflags_sl、--libs はPostgreSQL 8.1から追加されました。オプション--htmldirはPostgreSQL 8.4で追加されました。オプション--ldflags_exはPostgreSQL 9.0で追加されました。

例

使用中のPostgreSQLインストレーションの構築時の設定を再生成するには、以下のコマンドを実行します。

```
eval ./configure `pg_config --configure`
```

pg_config --configureの出力にはシェルの引用符が含まれますので、空白を含む引数も正しく表現することができます。したがって、正しく動作させるためにはevalが必要です。

pg_dump

pg_dump — PostgreSQLデータベースをスクリプトファイルまたは他のアーカイブファイルへ抽出する

概要

pg_dump [connection-option...] [option...] [dbname]

説明

pg_dumpはPostgreSQLデータベースをバックアップするユーティリティです。データベースを使用中であっても一貫性のあるバックアップを作成することができます。pg_dumpは他のユーザによるデータベースへのアクセス（読み書き）をブロックしません。

pg_dumpは単一のデータベースしかダンプしません。クラスタ全体、あるいは、クラスタ内の全データベースに共通のグローバルオブジェクト（ロールやテーブル空間など）をダンプするにはpg_dumpallを使用してください。

ダンプはスクリプト形式、または、アーカイブファイル形式で出力することができます。スクリプトダンプは、保存した時点の状態のデータベースを再構成するために必要なSQLコマンドが書き込まれた平文ファイルです。このスクリプトを使ってリストアを行うには、それをpsqlに読み込ませます。スクリプトファイルを使えば、ダンプを行ったのとは別のマシンや別のアーキテクチャ上でも、データベースを再構築することができます。また、多少編集すれば他のSQLデータベース製品上でもデータベースの再構築が可能です。

もう1つの形式であるアーカイブファイル形式を使ってデータベースを再構築するには、pg_restoreを使用しなければなりません。このファイルを使用すると、pg_restoreがリストア対象を選択したり、リストアするアイテムを並び替えたりできます。アーカイブファイルもまた、アーキテクチャを越えて移植できるように設計されています。

いずれかのアーカイブファイル形式をpg_restoreと組み合わせて使用する場合は、pg_dumpの柔軟なアーカイブ/転送機構が利用できます。具体的には、pg_dumpを使用してデータベース全体をバックアップし、pg_restoreを使用して、アーカイブの内容を検査したり、データベースの一部を選択してリストアしたりすることができます。最も柔軟な出力ファイル形式は「カスタム」形式（-Fc）と「ディレクトリ」形式（-Fd）です。これらはすべての保存された項目の選択や並び換えを行うことができ、並行リストアをサポートし、デフォルトで圧縮されます。「ディレクトリ」形式は、並行ダンプをサポートする唯一の形式です。

pg_dumpの実行中は、標準エラーに出力される警告（特に後述の制限に関する警告）が出力されていないか確認してください。

オプション

以下のコマンドラインオプションは出力内容とその形式を制御します。

dbname

ダンプするデータベースの名前を指定します。指定されていない場合は環境変数PGDATABASEが使われます。この変数も設定されていない場合は、接続のために指定されたユーザ名が使用されます。

-a
--data-only

データのみをダンプし、スキーマ(データ定義)はダンプしません。テーブルデータ、ラージオブジェクト、シーケンス値がダンプされます。

このオプションは--section=dataを指定することと似ていますが、歴史的な理由で同一ではありません。

-b
--blobs

ラージオブジェクトをダンプに含めます。--schema、--table、--schema-onlyが指定された場合を除き、これがデフォルトの動作です。したがって、-bオプションは特定のスキーマやテーブルのダンプにラージオブジェクトを追加する場合にのみ有用です。blobはデータとみなされるため、--data-onlyが使われたときは含まれますが、--schema-onlyが使われたときには含まれないことに注意してください。

-B
--no-blobs

ラージオブジェクトをダンプに含めません。

-bと-Bの両方が指定された場合は、データのダンプ時にラージオブジェクトを出力します。-bの説明を参照してください。

-c
--clean

データベースオブジェクトを作成するコマンドの前に、データベースオブジェクトを整理(削除)するコマンドを書き出します。(--if-existsも指定されなければ、リストア先のデータベースの中に存在しないオブジェクトがある場合に、害がないエラーがいくつか発生するかもしれません。)

このオプションは平文形式の場合にのみ有効です。アーカイブ形式では、pg_restoreを呼び出す時にこのオプションを指定することができます。

-C
--create

初めにデータベース自体を作成するコマンドを出力し、その後、作成したデータベースに接続するコマンドを出力します (このようなスクリプトを使用すると、スクリプトを実行する前に対象のインストレーションの中のどのデータベースに接続すればよいかという問題を考える必要がなくなります)。--cleanも同時に指定されている場合、このスクリプトは接続する前に対象データベースを削除し再作成します。

--createでは出力に、もしあるならデータベースのコメントも含まれます。また、あらゆる設定変数の設定、すなわち、このデーベースを対象としているALTER DATABASE ... SET ...とALTER ROLE ... IN DATABASE ... SET ...コマンドも含まれます。--no-aclが指定されていない限り、データベースに対するアクセス権限自体もダンプされます。

このオプションは平文形式の場合にのみ有効です。アーカイブ形式では、pg_restoreを呼び出す時にこのオプションを指定することができます。

-E encoding
--encoding=encoding

指定した文字セット符号化方式でダンプを作成します。デフォルトではダンプはデータベースの符号化方式で作成されます。(環境変数PGCLIENTENCODINGを好みのダンプ時の符号化方式に設定することで、同じ結果を得ることができます。)

-f file
--file=file

出力を指定のファイルに送ります。ファイルを基にする出力形式ではこのパラメータは省くことができます。省略時は標準出力が使用されます。しかしディレクトリ出力形式の場合は、省略することはできず、ファイルではなく対象ディレクトリを指定します。この場合、ディレクトリはpg_dumpが生成しますので、事前に存在してはなりません。

-F format
--format=format

出力形式を選択します。formatには以下のいずれかを取ることができます。

p
plain

平文のSQLスクリプトファイルを出力します(デフォルト)。

c
custom

pg_restoreへの入力に適したカスタム形式アーカイブを出力します。ディレクトリ出力形式と一緒に使用する場合、リストア時に手作業で保管された項目の選択、再順序付けできますので、これはもっとも柔軟な出力形式です。また、この形式はデフォルトで圧縮されます。

d
directory

pg_restoreへの入力に適したディレクトリ形式のアーカイブを出力します。これは、ダンプされる各テーブルおよびblobごとに1つのファイル、さらに、pg_restoreから読み取り可能な、機械的に読み取り易い書式でダンプしたオブジェクトを記述する目次ファイルと呼ばれるファイルを持つディレクトリを作成します。ディレクトリ形式アーカイブは標準Unixツールで操作することができます。例えば、未圧縮アーカイブ内のファイルをgzipツールを使用して圧縮することができます。この形式はデフォルトで圧縮されます。また並行ダンプをサポートします。

t
tar

pg_restoreへの入力に適したtar形式のアーカイブを出力します。このtar形式はディレクトリ形式と互換性があります。tar形式アーカイブを展開すると、有効なディレクトリ形式のアーカイブを生成し

ます。しかしtar形式は圧縮をサポートしません。またtar形式を使う場合、リストア時にテーブルデータ項目の相対的な順序を変更することはできません。

-j njobs
--jobs=njobs

njobs個のテーブルを同時にダンプすることによって、並行してダンプを実行します。このオプションはダンプを実行するのに必要な時間を減らすかもしれませんが、データベースサーバの負荷を増やします。このオプションはディレクトリ出力形式でのみ使用することができます。複数のプロセスが同時にそのデータを書き出すことができるのはディレクトリ形式だけだからです。

pg_dumpはnjobs+1個のデータベース接続を開きます。このため、すべての接続を収容できる程度にmax_connectionsが高いことを確認してください。

並行ダンプを実行している時にデータベースオブジェクトに対して排他ロックを要求すると、ダンプが失敗する可能性があります。ダンプ実行中に誰かがダンプ予定のオブジェクトを削除してそれがなくなってしまうことがないように、ワーカプロセスがダンプする予定のオブジェクトに対してpg_dumpのマスタプロセスが共有ロックを要求するのがその理由です。その後他のクライアントがテーブルに対する排他ロックを要求すると、そのロックは許可されませんが、マスタプロセスが共有ロックを解放することを待機するキューに保管されます。その結果、テーブルへのその他のアクセスは許可されず、排他ロック要求の後のキューに保管されます。これには、そのテーブルをダンプしようとする作業用プロセスも含まれます。何らかの注意をしないと、古典的なデッドロック状態になります。この競合を検知するためにpg_dumpの作業用プロセスはNOWAITオプションを使用する別の共有ロックを要求します。作業用プロセスによる共有ロックが許可されない場合、だれかがその時に排他ロックを要求していることになり、ダンプを継続することができません。pg_dumpには、ダンプを中断するしか選択肢がありません。

一貫したバックアップのためには、データベースサーバは、同期スナップショットをサポートする必要があります。これはプライマリサーバについてはPostgreSQL 9.2で、スタンバイについてはバージョン10で導入された機能です。この機能を用いれば、データベースクライアントは異なる接続を使用していたとしても、確実に同じデータセットを参照することができます。pg_dump -jは複数のデータベース接続を使用します。マスタプロセスで一度、また、作業用ジョブそれぞれでも一度、データベースと接続します。同期スナップショット機能がないと、異なる作業用ジョブがそれぞれの接続で同じデータを参照していることが保証されず、一貫性がないバックアップになってしまいます。

9.2より前のサーバで並行ダンプを実行させたいのであれば、データベースの内容が、マスタがデータベースに接続してから最後の作業用ジョブがデータベースに接続するまでの間に変更されないことを確実にしなければなりません。このためのもっとも簡単な方法は、バックアップを始める前にデータを変更するプロセス（DDLおよびDML）がデータベースにアクセスすることを止めさせることです。また、9.2より前のPostgreSQLに対してpg_dump -jを実行する場合は--no-synchronized-snapshotsパラメータを指定する必要もあります。

-n pattern
--schema=pattern

patternにマッチするスキーマのみをダンプします。これはスキーマ自体とそこに含まれるオブジェクトすべてを選択します。このオプションが指定されなければ、対象データベース内にあるシステム以外のスキーマ全てがダンプされます。複数の-nオプションを記述することで、複数のスキーマを選択することができます。patternパラメータはpsqlの\dコマンドと同じ規則に従うパターン（下記のPatterns参照）として解釈されます。ですので、ワイルドカード文字をパターン内に記述することで、複数のスキーマを選

択することもできます。ワイルドカードを使用する時は、シェルがそのワイルドカードを展開しないように、必要であればパターンを引用符で括ってください。Examplesを参照してください。

> **注記**
>
> -nが指定されると、pg_dumpは選択したスキーマ内のオブジェクトが依存する可能性がある、その他のデータベースオブジェクトのダンプを行いません。したがって、スキーマ指定のダンプ結果を初期状態のデータベースに正常にリストアできるという保証はありません。

> **注記**
>
> -nが指定されると、blobなどの非スキーマオブジェクトはダンプされません。--blobsオプションをつけてダンプを行うことでblobも追加されます。

-N pattern
--exclude-schema=pattern

patternにマッチするスキーマをダンプしません。このパターンは-nと同様の規則に従って解釈されます。-Nを複数指定して、複数のパターンのいずれかにマッチするスキーマを除外することができます。

-nと-Nの両方が指定された場合、少なくとも1つの-nにマッチし-Nオプションにマッチしないスキーマだけがダンプされます。-nなしで-Nが指定された場合、-Nにマッチするスキーマが通常のダンプから除外されます。

-O
--no-owner

オブジェクトの所有権を元のデータベースにマッチさせるためのコマンドを出力しません。デフォルトでは、pg_dumpは、ALTER OWNER文またはSET SESSION AUTHORIZATION文を発行して、作成したデータベースオブジェクトの所有権を設定します。スーパーユーザ（もしくは、そのスクリプト内の全てのオブジェクトを所有するユーザ）以外のユーザがスクリプトを実行した場合、これらの文は失敗します。任意のユーザがリストアできるスクリプトを作成するには、-Oを指定してください。ただし、この場合は、全てのオブジェクトの所有者がリストアしたユーザとなってしまいます。

このオプションは平文形式の場合にのみ有効です。アーカイブ形式では、pg_restoreを呼び出す時にこのオプションを指定することができます。

-R
--no-reconnect

このオプションは廃止されましたが、後方互換性を保持するため受け入れられます。

-s
--schema-only

データ定義（スキーマ）のみをダンプし、データはダンプしません。

このオプションは--data-onlyの逆です。これは--section=pre-data --section=post-dataを指定することと似ていますが、歴史的な理由のため同一ではありません。

(これと--schemaオプションと混乱しないでください。「schema」という単語を異なる意味で使用しています。)

データベース内の一部のみのテーブルのテーブルデータを除外するためには--exclude-table-dataを参照してください。

-S username
--superuser=username

トリガを無効にする場合に使用する、スーパーユーザのユーザ名を指定します。これは--disable-triggersを使う場合にのみ使用されます。(通常は このオプションを使うよりも、出力されたスクリプトをスーパーユーザ権限で実行する方が良いでしょう。)

-t pattern
--table=pattern

patternにマッチする名前のテーブルのみをダンプします。複数の-tオプションを記述することで複数のテーブルを選択することができます。patternパラメータはpsqlの \dコマンドで使用される規則(下記のPatterns参照)と同じ規則に従うパターンとして解釈されます。ですので、ワイルドカード文字をパターン内に記述することで、複数のテーブルを選択することもできます。ワイルドカードを使用する時は、シェルによりそのワイルドカードを展開させないように、パターンを引用符で括ってください。下記のExamplesを参照してください。

このオプションは、テーブルだけでなく、ビュー、マテリアライズドビュー、外部テーブル、シーケンスの定義をダンプするのにも使えます。ビューやマテリアライズドビューの内容はダンプしませんし、対応する外部サーバに--include-foreign-dataが指定されている場合にのみ、外部テーブルの内容がダンプされます。

-tが使用されると、-nおよび-Nオプションの効果はなくなります。-tで選択したテーブルが、これらのオプションとは関係なくダンプされ、また、非テーブルオブジェクトはダンプされないためです。

> **注記**
>
> -tが指定されると、pg_dumpは選択したテーブル内のオブジェクトが依存する可能性がある他のデータベースオブジェクトのダンプを行いません。したがって、テーブル指定のダンプ結果を初期化されたデータベースに正常にリストアできるという保証はありません。

> **注記**
>
> -tオプションの動作は8.2より前のバージョンのPostgreSQLと完全な互換性はありません。以前は、-t tabと記述することでtabという名前のテーブルをすべてダンプしていました。しかし現在は、デフォルトの検索パスで可視のものだけがダンプされます。過去の動作を行うためには、-t '*.tab'と記述してください。また、特定のスキーマ内のテーブルを選択するためには、以前は-n sch -t tabと記述していましたが、-t sch.tabなどと記述しなければなりません。

-T pattern
--exclude-table=pattern

patternにマッチするテーブルをダンプしません。このパターンは-tと同じ規則に従って解釈されます。-Tを複数指定することで、複数のパターンのいずれかにマッチするテーブルをすべて除外させることができます。

-tと-Tの両方が指定された場合、少なくとも1つの-tオプションにマッチし、-Tオプションにマッチしないテーブルのみがダンプされます。-tなしで-Tが指定された場合、通常のダンプから-Tにマッチするテーブルが除外されます。

-v
--verbose

冗長モードを指定します。これを指定すると、pg_dumpは、詳細なオブジェクトコメント、開始時刻、終了時刻をダンプファイルに、進行状況メッセージを標準エラーに出力します。

-V
--version

pg_dumpのバージョンを表示し、終了します。

-x
--no-privileges
--no-acl

アクセス権限(grant/revokeコマンド)のダンプを抑制します。

-Z 0..9
--compress=0..9

使用する圧縮レベルを指定します。ゼロは圧縮無しを意味します。カスタムアーカイブ形式では、これは個々のテーブルデータセグメントの圧縮を指定するもので、デフォルトでは中間レベルで圧縮されます。平文出力では、非ゼロの圧縮レベルの指定によりあたかもgzipに渡されたかのように出力ファイル全体が圧縮されます。しかしデフォルトは圧縮無しです。tarアーカイブ形式では現在圧縮を全くサポートしていません。

--binary-upgrade

このオプションは現位置でのアップグレード用のユーティリティにより使用されるものです。他の目的での使用は推奨されませんし、サポートもされません。このオプションの動作は、将来通知することなく変更される可能性があります。

--column-inserts
--attribute-inserts

明示的に列名を付けたINSERTコマンド(INSERT INTO table (column, ...)VALUES...)としてデータをダンプします。これによりリストアは非常に遅くなります。主にPostgreSQL以外のデータベースへロード可能なダンプを作成する時に有用です。再ロード中のエラーでは、テーブルの内容がまるごと失われることはなく、問題のあるINSERTの一部の行が失われるだけです。

--disable-dollar-quoting

このオプションは、関数本体用のドル引用符の使用を無効にし、強制的に標準SQLの文字列構文を使用した引用符付けを行います。

--disable-triggers

このオプションは、データのみのダンプを作成する場合にしか適用されません。データの再ロード中に、対象テーブル上のトリガを一時的に無効にするコマンドを出力するようpg_dumpに指示します。このオプションは、データの再ロード中には呼び出したくない参照整合性検査やその他のトリガがテーブル上にある場合に使用します。

現在のところ、--disable-triggersに対応するコマンドを実行するのは、スーパーユーザでなければなりません。そのため、-Sでスーパーユーザの名前を指定するか、あるいは、可能であれば、スーパーユーザ権限でスクリプトを開始するよう注意する必要があります。

このオプションは平文形式の場合にのみ有効です。他の形式では、pg_restoreを呼び出す時にこのオプションを指定することができます。

--enable-row-security

このオプションは、行セキュリティのあるテーブルの内容をダンプするときにのみ意味を持ちます。デフォルトでは、pg_dumpはrow_securityをoffに設定し、テーブルからすべてのデータがダンプされるようにします。ユーザが行セキュリティを無視できるだけの十分な権限を持っていない場合、エラーが発生します。このパラメータはpg_dumpがrow_securityをonに設定するようにすることで、テーブルの内容のうち、ユーザがアクセスできる部分をダンプすることを可能にします。

リストア時のCOPY FROMが行セキュリティをサポートしていないので、今、このオプションを使う場合は、ダンプをINSERT形式にするのがおそらく望ましいでしょう。

--exclude-table-data=pattern

patternにマッチするすべてのテーブルのデータをダンプしません。パターンは-t用の規則と同じ規則にしたがって解釈されます。複数のパターンのいずれかにマッチするテーブルを除外することができるように、--exclude-table-dataを複数回与えることができます。このオプションは、特定のテーブルに関してデータを格納する必要はないがテーブル定義は必要である場合に有用です。

データベース内のすべてのテーブルに関してデータを除外するためには、--schema-onlyを参照してください。

--extra-float-digits=ndigits

浮動小数点データをダンプする時に、利用できる最大の精度の代わりに、extra_float_digitsで指定された値を使います。バックアップ目的で作られたダンプルーチンは、このオプションを使うべきではありません。

--if-exists

データベースオブジェクトを初期化するときに、条件コマンドを使います(つまり、IF EXISTS句を追加します)。このオプションは、--cleanも指定されているのでなければ、有効にはなりません。

--include-foreign-data=foreignserver

foreignserverパターンとマッチする外部サーバの外部テーブルのデータをダンプします。複数の--include-foreign-dataスイッチを書くことで、複数の外部サーバを選択できます。また、foreignserverパラメータはpsqlの\dコマンドで使われているのと同じ規則に従うパターンとして解釈されます(下記のPatternsを参照してください)ので、パターンにワイルドカード文字を書くことで複数の外部サーバを選択することもできます。ワイルドカードを使う時は、シェルがワイルドカードを展開しないよう必要ならパターンを引用符で囲むことに注意してください。下記のExamplesを参照してください。唯一の例外は空のパターンが許されていないことです。

> **注記**
>
> --include-foreign-dataが指定された場合、pg_dumpは外部テーブルが書き込み可能かどうか確認しません。そのため、外部テーブルのダンプの結果をリストアするのに成功する保証はありません。

--inserts

データを(COPYではなく)INSERTコマンドの形式でダンプします。これを行うとリストアに非常に時間がかかります。主にPostgreSQL以外のデータベースへロード可能なダンプを作成する時に有用です。再ロード中のエラーでは、テーブルの内容がまるごと失われることはなく、問題のあるINSERTの一部の行が失われるだけです。列の順序を変更した場合はリストアが失敗する可能性があることに注意してください。--column-insertsはさらに処理が遅くなりますが、列の順序変更に対して安全です。

--load-via-partition-root

テーブルパーティションのデータをダンプするときには、COPYあるいはINSERT文の対象を、そのパーティションではなく、それを含むパーティション階層のルートにします。これにより、データが読み込まれるときに各行に対して適切なパーティションが再判断されます。これは行が元のサーバ上と同じパーティションに必ずしも落ちないようなサーバ上にデータを再読み込みするときに有用でしょう。例えば、パーティション列がtext型で二つのシステムがパーティション列のソートで使われる照合順序の異なる定義を持っている場合に、これはあり得ます。

pg_restoreは与えられたアーカイブデータ要素がデータを投入するパーティションがどれであるかを正確に知ることがないため、このオプションで作られたアーカイブからリストアを行うときには並列化しない方が良いです。並行ジョブ間でのロック競合のため、これは非効率になります。また、全ての関連データが読み込まれる前に外部キー制約が設定されることにより、場合によってはリロード失敗も起こります。

--lock-wait-timeout=timeout

ダンプの開始時に共有テーブルのロックを永遠に待ちません。代わりに指定したtimeout内にテーブルロックを獲得できなければ失敗します。タイムアウトはSET statement_timeoutで受け付けられる任意の書式で指定できます。(使用可能な形式はダンプの元となるサーバのバージョンに依存して異なりますが、すべてのバージョンにおいてミリ秒単位の整数値は受け付けられます。)

--no-comments

コメントをダンプしません。

--no-publications

パブリケーションをダンプしません。

--no-security-labels

セキュリティラベルをダンプしません。

--no-subscriptions

サブスクリプションをダンプしません。

--no-sync

デフォルトでは、pg_dumpはすべてのファイルが確実にディスクに書き出されるまで待機します。このオプションを使うとpg_dumpは待機せずに戻るため、より高速になりますが、これは、その後にオペレーティングシステムがクラッシュすると、ダンプが破損する可能性があることを意味します。一般的に言って、このオプションはテスト用には有用ですが、実運用の環境からデータをダンプするときには使用しない方が良いでしょう。

--no-synchronized-snapshots

このオプションにより、9.2より前のサーバに対してpg_dump -jを実行することができます。詳細については-jパラメータの説明を参照してください。

--no-tablespaces

テーブル空間を選択するコマンドを出力しません。このオプションを使用すると、すべてのオブジェクトはリストア時のデフォルトのテーブル空間の中に作成されます。

このオプションは平文書式でのみ有意です。アーカイブ書式では、pg_restoreを呼び出す時にこのオプションを指定することができます。

--no-unlogged-table-data

ログを取らないテーブルの内容をダンプしません。このオプションはテーブル定義(スキーマ)をダンプするかどうかには影響しません。そのテーブルデータのダンプを抑制するだけです。スタンバイサーバからダンプを行う場合、ログを取らないテーブル内のデータは常に除外されます。

--on-conflict-do-nothing

INSERTコマンドにON CONFLICT DO NOTHINGを追加します。このオプションは、--inserts、--column-insertsまたは--rows-per-insertが同時に指定されていなければ、有効ではありません。

--quote-all-identifiers

強制的にすべての識別子に引用符を付与します。このオプションは、pg_dumpのメジャーバージョンとは異なるメジャーバージョンのPostgreSQLのサーバーからデータベースをダンプするとき、あるいは出力を異なるメジャーバージョンのサーバにロードする予定であるときに推奨されます。デフォルトでは、pg_dumpは、それ自身のメジャーバージョンにおける予約語である識別子に対してのみ引用符を付与します。これは、他のバージョンのサーバを処理するときに互換性の問題を引き起こす場合があり

ます。他のバージョンのサーバでは予約語の集合が多少、異なる場合があるからです。--quote-all-identifiersを使用することで、ダンプのスクリプトが読みにくくなりますが、このような問題を防ぐことができます。

--rows-per-insert=nrows

(COPYではなく)INSERTコマンドでデータをダンプします。INSERTコマンド1つあたりの最大行数を制御します。指定する値は0より大きくなければなりません。再ロード中のエラーでは、テーブルの内容がまるごと失われることはなく、問題のあるINSERTの一部の行が失われるだけです。

--section=sectionname

指定した部分のみをダンプします。部分名はpre-data、data、post-dataのいずれかを取ることができます。複数の部分を選択するために、このオプションは複数回指定することができます。デフォルトではすべての部分をダンプします。

data部分には、実際のテーブルデータとラージオブジェクトの中身、シーケンス値が含まれます。post-data項目は、インデックス定義、トリガ定義、ルール定義、有効化された検査制約以外の制約定義が含まれます。pre-data項目は、他のすべてのデータ定義項目が含まれます。

--serializable-deferrable

使用されるスナップショットがその後のデータベース状態と一貫性を持つことを保証するために、ダンプ時にserializableトランザクションを使用します。ダンプが失敗したり、serialization_failureにより他のトランザクションがロールバックしたりする危険がないように、トランザクションストリーム内で異常が発生することがない時点まで待つことでこれを行います。トランザクション分離および同時実行性の制御については第13章を参照してください。

このオプションは障害対策のリカバリのみを目的とするダンプでは利点はありません。元のデータベースを継続して更新しながら、レポート処理や他の読み取りのみの負荷分散のためにデータベースのコピーをロードするために使用されるダンプとして有用になります。こうしないと、ダンプには 何らかのトランザクションの直列実行が最終的にコミットされた状態と一貫性がない状態が反映される可能性があります。例えば、バッチ処理技術が使用される場合、バッチは、バッチ内で存在するすべての項目を持たないダンプ内でクローズしたものと表示される可能性があります。

pg_dumpを始めた時に読み書きを行う実行中のトランザクションが存在しない場合、このオプションは何の差異ももたらしません。読み書きを行うトランザクションが実行中の場合、確定できない期間、ダンプの起動が遅延される可能性があります。動き出してからの性能は、このスイッチがある場合とない場合とで違いはありません。

--snapshot=snapshotname

データベースのダンプを作成する時に、指定した同期スナップショットを使用します（詳しくは表 9.88を参照して下さい）。

このオプションは、ダンプを論理レプリケーションスロット（第48章参照）あるいは同時実行セッションと同期させる必要がある時に役に立ちます。

並行ダンプの場合、新しいスナップショットを作る代わりに、このオプションで指定されたスナップショット名が使われます。

--strict-names

各スキーマ指定(-n/--schema)および各テーブル指定(-t/--table)が、ダンプされるデータベース内の少なくとも1つのスキーマおよびテーブルにマッチすることを必要とします。スキーマおよびテーブル指定のどれもがマッチするものを見つけられなかった場合は、--strict-namesがなくてもpg_dumpはエラーを発生させることに注意して下さい。

このオプションは-N/--exclude-schema、-T/--exclude-table、--exclude-table-dataには影響を与えません。除外のパターンがマッチするオブジェクトを見つけられないことは、エラーとはみなされません。

--use-set-session-authorization

オブジェクトの所有権を決定するために、ALTER OWNERコマンドの代わりに標準SQLのSET SESSION AUTHORIZATIONコマンドを出力します。これにより、ダンプの標準への互換性が高まりますが、ダンプ内のオブジェクトの履歴によっては正しくリストアされない可能性が生じます。また、SET SESSION AUTHORIZATIONを使用したダンプを正しくリストアするためには、確実にスーパーユーザ権限が必要となります。ALTER OWNERで必要な権限はこれよりも少なくなります。

-?
--help

pg_dumpのコマンドライン引数の使用方法を表示し、終了します。

以下のコマンドラインオプションは、データベース接続パラメータを制御します。

-d dbname
--dbname=dbname

接続するデータベースの名前を指定します。コマンドラインでオプション以外の最初の引数としてdbnameを指定することと同じです。dbnameは接続文字列でも構いません。その場合、接続文字列パラメータは衝突するコマンドラインオプションに優先します。

-h host
--host=host

サーバが稼働しているマシンのホスト名を指定します。この値がスラッシュから始まる場合、Unixドメインソケット用のディレクトリとして使用されます。デフォルトは、設定されていれば環境変数PGHOSTから取得されます。設定されていなければ、Unixドメインソケット接続とみなされます。

-p port
--port=port

サーバが接続を監視するTCPポートもしくはローカルUnixドメインソケットファイルの拡張子を指定します。デフォルトは、設定されている場合、環境変数PGPORTの値となります。設定されていなければ、コンパイル時のデフォルト値となります。

-U username
--username=username

接続ユーザ名です。

-w
--no-password

パスワードの入力を促しません。サーバがパスワード認証を必要とし、かつ、.pgpassファイルなどの他の方法が利用できない場合、接続試行は失敗します。バッチジョブやスクリプトなどパスワードを入力するユーザが存在しない場合にこのオプションは有用かもしれません。

-W
--password

データベースに接続する前に、pg_dumpは強制的にパスワード入力を促します。

サーバがパスワード認証を要求する場合pg_dumpは自動的にパスワード入力を促しますので、これが重要になることはありません。しかし、pg_dumpは、サーバにパスワードが必要かどうかを判断するための接続試行を無駄に行います。こうした余計な接続試行を防ぐために-Wの入力が有意となる場合もあります。

--role=rolename

ダンプを作成する際に使用するロール名を指定します。このオプションによりpg_dumpはデータベースに接続した後にSET ROLE rolenameコマンドを発行するようになります。認証に使用したユーザ(-Uで指定されたユーザ)がpg_dumpで必要とされる権限を持たないが、必要な権限を持つロールに切り替えることができる場合に有用です。一部のインストレーションではスーパーユーザとして直接ログインさせないポリシーを取ることがありますが、このオプションを使用することでポリシーに反することなくダンプを作成することができます。

環境

PGDATABASE
PGHOST
PGOPTIONS
PGPORT
PGUSER

デフォルトの接続パラメータです。

PG_COLOR

診断メッセージで色を使うかどうかを指定します。可能な値はalways、auto、neverです。

また、このユーティリティは、他のほとんどのPostgreSQLユーティリティと同様、libpqでサポートされる環境変数を使用します(33.14を参照してください)。

診断

pg_dumpは内部でSELECT文を実行します。pg_dumpの実行時に問題が発生する場合は、psqlなどを使用して、そのデータベースから情報をselectできることを確認してください。また、libpqフロントエンドライブラリで使用されるデフォルトの接続設定や環境変数も適用されます。

通常pg_dumpのデータベースに対する活動は統計情報コレクタにより収集されます。これを望まない場合、PGOPTIONSまたはALTER USERコマンドを使用してtrack_countsパラメータを偽に設定してください。

注釈

データベースクラスタにおいてtemplate1データベースに対し独自の変更を行っている場合、pg_dumpの出力は、確実に空のデータベースにリストアするように注意してください。そうしないと、おそらく追加されたオブジェクトの重複定義によってエラーが発生します。独自の追加が反映されていない空のデータベースを作成するには、template1ではなくtemplate0をコピーしてください。以下に例を示します。

```
CREATE DATABASE foo WITH TEMPLATE template0;
```

データのみのダンプを選択し、--disable-triggersオプションを使用する場合、pg_dumpはデータを挿入する前にユーザテーブルにトリガを無効にするコマンドを発行し、データの挿入が完了した後で、それらを再び有効にする問い合わせを発行します。リストアが途中で停止した場合、システムカタログが不適切な状態のままになっている可能性があります。

pg_dumpが生成するダンプファイルには、オプティマイザが問い合わせ計画を決定する際に使用される統計情報が含まれていません。そのため、最適な性能を発揮するために、ダンプファイルからリストアした後でANALYZEを実行することをお勧めします。詳しくは24.1.3および24.1.6を参照してください。

pg_dumpは新しいバージョンのPostgreSQLへのデータ移行に使用されますので、pg_dumpの出力はpg_dumpのバージョンより新しいバージョンのPostgreSQLデータベースへロード可能と想定できるようになっています。また、pg_dumpは自身より古いバージョンのPostgreSQLデータベースを読み取ることもできます。(現在はバージョン8.0までのサーバをサポートします。) しかし、pg_dumpはそれ自身のメジャーバージョンより新しいPostgreSQLサーバのダンプを取ることはできません。無効なダンプを作成するリスクは取らず、ダンプしようとさえしません。また、pg_dumpの出力がメジャーバージョンが古いサーバにロードできるとは、たとえ同じバージョンのサーバから取得したダンプであっても、保証されていません。より古いサーバへのダンプファイルのロードには、古いサーバでは理解できない構文を削除するために、ダンプファイルの手作業による修正が必要になることがあります。バージョンをまたがる場合は--quote-all-identifiersオプションの使用が推奨されます。これにより、PostgreSQLの異なるバージョンで、予約語のリストが変わることによって発生する問題を防ぐことができるからです。

論理レプリケーションのサブスクリプションをダンプするとき、pg_dumpはconnect = falseオプションを使用するCREATE SUBSCRIPTION生成するため、サブスクリプションのリストア時には、レプリケーションスロットの作成や初回のテーブルコピーのためのリモート接続が行われません。このため、リモートサーバへのネットワーク接続を必要とせずにダンプをリストアできます。その後でサブスクリプションを適切な方法で再有効化するのはユーザの責任です。関連するホストが変更されたときは、接続情報も変更しなければならないかもしれません。新しく完全なテーブルコピーを開始する前に、コピー先のテーブルを空にするのが適切なこともあります。

例

mydbという名前のデータベースをSQLスクリプトファイルにダンプします。

```
$ pg_dump mydb > db.sql
```

newdbという名前の（新規に作成した）データベースにスクリプトを再ロードします。

```
$ psql -d newdb -f db.sql
```

カスタム形式のアーカイブファイルにデータベースをダンプします。

```
$ pg_dump -Fc mydb > db.dump
```

ディレクトリ形式アーカイブにデータベースをダンプします。

```
$ pg_dump -Fd mydb -f dumpdir
```

5個の作業用ジョブを使用してデータベースをdirectory形式のアーカイブにダンプします。

```
$ pg_dump -Fd mydb -j 5 -f dumpdir
```

newdbという名前の（新規に作成した）データベースにアーカイブファイルを再ロードします。

```
$ pg_restore -d newdb db.dump
```

アーカイブファイルをダンプ元と同じデータベースに再ロードし、そのデータベースの現在の内容を捨てます。

```
$ pg_restore -d postgres --clean --create db.dump
```

mytabという名前の単一のテーブルをダンプします。

```
$ pg_dump -t mytab mydb > db.sql
```

detroitスキーマ内の名前がempで始まるすべてのテーブルをダンプします。ただし、employee_logという名前のテーブルは除きます。

```
$ pg_dump -t 'detroit.emp*' -T detroit.employee_log mydb > db.sql
```

eastまたはwestで始まりgsmで終わるスキーマをすべてダンプします。ただし、testという単語を含む場合は除きます。

```
$ pg_dump -n 'east*gsm' -n 'west*gsm' -N '*test*' mydb > db.sql
```

正規表現記法を使用してオプションをまとめた形で同じことを行います。

```
$ pg_dump -n '(east|west)*gsm' -N '*test*' mydb > db.sql
```

ts_から始まる名前のテーブルを除き、すべてのデータベースオブジェクトをダンプします。

```
$ pg_dump -T 'ts_*' mydb > db.sql
```

大文字または大文字小文字混在の名前を-tなどのスイッチに指定するには、名前を二重引用符で括らなければなりません。さもないと小文字に変換されます（下記のPatternsを参照してください）。しかし、二重引用符はシェルでも特別に扱われますので、これも引用符で括らなければなりません。したがって、大文字小文字混在の名前を持つテーブルを1つダンプするには、以下のようにしなければなりません。

```
$ pg_dump -t "\"MixedCaseName\"" mydb > mytab.sql
```

関連項目

pg_dumpall, pg_restore, psql

pg_dumpall

pg_dumpall — PostgreSQLのデータベースクラスタをスクリプトファイルへ抽出する

概要

pg_dumpall [connection-option...] [option...]

説明

pg_dumpallはクラスタの全てのPostgreSQLデータベースを、1つのスクリプトファイルに書き出す(「ダンプする」)ためのユーティリティです。スクリプトファイルには、データベースのリストアのためにpsqlへの入力として使うことのできるSQLコマンドが含まれています。これはクラスタ内の各データベースに対してpg_dumpを呼び出すことで行われます。さらに、pg_dumpallは、全てのデータベースに共通するグローバルオブジェクト、すなわちデータベースロールとテーブルスペースもダンプします (pg_dumpはこれらのオブジェクトを保存しません)。

pg_dumpallは全てのデータベースからテーブルを読み込むため、完全なダンプを作成するには、おそらくデータベーススーパーユーザとして接続する必要がある可能性が高いでしょう。さらに、保存されたスクリプトを実行する時には、ロールを追加したり、データベースを作成したりするので、スーパーユーザ権限が必要になるでしょう。

SQLスクリプトは標準出力に書き込まれます。それをファイルにリダイレクトするためには、-f/--fileオプションまたはシェルの演算子を使用して下さい。

pg_dumpallは、PostgreSQLサーバに何度か接続しなければなりません(データベースごとに接続することになります)。パスワード認証を使用している場合、その度にパスワード入力が促されます。そのような場合は~/.pgpassファイルを用意しておくと便利です。詳細は33.15を参照してください。

オプション

以下のコマンドラインオプションは出力内容や形式を制御します。

-a
--data-only

データのみをダンプし、スキーマ(データ定義)をダンプしません。

-c
--clean

データベースを再作成するコマンドの前に、データベースのクリーンアップ(削除)するコマンドを書き出します。ロールおよびテーブル空間用のDROPコマンドも同様に追加されます。

-E encoding
--encoding=encoding

指定された文字エンコーディングでダンプを作ります。デフォルトでは、ダンプはデータベースエンコーディングで作られます。(同じ結果を得る他の方法はPGCLIENTENCODING環境変数を望みのダンプエンコーディングに設定することです。)

-f filename
--file=filename

出力を指定したファイルに送ります。これが省略されると標準出力が使用されます。

-g
--globals-only

グローバルオブジェクト（ロールとテーブル空間）のみをダンプし、データベースのダンプを行いません。

-O
--no-owner

オブジェクトの所有権を元のデータベースに一致させるためのコマンドを出力しません。デフォルトでは、pg_dumpallはALTER OWNER文またはSET SESSION AUTHORIZATION文を発行して作成したスキーマ要素の所有権を設定します。スーパーユーザ（もしくは、スクリプト内の全てのオブジェクトを所有するユーザ）以外のユーザがスクリプトを実行した場合、これらの文は失敗します。任意のユーザがリストアできるスクリプトを作成するには、-Oを指定してください。ただし、この場合は、全てのオブジェクトの所有者がリストアしたユーザとなってしまいます。

-r
--roles-only

ロールのみをダンプし、データベースやテーブル空間のダンプを行いません。

-s
--schema-only

オブジェクト定義（スキーマ）のみをダンプし、データをダンプしません。

-S username
--superuser=username

トリガを無効にする際に使用するスーパーユーザのユーザ名を指定します。これは--disable-triggersを使用する場合にのみ使用されます（通常はこのオプションを使うよりも、出力されたスクリプトをスーパーユーザ権限で実行する方が良いでしょう）。

-t
--tablespaces-only

テーブル空間のみをダンプし、データベースやロールのダンプを行いません。

-v
--verbose

冗長モードを指定します。これを指定すると、pg_dumpallは開始時刻と終了時刻をダンプファイルに、進行メッセージを標準エラーに出力するようになります。また、これによりpg_dumpの冗長出力が有効になります。

-V
--version

pg_dumpallのバージョンを表示し、終了します。

-x
--no-privileges
--no-acl

アクセス権限のダンプ(grant/revokeコマンド)を行いません。

--binary-upgrade

このオプションは現位置でのアップグレード用のユーティリティにより使用されるものです。他の目的での使用は推奨されませんし、サポートもされません。このオプションの動作は、将来通知することなく変更される可能性があります。

--column-inserts
--attribute-inserts

明示的に列名を付けたINSERTコマンド(INSERT INTO table (column, ...) VALUES...)としてダンプします。これによりリストアは非常に遅くなります。主に、PostgreSQL以外のデータベースにロード可能なダンプを作成する時に有用です。

--disable-dollar-quoting

このオプションは、関数本体用のドル引用符の使用を無効にし、強制的に標準SQLの文字列構文を使用した引用符付けを行います。

--disable-triggers

このオプションは、データのみのダンプを作成する場合だけに使用します。データのリロード中に、対象とするテーブル上のトリガを一時的に使用不可にするためのコマンドを出力するようpg_dumpallに指示します。このオプションは、データのリロード中には呼び出したくない参照整合性検査やその他のトリガがテーブル上にある場合に使用します。

現在のところ、--disable-triggersを指定してコマンドを実行するのは、スーパーユーザでなければなりません。そのため、-Sでスーパーユーザの名前を指定するか、あるいは、可能であれば、スーパーユーザ権限でスクリプトを開始するよう注意する必要があります。

--exclude-database=pattern

名前がpatternにマッチするデータベースをダンプしません。複数の--exclude-databaseオプションを記述することで複数のパターンを除外できます。patternパラメータはpsqlの\dコマンドで使用される規則(以下のPatterns参照)と同じ規則に従うパターンとして解釈されます。ですので、ワイルドカード文字をパターン内に記述することで、複数のデータベースを除外することもできます。ワイルドカードを使用する時は、シェルによりそのワイルドカードを展開させないように、パターンを引用符で括ってください。

--extra-float-digits=ndigits

浮動小数点データをダンプする時に、利用できる最大の精度の代わりに、extra_float_digitsで指定された値を使います。バックアップ目的で作られたダンプルーチンは、このオプションを使うべきではありません。

--if-exists

データベースおよび他のオブジェクトを初期化するときに、条件コマンドを使います(つまりIF EXISTS句を追加します)。このオプションは、--cleanも指定されているのでなければ、有効にはなりません。

--inserts

データを（COPYではなく）INSERTコマンドとしてダンプします。これを行うとリストアが非常に遅くなります。主にPostgreSQL以外のデータベースにロード可能なダンプを作成する時に有用です。列の順序を変更した場合はリストアが失敗する可能性があることに注意してください。さらに低速になりますが、--column-insertsオプションの方が安全です。

--load-via-partition-root

テーブルパーティションに対するデータをダンプするとき、COPYやINSERT文をパーティション自体ではなく、それを含むパーティション階層のルートに向けさせます。これにより、データが読み込まれるときに各行に対して適切なパーティションが再判断されるようになります。これは、行が必ずしも元のサーバと同じパーティションに落ちないようなサーバにデータを再読み込みするときに有用でしょう。例えば、パーティション列がtext型で二つのシステムがパーティション列をソートするのに使われる照合順序の異なった定義を持っている場合に、これは起こりえることです。

--lock-wait-timeout=timeout

ダンプ開始時に共有テーブルロックの獲得のために永遠に待機しません。指定したtimeoutの間にテーブルをロックすることができない場合は失敗します。タイムアウトはSET statement_timeoutで受け付けられる任意の書式で指定することができます。許される値はダンプ元のサーババージョンに依存して変わります。しかし、7.3以降のすべてのバージョンでミリ秒単位の整数値は受け付けられます。このオプションは7.3より前のサーバからダンプされた場合、無視されます。

--no-comments

コメントをダンプしません。

--no-publications

パブリケーションをダンプしません。

--no-role-passwords

ロールのパスワードをダンプしません。リストア時にロールのパスワードはNULLになるため、パスワードが設定されるまでは、パスワード認証は常に失敗します。このオプションが指定された場合、パスワードの値が不要となるので、ロール情報はpg_authidではなく、カタログビューのpg_rolesから読み取られます。従って、pg_authidへのアクセスが何らかのセキュリティーポリシーによって制限されている場合にも、このオプションは役立ちます。

--no-security-labels

セキュリティラベルをダンプしません。

--no-subscriptions

サブスクリプションをダンプしません。

--no-sync

デフォルトでは、pg_dumpallはすべてのファイルが確実にディスクに書き込まれるまで待機します。このオプションを使うとpg_dumpallは待機せずに戻るため、より高速になりますが、これは、その後にオペレーティングシステムがクラッシュすると、ダンプが破損する可能性があることを意味します。一般的に

言って、このオプションはテスト用には有用ですが、実運用の環境からデータをダンプするときには使用しないほうが良いでしょう。

--no-tablespaces

オブジェクト用にテーブル空間を作成または選択するコマンドを出力しません。このオプションを付けると、すべてのオブジェクトはリストア時のデフォルトのテーブル空間内に作成されます。

--no-unlogged-table-data

ログを取らないテーブルの内容をダンプしません。このオプションはテーブル定義(スキーマ)をダンプするかどうかには影響しません。そのテーブルデータのダンプを抑制するだけです。

--on-conflict-do-nothing

INSERTコマンドにON CONFLICT DO NOTHINGを追加します。このオプションは、--insertsまたは--column-insertsが同時に指定されていなければ、有効ではありません。

--quote-all-identifiers

強制的にすべての識別子に引用符を付与します。このオプションは、pg_dumpallのメジャーバージョンとは異なるメジャーバージョンのPostgreSQLのサーバーからデータベースをダンプするとき、あるいは出力を異なるメジャーバージョンのサーバにロードする予定であるときに推奨されます。デフォルトでは、pg_dumpallは、それ自身のメジャーバージョンにおける予約語である識別子に対してのみ引用符を付与します。これは、他のバージョンのサーバを処理するときに互換性の問題を引き起こす場合があります。他のバージョンのサーバでは予約語の集合が多少、異なる場合があるからです。--quote-all-identifiersを使用することで、ダンプのスクリプトが読みにくくなりますが、このような問題を防ぐことができます。

--rows-per-insert=nrows

(COPYではなく)INSERTコマンドでデータをダンプします。INSERTコマンド1つあたりの最大行数を制御します。指定する値は0より大きくなければなりません。再ロード中のエラーでは、テーブルの内容がまるごと失われることはなく、問題のあるINSERTの一部の行が失われるだけです。

--use-set-session-authorization

ALTER OWNERコマンドの代わりに標準SQLのSET SESSION AUTHORIZATIONコマンドを出力します。これにより、ダンプの標準への互換性が高まりますが、ダンプ内のオブジェクトの履歴によっては正しくリストアされない可能性があります。

-?
--help

pg_dumpallコマンドライン引数の使用方法を表示し、終了します。

以下のコマンドラインオプションは、データベース接続パラメータを制御します。

-d connstr
--dbname=connstr

サーバに接続するために使用されるパラメータを接続文字列として指定します。これは衝突するコマンドラインオプションよりも優先します。

このオプションは、他のクライアントアプリケーションとの一貫性のために--dbnameと呼ばれます。しかしpg_dumpallは多くのデータベースに接続しなければなりませんので、接続文字列内のデータベース名は無視されます。グローバルオブジェクトをダンプするために使用されるデータベースの名前を指定するため、または他のどのデータベースをダンプしなければならないかを見つけるためには-lを使用してください。

-h host
--host=host

データベースサーバが稼働しているマシンのホスト名を指定します。この値がスラッシュから始まる場合、Unixドメインソケット用のディレクトリとして使用されます。デフォルトは、設定されていれば環境変数PGHOSTから取得されます。設定されていなければ、Unixドメインソケット接続とみなされます。

-l dbname
--database=dbname

グローバルオブジェクトをダンプし、他のどのデータベースをダンプすべきかを見つけるために接続するデータベースの名前を指定します。指定されなかった場合、postgresが使用されます。もしこれも存在しない場合はtemplate1が使用されます。

-p port
--port=port

サーバが接続を監視するTCPポートもしくはローカルUnixドメインソケットファイルの拡張子を指定します。デフォルトは、設定されている場合、環境変数PGPORTの値になります。設定されていなければ、コンパイル時のデフォルト値となります。

-U username
--username=username

接続ユーザ名です。

-w
--no-password

パスワードの入力を促しません。サーバがパスワード認証を必要とし、かつ、.pgpassファイルなどの他の方法が利用できない場合、接続試行は失敗します。バッチジョブやスクリプトなどパスワードを入力するユーザが存在しない場合にこのオプションは有用かもしれません。

-W
--password

データベースに接続する前に、pg_dumpallは強制的にパスワード入力を促します。

サーバがパスワード認証を要求する場合pg_dumpallは自動的にパスワード入力を促しますので、これが重要になることはありません。しかし、pg_dumpallは、サーバにパスワードが必要かどうかを判断するための接続試行を無駄に行います。こうした余計な接続試行を防ぐために-Wの入力が有意となる場合もあります。

パスワードの入力はダンプするデータベース毎に繰り返し促されます。通常は、手作業のパスワード入力に依存するよりも~/.pgpassを設定する方が良いでしょう。

--role=rolename

ダンプを作成する際に使用するロール名を指定します。このオプションによりpg_dumpallはデータベースに接続した後にSET ROLE rolenameコマンドを発行するようになります。認証に使用したユーザ(-Uで指定されたユーザ)がpg_dumpallで必要とされる権限を持たないが、必要な権限を持つロールに切り替えることができる場合に有用です。一部のインストレーションではスーパーユーザとして直接ログインさせないポリシーを取ることがありますが、このオプションを使用することでポリシーに反することなくダンプを作成することができます。

環境

PGHOST
PGOPTIONS
PGPORT
PGUSER

デフォルトの接続パラメータです。

PG_COLOR

診断メッセージで色を使うかどうかを指定します。可能な値はalways、auto、neverです。

また、このユーティリティは、他のほとんどのPostgreSQLユーティリティと同様、libpqでサポートされる環境変数を使用します(33.14を参照してください)。

注釈

pg_dumpallは、内部でpg_dumpを呼び出すので、診断メッセージの一部ではpg_dumpを参照しています。

--cleanオプションは、ダンプスクリプトを新しいクラスタにリストアする意図のときであっても、有用なことがあります。--cleanの使用は、スクリプトに組み込みのpostgresおよびtemplate1データベースを削除して再作成する権限を与え、これらのデータベースが元のクラスタに持っていたものと同じ属性(例えばロケールやエンコーディング)を保つことを保証します。このオプションが無いと、これらのデータベースは既存のデータベースレベルの属性を維持します。

オプティマイザが正確な統計情報を使用できるように、リストア後は、リストアしたテーブルそれぞれに対してANALYZEを実行することを勧めます。また、vacuumdb -a -zを実行すると、全てのデータベースを解析することができます。

ダンプスクリプトはエラー無しに完全に実行すると期待すべきではありません。特にスクリプトは存在する全てのロールに対してCREATE ROLEを発行するので、宛先のクラスタが異なるベースストラップスーパーユーザ名で初期化されていない限り、ブートストラップスーパーユーザについて「role already exists」エラーを受け取ることは確実です。このエラーは無害であり、無視してください。--cleanオプションの使用は、追加で無害な存在しないオブジェクトについてのエラーメッセージを発生させますが、--if-existsを加えることでエラー発生を最小化できます。

pg_dumpallでは、必要なテーブル空間用のディレクトリがリストア前に存在していることを要求します。存在しないと、デフォルト以外の場所にあるデータベースに関して、そのデータベース生成が失敗します。

例

全てのデータベースを書き出す場合、以下のようにします。

```
$ pg_dumpall > db.out
```

上記のファイルからデータベースを読み込む場合、以下のようにします。

```
$ psql -f db.out postgres
```

ここではどのデータベースに接続するかということは重要ではありません。なぜならpg_dumpallが作成するスクリプトファイルには、保存されたデータベースの作成および接続のための適切なコマンドが含まれているからです。例外は--cleanを指定した場合で、最初にpostgresデータベースに接続しなければなりません。このときのスクリプトは即座に他のデータベースを削除しようとし、接続中のデータベースに対しては失敗するでしょう。

関連項目

発生し得るエラーの原因については、pg_dumpを参照してください。

pg_isready

pg_isready — PostgreSQLサーバの接続状態を検査する

概要

pg_isready [*connection-option*...] [*option*...]

説明

pg_isreadyはPostgreSQLデータベースサーバの接続状態を検査するためのユーティリティです。終了ステータスが接続検査の結果を示します。

オプション

-d *dbname*
--dbname=*dbname*

接続するデータベースの名前を指定します。*dbname*は接続文字列でも構いません。その場合、接続文字列パラメータは衝突するコマンドラインオプションよりも優先します。

-h *hostname*
--host=*hostname*

サーバが稼働しているマシンのホスト名を指定します。値がスラッシュから始まる場合は、Unixドメインソケット用のディレクトリとして使用されます。

-p *port*
--port=*port*

サーバが接続を監視しているTCPポートまたはUnixドメインソケットファイルの拡張子を指定します。デフォルトは環境変数PGPORTの値、もし設定されていなければ、コンパイル時に指定したポート、通常は5432です。

-q
--quiet

状態メッセージを表示しません。これはスクリプト処理の際に有用です。

-t *seconds*
--timeout=*seconds*

サーバが応答しないことを返す前に、接続試行時に待機する最大秒数です。ゼロに設定すると無効になります。デフォルトは3秒です。

-U username
--username=username

デフォルトではなくusernameユーザとしてデータベースに接続します。

-V
--version

pg_isreadyのバージョンを表示し終了します。

-?
--help

pg_isreadyのコマンドライン引数に関する説明を表示し終了します。

終了ステータス

pg_isreadyは、サーバが通常通り接続を受け付けている場合は0を、サーバが接続を拒絶している(例えば起動時)場合は1を、接続試行に対する応答がない場合は2を、試行が行われなかった(例えば無効なパラメータが原因)場合は3をシェルに返します。

環境

他のほとんどのPostgreSQLユーティリティと同様、pg_isreadyはlibpqによってサポートされる環境変数(33.14参照)を使用します。

環境変数PG_COLORは診断メッセージで色を使うかどうかを指定します。可能な値はalways、auto、neverです。

注釈

サーバの状態を取得するのに、正しいユーザ名、パスワード、データベース名の値を使う必要はありません。しかし、正しくない値が使われた場合、サーバは接続試行に失敗したことをログに記録します。

例

標準的な使用方法を示します。

```
$ pg_isready
/tmp:5432 - accepting connections
$ echo $?
0
```

起動中のPostgreSQLクラスタに対して接続パラメータを付けて実行します。

```
$ pg_isready -h localhost -p 5433
```

```
localhost:5433 - rejecting connections
$ echo $?
1
```

応答しないPostgreSQLクラスタに対して接続パラメータを付けて実行します。

```
$ pg_isready -h someremotehost
someremotehost:5432 - no response
$ echo $?
2
```

pg_receivewal

pg_receivewal — PostgreSQLサーバから先行書き込みログをストリームする

概要

```
pg_receivewal [option...]
```

説明

pg_receivewalは実行中のPostgreSQLクラスタから先行書き込みログをストリームするために使用されます。先行書き込みログはストリーミングレプリケーションプロトコルを使用してストリームされ、ローカルディレクトリのファイルとして書き出されます。このディレクトリはポイントインタイムリカバリ(25.3参照)を用いてリストアする際のアーカイブ場所として使用することができます。

pg_receivewalは先行書き込みログをサーバで生成に合わせてリアルタイムでストリームし、archive_commandとは異なり、セグメントが完了するまで待機することはありません。このため、pg_receivewalを使用する場合にはarchive_timeoutを設定する必要はありません。

PostgreSQLのスタンバイサーバのWALレシーバと異なり、pg_receivewalはデフォルトでは、WALファイルがクローズされた時にのみ、WALデータをフラッシュします。WALデータをリアルタイムでフラッシュするには--synchronousオプションを指定する必要があります。pg_receivewalはWALを適用しないので、synchronous_commitがremote_applyのときにこれを同期スタンバイにすることはできません。そのようにした場合、決してキャッチアップすることのないスタンバイになり、トランザクションコミットのブロックをひき起こします。これを避けるには、synchronous_standby_namesに適切な値を設定するか、pg_receivewalに対して一致しないapplication_nameを指定する、あるいは、synchronous_commitの値をremote_apply以外の値に変更してください。

先行書き込みログは通常のPostgreSQL接続を経由して、そしてレプリケーションプロトコルを使用して、ストリームされます。この接続はREPLICATION権限(21.2参照)を持つユーザまたはスーパーユーザによって確立されなければなりません。そしてpg_hba.confでレプリケーション用の接続を許可しなければなりません。またサーバではストリーム用に利用できるセッションが少なくとも1つ存在できるためにmax_wal_sendersを十分大きく設定しなければなりません。

接続が失われた場合、または、致命的ではないエラーで初期確立ができなかった場合、pg_receivewalは無期限に接続を再試行しできるだけ早くストリーミングを再確立します。この動作を止めるためには-nパラメータを使用してください。

致命的なエラーが無い場合、pg_receivewalはSIGINTシグナル(**Control**+**C**)で停止されるまで実行を続けます。

オプション

```
-D directory
--directory=directory
```

出力を書き出すディレクトリです。

このパラメータは必須です。

-E lsn
--endpos=lsn

受信が指定したLSNに達したなら、自動的にレプリケーションを停止して、通常の終了ステータス0で終了します。

lsnとちょうど等しいLSNのレコードがある場合、そのレコードは処理されます。

--if-not-exists

--create-slotが指定され、指定された名前のスロットが既に存在していた場合に、エラーを発生させません。

-n
--no-loop

接続エラー時に繰り返しません。代わりにエラーですぐに終了します。

--no-sync

このオプションはpg_receivewalがWALデータをディスクに強制的にフラッシュさせないようにします。これはより高速ですが、オペレーションシステムのクラッシュ後にWALセグメントが破損している可能性があります。一般に、このオプションはテストには有益ですが、本番配備でWALのアーカイビングを行うときに使うべきではありません。

このオプションは--synchronousと両立しません。

-s interval
--status-interval=interval

サーバに状態パケットを返答する間隔を秒単位で指定します。これによりサーバからより簡単に進行状況を監視することができます。ゼロという値は状態の定期的な更新を完全に無効にします。しかしタイムアウトによる切断を防ぐために、サーバから要求された時には更新を送信します。デフォルト値は10秒です。

-S slotname
--slot=slotname

pg_receivewalが既存のレプリケーションスロットを使うようにします(26.2.6を参照してください)。このオプションが使われると、pg_receivewalはフラッシュ位置をサーバに報告します。これは、各セグメントがいつディスクに同期されたかを示し、それによりサーバが必要のなくなったセグメントを削除できるようになります。

pg_receivewalのレプリケーションクライアントが同期スタンバイとしてサーバ上で構成されている場合、レプリケーションスロットを利用するとフラッシュ位置がサーバに報告されますが、それはWALファイルがクローズされる時のみです。したがって、その構成ではプライマリ上のトランザクションが長時間待たされることになり、結果的に満足する動作を得られません。これを正しく動作させるには、追加で--synchronousオプション(以下を参照)を指定する必要があります。

--synchronous

WALデータを受け取ると即座にディスクにフラッシュします。またフラッシュした直後に、--status-intervalの値が何であれ、ステータスパケットをサーバに送り返します。

pg_receivewalのレプリケーションクライアントが同期スタンバイとしてサーバ上で構成されている場合、フィードバックが遅延なくサーバに送り返されることを確実にするため、このオプションを指定すべきです。

-v
--verbose

冗長モードを有効にします。

-Z level
--compress=level

先行書き込みログのgzip圧縮を有効にし、圧縮レベルを指定します（0から9まで、0は圧縮なし、9が最大圧縮）。すべてのファイル名に、拡張子.gzが自動的に追加されます。

以下のコマンドラインオプションはデータベース接続パラメータを制御します。

-d connstr
--dbname=connstr

サーバに接続するために使用するパラメータを、接続文字列として指定します。これは衝突するコマンドラインオプションよりも優先します。

このオプションは他のクライアントアプリケーションとの整合性のために--dbnameと呼ばれます。しかし、pg_receivewalはクラスタ内のどの特定のデータベースにも接続しませんので、接続文字列内のデータベース名は無視されます。

-h host
--host=host

サーバが稼働しているマシンのホスト名を指定します。名前がスラッシュから始まる場合、Unixドメインソケット用のディレクトリとして使用されます。デフォルトは環境変数PGHOSTが設定されていればその値から取られ、設定されていない場合はUnixドメインソケット接続が試行されます。

-p port
--port=port

サーバが接続を待ち受けるTCPポートまたはUnixドメインソケットファイルの拡張子を指定します。デフォルトは環境変数PGPORTが指定されていればその値から取られ、設定されていない場合はコンパイル時のデフォルト値から取られます。

-U username
--username=username

接続するユーザ名です。

-w
--no-password

パスワード入力を促しません。サーバがパスワード認証を必要とし、かつ、.pgpassファイルなど他の手段でパスワードが入手できない場合、接続試行は失敗します。このオプションは、パスワードを入力するユーザが存在しないバッチジョブやスクリプトで有用になります。

-W
--password

pg_receivewalはデータベースに接続する前にパスワード入力を強制的に促します。

このオプションは重要ではありません。pg_receivewalは、サーバがパスワード認証を要求した場合に自動的にパスワードを促すためです。しかしpg_receivewalは、サーバがパスワードを要求するかどうかを確認するために接続試行を浪費します。-Wと入力して無駄な接続試行を防止することが有意である場合があります。

pg_receivewalは物理的なレプリケーションスロットを制御するため、以下の2つの動作のうちの1つを実行できます。

--create-slot

--slotで指定された名前の新しい物理的なレプリケーションスロットを作成して終了します。

--drop-slot

--slotで指定された名前の物理的なレプリケーションスロットを削除して終了します。

この他に以下のオプションも使用することができます。

-V
--version

pg_receivewalのバージョンを表示し、終了します。

-?
--help

pg_receivewalコマンドライン引数についてのヘルプを表示し、終了します。

終了ステータス

pg_receivewalはSIGINTシグナルで終了されたとき、ステータス0で終了します。(これは止めるための通常の方法です。そのためエラーではありません。) 致命的エラーや他のシグナルに対しては、終了ステータスは非ゼロになります。

環境

他のほとんどのPostgreSQLユーティリティと同様、このユーティリティはlibpqでサポートされる環境変数(33.14参照)を使用します。

環境変数PG_COLORは診断メッセージで色を使うかどうかを指定します。可能な値はalways、auto、neverです。

注釈

archive_commandの代わりにpg_receivewalをWALのバックアップのメインの方法として使用する場合、レプリケーションスロットを使用することを強く推奨します。そうしなければ、サーバはarchive_commandと

レプリケーションスロットのいずれからもWALのストリームがどこまでアーカイブされているかの情報を得られないため、先行書き込みログファイルがバックアップされる前にそれを再利用または削除するかもしれないのです。しかし、WALデータを受け取る側がそのフェッチに追いつけない場合、レプリケーションスロットがサーバのディスクスペースを一杯にしてしまうかもしれないことに注意してください。

pg_receivewalは、ソースクラスタでグループパーミッションが有効である場合、受け取ったWALファイルのグループパーミッションを維持します。

例

先行書き込みログをmydbserverにあるサーバからストリームし、それをローカルディレクトリ/usr/local/pgsql/archiveに格納します。

```
$ pg_receivewal -h mydbserver -D /usr/local/pgsql/archive
```

関連項目

pg_basebackup

pg_recvlogical

pg_recvlogical — PostgreSQLのストリームの論理デコードを制御する

概要

pg_recvlogical [option...]

説明

pg_recvlogicalはレプリケーションスロットの論理デコードを制御し、またレプリケーションスロットからデータを流します。

これはレプリケーションモードの接続をするため、pg_receivewalと同じ制約に加えて、論理レプリケーション(第48章を参照)と同じ制約も受けます。

pg_recvlogicalには、論理デコードのSQLインタフェースのpeekとgetのモードに対応するものがありません。データを受信する度、および終了時にダラダラとその再生確認を送信します。スロット上の未処理のデータを処理せずに検査するには、pg_logical_slot_peek_changesを使用してください。

オプション

動作を選択するため、以下のオプションのうち少なくとも1つを指定しなければなりません。

--create-slot

新しい論理レプリケーションスロットを--slotで指定した名前で、--pluginの出力プラグインを使い、--dbnameで指定したデータベースに対して作成します。

--drop-slot

--slotで指定された名前のレプリケーションスロットを削除して、終了します。

--start

--slotで指定した論理レプリケーションスロットからストリームの変更を開始し、シグナルを受けて終了するまで継続します。サーバ側の変更ストリームがサーバのシャットダウンまたは接続断によって終了した場合は、--no-loopが指定されていなければ、ループ内でリトライします。

ストリームのフォーマットは、スロットが作成された時に指定された出力プラグインによって決定されます。

接続はスロットの作成時に使用したのと同じデータベースに対してでなければなりません。

--create-slotと--startは同時に指定することができます。--drop-slotは他の動作と組み合わせることができません。

以下のコマンド行オプションは出力の場所とフォーマット、およびその他のレプリケーションの動作を制御します。

-E lsn
--endpos=lsn

--startモードでは、自動的にレプリケーションを停止し、受信が指定のLSNに到達したら正常な終了ステータス0で終了します。--start以外のモードの時に指定された場合は、エラーが発生します。

LSNがlsnと正確に一致するレコードがある場合、そのレコードは出力されます。

--endposオプションはトランザクションの境界を意識しないため、トランザクションの途中で出力を切り捨てるかもしれません。部分的に出力されたトランザクションはいずれも処理されず、スロットが次回、読み込まれた時に再び再生されます。個々のメッセージが切り捨てられることはありません。

-f filename
--file=filename

受け取り、デコードしたトランザクションデータをこのファイルに書き込みます。stdoutに出力するには-を使います。

-F interval_seconds
--fsync-interval=interval_seconds

出力ファイルがディスクに安全にフラッシュされることを確実にするため、pg_recvlogicalがfsync()の呼び出しを実行する頻度を指定します。

サーバはクライアントに対して、フラッシュを実行し、またフラッシュ位置をサーバに報告するように、ときどき要求します。この設定はそれに加えて、フラッシュをより高頻度で実行するものです。

0という間隔を指定すると、fsync()の呼び出しをまったく実行しなくなりますが、それでも状況をサーバに報告はします。この場合、クラッシュするとデータが失われるかもしれません。

-I lsn
--startpos=lsn

--startモードでは、レプリケーションを指定のLSNから開始します。この効果の詳細については第48章および52.4を参照してください。その他のモードでは無視されます。

--if-not-exists

--create-slotが指定され、指定された名前のスロットが既に存在している場合に、エラーを発生させません。

-n
--no-loop

サーバへの接続が失われたとき、ループ内でリトライせず、単に終了します。

-o name[=value]
--option=name[=value]

オプションnameと(指定されていれば)オプション値valueを出力プラグインに渡します。存在するオプションとその効果は、利用する出力プラグインに依存します。

-P plugin
--plugin=plugin

スロットを作成するとき、指定された論理デコードの出力プラグインを使います。第48章を参照してください。このオプションは、スロットが既に存在する時は、何の効果もありません。

-s interval_seconds
--status-interval=interval_seconds

このオプションはpg_receivewalの同じ名前のオプションと同じ効果があります。そちらの説明を参照してください。

-S slot_name
--slot=slot_name

--startモードでは、slot_nameという名前の既存の論理レプリケーションスロットを使います。--create-slotでは、この名前のスロットを作成します。--drop-slotモードでは、この名前のスロットを削除します。

-v
--verbose

冗長モードを有効にします。

以下のコマンド行オプションはデータベース接続パラメータを制御します。

-d dbname
--dbname=dbname

接続するデータベースです。この意味の詳細は動作の説明を参照してください。dbnameは接続文字列でも構いません。その場合、接続文字列パラメータは衝突するコマンドラインオプションよりも優先します。デフォルトはユーザ名です。

-h hostname-or-ip
--host=hostname-or-ip

サーバが動作しているマシンのホスト名を指定します。値がスラッシュで始まるときは、Unixドメインソケットのディレクトリとみなされます。デフォルト値は、環境変数PGHOSTが設定されていればそれが使用され、設定されていなければUnixドメインソケット接続を試みます。

-p port
--port=port

サーバが接続を監視しているTCPポート番号またはローカルのUnixドメインソケットのファイル拡張子を指定します。デフォルトは環境変数PGPORTが設定されていればそれを使用し、そうでなければコンパイル時に設定されたデフォルト値です。

-U user
--username=user

接続で使用するユーザ名です。デフォルトは現在のOSのユーザ名です。

-w
--no-password

パスワード入力を促しません。サーバがパスワード認証を必要とし、.pgpassファイルなど他の手段によるパスワードが利用できない場合は、接続試行は失敗します。このオプションはバッチジョブやスクリプトなど、パスワードを入力するユーザがいない場合に有用でしょう。

-W
--password

pg_recvlogicalがデータベースに接続する前に、強制的にパスワード入力を促します。

サーバがパスワード認証を要求する場合、pg_recvlogicalは自動的にパスワード入力を促しますから、このオプションは本質的ではありません。しかし、サーバがパスワードを必要とすることを確認するためにpg_recvlogicalが無駄な接続試行を行うことになります。無駄な接続試行を避けるために-Wを指定するのが有効になる場合もあるでしょう。

この他に、以下のオプションが利用できます。

-V
--version

pg_recvlogicalのバージョンを出力して、終了します。

-?
--help

pg_recvlogicalのコマンド行引数に関するヘルプを表示して、終了します。

環境

このユーティリティは、他のほとんどのPostgreSQLユーティリティと同様、libpqでサポートされる環境変数を使用します。(33.14を参照してください)。

環境変数PG_COLORは診断メッセージで色を使うかどうかを指定します。可能な値はalways、auto、neverです。

注意

pg_recvlogicalは、ソースクラスタでグループパーミッションが有効である場合、受け取ったWALファイルのグループパーミッションを維持します。

例

例については48.1を参照してください。

関連項目

pg_receivewal

pg_restore

pg_restore — pg_dumpによって作成されたアーカイブファイルからPostgreSQLデータベースをリストアする

概要

pg_restore [connection-option...] [option...] [filename]

説明

pg_restoreは、pg_dumpによって作成された平文形式以外のアーカイブファイルを使って、PostgreSQLデータベースをリストアするためのユーティリティです。このコマンドは、データベースを再構成して保存された時点の状態にするために必要なコマンドを発行します。また、pg_restoreは、アーカイブファイルから、リストアする内容を選択したり、リストアする前にアイテムの並び替えを行うこともできます。アーカイブファイルはアーキテクチャに依存しない移植性を持つように設計されています。

pg_restoreの操作には2つのモードがあります。データベース名が指定された場合、pg_restoreはそのデータベースに接続し、アーカイブを直接そのデータベースにリストアします。データベース名が指定されなかった場合は、データベースを再構築するために必要となるSQLコマンドが含まれたスクリプトが作成されます（ファイルもしくは標準出力に書き出されます）。このスクリプトの出力はpg_dumpの平文形式の出力と同じです。従って、出力を制御するオプションの中には、pg_dumpのオプションに類似したものがあります。

当然ながら、pg_restoreによって、アーカイブファイルに存在しない情報をリストアすることはできません。例えば、アーカイブが「INSERTコマンドの形式でデータダンプ」を行うオプションを使用して作成されたものであった場合、pg_restoreはCOPY文を使用してデータを読み込むことはできません。

オプション

pg_restoreは以下のコマンドライン引数を受け付けます。

filename

リストアするアーカイブファイル（ディレクトリ書式アーカイブの場合はディレクトリ）の場所を指定します。指定がない場合は、標準入力が使用されます。

-a
--data-only

データのみをリストアし、スキーマ（データ定義）はリストアしません。アーカイブ内にある、テーブルデータ、ラージオブジェクト、シーケンス値がリストアされます。

このオプションは--section=dataを指定することと似ていますが、歴史的な理由により同一ではありません。

-c
--clean

再作成前にデータベースオブジェクトを整理(削除)します。(--if-existsが使われなければ、対象のデータベースの中にオブジェクトがない場合に、害のないメッセージをいくつか出力するかもしれません。)

-C
--create

リストア前にデータベースを作成します。--cleanも同時に指定されている場合、接続する前に対象データベースを削除し再作成します。

--createでは、pg_restoreは、もしあるならデータベースのコメントもリストアします。また、あらゆる設定変数の当データベースに対する設定、すなわち、このデータベースを対象にしたALTER DATABASE ... SET ...とALTER ROLE ... IN DATABASE ... SET ...コマンドもリストアします。--no-aclが指定されていない限り、データベース自体に対するアクセス権限もリストアされます。

このオプションがある場合、-dで指定したデータベースは最初のDROP DATABASEとCREATE DATABASEコマンドの発行時にのみ使用されます。そして、すべてのデータはアーカイブ内に記述された名前のデータベースにリストアされます。

-d dbname
--dbname=dbname

dbnameデータベースに接続し、このデータベースに直接リストアします。dbnameは接続文字列でも構いません。その場合、接続文字列パラメータは衝突するコマンドラインオプションよりも優先します。

-e
--exit-on-error

データベースにSQLコマンドを送信中にエラーが発生した場合、処理を終了します。デフォルトでは、処理を続行し、リストア処理の最後に発生したエラーの数を表示します。

-f filename
--file=filename

作成するスクリプト(-lを使用した場合はアーカイブの一覧)の出力ファイルを指定します。stdout(標準出力)に出力するには-を使ってください。

-F format
--format=format

アーカイブの形式を指定します。pg_restoreは形式を自動認識するので、このオプションは必須ではありません。指定する値は以下のいずれかになります。

c
custom

アーカイブがpg_dumpのカスタム形式であることを表します。

d
directory

アーカイブがディレクトリアーカイブであることを表します。

t
tar

アーカイブがtarアーカイブであることを表します。

-I index
--index=index

指定したインデックスの定義のみをリストアします。複数の-Iスイッチをつけることで複数のインデックスを指定できます。

-j number-of-jobs
--jobs=number-of-jobs

pg_restoreのもっとも時間がかかる部分、つまり、データのロード、インデックスの作成、制約の作成部分を最大number-of-jobsの並行セッションを使用して実行します。このオプションは、複数プロセッサマシンで稼働するサーバに大規模なデータベースをリストアする時間を劇的に減らすことができます。データベースサーバに直接接続するのではなくスクリプトを生成する場合には、このオプションは無視されます。

各ジョブは1プロセスまたは1スレッド(オペレーティングシステムに依存)です。各ジョブはサーバへの別々の接続を使用します。

このオプションの最適値はサーバ、クライアント、ネットワークのハードウェア構成に依存します。要素にはCPUコア数やディスク構成も含まれます。試行する最初の値としてサーバのCPUコア数を勧めます。しかし、多くの場合これより大きな値でもリストア時間を高速化することができます。当然ながらあまりに大きな値を使用すると、スラッシングのために性能が劣化することになります。

カスタムアーカイブ書式およびディレクトリアーカイブ書式のみがこのオプションをサポートします。入力ファイルは通常のファイルまたはディレクトリでなければなりません(例えばパイプや標準入力はいけません)。また、複数ジョブは--single-transactionオプションといっしょに使用することはできません。

-l
--list

アーカイブの内容を一覧表として出力します。このコマンドが出力する一覧は、-Lオプションに対する入力として使用することができます。-nや-tなどのフィルタオプションを-lといっしょに使用すると、一覧出力する項目が制限されます。

-L list-file
--use-list=list-file

list-file内で指定したアーカイブ要素のみをリストアします。また、それらはそのファイルの出現順にリストアされます。-nや-tなどのフィルタオプションを-Lといっしょに使用すると、リストアする項目がさらに制限されます。

list-fileは通常、事前に行った-l操作の出力を編集して作成されます。行の移動や削除、または、行の先頭にセミコロン(;)を付けてコメントアウトすることが可能です。後述の例を参照してください。

-n schema
--schema=schema

指定されたスキーマ内のオブジェクトのみをリストアします。複数の-nスイッチをつけることで複数のスキーマを指定できます。これは特定のテーブルのみをリストアするために-tオプションと組み合わせることができます。

-N schema
--exclude-schema=schema

指定したスキーマ内にあるオブジェクトをリストアしません。-Nオプションを複数回指定することで、複数のスキーマを除外することができます。

同じスキーマ名が-nと-Nの両方で指定された場合は、-Nオプションが優先し、そのスキーマは除外されます。

-O
--no-owner

オブジェクトの所有者を元のデータベースに合わせるためのコマンドを出力しません。デフォルトでは、pg_restoreは、ALTER OWNERまたはSET SESSION AUTHORIZATIONを発行して、作成したスキーマ要素の所有者を設定します。データベースに最初に接続したのがスーパーユーザ(もしくは、そのスクリプト内の全てのオブジェクトを所有するユーザ)でない場合、これらの文は失敗します。-Oを付与すると、初期接続に任意のユーザ名を使用できるようになります。ただし、この場合は、全てのオブジェクトの所有者がリストアしたユーザになります。

-P function-name(argtype [, ...])
--function=function-name(argtype [, ...])

指定した関数のみをリストアします。関数や引数の名前は、ダンプファイルの一覧で出力される通りのスペルで正確に入力するよう注意してください。複数の-Pスイッチをつけることで複数の関数を指定できます。

-R
--no-reconnect

このオプションは廃止されました。後方互換性を保持するために受け入れられています。

-s
--schema-only

アーカイブ内にあるスキーマ項目の範囲でスキーマ(データ定義)のみをリストアし、データ(テーブルの内容)をリストアしません。

このオプションは--data-onlyの逆です。このオプションは--section=pre-data --section=post-dataを指定することと似ていますが、歴史的な理由により同一ではありません。

(これと--schemaオプションと混同しないでください。「schema」という単語を異なる意味で使用しています。)

-S username
--superuser=username

トリガを無効にする場合に使用する、スーパーユーザのユーザ名を指定します。これは--disable-triggersを使う場合にのみ使用されます。

-t table
--table=table

指定されたテーブルのみについて、定義、データまたはその両方をリストアします。この目的において、「テーブル」にはビュー、マテリアライズドビュー、シーケンス、外部テーブルが含まれます。複数の-tスイッチを指定することで複数のテーブルを指定することができます。-nオプションと組み合わせることでスキーマを指定することができます。

> **注記**
>
> -tが指定された場合、pg_restoreは選択されたテーブルが依存するその他のデータベースオブジェクトについてリストアしようとはしません。そのため、初期化されたデータベースに特定のテーブルをリストアすることが成功する保証はありません。

> **注記**
>
> このフラグはpg_dumpの-tフラグと同じ動作をするわけではありません。現在のところ、pg_restoreでワイルドカードマッチを提供する予定はありませんし、-tでスキーマ名を含めることもできません。加えて、pg_dumpの-tフラグは選択されたテーブルの（インデックスなどの）従属オブジェクトもダンプしますが、pg_restoreの-tフラグではそのような従属オブジェクトを含めません。

> **注記**
>
> PostgreSQLの9.6より前のバージョンでは、このフラグはテーブルにのみマッチし、その他の種類のリレーションとはマッチしませんでした。

-T trigger
--trigger=trigger

指定されたトリガだけをリストアします。複数の-Tスイッチをつけることで、複数のトリガを指定できます。

-v
--verbose

冗長モードを指定します。

-V
--version

pg_restoreのバージョンを表示し、終了します。

-x
--no-privileges
--no-acl

アクセス権限（grant/revokeコマンド）のリストアを行いません。

-1
--single-transaction

リストアを単一トランザクションとして実行します（つまり発行するコマンドをBEGIN/COMMITで囲みます）。これにより確実に、すべてのコマンドが完全に成功するか、まったく変更がなされないかのどちらかになります。このオプションは--exit-on-errorを意味します。

--disable-triggers

このオプションは、データのみのダンプからリストアする際にしか適用されません。データの再ロード中、pg_restoreに対し、対象テーブル上のトリガを一時的に無効にするコマンドを実行するよう指示します。このオプションは、データの再ロード中には呼び出したくない参照整合性検査やその他のトリガがある場合に使用します。

現在のところ、--disable-triggersが生成するコマンドを実行するのは、スーパーユーザでなければなりません。そのため、-Sでスーパーユーザの名前を指定するか、あるいは、可能であれば、PostgreSQLのスーパーユーザ権限でpg_restoreを実行する必要があります。

--enable-row-security

このオプションは、行セキュリティのあるテーブルの内容をリストアするときにのみ意味を持ちます。デフォルトではpg_restoreはrow_securityをoffに設定し、すべてのデータが確実にテーブルにリストアされるようにします。ユーザが行セキュリティを回避できるだけの十分な権限がないときはエラーが発生します。このパラメータはpg_restoreがrow_securityをonに設定するようにし、ユーザが行セキュリティが有効なテーブルの内容をリストアできるようにします。それでも、ユーザがダンプからテーブルに行を挿入する権限を持っていなければ、これは失敗します。

COPY FROMは行セキュリティをサポートしないので、このオプションは今のところ、ダンプがINSERT形式である必要があることに注意してください。

--if-exists

データベースオブジェクトを削除するときに、条件コマンドを使います（つまりIF EXISTS句を追加します）。このオプションは、--cleanも指定されているのでなければ有効にはなりません。

--no-comments

たとえアーカイブにコメントが含まれていても、コメントをリストアするコマンドを出力しません。

--no-data-for-failed-tables

デフォルトでは、関連するテーブルの作成に失敗した（たとえば、既に存在するなどの理由により）としてもテーブルデータオブジェクトはリストアされます。このオプションにより、こうしたテーブルデータは単に無視されるようになります。これは対象のデータベースに目的のテーブルの中身が含まれている時に便利です。たとえばPostGISなどのPostgreSQL拡張用の補助テーブルが既に対象のデータベース内に存在する可能性があります。このオプションを指定すれば、二重ロードや古いデータのロードを防ぐことができます。

このオプションは直接データベースにリストアする時にのみ有効で、SQLスクリプト出力を生成する時は無効です。

--no-publications

アーカイブにパブリケーションが含まれていたとしても、それをリストアするコマンドを出力しません。

--no-security-labels

アーカイブにセキュリティラベルが含まれている場合であっても、セキュリティラベルを戻すコマンドを出力しません。

--no-subscriptions

アーカイブにサブスクリプションが含まれていたとしても、それをリストアするコマンドを出力しません。

--no-tablespaces

テーブル空間を選択するコマンドを出力しません。このオプションを付けると、すべてのオブジェクトはリストア時にデフォルトとなっているテーブル空間内に作成されます。

--section=sectionname

指定された部分のみをリストアします。部分名はpre-data、data、post-dataのいずれかを取ることができます。複数の部分を選択するために、このオプションを複数指定することができます。デフォルトではすべての部分をリストアします。

data部分には、実際のテーブルデータやラージオブジェクト定義が含まれます。post-data項目は、インデックス定義、トリガ定義、ルール定義、有効化された検査制約以外の制約定義から構成されます。pre-data項目は、他のすべてのデータ定義項目から構成されます。

--strict-names

スキーマ指定(-n/--schema)およびテーブル指定(-t/--table)がバックアップファイル内の少なくとも1つのスキーマあるいはテーブルにマッチすることを必要とします。

--use-set-session-authorization

ALTER OWNERコマンドの代わりに、標準SQLのSET SESSION AUTHORIZATIONコマンドを出力して、オブジェクトの所有権を決定します。これにより、ダンプの標準への互換性が高まりますが、ダンプ内のオブジェクトの履歴によっては正しくリストアされない可能性が生じます。

-?
--help

pg_restoreコマンドライン引数の使用方法を表示し、終了します。

pg_restoreはさらに以下のコマンドライン引数を接続パラメータとして受け付けます。

-h host
--host=host

サーバが稼働しているマシンのホスト名を指定します。この値がスラッシュから始まる場合、Unixドメインソケット用のディレクトリとして使用されます。デフォルトは、設定されていれば環境変数PGHOSTから取得されます。設定されていなければ、Unixドメインソケット接続とみなされます。

-p port
--port=port

サーバが接続を監視するTCPポートもしくはローカルUnixドメインソケットファイルの拡張子を指定します。デフォルトは、設定されている場合、環境変数PGPORTの値となります。設定されていなければ、コンパイル時のデフォルト値となります。

-U username
--username=username

接続ユーザ名です。

-w
--no-password

パスワードの入力を促しません。サーバがパスワード認証を必要とし、かつ、.pgpassファイルなどの他の方法が利用できない場合、接続試行は失敗します。バッチジョブやスクリプトなどパスワードを入力するユーザが存在しない場合にこのオプションは有用かもしれません。

-W
--password

データベースに接続する前に、pg_restoreは強制的にパスワード入力を促します。

サーバがパスワード認証を要求する場合pg_restoreは自動的にパスワード入力を促しますので、これが重要になることはありません。しかし、pg_restoreは、サーバにパスワードが必要かどうかを判断するための接続試行を無駄に行います。こうした余計な接続試行を防ぐために-Wの入力が有意となる場合もあります。

--role=rolename

リストアを実行する際に使用するロール名を指定します。このオプションによりpg_restoreはデータベースに接続した後にSET ROLE rolenameコマンドを発行するようになります。認証に使用したユーザ（-Uで指定されたユーザ）がpg_restoreで必要とされる権限を持たないが、必要な権限を持つロールに切り替えることができる場合に有用です。一部のインストレーションではスーパーユーザとして直接ログインさせないポリシーを取ることがありますが、このオプションを使用することでポリシーに反することなくリストアを行うことができます。

環境

PGHOST
PGOPTIONS
PGPORT
PGUSER

デフォルトの接続パラメータです。

PG_COLOR

診断メッセージで色を使うかどうかを指定します。可能な値はalways、auto、neverです。

また、このユーティリティは、他のほとんどのPostgreSQLユーティリティと同様、libpqでサポートされる環境変数を使用します（33.14を参照してください）。しかしデータベース名が指定されていない場合はPGDATABASEは読み取られません。

診断

-dオプションによってデータベースに直接接続するよう指定されている場合、pg_restoreは内部でSQL文を実行します。pg_restoreの実行時に問題が発生する場合は、psqlなどを使用して、そのデータベースから情報を選択できることを確認してください。また、libpqフロントエンドライブラリで使用されるデフォルト接続設定や環境変数もすべて適用されます。

注釈

template1データベースに対し独自の変更を行っている場合、pg_restoreの出力は、確実に空のデータベースにロードするよう注意してください。そうしないと、おそらく追加されたオブジェクトの重複定義によってエラーが発生します。独自の追加が反映されていない空のデータベースを作成するには、template1ではなくtemplate0をコピーしてください。以下に例を示します。

```
CREATE DATABASE foo WITH TEMPLATE template0;
```

pg_restoreの制限を以下に示します。

- 既存のテーブルにデータをリストアする際に--disable-triggersオプションを使用すると、pg_restoreは、データを挿入する前に、ユーザテーブル上のトリガを無効にするコマンドを発行し、データの挿入が完了した後で、それらを再び有効にする問い合わせを発行します。リストアが途中で停止した場合、システムカタログが不適切な状態のままになっている可能性があります。
- pg_restoreは特定のテーブルのみのラージオブジェクトなどといった、ラージオブジェクトを選択してリストアすることはできません。アーカイブにラージオブジェクトが含まれる場合、すべてのラージオブジェクトをリストアします。もし-L、-tなどのオプションで除外が指定されていた場合は、全くリストアしません。

pg_dumpの制限についての詳細は、pg_dumpの文書も参照してください。

リストア後は、オプティマイザが有用な統計情報を持つように、リストアしたテーブルそれぞれに対してANALYZEを実行することをお勧めします。詳しくは24.1.3および24.1.6を参照してください。

例

mydbという名前のデータベースをカスタム形式のダンプファイルにダンプしているものと仮定します。

```
$ pg_dump -Fc mydb > db.dump
```

データベースを削除し、ダンプファイルから再作成します。

```
$ dropdb mydb
$ pg_restore -C -d postgres db.dump
```

-dオプションのデータベース名は、クラスタに存在する任意のデータベースで良いです。pg_restoreは、mydbに対するCREATE DATABASEコマンドを発行するためだけに、このデータベース名を使用します。-Cを付けると、データは常にダンプファイル内に記載された名前のデータベースにリストアされます。

newdbという新しいデータベースにダンプファイルをリストアします。

```
$ createdb -T template0 newdb
$ pg_restore -d newdb db.dump
```

-Cを使用していないことに注意してください。代わりにリストアするデータベースに直接接続しています。また、新しいデータベースをtemplate1ではなくtemplate0からコピーして作成している点にも注意してください。確実に初期状態を空にするためです。

データベースのアイテムを並び換えるには、まずこのアーカイブの内容の一覧をダンプしなければなりません。

```
$ pg_restore -l db.dump > db.list
```

一覧ファイルは、ヘッダと各アイテムを1行で表したものから構成されます。

```
;
; Archive created at Mon Sep 14 13:55:39 2009
;     dbname: DBDEMOS
;     TOC Entries: 81
;     Compression: 9
;     Dump Version: 1.10-0
;     Format: CUSTOM
;     Integer: 4 bytes
;     Offset: 8 bytes
;     Dumped from database version: 8.3.5
;     Dumped by pg_dump version: 8.3.8
;
;
; Selected TOC Entries:
;
3; 2615 2200 SCHEMA - public pasha
1861; 0 0 COMMENT - SCHEMA public pasha
1862; 0 0 ACL - public pasha
317; 1247 17715 TYPE public composite pasha
319; 1247 25899 DOMAIN public domain0 pasha
```

セミコロンで始まる行はコメントです。行の先頭の番号は、各アイテムに割り当てられた内部アーカイブIDを示します。

このファイルの各行に対して、コメントアウト、削除、並び替えを行うことができます。以下に例を示します。

```
10; 145433 TABLE map_resolutions postgres
;2; 145344 TABLE species postgres
```

```
;4; 145359 TABLE nt_header postgres
6; 145402 TABLE species_records postgres
;8; 145416 TABLE ss_old postgres
```

このファイルをpg_restoreの入力として利用すれば、アイテム10と6だけを、この順番でリストアすることができます。

```
$ pg_restore -L db.list db.dump
```

関連項目

pg_dump, pg_dumpall, psql

pg_verifybackup

pg_verifybackup — PostgreSQLクラスタのベースバックアップの完全性を確認します

概要

pg_verifybackup [option...]

説明

pg_verifybackupは、pg_basebackupを使って取られたデータベースクラスタバックアップの完全性を、バックアップ時にサーバで生成されたbackup_manifestと比較して確認するために使われます。バックアップは"plain"形式で保管されていなければなりません。"tar"形式のバックアップは展開後に確認できます。

pg_verifybackupにより実行される検証は、バックアップを利用しようとしている動作中のサーバにより実行される検査をすべて含む訳ではありませんし、含むことができる訳でもないことに注意するのは重要です。このツールを使ったとしても、テストリストアを実行したり、結果のデータベースが期待した通りに動作し、正しいデータを含んでいるように見えることを検証したりすべきです。しかしながら、pg_verifybackupは、ストレージの問題や利用者のエラーによる多くの問題を検出できます。

バックアップの検証は4つの段階を経て進みます。第1に、pg_verifybackupがbackup_manifestファイルを読みます。そのファイルが存在しない、読むことができない、不正である、または、自身の内部チェックサムの検証に失敗した場合には、pg_verifybackupは致命的なエラーで終了します。

第2に、pg_verifybackupは、以下に記すようないくつかの例外を除いて、ディスクに保管されているデータファイルがサーバの送ろうと意図したデータファイルと完全に同一かを検証します。2、3の例外では、余分な、もしくは、失われたファイルが検出されます。この段階ではpostgresql.auto.conf、standby.signal、recovery.signalの存在、欠如、修正は無視されます。なぜなら、バックアップを取る過程の一部として、このファイルは作成されていたり、修正されていたりすることが予想されるからです。また、たとえそのファイルがバックアップマニフェストの一覧に載っていたとしても、対象ディレクトリ内のbackup_manifestファイルやpg_wal内のものについて問題視しません。ファイルだけが検査されます。ディレクトリの存在や欠如は、ディレクトリがなければ、そこに含まれるはずのファイルも必ずないという間接的なものを除き検証されません。

次に、pg_verifybackupは、すべてのファイルのチェックサムを取り、そのチェックサムをマニフェスト内の値と比較し、計算されたチェックサムがマニフェストに保管されたチェックサムと一致しないファイルに対してエラーを出力します。この段階は、前の段階でエラーとなったファイルに対しては実行されません。既に問題があると分かっているからです。前の段階で無視されたファイルは、この段階でも無視されます。

最後にpg_verifybackupは、マニフェストを使って、バックアップを回復するのに必要な先行書き込みログレコードが存在し、それが読み込めて解析できるかを検証します。backup_manifestは、必要となる先行書き込みログレコードに関する情報を含んでおり、pg_verifybackupは、その情報を使って、先行書き込みログレコードを解析するようpg_waldumpを呼び出します。pg_waldumpがエラーだけを報告し、それ以外の出力をしないよう--quietフラグが使われます。この水準の検証は、存在しないファイルや内部チェックサムが一致しないなどの明らかな問題を検出するには十分ですが、回復しようとする時に起こりうる問題をすべて検出す

るほど広範囲に十分なものではありません。例えば、正しいチェックサムを持つものの馬鹿げた動作を指定する先行書き込みログレコードを生成するサーバのバグは、この方法では検出できません。

バックアップからの回復に必要でない余分なWALが存在すると、それはこのツールでは検査されないことに注意してください。その目的のためにpg_waldumpを別に呼び出して使うことはできます。また、WALの検証はバージョン固有のものだということにも注意してください。検査するバックアップに付属したバージョンのpg_verifybackupと、それゆえ、pg_waldumpも使わないとなりません。対照的に、データファイルの完全性の検査は、backup_manifestファイルを生成したサーバのバージョンが何であれ動作します。

オプション

pg_verifybackupは以下のコマンドライン引数を受け付けます。

-e
--exit-on-error

バックアップで問題が検出され次第、終了します。このオプションが指定されていなければ、pg_verifybackupは問題が検出された後もバックアップの検査を続け、検出した問題をすべてエラーとして報告します。

-i path
--ignore=path

バックアップ内に実際に存在するデータファイルの一覧とbackup_manifestファイル内の一覧を比較する時に、指定されたファイルやディレクトリを無視します。相対パス名で指定してください。ディレクトリが指定された場合、このオプションは、その位置をルートとするサブツリー全体に影響します。相対パス名が指定されたパス名に一致する場合、余分なファイル、足りないファイル、ファイルサイズの違い、チェックサムの不一致の報告は抑制されます。このオプションは複数回指定できます。

-m path
--manifest-path=path

バックアップディレクトリのルートにあるものではなく、指定されたパスのマニフェストファイルを使用します。

-n
--no-parse-wal

このバックアップからの回復に必要な先行書き込みログデータを解析しません。

-q
--quiet

バックアップの検証に成功した場合、何も表示しません。

-s
--skip-checksums

データファイルのチェックサムを検証しません。ファイルの存在、欠如とファイルのサイズは検査されます。ファイル自身を読み込む必要がありませんので、これはずっと速いです。

-w path
--wal-directory=path

pg_walではなく、指定されたディレクトリのWALファイルを解析しようとします。バックアップがWALアーカイブとは別の場所に保管されている場合、これは有用でしょう。

他のオプションも使用可能です。

-V
--version

pg_verifybackupのバージョンを表示し、終了します。

-?
--help

pg_verifybackupのコマンドライン引数に関するヘルプを表示し、終了します。

例

mydbserverでサーバのバックアップを作成し、バックアップの完全性を検証します。

```
$ pg_basebackup -h mydbserver -D /usr/local/pgsql/data
$ pg_verifybackup /usr/local/pgsql/data
```

mydbserverでサーバのバックアップを作成し、マニフェストをバックアップディレクトリの外のどこかに移動し、バックアップを検証します。

```
$ pg_basebackup -h mydbserver -D /usr/local/pgsql/backup1234
$ mv /usr/local/pgsql/backup1234/backup_manifest /my/secure/location/backup_manifest.1234
$ pg_verifybackup -m /my/secure/location/backup_manifest.1234 /usr/local/pgsql/backup1234
```

バックアップディレクトリに手で追加されたファイルを無視し、チェックサムの検証も省略してバックアップを検証します。

```
$ pg_basebackup -h mydbserver -D /usr/local/pgsql/data
$ edit /usr/local/pgsql/data/note.to.self
$ pg_verifybackup --ignore=note.to.self --skip-checksums /usr/local/pgsql/data
```

関連項目

pg_basebackup

psql

psql — PostgreSQLの対話的ターミナル

概要

psql [option...] [dbname [username]]

説明

psqlとはPostgreSQLのターミナル型フロントエンドです。対話的に問い合わせを入力し、それをPostgreSQLに対して発行して、結果を確認することができます。また、ファイルから入力を読み込むことも可能です。さらに、スクリプトの記述を簡便化したり、様々なタスクを自動化したりする、いくつものメタコマンドとシェルに似た各種の機能を備えています。

オプション

-a
--echo-all

読み込んだ全ての空でない入力行を標準出力に表示します。(これは対話式に読み込まれる行には適用されません。) これはECHO変数をallに設定するのと同じ意味を持ちます。

-A
--no-align

位置揃えなしの出力モードに切り替えます。(デフォルトの出力モードはaligned(位置揃えあり)です。) これは\pset format unalignedと同等です。

-b
--echo-errors

失敗したSQLコマンドを標準エラー出力に出力します。これはECHO変数をerrorsに設定するのと同等です。

-c command
--command=command

psqlに対し、指定のコマンド文字列commandを実行するよう指示します。このオプションは繰り返すことができ、また-fオプションと任意の順序で組み合わせることができます。-cまたは-fが指定されると、psqlは標準入力からコマンドを読み取りません。その代わりに、すべての-cオプションおよび-fオプションを順に処理した後、終了します。

commandは、サーバで完全に解析可能な(つまり、psql特有の機能は含まない)コマンド文字列、もしくは、バックスラッシュコマンド1つである必要があります。このため、-cオプション内ではSQLとpsqlメタコマンドを混在させることはできません。これらを同時に使用するには、-cオプションを繰り返し利用するか、あるいはパイプを使って文字列をpsqlに渡します。例えば、

```
psql -c '\x' -c 'SELECT * FROM foo;'
```

あるいは

```
echo '\x \\ SELECT * FROM foo;' | psql
```

のようにします(\\はメタコマンドの区切り文字です)。

-cに渡される各SQLのコマンド文字列は、単一の要求としてサーバに送信されます。このため、トランザクションを複数に分けるBEGIN/COMMITコマンドが明示的に文字列内に含まれない限り、文字列内に複数のSQLコマンドが含まれていたとしても、サーバはそれを1つのトランザクションとして実行します。(複数問い合わせの文字列をどのようにサーバが処理するかについて、詳しくは52.2.2.1を参照してください。)また、psqlは文字列内の最後のSQLコマンドの結果しか出力しません。同じ文字列をファイル、あるいはpsqlの標準入力として渡した場合、psqlは各SQLコマンドを別々に送信しますので、この場合とは動作が異なります。

この動作のため、1つの-cの文字列に2つ以上のSQLコマンドを指定すると、予期しない結果をもたらすことがあります。-cのコマンドを繰り返し使うか、あるいは、上記のようにechoを使うか、以下の例のようにシェルのヒアドキュメントを使うことで、psqlの標準入力に複数のコマンドを入力する方が良いでしょう。

```
psql <<EOF
\x
SELECT * FROM foo;
EOF
```

--csv

CSV(コンマ区切り値)出力モードに切り替えます。これは\pset format csvと同等です。

-d dbname
--dbname=dbname

接続するデータベースの名前を指定します。コマンドラインでオプション以外の最初の引数としてdbnameを指定するのと同じ効力を持ちます。dbnameは接続文字列でも構いません。その場合、接続文字列パラメータは衝突するコマンドラインオプションに優先します。

-e
--echo-queries

サーバに送られるすべてのSQLコマンドを標準出力にも送ります。ECHO変数をqueriesに設定するのと同じ効力を持ちます。

-E
--echo-hidden

\dやその他のバックスラッシュコマンドによって生成される実際の問い合わせを表示します。これを使って、psqlの内部動作を調べることができます。これは変数ECHO_HIDDENをonに設定するのと同じ効力を持ちます。

-f filename
--file=filename

標準入力ではなく、ファイルfilenameからコマンドを読み取ります。このオプションは繰り返すことができ、また-cオプションと任意の順序で組み合わせることができます。-cまたは-fが指定されると、psqlは標準入力からコマンドを読み取りません。その代わりに、すべての-cオプションおよび-fオプションを順に処理した後、終了します。その点を除けば、このオプションは\iメタコマンドとほぼ同等です。

filenameに-（ハイフン）を指定すると、標準入力からEOFを示すもの、または\qメタコマンドまで読み取られます。これは対話的入力をファイルからの入力と混在させるために使うことができます。ただし、この場合、Readlineは使われないことに注意してください（-nが指定された場合と同様です）。

このオプションを指定するのと、psql < filenameと入力するのでは、微妙に動作が異なります。一般的には、両者とも期待通りの動作を行いますが、-fを使用した場合は、エラーメッセージに行番号を付けるなどいくつか便利な機能が有効になります。また、このオプションを使用した場合、起動時のオーバーヘッドが減少する可能性が若干あります。一方、シェルの入力リダイレクションを使用する方法では、（理論的には）全て手作業で入力した場合の出力とまったく同一な出力になることが保証されます。

-F separator
--field-separator=separator

separatorを位置揃えを行わない出力におけるフィールド区切り文字として使用します。\pset fieldsepもしくは\fと同じ効力を持ちます。

-h hostname
--host=hostname

サーバを実行しているマシンのホスト名を指定します。この値がスラッシュから始まる場合、Unixドメインソケット用のディレクトリとして使用されます。

-H
--html

HTML出力モードに切り替えます。これは、\pset format htmlもしくは\Hコマンドと同等です。

-l
--list

利用可能な全てのデータベースを一覧表示し、終了します。この他の接続に関連しないオプションは無視されます。\listメタコマンドと似た効力を持ちます。

このオプションが使用されるときは、別のデータベース名がコマンドラインで指定されない限り（オプション-dまたは、環境変数ではない非オプション引数、おそらくサービスエントリーを通じて）、psqlは、postgresデータベースに接続します。

-L filename
--log-file=filename

すべての問い合わせの出力を通常の出力先に出力し、さらにファイルfilenameに書き出します。

-n
--no-readline

行編集用のReadlineを使用しません。またコマンド履歴も使用しません。コピー&ペースト時のタブ展開を無効にするために有用かもしれません。

-o filename
--output=filename

全ての問い合わせの出力をfilenameファイルに書き込みます。これは\oコマンドと同じ効力を持ちます。

-p port
--port=port

サーバが接続監視を行っているTCPポートもしくはローカルUnixドメインソケットファイルの拡張子を指定します。環境変数PGPORTの値、環境変数が設定されていない場合はコンパイル時に指定した値（通常は5432）がデフォルト値となります。

-P assignment
--pset=assignment

\pset形式により表示オプションを指定します。ここでは空白ではなく等号を使って名前と値を区切っていることに注意してください。たとえば、出力形式をLaTeXにする場合、-P format=latexと入力します。

-q
--quiet

psqlがメッセージ出力なしで処理を行うように指示します。デフォルトでは、ウェルカム（welcome）メッセージや各種の情報が表示されますが、このオプションを使用した場合、これらのメッセージが表示されません。-cオプションと併用すると便利です。これは変数QUIETをonに設定するのと同じ効力を持ちます。

-R separator
--record-separator=separator

separatorを位置揃えを行わない出力におけるレコード区切り文字として使用します。これは\pset recordsepと同じです。

-s
--single-step

シングルステップモードで実行します。これは各コマンドがサーバに送信される前に、ユーザに対して実行するかキャンセルするかについて確認を求めることを意味します。スクリプトのデバッグを行う時に使用してください。

-S
--single-line

シングル行モードで実行します。このモードでは、セミコロンと同じように改行もSQLコマンドの終端として扱われます。

> **注記**
>
> このモードはどうしてもこのような方式を使用したいユーザ向けに用意されたもので、必ずしも使用が推奨されるわけではありません。特に、1行にSQLとメタコマンドを混在させる場合、経験の浅いユーザにとってその実行順番は必ずしもわかりやすいものではありません。

-t
--tuples-only

列名と結果の行数フッタなどの表示を無効にします。これは、\tおよび\pset tuples_onlyと同等です。

-T table_options
--table-attr=table_options

HTMLのtableタグで使用されるオプションを指定します。詳細は\pset tableattrを参照してください。

-U username
--username=username

デフォルトのユーザではなくusernameユーザとしてデータベースに接続します（当然、そうする権限を持っていなければなりません）。

-v assignment
--set=assignment
--variable=assignment

\setメタコマンドのように、変数の代入を行います。値がある場合、コマンド行上では、名前と値を等号(=)で区切る必要があることに注意してください。変数を未設定の状態にするには、等号を指定しないでください。値が空の変数を設定するには、値を指定しないで等号のみ使用してください。これらの代入はコマンド行処理の段階で行われます。そのため、接続状態を表す変数は後で上書きされる可能性があります。

-V
--version

psqlのバージョンを表示し、終了します。

-w
--no-password

パスワードの入力を促しません。サーバがパスワード認証を必要とし、かつ、.pgpassファイルなどの他の情報源からパスワードが入手可能でない場合、接続試行は失敗します。バッチジョブやスクリプトなどパスワードを入力するユーザが存在しない場合にこのオプションは有用かもしれません。

このオプションはセッション全体にわたって設定されたままであることに注意してください。このため\connectメタコマンドの使用に関しても初期接続試行と同様に影響します。

-W
--password

パスワードが使われない場合であっても、データベースに接続する前にpsqlは強制的にパスワード入力を促します。

サーバがパスワード認証を必要とし、かつ、.pgpassファイルなどの他の情報源からパスワードが入手可能でない場合、psqlは常にパスワード入力を促します。しかし、psqlは、サーバにパスワードが必要かどうかを判断するための接続試行を無駄に行います。こうした余計な接続試行を防ぐために-Wの入力が有意となる場合もあります。

このオプションはセッション全体に対して設定されたままであることに注意してください。このため初期接続試行と同様に\connectメタコマンドの使用にも影響を与えます。

-x
--expanded

拡張テーブル形式モードを有効にします。これは\xおよび\pset expandedと同じです。

-X,
--no-psqlrc

起動用ファイル(psqlrcファイルおよびユーザ用の~/.psqlrcファイルのどちらも)を読み込みません。

-z
--field-separator-zero

位置揃えを行わない出力用のフィールド区切り文字をゼロバイトに設定します。これは\pset fieldsep_zeroと同じです。

-0
--record-separator-zero

位置揃えを行わない出力用のレコード区切り文字をゼロバイトに設定します。これは例えばxargs -0と連携する時に有用です。これは\pset recordsep_zeroと同じです。

-1
--single-transaction

このオプションは、1つ以上の-cオプションや-fオプションと組み合わせてのみ使うことができます。これによりpsqlは、最初のそのようなオプションの前にBEGINコマンドを発行し、最後のオプションの後にCOMMITコマンドを発行するようになります。そうすることで、全てのコマンドが単一のトランザクションに囲まれます。これによりすべてのコマンドが成功して完了するか、変更がまったく行われないかのいずれかになります。

コマンド自体がBEGIN、COMMIT、ROLLBACKを含んでいる場合、このオプションは期待した効果を得ることができません。また、個別のコマンドがトランザクションブロック内部で実行できない場合、このオプションを指定することで、そのトランザクション全体が失敗します。

-?
--help[=topic]

psqlに関するヘルプを表示し、終了します。オプションのtopicパラメータ(デフォルトはoptions)はpsqlのどの部分を説明するかを選択します。commandsはpsqlのバックスラッシュコマンドについて、optionsはpsqlに渡すことができるコマンド行オプションについて、variablesはpsqlの設定変数についてのヘルプを表示します。

終了ステータス

psqlは、正常に終了した時には0を、psqlにとって致命的なエラー（メモリ不足やファイルが見つからないなど）が発生した時には1を、セッションが対話式でない状態でサーバとの接続が不完全になった時には2を、ON_ERROR_STOP変数が設定されている状態でスクリプトでエラーが発生した時には3をシェルに返します。

使用方法

データベースへの接続

psqlはPostgreSQLの正式なクライアントアプリケーションです。データベースに接続するには、接続するデータベース名、ホスト名、サーバのポート番号、接続する際に使用するユーザ名がわかっていなければなりません。psqlでは、それらをコマンドラインオプションで指定することができます。接続するデータベース名は-d、ホスト名は-h、サーバのポート番号は-p、接続するユーザ名は-Uを使用してそれぞれ指定します。オプションでない引数がある場合、それはデータベース名（データベース名が与えられている場合にはユーザ名）とみなされます。これらのオプションは全て指定されている必要はありません。便利なデフォルト値があります。ホスト名を省略した場合、psqlはUnixドメインソケット経由でローカルホスト上のサーバに、Unixドメインソケットを持たないマシンではlocalhostにTCP/IP経由で接続します。デフォルトのポート番号はコンパイル時に設定されます。データベースサーバは同じデフォルト値を使用するので、多くの場合、ポートは指定する必要はありません。デフォルトのユーザ名とデータベース名は、OSのユーザ名です。任意のユーザ名で全てのデータベースに接続できるわけではありません。データベース管理者は、接続権限をユーザに知らせておかなければなりません。

デフォルトが完全には適用できない時は、入力の手間を省くために、環境変数PGDATABASE、PGHOST、PGPORT、PGUSERに適当な値を設定することもできます。（この他の環境変数については、33.14を参照してください。）また、~/.pgpassファイルを使用すれば、定常的なパスワードの入力を省略でき、便利です。詳細は33.15を参照してください。

他の接続パラメータの指定方法としてconninfo文字列またはURIがあります。これは、データベース名の代わりに使用されます。この機構により、接続全体に関する非常に幅広い制御を行うことができます。以下に例を示します。

```
$ psql "service=myservice sslmode=require"
$ psql postgresql://dbmaster:5433/mydb?sslmode=require
```

この方法では接続パラメータの検索に、33.17で説明するLDAPを使用することもできます。利用できる接続オプションのすべてについての詳細は、33.1.2を参照してください。

何らかの原因（権限がない、指定したホストでサーバが稼働していないなど）で接続ができなかった場合は、psqlはエラーメッセージを表示し、終了します。

標準入力および標準出力の両方が端末である場合、psqlはクライアントの符号化方式を「auto」に設定します。これはロケール設定（Unixシステムでは環境変数LC_CTYPE）から適切なクライアント符号化方式を決定

します。想定した通りに動作しない場合、環境変数PGCLIENTENCODINGを使用してクライアント符号化方式を上書きすることができます。

SQLコマンドの入力

通常の操作において、psqlは、psqlが現在接続しているデータベース名の後に=>の文字列が付いたプロンプトを表示します。以下に例を示します。

```
$ psql testdb
psql (13.1)
Type "help" for help.

testdb=>
```

プロンプトに対しユーザはSQLコマンドを入力することができます。通常、入力された行はコマンド終了を意味するセミコロンに達した時点でサーバへと送信されます。改行はコマンドの終了とはみなされません。したがって、わかりやすくするために、コマンドは複数の行にわたって記述することができます。コマンドが送信され問題なく実行されると、画面にコマンドの結果が表示されます。

安全なスキーマの利用パターンを適用していないデータベースに信頼できないユーザがアクセス可能な場合は、セッションの開始時にsearch_pathから、誰でも書き込みができるスキーマを削除してください。options=-csearch_path=を接続文字列に追加するか、SELECT pg_catalog.set_config('search_path', '', false)を他のSQLの前に発行することができます。この配慮はpsqlに固有のものではありません。任意のSQLを実行するすべてのインタフェースに適用されるものです。

また、コマンドが実行される度に、psqlはLISTENとNOTIFYによって生成された非同期通知イベントを検査します。

Cの形式のブロックコメントは、サーバに送信され、サーバによって取り除かれますが、SQL標準のコメントはpsqlによって取り除かれます。

メタコマンド

psql内で入力されたコマンドのうち、引用符で囲まれていないバックスラッシュで始まるものは、psql自身が実行するpsqlのメタコマンドとして扱われます。これらのコマンドを使うと、データベースを管理したりスクリプトを作成するにあたって、psqlがより便利になります。メタコマンドはよくスラッシュコマンド、またはバックスラッシュコマンドとも呼ばれます。

psqlコマンドは、バックスラッシュ、コマンド本体、引数の順につなげた形式になっています。引数とコマンド本体の間および引数と引数の間は、空白文字によって分割されています。

引数に空白を含める場合は単一引用符で囲みます。単一引用符を引数に含める場合には、単一引用符で括られた文字列の中で、その単一引用符を2つ続けてください。単一引用符で囲われた文字は、C言語と同じような置換の対象となります。このような文字には、\n(改行)、\t(タブ)、\b (後退)、\r(復帰)、\f (改頁)、

\digits(8進数で表された文字)、\xdigits(16進数で表された文字)があります。単一引用符で括られたテキスト内でその他の任意の文字の前にバックスラッシュを付けた場合は、その文字が何であろうとその一文字だけとして扱われます。

SQL Interpolationで説明されているとおり、引数の中に引用符で囲まれていないコロン(:)とそれに続くpsql変数がある場合、その部分は変数の値で置換されます。そこで説明されている:'variable_name'および:"variable_name"という形式も同様に機能します。:{?variable_name}構文は、変数が定義済みかどうかをテストできます。これはTRUEかFALSEに置き換えられます。コロンをバックスラッシュでエスケープすると置換が防止されます。

引数の中で逆引用符(`)に囲まれた文字列は、シェルに渡されるコマンド行として解釈されます。逆引用符に囲まれた文字列は、コマンドの出力(行末の改行はすべて削除されます)で置換されます。逆引用符に囲まれた文字列内では、:variable_nameという形式でvariable_nameがpsqlの変数名であるものが、その変数の値で置換されることを除いて、特別な引用やその他の処理は起きません。また:'variable_name'という形式なら、それが変数値で置換された上で、それがシェルコマンドの単一の引数となるよう適切に引用符が付けられます。(変数に何が入っているのか正確に理解しているのでなければ、ほとんどすべての場合で後者の形式の方が望ましいでしょう。) 復帰文字、改行文字をすべてのプラットフォームで安全に引用することはできないので、そのような文字が変数値に含まれていた場合は、:'variable_name'はエラーメッセージを表示し、変数値による置換を行いません。

コマンドには、引数として(テーブル名などの)SQLの識別子を取るものがあります。これらの引数は次のようなSQLの構文規則に従います。引用符を伴わない文字は強制的に小文字になります。しかし、二重引用符(")で囲まれると、大文字小文字変換が行われず、空白文字を識別子内に含めることができます。さらに、二重引用符内では、連続する2つの二重引用符は1つの二重引用符とみなされます。例えば、FOO"BAR"BAZはfooBARbazと解釈され、"A weird"" name"はA weird" nameになります。

引数の解析は行末または引用符で囲まれていないもう1つのバックスラッシュが見つかると終了します。引用符がないバックスラッシュは新しいメタコマンドの始まりと解釈されます。\\(バックスラッシュ2つ)という特別な文字の並びは引数の終わりを意味するので、SQLコマンドが残されている場合は、その解析を続けます。このように、SQLコマンドとpsqlコマンドは1つの行に自由に混合して記述することができます。しかし、あらゆる場合において、メタコマンドの引数は行をまたぐことはできません。

メタコマンドの多くは問い合わせバッファの上で動作します。これは入力されたSQLコマンド文字列で、まだ実行のためにサーバに送信されていないものをすべて保持するだけのバッファです。これには以前の入力行や、同じ行のメタコマンドより前に入力されたすべての文字列も含まれます。

以下のメタコマンドが定義されています。

\a

現在のテーブルの出力形式が「揃えない」になっていれば「揃える」に、「揃える」になっていれば「揃えない」に設定します。このコマンドは後方互換性を保持するためにあります。より一般的な解決策は\psetを参照してください。

\c または \connect [-reuse-previous=on|off] [dbname [username] [host] [port] | conninfo]

PostgreSQLサーバへの新規の接続を確立します。接続のパラメータは、位置の構文(1つ以上のデータベース名、ユーザ、ホスト、ポート)、あるいはconninfo接続文字列で指定できます。後者の詳細

は33.1.1で説明します。引数が与えられなければ、新しい接続は以前と同じパラメータを使って作られます。

dbname、username、host、portのいずれについても-を指定するのは、パラメータを省略するのと同じになります。

新しい接続では以前の接続での接続パラメータを再利用できます。データベース名、ユーザ、ホスト、ポートだけでなく、sslmodeのようなその他の設定もです。デフォルトでは、パラメータは位置の構文では再利用されますが、conninfo文字列が与えられた場合はそうではありません。第一引数で-reuse-previous=onあるいは-reuse-previous=offを渡すことで、このデフォルトと異なる動作をさせることができます。パラメータが再利用される場合、文字列として明示的に指定されなかったパラメータは既存の接続のパラメータから取得されます。例外は、host設定が位置の構文を使った以前の値から変更された場合に、既存の接続のパラメータにあるhostaddr設定が削除されることです。コマンドで特定のパラメータを指定せず、かつ再利用もしない場合は、libpqのデフォルトが使用されます。

新規接続に成功した場合、以前の接続は閉じられます。接続の試行が（ユーザ名の間違いやアクセス拒否などの理由で）失敗した場合、psqlが対話式モードである場合、それまでの接続が保持されます。非対話式スクリプトを実行している場合は、処理はエラーとなり、即座に停止します。この実装の違いは、対話モードでは入力ミスに対するユーザの利便性を考慮し、非対話モードではスクリプトによって間違ったデータベースを操作することを防ぐための安全策を考慮した結果決められました。

例：

```
=> \c mydb myuser host.dom 6432
=> \c service=foo
=> \c "host=localhost port=5432 dbname=mydb connect_timeout=10 sslmode=disable"

=> \c -reuse-previous=on sslmode=require    -- sslmodeのみ変更
=> \c postgresql://tom@localhost/mydb?application_name=myapp
```

\C [title]

問い合わせ結果として表示されるテーブルのタイトルの設定、または、タイトルの設定解除を行います。このコマンドは、\pset title titleと同じ効力を持ちます。（このコマンド名は「標題(caption)」に由来します。以前はHTMLのテーブルの標題を設定するためだけに使われていたためです。）

\cd [directory]

現在の作業ディレクトリをdirectoryに変更します。引数がない場合は、現在のユーザのホームディレクトリに変更します。

> **ヒント**
>
> 現在の作業ディレクトリを表示するには、\! pwdを使用してください。

\conninfo

現在のデータベース接続に関する情報を出力します。

```
\copy { table [ ( column_list ) ] } from { 'filename' | program 'command' | stdin |
pstdin } [ [ with ] ( option [, ...] ) ] [ where condition ]
\copy { table [ ( column_list ) ] | ( query ) } to { 'filename' | program 'command' |
stdout | pstdout } [ [ with ] ( option [, ...] ) ]
```

フロントエンド(クライアント)コピーを行います。これはCOPY SQLコマンドを実行する操作ですが、サーバで指定ファイルに対する読み込みまたは書き込みを行うのではなく、psqlがファイルの読み書きや、サーバとローカルファイルシステム間のデータ送信を行います。この場合、ファイルへのアクセス権限はサーバではなくローカルユーザのものを使用するので、SQLのスーパーユーザ権限は必要ありません。

programが指定された場合、commandがpsqlにより実行され、commandから、または、commandへのデータはサーバとクライアント間を行き来します。ここでも、実行権限はローカル側のユーザであり、サーバ側ではなく、SQLスーパーユーザ権限は必要とされません。

\copy ... from stdinでは、データ行は、コマンドの発行源と同じところから、\.を読み取るまで、あるいは、ストリームがEOFに達するまで読み続けます。このオプションは、SQLスクリプトファイルの内部でテーブルにデータを投入する場合に便利です。\copy ... to stdoutでは、出力はpsqlコマンドの出力と同じところに送られますが、COPY countコマンドのステータスは表示されません(これはデータ行と混同してしまうかもしれないからです)。コマンドの入力元や\oオプションに関わらず、psqlの標準入力や標準出力を読み書きするには、from pstdinあるいはto pstdoutと書いてください。

このコマンドの構文はSQLのCOPYコマンドに似ています。データの入力元と出力先以外のすべてのオプションはCOPYと同じです。このため\copyメタコマンドには特別な解析規則が適用されていることに注意してください。他のほとんどのメタコマンドとは異なり、行の残り部分の全体は常に\copyの引数として解釈され、引数内の変数の置換や逆引用符の展開は行われません。

> **ヒント**
>
> \copy ... toと同じ結果を得る他の方法はSQLのCOPY ... TO STDOUTコマンドを使って、\g filenameか\g |programで終了することです。\copyと違い、この方法はコマンドが複数行にわたっても良いですし、変数の置換や逆引用符の展開式も使用できます。

> **ヒント**
>
> 全データがクライアント/サーバ接続を通らなければならないため、これらの操作は、ファイルやデータソースまたは宛先のプログラムを指定したSQLのCOPYコマンドほどには効率的ではありません。大量のデータにはSQLコマンドがより望ましいでしょう。

```
\copyright
```

PostgreSQLの著作権および配布条項を表示します。

```
\crosstabview [ colV [ colH [ colD [ sortcolH ] ] ] ]
```

問い合わせのバッファを実行し(\gと同様)、その結果をクロス表形式で表示します。問い合わせは少なくとも3つの列を返す必要があります。colVで特定される出力列が縦方向のヘッダになり、colHで特定

される出力列が横方向のヘッダになります。colDは表内に表示される出力列を特定します。オプションのsortcolHで水平方向のヘッダをソートする列を指定できます。

それぞれの列の指定は、列番号（1から始まります）でも列名でも可能です。列名については、通常のSQLの大文字小文字変換および引用の規則が適用されます。省略した場合、colVは列1、colHは列2となります。colHはcolVとは異なるものでなければなりません。colDを指定しない場合、問い合わせの結果にはちょうど3つの列がなければならず、colVでもcolHでもない列がcolDとなります。

縦方向のヘッダは一番左の列に表示され、colVの列にある値が問い合わせ結果と同じ順序で現れますが、重複するものは除かれます。

横方向のヘッダは1行目に表示され、colHの列にある値が現れますが、重複するものは除かれます。デフォルトでは、これらは問い合わせの結果と同じ順序で表示されます。しかしオプションのsortcolH引数が指定された場合は、colHの値は対応するsortcolHの値に従ってソートされて横方向のヘッダに現れますが、sortcolHの列の値は整数値でなければなりません。

クロス表の内側では、colHのそれぞれの個別値xとcolVのそれぞれの個別値yに対して、その交点(x,y)に位置するセルに、問い合わせの結果のcolHの値がxでcolVの値がyである行のcolD列の値が現れます。そのような行がなければ、セルは空欄になります。そのような行が複数あると、エラーが報告されます。

\d[S+] [pattern]

patternにマッチする各リレーション（テーブル、ビュー、マテリアライズドビュー、インデックス、シーケンス、外部テーブル）または複合型について、全ての列、列の型、テーブル空間（デフォルト以外を使用している場合）、NOT NULLやデフォルトなどの特別な属性を表示します。関連付けられているインデックス、制約、ルールおよびトリガも表示されます。外部テーブルでは関連する外部サーバも表示されます。（「パターンのマッチング」については後述のPatternsで定義されています。）

一部の種類のリレーションでは、\dは各列について追加の情報を表示します。例えば、シーケンスでは列の値、インデックスではインデックス式、外部テーブルでは外部データラッパのオプションです。

\d+というコマンド形式も同一ですが、より多くの情報を表示します。こちらでは、テーブルの列に関連付けられたコメントやテーブルにOIDが存在するかどうか、リレーションがビューの場合はビューの定義、デフォルトと異なるreplica identityの設定も表示されます。

デフォルトではユーザが作成したオブジェクトのみが表示されます。システムオブジェクトを含めるためには、パターンまたはS修飾子を付与してください。

> **注記**
>
> \dがpattern引数なしで使用された場合は、\dtvmsEと同じ意味となり、可視である全てのテーブル、ビュー、マテリアライズドビュー、シーケンス、外部テーブルの一覧が表示されます。これは純粋に利便性のためです。

\da[S] [pattern]

集約関数と、その戻り値のデータ型、演算対象となるデータ型の一覧を表示します。patternが指定された場合、そのパターンに名前がマッチする集約のみが表示されます。デフォルトではユーザが作成し

たオブジェクトのみが表示されます。システムオブジェクトを含めるためには、パターンまたはS修飾子を付与してください。

\dA[+] [pattern]

アクセスメソッドの一覧を表示します。patternが指定された場合は、そのパターンにマッチする名前のアクセスメソッドのみが表示されます。コマンド名の後に+が付加された場合は、各メソッドに関連付けられたハンドラ関数および説明も表示されます。

\dAc[+] [access-method-pattern [input-type-pattern]]

演算子クラスの一覧を表示します(37.16.1を参照してください)。access-method-patternが指定されていれば、そのパターンにマッチする名前のアクセスメソッドと関係する演算子クラスのみが表示されます。input-type-patternが指定されていれば、そのパターンにマッチする名前の入力型と関係する演算子クラスのみが表示されます。コマンド名の後に+が付加された場合は、各演算子クラスに関連付けられた演算子族および所有者も表示されます。

\dAf[+] [access-method-pattern [input-type-pattern]]

演算子族の一覧を表示します(37.16.5を参照してください)。access-method-patternが指定されていれば、そのパターンにマッチする名前のアクセスメソッドと関係する演算子族のみが表示されます。input-type-patternが指定されていれば、そのパターンにマッチする名前の入力型と関係する演算子族のみが表示されます。コマンド名の後に+が付加された場合は、各演算子族に関連付けられた所有者も表示されます。

\dAo[+] [access-method-pattern [operator-family-pattern]]

演算子族に関連する演算子の一覧を表示します(37.16.2を参照してください)。access-method-patternが指定されていれば、そのパターンにマッチする名前のアクセスメソッドと関係する演算子族のメンバーのみが表示されます。operator-family-patternが指定されていれば、そのパターンにマッチする名前の演算子族のメンバーのみが表示されます。コマンド名の後に+が付加された場合は、(順序演算子であれば)ソート演算子族も表示されます。

\dAp[+] [access-method-pattern [operator-family-pattern]]

演算子族に関連するサポート関数の一覧を表示します(37.16.3を参照してください)。access-method-patternが指定されていれば、そのパターンにマッチする名前のアクセスメソッドと関係する演算子族の関数のみが表示されます。operator-family-patternが指定されていれば、そのパターンにマッチする名前の演算子族の関数のみが表示されます。コマンド名の後に+が付加された場合は、実際のパラメータの一覧を伴って、冗長に表示されます。

\db[+] [pattern]

テーブル空間を一覧表示します。patternが指定された場合、そのパターンに名前がマッチするテーブル空間のみが表示されます。コマンド名に+が付与された場合、各テーブル空間に関連付けされたオプション、ディスク上のサイズ、権限、摘要についても表示します。

\dc[S+] [pattern]

文字セット符号化方式間の変換の一覧を表示します。patternが指定された場合、そのパターンに名前がマッチする変換のみが表示されます。デフォルトではユーザが作成したオブジェクトのみが表示され

ます。システムオブジェクトを含めるためには、パターンまたはS修飾子を付与してください。コマンド名に+を付与すると、各オブジェクトに関連する説明を付けて表示します。

\dC[+] [pattern]

型キャストの一覧を表示します。patternが指定された場合、そのパターンに元データ型または変換先データ型がマッチするキャストのみが表示されます。コマンド名に+を付与すると、各オブジェクトに関連する説明を付けて表示します。

\dd[S] [pattern]

constraint、operator class、operator family、rule、triggerという種類のオブジェクトについての説明を表示します。他のコメントはすべて、これらのオブジェクト種類用の対応するバックスラッシュコマンドによって表示されます。

\ddはpatternにマッチするオブジェクトの説明を表示します。引数が指定されていない場合は、適切な種類の可視なオブジェクトの説明を表示します。どちらの場合でも、一覧に表示されるのは説明を持つオブジェクトのみです デフォルトではユーザが作成したオブジェクトのみが表示されます。システムオブジェクトを含めるためには、パターンまたはS修飾子を付与してください。

オブジェクトの説明はCOMMENT SQLコマンドを使用して作成することができます。

\dD[S+] [pattern]

ドメインの一覧を表示します。patternが指定された場合、ドメイン名がそのパターンにマッチするもののみが表示されます。デフォルトではユーザが作成したオブジェクトのみが表示されます。システムオブジェクトを含めるには、パターンを指定するか、あるいはS修飾子を付けてください。コマンド名の後に+を付けた場合は、各オブジェクトの権限と説明も表示されます。

\ddp [pattern]

デフォルトのアクセス権限設定を一覧表示します。組み込みのデフォルトから権限設定が変更されたロール（および適切ならばスキーマも）ごとに1項目示されます。patternが指定された場合、パターンにマッチするロール名またはスキーマ名の項目のみが表示されます。

ALTER DEFAULT PRIVILEGESコマンドを使用して、デフォルトのアクセス権限を設定します。権限表示の意味は5.7で説明します。

\dE[S+] [pattern]
\di[S+] [pattern]
\dm[S+] [pattern]
\ds[S+] [pattern]
\dt[S+] [pattern]
\dv[S+] [pattern]

このコマンド群において、E、i、m、s、t、vという文字はそれぞれ、外部テーブル、インデックス、マテリアライズドビュー、シーケンス、テーブル、ビューを表します。これらの種類のオブジェクトの一覧を表示するために、これらの文字の中の任意の文字またはすべてを任意の順番で指定することができます。例えば、\dtiはテーブルとインデックスを列挙します。+がコマンド名に付与された場合、各オブジェクトは、もしあれば永続性の状態(permanent、temporary、またはunlogged)、ディスク上の物理容量、

関連する説明をつけて表示されます。patternが指定されている場合は、パターンに名称がマッチする項目のみが表示されます。デフォルトでは、ユーザが作成したオブジェクトのみが表示されます。システムオブジェクトを含めるためにはパターンまたはS修飾子を付与してください。

\des[+] [pattern]

外部(foreign)サーバ(つまり「external servers」)を一覧表示します。patternが指定されている場合は、名前がパターンにマッチするサーバのみが表示されます。\des+構文が使用された場合、サーバのアクセス権限、型、バージョン、オプション、説明など各サーバの完全な説明が表示されます。

\det[+] [pattern]

外部(foreign)テーブル(つまり「external tables」)を一覧表示します。patternが指定された場合、パターンにテーブル名またはスキーマ名がマッチするもののみが表示されます。\det+が使用された場合、汎用オプションと外部テーブルの説明も表示されます。

\deu[+] [pattern]

ユーザマップ(つまり「external users」)を一覧表示します。patternが指定されている場合は、名前がパターンにマッチするユーザのみが表示されます。\deu+構文が使用された場合、各マップについて追加情報が表示されます。

> **注意**
>
> \deu+ではリモートユーザのユーザ名とパスワードも表示される可能性があります。これらを外部に曝さないように注意しなければなりません。

\dew[+] [pattern]

外部データラッパ(つまり「external wrappers」)を一覧表示します。patternが指定されている場合、名前がパターンにマッチする外部データラッパのみが表示されます。\dew+構文が使用された場合、外部データラッパのアクセス権限、オプションおよび説明も表示されます。

\df[anptwS+] [pattern]

関数とその結果のデータ型、引数のデータ型、および、「agg」(集約)、「normal」、「procedure」、「trigger」、「window」で分類される関数の種類の一覧を表示します。特定種類の関数のみを表示させるには、対応する文字a、n、p、t、wをコマンドに付けて下さい。patternが指定されている場合は、そのパターンに名前がマッチする関数のみが表示されます。デフォルトではユーザが作成したオブジェクトのみが表示されます。システムオブジェクトを含めるためには、パターンまたはS修飾子を付与してください。\df+構文が使われた場合、各関数の、揮発性、並列処理での安全性、所有者、セキュリティ分類、アクセス権限、言語、ソースコードや説明を含む付加的情報も表示されます。

> **ヒント**
>
> 特定のデータ型を引数とする関数や特定のデータ型を返す関数を検索するには、ページャの検索機能を使用して\dfの出力をスクロールしてください。

\dF[+] [pattern]

全文検索設定を一覧表示します。patternが指定された場合、このパターンにマッチする名前の設定のみが表示されます。\dF+形式が使用された場合、使用される全文検索パーサや各パーサトークン型についての辞書リストなど各設定の完全な説明が表示されます。

\dFd[+] [pattern]

全文検索辞書を一覧表示します。patternが指定された場合、このパターンにマッチする名前の辞書のみが表示されます。\dFd+形式が使用された場合、選択された辞書それぞれについて使用される全文検索テンプレートやオプション値など更なる情報が表示されます。

\dFp[+] [pattern]

全文検索パーサを一覧表示します。patternが指定された場合、このパターンにマッチする名前のパーサのみが表示されます。\dFp+形式が使用された場合、使用される関数や認知されるトークン型のリストなど各パーサの完全な説明が表示されます。

\dFt[+] [pattern]

テキスト検索テンプレートを一覧表示します。patternが指定された場合、このパターンにマッチする名前のテンプレートのみが表示されます。\dFt+形式が使用された場合、テンプレートそれぞれについて使用される関数名など更なる情報が表示されます。

\dg[S+] [pattern]

データベースロールを一覧表示します。(「ユーザ」と「グループ」という概念は「ロール」に統合されましたので、このコマンドは\duと同じものになりました。) デフォルトでは、ユーザによって作成されたロールのみが表示されます。システムロールを含めるにはS修飾子を付与してください。patternが指定されている場合は、そのパターンに名前がマッチするロールのみが表示されます。\dg+構文が使用された場合、ロールそれぞれについて更なる情報が表示されます。現時点では各ロールのコメントが追加されます。

\dl

\lo_listの別名で、ラージオブジェクトの一覧を表示します。

\dL[S+] [pattern]

手続き言語を一覧表示します。patternを指定すると、パターンに名前がマッチする言語のみが表示されます。デフォルトではユーザが作成した言語のみが表示されます。システムオブジェクトを含めるためには、パターンまたはS修飾子を付与してください。+をコマンド名に追加すると、呼び出しハンドラ、有効性検証関数、アクセス権限、システムオブジェクトか否かという情報を付けて各言語が表示されます。

\dn[S+] [pattern]

スキーマ(名前空間)の一覧を表示します。patternを指定すると、パターンに名前がマッチするスキーマのみが表示されます。デフォルトではユーザが作成したオブジェクトのみが表示されます。パターンまたはS修飾子を追加すると、システムオブジェクトが表示に追加されます。コマンド名の後に+を付加すると、各オブジェクトに関連付けられている権限と説明が(存在すれば)表示されます。

\do[S+] [pattern]

演算子と、その演算項目と結果の型を一覧表示します。patternを指定すると、パターンに名前がマッチする演算子のみが表示されます。デフォルトではユーザが作成したオブジェクトのみが表示されます。システムオブジェクトを含めるためには、パターンまたはS修飾子を付与してください。コマンド名に+を付加すると、各演算子についての追加情報が表示されますが、現在はその元になっている関数の名前だけです。

\dO[S+] [pattern]

照合順序を一覧表示します。patternを指定すると、パターンに名前がマッチする照合順序のみが表示されます。デフォルトではユーザが作成したオブジェクトのみが表示されます。システムオブジェクトを含めるためには、パターンまたはS修飾子を付与してください。コマンド名の後に+を付加すると、各照合順序に関連付けられている説明が(存在すれば)表示されます。現在のデータベースの符号化方式で使用できる照合順序のみが表示されることに注意してください。このため同じインストレーションであってもデータベースによって結果が異なる可能性があります。

\dp [pattern]

テーブル、ビュー、シーケンスを、関連付けられているアクセス権限とともに一覧表示します。patternを指定すると、パターンに名前がマッチするテーブル、ビュー、シーケンスのみが表示されます。

アクセス権限の設定にはGRANTコマンドとREVOKEコマンドが使われます。権限の表示に関する意味は5.7で説明します。

\dP[itn+] [pattern]

パーティション化されたリレーションの一覧を表示します。patternが指定されている場合は、名前がパターンにマッチするエントリのみが表示されます。修飾子t(テーブル)とi(インデックス)をコマンドに付けて、表示されるリレーションの種類を限定できます。デフォルトでは、パーティション化されたテーブルとインデックスの一覧が表示されます。

修飾子n(「nested」)が使われた、もしくはパターンが指定された場合、ルートでないパーティション化されたリレーションが含められ、各パーティション化されたリレーションの親を表示しながら列が表示されます。

コマンド名の後に+が付けられた場合、リレーションの説明と一緒に、各リレーションのパーティションの大きさの合計も表示されます。nが+と組み合わされた場合、大きさが2つ表示されます。1つは直接アタッチされたリーフパーティションの合計の大きさで、もう1つは間接的にアタッチされたサブパーティションを含む全パーティションの合計の大きさです。

\drds [role-pattern [database-pattern]]

定義済み設定に関する設定を一覧表示します。これらの設定はロール固有、データベース固有、またはその両方です。role-patternおよびdatabase-patternはそれぞれ特定のロールやデータベースを選択するために使用します。パターンが省略された場合、または*が指定された場合、ロール固有ではない、または、データベース固有ではない設定を含め、すべての設定を表示します。

ロール単位およびデータベース単位の設定を定義するにはALTER ROLEおよびALTER DATABASEコマンドを使用します。

\dRp[+] [pattern]

レプリケーションのパブリケーションを一覧表示します。patternが指定された場合、名前がそのパターンにマッチするパブリケーションのみが表示されます。コマンド名の後に+が付けられた場合、各パブリケーションに関連付けられているテーブルも表示されます。

\dRs[+] [pattern]

レプリケーションのサブスクリプションを一覧を表示します。patternが指定された場合、名前がそのパターンにマッチするサブスクリプションのみが表示されます。コマンド名の後に+が付けられた場合、サブスクリプションの追加属性も表示されます。

\dT[S+] [pattern]

データ型を一覧表示します。patternを指定すると、パターンにマッチする名前を持つ型のみを表示します。+をコマンド名に付けると、型ごとに、型の内部名、サイズ、enum型では許される値、関連する権限も表示されます。デフォルトではユーザが作成したオブジェクトのみが表示されます。システムオブジェクトを含めるためには、パターンまたはS修飾子を付与してください。

\du[S+] [pattern]

データベースロールを一覧表示します。(「ユーザ」と「グループ」という概念は「ロール」に統合されましたので、このコマンドは\dgと同じものになりました。) patternが指定されている場合は、そのパターンに名前がマッチするロールのみが表示されます。デフォルトでは、ユーザによって作成されたロールのみが表示されます。システムロールを含めるにはS修飾子を付与してください。\du+構文が使用された場合、ロールそれぞれについて更なる情報が表示されます。現時点では各ロールのコメントが追加されます。

\dx[+] [pattern]

インストールされた拡張を一覧表示します。patternを指定すると、パターンにマッチする名前の拡張のみを表示します。\dx+形式が使用された場合、マッチする拡張それぞれについて拡張に属するすべてのオブジェクトが表示されます。

\dy[+] [pattern]

イベントトリガを一覧表示します。patternを指定すると、パターンにマッチする名前のイベントトリガのみを表示します。+をコマンド名に追記すると、関連する説明を付けて各オブジェクトを表示します。

\eまたは\edit [filename] [line_number]

filenameが指定された場合、このファイルが編集されます。エディタを終了した後、ファイルの内容は問い合わせバッファにコピーされます。filenameが指定されない場合、現在の問い合わせバッファが一時ファイルにコピーされ、同様に編集されます。現在の問い合わせバッファが空の場合、最も最近に実行された問い合わせが一時ファイルにコピーされ、同様に編集されます。

次に、問い合わせバッファの新しい内容が、psqlの通常の規則に従い、全体を1行として再解析されます。完全な問い合わせはすべて即座に実行されます。つまり、問い合わせバッファにセミコロンが含まれるか、セミコロンで終わっている場合、そこまでの部分すべてが実行され、問い合わせバッファから削除されます。問い合わせバッファ内に残ったものはすべて再表示されます。送信するにはセミコロンまたは\gを、問い合わせバッファをクリアしてキャンセルするには\rを入力してください。

バッファ全体を1行として扱うので、特にメタコマンドに影響があります。バッファ内でメタコマンドより後にある部分はすべて、それが複数行にまたがっていたとしても、メタコマンドの引数として解釈されます。(従って、この方法ではメタコマンドを使用するスクリプトを作成できません。その目的の場合は、\iを使ってください。)

行番号(line_number)が指定された場合、psqlはファイルまたは問い合わせバッファ内の指定行にカーソルを位置づけます。すべてが数字の引数が1つだけ指定された場合、psqlはそれをファイル名ではなく行番号であるとみなすことに注意してください。

> **ヒント**
>
> 使用するエディタを設定、カスタマイズする方法については、下記のEnvironmentを参照してください。

\echo text [...]

評価された引数を空白で区切り、標準出力に出力し、改行します。スクリプトが出力するところどころに情報を記載する場合に有用です。使用例を次に示します。

```
=> \echo `date`
Tue Oct 26 21:40:57 CEST 1999
```

最初の引数が引用符で囲まれていない-nである場合、最後の改行は出力されません(し、最初の引数も出力されません)。

> **ヒント**
>
> \oコマンドを使用して問い合わせの出力先を変更した場合、このコマンドではなく、\qechoを使用した方が良いかもしれません。\warnコマンドも参照してください。

\ef [function_description [line_number]]

このコマンドは指定された関数やプロシージャの定義をCREATE OR REPLACE FUNCTIONやCREATE OR REPLACE PROCEDUREコマンド構文で取り出し、編集します。編集は\editと同様の方法で行われます。エディタ終了後、更新されたコマンドはセミコロンを付けていれば即座に実行されます。そうでなければ再表示されます。送信するにはセミコロンあるいは\gを、キャンセルするには\rを入力してください。

対象の関数は名前だけ、または、たとえばfoo(integer, text)のように名前と引数で指定することができます。同じ名前の関数が複数存在する場合、引数の型を指定しなければなりません。

関数が指定されなかった場合、空のCREATE FUNCTIONのテンプレートが編集用に表示されます。

行番号が指定された場合、psqlは関数本体における指定行にカーソルを移動します。(関数本体は通常、ファイルの先頭から始まらないことに注意してください。)

他のほとんどのメタコマンドと異なり、行の残り部分はすべて\efの引数であると常に解釈され、引数内の変数の置換も逆引用符の展開も行われません。

> **ヒント**
>
> 使用するエディタを設定、カスタマイズする方法については下記のEnvironmentを参照してください。

\encoding [encoding]

クライアント側の文字セット符号化方式を設定します。引数を指定しない場合、このコマンドは現在の符号化方式を表示します。

\errverbose

最も最近のサーバのエラーメッセージを最大の冗長さ、つまりVERBOSITYがverboseに、そしてSHOW_CONTEXTがalwaysに設定されているかのようにして、繰り返します。

\ev [view_name [line_number]]

このコマンドは、指定したビューの定義をCREATE OR REPLACE VIEWコマンドの形式で取得して、編集します。編集は\editの場合と同じ方法で行われます。エディタ終了後、更新されたコマンドはセミコロンを付けていれば即座に実行されます。そうでなければ再表示されます。送信するにはセミコロンあるいは\gを、キャンセルするには\rを入力してください。

ビューを指定しなかった場合は、空のCREATE VIEWテンプレートが編集用に提供されます。

行番号を指定した場合、psqlはカーソルをビュー定義の指定した行に位置づけます。

他のほとんどのメタコマンドと異なり、行の残り部分はすべて\evの引数であると常に解釈され、引数内の変数の置換も逆引用符の展開も行われません。

\f [string]

位置揃えされていない問い合わせの出力用の、フィールドの区切り文字を設定します。デフォルトは、縦棒(|)です。これは\pset fieldsepと同じです。

\g [(option=value [...])] [filename]
\g [(option=value [...])] [|command]

現在の問い合わせ入力バッファをサーバに送って実行します。

\gの後に括弧が現れる場合は、括弧はoption=value書式オプション句の空白で区切られた一覧を囲んでいます。書式オプション句は\pset option valueコマンドと同じように解釈されますが、この問い合わせの間でのみ有効です。この一覧の中では、空白は=の周りでは許されていませんが、オプション句の間には必要です。=valueが省略された場合、指名されたoptionは、明示されたvalueがない\pset optionと同じように変更されます。

filenameや|command引数を指定すると、問い合わせ出力を通常通りに表示する代わりに、指定したファイルに書き込んだり、指定のシェルコマンドにパイプで渡します。問い合わせが成功しゼロ以上のタプルが返る場合にのみファイルまたはコマンドに書き出されます。問い合わせが失敗する場合やデータを返さないSQLコマンドでは書き出されません。

現在の問い合わせバッファが空の場合、最も最近に送信された問い合わせが再実行されます。その点を除けば、\gだけを指定した場合は、セミコロンと実質的に同じです。\gに引数を指定した場合は、\oコマンドの「一度限りの」代替手段として使用でき、さらに通常は\psetで設定される出力書式のオプションの一度限りの調整もできます。

最後の引数が|で始まっている場合、行の残りの部分はすべて実行するcommandであると解釈され、その中では変数の置換も逆引用符の展開も行なわれません。行の残り部分は、単にあるがままにシェルに渡されます。

\gdesc

現在の問い合わせバッファの結果の説明(列名とデータ型)を表示します。問い合わせは実際には実行されませんが、ある種の構文エラーが含まれている場合、そのエラーは通常の方法で報告されます。

現在の問い合わせバッファが空の場合、直近に送った問い合わせの説明が代わりに出力されます。

\gexec

現在の問い合わせバッファをサーバに送信し、問い合わせの出力(あれば)の各行の各列をSQL文として実行します。例えば、my_tableの各列にインデックスを作成するには次のようにします。

```
=> SELECT format('create index on my_table(%I)', attname)
-> FROM pg_attribute
-> WHERE attrelid = 'my_table'::regclass AND attnum > 0
-> ORDER BY attnum
-> \gexec
CREATE INDEX
CREATE INDEX
CREATE INDEX
CREATE INDEX
```

生成された問い合わせは行が返された順番で実行され、また2つ以上の列が返された場合は、各行の中で左から右に実行されます。NULLのフィールドは無視されます。生成された問い合わせは、そのままサーバに送信されて処理されるため、psqlのメタコマンドとすることはできず、またpsqlの変数の参照を含むこともできません。個別の問い合わせで失敗した場合、残りの問い合わせの実行はON_ERROR_STOPが設定されているのでなければ継続します。個々の問い合わせの実行はECHOの処理に従います。(\gexecを使う場合、ECHOをallあるいはqueriesに設定することが推奨されることが多いでしょう。) 問い合わせのログ出力、シングルステップモード、時間表示(timing)、およびその他の問い合わせ実行に関する機能は、生成された各問い合わせにも適用されます。

現在の問い合わせバッファが空の場合、最も最近に送信された問い合わせが再実行されます。

\gset [prefix]

現在の問い合わせバッファをサーバに送信し、問い合わせの出力をpsql変数(下記のVariables参照)に格納します。実行される問い合わせは正確に1行を返さなければなりません。行の各列は、列と同じ名前を持つ別々の変数に格納されます。例えば、以下のようになります。

```
=> SELECT 'hello' AS var1, 10 AS var2
```

```
-> \gset
=> \echo :var1 :var2
hello 10
```

prefixを指定した場合、使用する変数の名前を作成する時にその文字列が問い合わせの列名の前に付けられ、次のようになります。

```
=> SELECT 'hello' AS var1, 10 AS var2
-> \gset result_
=> \echo :result_var1 :result_var2
hello 10
```

列の結果がNULLである場合、対応する変数は設定されず未設定状態となります。

問い合わせが失敗、または1行を返さない場合、変数は変更されません。

現在の問い合わせバッファが空の場合、最も最近に送信された問い合わせが再実行されます。

\gx [(option=value [...])] [filename]
\gx [(option=value [...])] [|command]

\gxは、\psetオプションの一覧にexpanded=onが含まれているかのように、この問い合わせに対して拡張出力モードを使用することを除いて\gと同じです。\xも参照してください。

\hまたは\help [command]

指定したSQLコマンドの構文に関するヘルプを表示します。commandが指定されていない場合は、psqlは構文ヘルプが存在する全てのコマンドの一覧を表示します。commandをアスタリスク(*)にすると、全てのSQLコマンドの構文ヘルプが表示されます。

他のほとんどのメタコマンドと異なり、行の残り部分はすべて\helpの引数であると常に解釈され、引数内の変数の置換も逆引用符の展開も行われません。

> **注記**
>
> 入力を簡単にするため、複数の単語からなるコマンドを引用符で囲む必要はありません。**\help alter table**と入力するだけで十分です。

\Hまたは\html

HTML問い合わせ出力形式を有効にします。HTML形式が有効になっている場合は、デフォルトの位置揃えされたテキスト形式に戻します。このコマンドは互換性と簡便性のために存在します。他の出力オプションについては、\psetを参照してください。

\iまたは\include filename

filenameファイルから入力を読み取り、キーボードから入力された場合と同じように実行します。

filenameが-(ハイフン)の場合、EOFを示すもの、または\qメタコマンドが読まれるまで標準入力から読み込みます。これは対話的な入力とファイルからの入力を混在させるために使うことができます。

Readlineと同じ挙動は、それが最も外部のレベルで動作している場合にのみ利用されることに注意してください。

> **注記**
>
> 読み取られた行を画面に表示させる場合は、ECHO変数をallに設定する必要があります。

\if *expression*
\elif *expression*
\else
\endif

このコマンド群は入れ子にすることができる条件ブロックを実現します。条件ブロックは\ifで始まり、\endifで終わらなければなりません。その間には\elif句をいくつでも置くことができ、さらにその後に\else句を1つだけ置くことができます。条件ブロックを構成するコマンドの間には、通常の問い合わせや他の種類のバックスラッシュコマンドを置くことができます（通常は置きます）。

\ifコマンドおよび\elifコマンドはその引数を読み取り、それを論理式であるとして評価します。式の結果がtrueであれば、通常通りに処理が続きますが、そうでないときは、対応する\elif、\elseまたは\endifに到達するまで行をスキップします。ひとたび\ifまたは\elifの評価が真になったら、同じブロック内のそれより後にある\elifコマンドの引数は評価されず、偽であるとして扱われます。\elseより後にある行は、それより前の対応する\ifと\elifが一つも真にならなかった時にのみ実行されます。

\ifコマンドおよび\elifコマンドのexpression引数は、他のバックスラッシュコマンドの引数と同様、変数置換と逆引用符展開の対象となります。その後で、on/offのオプション変数の値のように評価されます。従って有効な値は、true、false、1、0、on、off、yes、noのいずれかに曖昧性なしに、大文字小文字を区別せずにマッチするものです。例えば、t、T、tRはすべてtrueであるとみなされます。

真にも偽にも適切に評価できない式には警告を発行し、偽として扱います。

スキップされる行も、問い合わせとバックスラッシュコマンドを特定するため、通常通り解析されますが、問い合わせはサーバには送信されず、条件コマンド（\if、\elif、\else、\endif）以外のバックスラッシュコマンドは無視されます。条件コマンドは入れ子の有効性の確認のためだけに検査されます。スキップされる行の変数参照は展開されず、逆引用符の展開も実行されません。

条件ブロック内のすべてのバックスラッシュコマンドは同じファイル内になければなりません。ファイル内のすべての\ifブロックが閉じられるより前に、メインの入力ファイルまたは\includeされたファイルの終端に到達した場合、psqlはエラーを発生させます。

例を示します。

```
-- データベース内に2つのレコードが存在するかどうかを検査し、
-- その結果を2つのpsql変数に格納する。
SELECT
    EXISTS(SELECT 1 FROM customer WHERE customer_id = 123) as is_customer,
    EXISTS(SELECT 1 FROM employee WHERE employee_id = 456) as is_employee
```

```
\gset
\if :is_customer
    SELECT * FROM customer WHERE customer_id = 123;
\elif :is_employee
    \echo 'is not a customer but is an employee'
    SELECT * FROM employee WHERE employee_id = 456;
\else
    \if yes
        \echo 'not a customer or employee'
    \else
        \echo 'this will never print'
    \endif
\endif
```

\irまたは\include_relative filename

\irコマンドは\iと似ていますが、相対ファイル名の解決方法が異なります。対話モードで実行している場合は2つのコマンドの動作は同一です。しかし、スクリプトから呼び出す場合、\irは、現在の作業ディレクトリではなく、そのスクリプトの格納ディレクトリから見た相対ファイル名として解釈します。

\l[+] または \list[+] [pattern]

サーバ内のデータベースについて、その名前、所有者、文字セット符号化方式、アクセス権限を一覧表示します。patternを指定すると、パターンにマッチする名前を持つデータベースのみを表示します。コマンド名に+を付けると、データベースのサイズ、デフォルトのテーブル空間、説明も表示します。(サイズ情報は現在のユーザが接続可能なデータベースでのみ表示されます。)

\lo_export loid filename

データベースからOIDがloidであるラージオブジェクトを読み取り、filenameに書き出します。これはlo_exportサーバ関数とは微妙に異なります。lo_export関数は、データベースサーバを実行しているユーザ権限で、サーバ上のファイルシステムに対して動作します。

> **ヒント**
>
> ラージオブジェクトのOIDを確認するには、\lo_listを使用してください。

\lo_import filename [comment]

ファイルをPostgreSQLのラージオブジェクトに保存します。オプションで、そのオブジェクトに指定したコメントを関連付けることができます。下記に例を示します。

```
foo=> \lo_import '/home/peter/pictures/photo.xcf' 'a picture of me'
lo_import 152801
```

上の応答は、指定したラージオブジェクトがオブジェクトID 152801として受け付けられたことを示します。今後この新規作成されたラージオブジェクトにアクセスする場合に、この番号が使用できます。可読

性を高めるために、常に全てのオブジェクトに人間がわかるようなコメントを関連付けることが推奨されます。\lo_listコマンドではOIDとコメントの両方が表示されます。

このコマンドは、ローカルなユーザによってローカルなファイルシステムに対して動作します。一方、サーバ側のlo_importは、サーバのユーザによってサーバ上のファイルシステムに対して動作します。このコマンドとサーバ側のlo_importは、この点で微妙に異なっています。

\lo_list

現在データベースに保存されている全てのPostgreSQLラージオブジェクトの一覧を、そのオブジェクトに付けられたコメントと一緒に表示します。

\lo_unlink loid

OIDがloidであるラージオブジェクトをデータベースから削除します。

> **ヒント**
>
> ラージオブジェクトのOIDを確認するには、\lo_listを使用してください。

\oまたは\out [filename]
\oまたは\out [|command]

以降の問い合わせの結果を、filenameで指定されたファイルに保存するか、またはシェルコマンドcommandにパイプで渡すようにします。引数がない場合、問い合わせの出力はリセットされて標準出力になります。

引数が|で始まっている場合、行の残りの部分はすべて実行するcommandであると解釈され、その中では変数の置換も逆引用符の展開も行われません。行の残り部分は、単にあるがままにシェルに渡されます。

「問い合わせの結果」には、全てのテーブル、コマンドの応答、データベースサーバからの注意メッセージだけでなく、データベースに問い合わせを行う(\dのような)各種バックスラッシュコマンドの出力が含まれます。ただし、エラーメッセージは含まれません。

> **ヒント**
>
> 問い合わせの結果の間にテキストを挿入するには、\qechoを使用してください。

\pまたは\print

現在の問い合わせバッファを標準出力に書き出します。現在の問い合わせバッファが空の場合、最も最近に実行された問い合わせが書き出されます。

\password [username]

指定したユーザ(デフォルトは現在のユーザ)のパスワードを変更します。このコマンドは、新しいパスワードを促し、暗号化して、それをALTER ROLEコマンドとしてサーバに送信します。これによりコマンド履歴やサーバログなどどこにも新しいパスワードが平文では残りません。

\prompt [text] name

変数nameに代入するテキストを入力するようにユーザを促します。プロンプトtextをオプションで指定することができます。(複数の単語をプロンプトで使用する場合はそのテキストを単一引用符でくくってください。)

デフォルトでは\promptは入出力に端末を使用します。しかし、-fコマンドラインスイッチが使用されている場合、\promptは標準入力、標準出力を使用します。

\pset [option [value]]

このコマンドは問い合わせ結果のテーブル出力に影響するオプションを設定します。optionには、どのオプションを設定するのかを記述します。valueの意味は選択したオプションにより変わります。以下のオプション別の説明の通り、オプションの中にはvalueを省略することでトグルや設定解除を行うものがあります。こうした動作の記載がなければ、valueを省略すると、単に現在の設定値が表示されることになります。

何も引数をつけずに\psetを実行すると、すべての表示オプションの現在の状態を表示します。

以下は、表示の調整に関するオプションです。

border

valueは数値でなければなりません。基本的には、この数字が大きくなればなるほど、表示するテーブルが持つ境界線は増えますが、詳細はそれぞれの出力形式に依存しています。HTML書式では、この値は直接border=...属性に反映されます。他のほとんどの書式の場合は、0(境界線なし)、1(内側の境界線)、2(テーブル枠)という3つの数値のみ意味を持ち、2より大きな値はborder = 2と同じとして扱われます。latexおよびlatex-longtable書式では、さらにデータ行の間に境界線を付ける、3という値をとることができます。

columns

wrapped書式の対象幅を設定し、そして、ページャを必要とする、拡張自動モードにおける縦表示への切替えに十分な幅で出力するかどうかを決定する幅制限を設定します。ゼロ(デフォルト)では、環境変数COLUMNS、もしCOLUMNSが設定されていなければ、検出したスクリーンの幅、により対象幅が制御されます。さらにcolumnsがゼロの場合、wrapped書式はスクリーン出力にのみ影響を与えることになります。columnsが非ゼロの場合は、ファイルやパイプへの出力も同様に折り返されます。

csv_fieldsep

CSV出力形式で使われるフィールド区切り文字を指定します。区切り文字がフィールドの値中に現れる場合には、標準のCSV規則に従ってそのフィールドは二重引用符内に出力されます。デフォルトはコンマです。

expanded (またはx)

valueを指定する場合は、拡張(expanded)モードを有効または無効にするonまたはoff、あるいはautoのいずれかでなければなりません。valueを省略した場合、このコマンドは通常モードと拡張モードの設定をトグルします。拡張モードを有効にした場合、問い合わせ結果は左に列名、右にデータという2つの列で出力されます。このモードは、データが通常の「水平(horizontal)」モードによる画面表示に適していない場合に有用です。自動(auto)設定の場合、問い合わせの出力が2列以上で

かつ画面幅より広ければ拡張モードが使用され、そうでなければ通常モードが使用されます。自動設定は位置揃え書式または折り返し書式でのみ有効です。この他の書式では、常に拡張モードが無効の場合と同様に動作します。

`fieldsep`

位置揃えなしの出力書式で使用されるフィールド区切り文字を指定します。これにより、例えばタブ区切りといった他プログラムに要求される形式を作成することができます。タブをフィールド区切り文字として使用するには、`\pset fieldsep '\t'`と入力します。デフォルトのフィールド区切り文字は`'|'`(縦棒)です。

`fieldsep_zero`

位置揃えなしの出力書式で使用されるフィールド区切り文字をゼロバイトに指定します。

`footer`

`value`を指定する場合、それぞれテーブルフッタの表示((`n rows`)数)を有効にするonまたは無効にするoffのいずれかでなければなりません。`value`を省略した場合、このコマンドはフッタの表示、非表示をトグルします。

`format`

出力形式をaligned、asciidoc、csv、html、latex、latex-longtable、troff-ms、unaligned、wrappedのいずれかに設定します。一意に判別できる範囲で省略が可能です。

aligned書式は、標準的で人間が読みやすいように、美しく整形されたテキスト出力です。これがデフォルトです。

unaligned書式は、表示行の1行に1つの行の全列を、現在有効なフィールド区切り文字で区切って書き出します。これは他のプログラムに読み込ませることを目的とした出力、例えばタブ区切りやカンマ区切り書式を生成する場合に有用です。しかし、列の値にフィールド区切り文字が現れても、特別扱いはしません。ですので、そのような目的には、CSV書式の方がより相応しいでしょう。

csv書式は RFC 4180[1]で記述された引用規則を適用して、列の値をコンマで区切って書きます。この出力はサーバのCOPYコマンドのCSV書式と互換性があります。列名が書かれたヘッダ行は、tuples_onlyがonでなければ生成されます。タイトルとフッタは出力されません。各行はシステム依存の改行文字で終わります。改行文字は、Unix系のシステムでは典型的には単独の改行(\n)であり、Microsoft Windowsでは復帰と改行の並び(\r\n)です。コンマ以外のフィールド区切り文字は`\pset csv_fieldsep`で選べます。

wrapped書式はalignedと似ていますが、幅の広いデータ値を複数行に折り返して対象の列幅に合うように出力します。対象の幅はcolumnsオプションの項に記述されているように決定されます。psqlは列ヘッダタイトルを折り返さないことに注意して下さい。このためwrapped書式は列ヘッダに必要とする幅全体が対象より長い場合、alignedと動作が同じになります。

asciidoc、html、latex、latex-longtableおよびtroff-ms書式は対応するマークアップ言語を使用する文書内に含めることを目的とした表を出力します。出力自体は完全な文書ではありません。HTMLでは必要性がないかもしれませんが、LaTeXでは完全な文書ラッパを持たせなけれ

[1] https://tools.ietf.org/html/rfc4180

ばなりません。latex書式はLaTeXのtabular環境を使います。latex-longtable書式ではLaTeXのlongtableおよびbooktabsパッケージも必要です。

linestyle

境界線の表示形式をascii、old-asciiまたはunicodeのいずれかに設定します。一意になれば省略形が許されます。(つまり一文字で十分であることを意味します。) デフォルトの設定はasciiです。このオプションはalignedおよびwrapped出力書式のみで有効です。

ascii形式は通常のASCIIを使用します。データ内の改行は右側余白に+を使用して表します。wrapped書式で、改行文字のない行が2行にまたがるときは、先頭行の右側余白にドット(.)を表示し、次の行の左側余白にもドットを表示します。

old-ascii形式は通常のASCII文字を使用して、PostgreSQL 8.4以前で使用されていた方法で整形します。データ内の改行は列区切りの左側に:記号を使用して表します。データを改行文字なしに折り返す際には、列区切りの左側に;記号を使用して表します。

unicode形式はUnicode矩形描画文字を使用します。データ内の改行は右側の余白に復帰記号を使用して表します。データを改行文字なしに折り返す際には、省略記号を先頭行の右側余白に表示し、次の行の左側余白にも表示します。

border設定がゼロより大きい場合、linestyleオプションはまた、境界線を描画する文字も決定します。通常のASCII文字はどのような場合でも動作しますが、Unicode文字が表示できる環境では、その方が見た目が良くなります。

null

null値の代わりに表示する文字列を設定します。デフォルトでは何も表示しません。そのため、よく空の文字列と間違うことがあります。例えば\pset null '(null)'とする人もいます。

numericlocale

valueを指定する場合、それぞれ10進数マーカの左に桁のくくりを分離するロケール固有の文字を表示するonまたは表示しないoffのいずれかでなければなりません。valueを省略した場合、このコマンドは通常出力かロケール固有の数値出力かをトグルします。

pager

問い合わせおよびpsqlのヘルプを出力する際の、ページャプログラムの使用を制御します。環境変数PSQL_PAGERまたはPAGERが設定されている場合、出力は指定したプログラムにパイプで渡されます。設定されていない場合は、プラットフォーム依存のデフォルト(moreなど)が使用されます。

pagerオプションがoffの場合、ページャプログラムは使用されません。pagerオプションがonの場合、ページャは適切な場合、つまり出力先が端末であり、その画面に収まらない場合に使用されます。またpagerオプションはalwaysに設定することもできます。こうすると画面に収まるかどうかに関わらずすべての端末出力でページャが使用されます。valueを指定しない\pset pagerはページャの使用をトグルします。

pager_min_lines

pager_min_linesがページ高より大きな数に設定されている場合、少なくともこれに設定されている行数の出力がなければ、ページャプログラムを呼び出しません。デフォルトの設定は0です。

recordsep

位置揃えなしの出力書式で使用されるレコード（行）の区切り文字を指定します。デフォルトは改行文字です。

recordsep_zero

位置揃えなしの出力書式で使用されるレコードの区切り文字をゼロバイトに指定します。

tableattr (または T)

HTML出力書式では、これはtableタグ内に記述する属性を指定します。これを使用して、例えば、cellpaddingやbgcolorを指定することができます。border属性は既に\pset borderによって処理されているので、このコマンドでborderを指定する必要はないでしょう。valueの指定がない場合、テーブル属性の設定は解除されます。

latex-longtable書式では、これは 左揃えされたデータ型を含む各列の幅の比率を制御します。空白文字で区切られた値のリスト、例えば'0.2 0.2 0.6'として指定します。指定がない出力列は最後に指定された値を使用します。

title (または C)

今後表示される全てのテーブル用にテーブルタイトルを設定します。これは出力に説明のためのタグを付けたい場合に有用です。valueがない場合、タイトルの設定が解除されます。

tuples_only (または t)

valueを指定する場合、それぞれタプルのみの表示を有効にする、onまたは無効にするoffのいずれかでなければなりません。valueを省略した場合、このコマンドはタプルのみの表示と通常表示をトグルします。通常表示では列のヘッダ、タイトル、各種フッタなどのその他の情報が追加されます。タプルのみのモードでは、テーブルの実データのみが表示されます。

unicode_border_linestyle

unicodeの線の形式の境界の形式をsingleまたはdoubleのどちらかに設定します。

unicode_column_linestyle

unicodeの線の形式の列の形式をsingleまたはdoubleのどちらかに設定します。

unicode_header_linestyle

unicodeの線の形式のヘッダの形式をsingleまたはdoubleのどちらかに設定します。

これらの異なる書式がどのように見えるかを示した実例が、下記のExamplesにあります。

ヒント

\psetには各種のショートカットコマンドがあります。\a、\C、\f、\H、\t、\T、\xを参照してください。

\qまたは\quit

psqlプログラムを終了します。スクリプトファイルでは、そのスクリプトの実行のみが終了します。

\qecho text [...]

このコマンドは、\echoと同じです。ただし、出力が\oにより設定された問い合わせ出力チャネルに書き出される点が異なります。

\rまたは\reset

問い合わせバッファをリセット(クリア)します。

\s [filename]

psqlのコマンドラインの履歴をfilenameに出力します。filenameが省略された場合、履歴は標準出力に書き出されます(適切であればページャを使います)。このコマンドは、psqlがReadlineサポートなしの状態でビルドされた場合は利用できません。

\set [name [value [...]]]

psqlの変数nameをvalue、または複数のvalueが与えられた場合はそれらを連結したものに設定します。第一引数しか指定されない場合は、変数に空文字列の値が設定されます。変数を未設定とするには、\unsetコマンドを使用してください。

引数をまったく取らない\setは、現在設定されているpsql変数すべての名前と値を表示します。

変数名には、文字、数字、アンダースコアを使用することができます。詳細は、後述のVariablesを参照してください。変数名は大文字小文字を区別します。

psqlの動作を制御する、あるいは接続状態を表す値に自動的に設定される、という点で特別な変数がいくつかあります。これらの変数については、以下のVariablesに記載されています。

> **注記**
>
> このコマンドはSQLのSETコマンドとは関係ありません。

\setenv name [value]

環境変数nameをvalueに設定します。valueが与えられない場合は、その環境変数を未設定状態にします。以下に例を示します。

```
testdb=> \setenv PAGER less
testdb=> \setenv LESS -imx4F
```

\sf[+] function_description

このコマンドは、CREATE OR REPLACE FUNCTIONコマンドやCREATE OR REPLACE PROCEDUREコマンドの形式で、指定された関数やプロシージャの定義を抽出し表示します。この定義は、\oで設定された現在の問い合わせ出力チャネルに出力されます。

対象の関数は、名前だけまたは、例えばfoo(integer, text)のように名前と引数で指定することができます。同じ名前の関数が複数存在する場合は、引数の型を指定しなければなりません。

コマンド名に+を付けると、出力行に関数本体の先頭行を1行目と数える行番号が付けられます。

他のほとんどのメタコマンドと異なり、行の残り部分はすべて\sfの引数であると常に解釈され、引数内の変数の置換も逆引用符の展開も行われません。

\sv[+] view_name

このコマンドは、指定したビューの定義をCREATE OR REPLACE VIEWコマンドの形式で取得して表示します。定義は現在の問い合わせの出力チャネルに表示されます。これは\oで設定できます。

コマンド名の後に+が付加された場合は、出力行に1から番号が付けられます。

他のほとんどのメタコマンドと異なり、行の残り部分はすべて\svの引数であると常に解釈され、引数内の変数の置換も逆引用符の展開も行われません。

\t

出力列名ヘッダと行数フッタの表示を切り替えます。このコマンドは\pset tuples_onlyと同じで、簡便性のために用意されています。

\T table_options

HTML出力書式におけるtableタグ内部に記述する属性を指定します。このコマンドは\pset tableattr table_optionsと同じ効力を持ちます。

\timing [on | off]

パラメータがある場合、各SQL文にかかる時間の表示の有無をonまたはoffに設定します。パラメータがない場合、表示をonとoffの間で切り替えます。表示はミリセカンド単位です。1秒より長い時間は 分:秒 の形式で表示され、必要なら時間と日のフィールドが追加されます。

\unset name

psql変数nameを未設定状態にします（削除します）。

psqlの動作を制御するほとんどの変数は未設定にすることができず、\unsetはそれをデフォルト値に設定するものとして解釈されます。以下のVariablesを参照してください。

\wまたは\write filename
\wまたは\write |command

現在の問い合わせバッファを、filenameファイルに書き出すか、もしくは、シェルコマンドcommandにパイプで渡します。現在の問い合わせバッファが空の場合、最も最近に実行された問い合わせを書き出します。

引数が|で始まっている場合、行の残りの部分はすべて実行するcommandであると解釈され、その中では変数の置換も逆引用符の展開も行われません。行の残り部分は、単にあるがままにシェルに渡されます。

\warn text [...]

このコマンドは、出力が標準出力ではなくpsqlの標準エラーチャネルに書かれることを除いて\echoと同一です。

\watch [seconds]

中断するか問い合わせが失敗するまで、現在の問い合わせバッファを繰り返し(\gと同じように)実行します。実行の間に指定秒数(デフォルトは2)の休止が入ります。各問い合わせの結果には、\pset titleの文字列(あれば)、問い合わせ開始時の時刻、および遅延間隔を含むヘッダとともに表示されます。

現在の問い合わせバッファが空の場合、最も最近に送信された問い合わせが再実行されます。

\x [on | off | auto]

拡張テーブル形式モードを設定またはトグルします。従って、このコマンドは\pset expandedと同じ効力を持ちます。

\z [pattern]

テーブル、ビュー、シーケンスを、関連付けられているアクセス権限とともに一覧表示します。patternを指定すると、パターンに名前がマッチするテーブル、ビュー、シーケンスのみが表示されます。

これは\dp(「権限の表示(display privileges)」)の別名です。

\! [command]

引数がないときはサブシェルに制御を渡し、サブシェルが終了したらpsqlが再開されます。引数があるときは、シェルコマンドcommandを実行します。

他のほとんどのメタコマンドと異なり、行の残り部分はすべて\!の引数であると常に解釈され、引数内の変数の置換も逆引用符の展開も行われません。行の残りの部分は単にあるがままにシェルに渡されます。

\? [topic]

ヘルプ情報を表示します。オプションのtopicパラメータ(デフォルトはcommands)はpsqlのどの部分を説明するかを選択します。commandsはpsqlのバックスラッシュコマンドについて、optionsはpsqlに渡すことができるコマンド行オプションについて、variablesはpsqlの設定変数についてのヘルプを表示します。

\;

バックスラッシュ-セミコロンは前述のコマンドと同じ位置づけのメタコマンドではありません。そうではなく単にセミコロンを追加処理無しで問い合わせバッファに加えます。

通常、psqlは、コマンド終了のセミコロンに到達したら、更なる入力が現在行に残っていても、SQLコマンドをすぐにサーバに送ります。よって、例えば、

```
select 1; select 2; select 3;
```

上記は3つのSQLコマンドが個々にサーバに送られることになり、各々の結果は続く次コマンドの前に表示されます。しかしながら、\;として挿入されたセミコロンはコマンド処理を誘発せず、その後、そのさらに後のコマンドは事実上結合されて、サーバに一つのリクエストとして送られます。したがって、例えば、

```
select 1\; select 2\; select 3;
```

上記は、バックスラッシュの無いセミコロンに達したときに、3つのSQLコマンドがサーバに単一リクエストでサーバに送られます。文字列にそれを複数トランザクションに分けるための明示的なBEGIN/COMMITが含まれているのでない限り、サーバはこのようなリクエストを単一トランザクションとして実行します。(複数問い合わせの文字列をどのようにサーバが処理するかについて、詳しくは52.2.2.1を参照してください。) psqlは各リクエストに対して受け取った最後の問い合わせ結果だけを出力します。この例では、3つ全てのSELECTが実際に実行されますが、psqlは3だけ出力します。

パターン

各種\dコマンドでは、patternパラメータを渡して、表示するオブジェクト名を指定することができます。最も単純な場合では、パターンが正確にオブジェクト名に一致します。パターン内の文字は、SQLの名前と同様、通常小文字に変換されます。例えば\dt FOOはfooという名前のテーブルを表示します。SQLの名前と同様、パターンを二重引用符で括ることで小文字への変換が取り止められます。二重引用符自体をパターン内に含めなければならない場合、二重引用符で括った文字列の中で二重引用符を二重に記載してください。これもSQLの引用符付き識別子の規則に従ったものです。例えば、\dt "FOO""BAR"はFOO"BARという名前のテーブルを表示します(foo"barではありません)。SQLの名前と異なり、パターンの一部を二重引用符で括ることができます。例えば、\dt FOO"FOO"BARはfooFOObarという名前のテーブルを表示します。

patternパラメータが完全に省略されている場合、\dコマンドは現在のスキーマ検索パス内で可視のオブジェクトを全て表示します。これは*というパターンを使用することと同じです。(オブジェクトを含むスキーマが検索パス上にあり、同じ種類かつ同じ名前のオブジェクトが検索パス上それより前に存在しない場合、そのオブジェクトは可視であるといいます。これは明示的なスキーマ修飾がない名前でオブジェクトを参照できるということと同じです。) 可視か否かに関わらずデータベース内の全てのオブジェクトを表示するには、*.*というパターンを使用します。

パターン内部では、*は(0文字を含む)任意の文字の並びにマッチし、?は任意の1文字にマッチします。(この記法はUnixシェルのファイル名パターンと似ています。) 例えば、\dt int*は、intから始まる名前を持つテーブルを表示します。しかし、二重引用符の中では、*と?はその特別な意味を失い、文字そのものにマッチします。

ドット(.)を含むパターンは、スキーマ名にオブジェクト名が続くパターンとして解釈されます。例えば、\dt foo*.*bar*は、スキーマ名がfooで始まるスキーマ内のテーブル名がbarを含むテーブルを全て表示します。ドットがない場合、パターンは現行のスキーマ検索パス内で可視的なオブジェクトのみにマッチします。ここでも、二重引用符で括られた文字列内のドットは特別な意味を失い、文字そのものにマッチすることになります。

上級者は文字クラス(例えば任意の数にマッチする[0-9])などの正規表現を使用することができます。ほぼすべての正規表現の特殊文字は9.7.3の規定通りに動作しますが、上述のように.が区切り文字となる点、*は正規表現の.*になる点、?が.になる点、$がそのまま扱われる点は例外です。.の代わりに?と、R*の代わりに(R+|)と、R?の代わりに(R|)と記述することで、これらのパターン文字を模擬することができます。通常の正規表現の解釈と異なり、パターンは常に名前全体にマッチするため、$を正規表現文字として扱う必要はありません。(言い替えると、$は自動的にパターンに追加されます。) パターンの適用位置を決められない

場合は、*を先頭や末尾に記載してください。二重引用符の内側では、正規表現の特殊文字はその意味を失い、文字そのものにマッチすることになる点に注意してください。また、正規表現の特殊文字は、演算子名のパターン（つまり\doの引数）では文字そのものにマッチします。

高度な機能

変数

psqlは一般的なUnixコマンドシェルに似た変数の置換機能を提供します。変数とは名前と値の組み合わせです。値として任意の長さの任意の文字列を使用できます。名前は文字(非ラテン文字を含む)、数字、アンダースコアから構成されなければなりません。

変数を設定するには、psqlの\setメタコマンドを以下のように使用します。

```
testdb=> \set foo bar
```

この例では、変数fooをbarという値に設定しています。変数の内容を取り出すには、以下のように変数名の前にコロンを付けます。

```
testdb=> \echo :foo
bar
```

これは通常のSQLコマンド内とメタコマンド内の両方で動作します。後述のSQL Interpolationで詳しく説明します。

第二引数なしで\setを呼び出すと、その変数には空文字列の値が設定されます。変数を未設定状態にする(つまり削除する)ためには、\unsetコマンドを使用してください。すべての変数の値を表示するためには、引数なしで\setを呼び出してください。

> **注記**
>
> \setの引数は他のコマンドと同じ置換規則に従います。このため、\set :foo 'something'のような参照を作成して、Perlにおける「ソフトリンク」やPHPにおける「可変変数」に当たるものを得られます。しかし、残念ながら（あるいは幸運にも）、このような構成をうまく使用する方法はありません。一方、\set bar :fooのようにして変数をコピーするのは、完全に有効な方法です。

これらの変数の多くは、psqlに特別扱いされています。これらは、変数の値を変更することにより、実行時に変更可能なオプションの設定を表現します。またpsqlの変更可能な状態を表現しているものもあります。慣習上、特別視される変数の名前はすべてASCII大文字（と数字とアンダースコア）からなります。将来的な互換性を最大限考慮するために、自分で作成した変数にはこのような変数名を使用しないでください。

psqlの動作を制御する変数は、一般に未設定にしたり、無効な値に設定したりすることができません。\unsetコマンドの実行は許されますが、変数をデフォルト値に設定するものとして解釈されます。第2引数なしの\setコマンドは、onの値を受け付ける制御変数では変数をonに設定するものとして解釈され、それ以外の場合は拒絶されます。また、onとoffの値を受け付ける制御変数では、trueやfalseなどそれ以外の論理値の共通で使われる綴りも受け付けます。

特別に扱われる変数を以下に示します。

AUTOCOMMIT

この変数の値がonの場合（これがデフォルトです）、各SQLコマンドの実行が成功すると、自動的にコミットされます。コミットを延期するには、BEGINもしくはSQLのSTART TRANSACTIONコマンドを入力する必要があります。値がoffもしくは未設定の場合、明示的にCOMMITもしくはENDを発行するまで、SQLコマンドはコミットされません。自動コミット無効モードでは、トランザクションブロック以外でコマンドが発行されると、そのコマンドを実行する前に、自動的にBEGINコマンドが発行されます（ただし、そのコマンド自体がBEGINコマンドやその他のトランザクション制御コマンドである場合、トランザクションブロック内で実行することができないコマンド（VACUUMなど）である場合は除きます）

注記

自動コミット無効モードでは、ABORTやROLLBACKを発行して、明示的に失敗したトランザクションを放棄しなければなりません。また、コミットせずにセッションを終了した場合は、作業が失われてしまうので注意してください。

注記

PostgreSQLは、伝統的に自動コミット有効モードで動作していましたが、自動コミット無効モードの方がよりSQLの仕様に近いものです。自動コミット無効モードは、システム全体に対するpsqlrcファイル、もしくは、個人用の.psqlrcファイルで設定すれば実現できます。

COMP_KEYWORD_CASE

SQLキーワードを補完する時に大文字小文字のどちらを使用するかを決定します。lowerまたはupperが設定された場合、補完された単語はそれぞれ小文字または大文字になります。preserve-lowerまたはpreserve-upper（デフォルト）が設定された場合、補完された単語は入力済みの文字の大文字小文字を引き継ぎますが、何も入力されていない場合はそれぞれ小文字または大文字に補完されます。

DBNAME

現在接続しているデータベース名です。この変数は（プログラム起動時も含め）データベースに接続する度に設定されますが、変更したり、未設定にすることもできます。

ECHO

allに設定された場合、空でない全ての入力行は、標準出力に書き出されます。(これは対話式に読み込まれる行には適用されません。) この動作をプログラム起動時に設定するには、-aスイッチを使用してください。queriesに設定された場合、psqlは各問い合わせがサーバに送信されるときに表示します。この動作を選択するオプションは-eです。errorsに設定された場合、失敗した問い合わせのみが標準エラー出力に出力されます。この動作に対応するオプションは-bです。none（デフォルトです）に設定された場合、どの問い合わせも表示されません。

ECHO_HIDDEN

この変数がonに設定されている場合、バックスラッシュコマンドがデータベースに問い合わせを行う時、最初にその問い合わせが表示されます。この機能は、PostgreSQL内部動作について調べたり、自作プ

ログラム内で同様の関数機能を用意したりするのに役立つでしょう。(この動作をプログラム起動時に選択するには-Eスイッチを使用してください)。この変数をnoexecという値に設定した場合、問い合わせは実際にサーバに送信、実行されずに、単に表示されるだけになります。デフォルト値はoffです。

ENCODING

現在のクライアント側の文字セット符号化方式です。これは(プログラムの起動時を含め)データベースに接続する度に、また符号化方式を\encodingで変更した時に設定されますが、変更したり、未設定にすることができます。

ERROR

最後のSQL問い合わせが失敗したならtrue、成功したならfalse。SQLSTATEも参照してください。

FETCH_COUNT

この変数が0より大きな整数値に設定されている場合、SELECT問い合わせの結果は、指定した行数の集合として取り出され、表示されます。デフォルトの動作では、表示する前にすべての結果が取り出されます。したがって、結果セットの大きさに関係なくメモリの使用量が限定されます。この機能を有効とする場合に100から1000までの値がよく使用されます。この機能を使用する際には、既に一部の行が表示されている場合、問い合わせが失敗する可能性があることに注意してください。

> **ヒント**
>
> 任意の出力書式でこの機能を使用することができますが、デフォルトのaligned書式は適していません。FETCH_COUNT行のグループそれぞれが別々に整形されてしまい、行のグループによって列幅が異なることになるためです。他の出力書式は適切に動作します。

HIDE_TABLEAM

この変数がtrueに設定されていれば、テーブルのアクセスメソッドの詳細は表示されません。これは主にリグレッションテストで有用です。

HISTCONTROL

この変数をignorespaceに設定した場合、空白文字から始まる行は履歴リストには入りません。ignoredupsに設定した場合、直前の履歴と同じ行は履歴リストに入りません。ignorebothに設定した場合は、上記の2つを組み合わせたものになります。none(デフォルトです)に設定した場合は、対話モードで読まれる全ての行が履歴リストに保存されます。

> **注記**
>
> この機能はBashの機能を真似たものです。

HISTFILE

履歴を格納するために使用されるファイルの名前です。未設定にすると、ファイル名は環境変数PSQL_HISTORYから取得されます。それも設定されていない場合、デフォルトは~/.psql_history、またはWindowsでは%APPDATA%\postgresql\psql_historyです。例えば、次のように入力します。

```
\set HISTFILE ~/.psql_history- :DBNAME
```

> **注記**
>
> この機能はBashの機能を真似たものです。

HISTSIZE

コマンド履歴に保存するコマンドの最大数（デフォルトは500）です。負の値に設定すると、制限がなくなります。

> **注記**
>
> この機能はBashの機能を真似たものです。

HOST

接続中のデータベースサーバホストです。この変数は（プログラム起動時も含め）データベースに接続する度に設定されますが、変更したり、未設定にすることもできます。

IGNOREEOF

この変数を1以下に設定すると、対話式セッションにEOF文字（通常**Control**+**D**）が送信された時、psqlが終了します。1より大きな数値を設定すると、対話的セッションを終了するには、指定された数だけ、続けてEOF文字を送信しなければなりません。数値以外を設定した場合は、10と解釈されます。デフォルトは0です。

> **注記**
>
> この機能はBashの機能を真似たものです。

LASTOID

INSERTや\lo_insertコマンドによって返された、最後に影響を受けたOIDの値です。この変数は、次のSQLコマンドの結果が表示されるまでの間のみ保証されています。バージョン12からPostgreSQLサーバはOIDシステム列をサポートしませんので、そのようなサーバを対象とした場合INSERTの後は、LASTOIDは常に0です。

LAST_ERROR_MESSAGE
LAST_ERROR_SQLSTATE

現在のpsqlセッションの直近の失敗した問い合わせに対する主エラーメッセージと関連するSQLSTATEコード。あるいは、現在セッションでエラーが無い場合、空文字列と00000。

ON_ERROR_ROLLBACK

onに設定されている場合、トランザクションブロック内である文がエラーとなった時に、そのエラーは無視され、トランザクションは継続します。interactiveに設定されている場合、対話式セッション内の場

合にのみエラーは無視されます。スクリプトファイルを読み込んでいる場合は無視されません。off（デフォルトです）に設定されている場合、トランザクションブロック内の文がエラーになると、トランザクション全体をアボートします。エラーロールバックのモードは、トランザクションブロック内で各コマンドの実行直前に暗黙的なSAVEPOINTを行い、コマンドが失敗した時にこのセーブポイントにロールバックすることで実現されています。

ON_ERROR_STOP

デフォルトではエラーの後もコマンド処理は続行されます。この変数がonに設定されていると、代わりに即座に停止します。対話モードではpsqlはコマンドプロンプトに戻ります。これ以外ではpsqlは終了し、 エラーコード1を返す致命的エラー条件と区別できるように、エラーコード3を返します。どちらの場合でも、現在実行中のスクリプト（トップレベルのスクリプト、もしあれば関連性を持つ他のスクリプトすべて）は即座に終了します。トップレベルのコマンド文字列が複数のSQLコマンドを含む場合、その時点のコマンドで処理は終了します。

PORT

接続中のデータベースサーバのポートです。この変数は（プログラム起動時も含め）データベースに接続する度に設定されますが、変更することも未設定にすることもできます。

PROMPT1
PROMPT2
PROMPT3

これらの変数は、psqlが発行するプロンプトの見た目を指定します。後述のPromptingを参照してください。

QUIET

この変数をonに設定することはコマンドラインオプション-qと同じ効力を持ちます。対話式モードではあまり役立ちません。

ROW_COUNT

最後のSQL問い合わせにより、返された、あるいは、影響をうけた行数。あるいは、問い合わせが失敗したか行数が報告されていない場合、0。

SERVER_VERSION_NAME
SERVER_VERSION_NUM

文字列としてのサーバーのバージョン番号、例えば9.6.2、10.1、11beta1など、および数値形式でのバージョン番号、例えば90602、100001などです。これらは（プログラムの起動時を含め）データベースに接続する度に設定されますが、変更することも未設定にすることもできます。

SHOW_CONTEXT

この変数は値never、errors、あるいはalwaysに設定することができ、CONTEXTフィールドがサーバからのメッセージに表示されるかどうかを制御します。デフォルトはerrorsです（CONTEXTはエラーメッセージ内では表示されますが、注意や警告メッセージでは表示されません）。この設定はVERBOSITYがterseまたはsqlstateに設定されている場合は効果がありません。（\errverboseも参照してください。こちらは受け取ったばかりのエラーについて、冗長なメッセージが必要なときに使えます。）

SINGLELINE

この変数をonに設定することはコマンドラインオプション-Sと同じ効力を持ちます。

SINGLESTEP

この変数をonに設定することはコマンドラインオプション-sと同じ効力を持ちます。

SQLSTATE

最後のSQL問い合わせの失敗に関するエラーコード（付録Aを参照）、あるいは、SQLが成功した場合には00000。

USER

接続中のデータベースユーザです。この変数は（プログラム起動時も含め）データベースに接続する度に設定されますが、変更することも未設定にすることもできます。

VERBOSITY

この変数をdefault、verbose、terse、sqlstateのいずれかに設定することで、エラー報告の冗長性を制御できます。（\errverboseも参照してください。こちらは受け取ったばかりのエラーについて、冗長なメッセージが必要なときに使えます。）

VERSION
VERSION_NAME
VERSION_NUM

これらの変数はプログラムの起動時にpsqlのバージョンを表すために設定され、それぞれ冗長な文字列、短い文字列（例：9.6.2、10.1、11beta1）、数字（例：90602、100001）です。これらは変更することも未設定にすることもできます。

SQL差し替え

psqlの変数には、通常のSQL文やメタコマンドの引数の中で使用（「差し替え：interpolate」）できるという重要な機能があります。さらにpsqlは、SQLリテラルと識別子として使用される変数の値が適切に引用符付けされていることを保証する機能を提供します。引用符付けをまったく行わずに差し替えるための構文は、変数名の前にコロン(:)を付けることです。以下に例を示します。

```
testdb=> \set foo 'my_table'
testdb=> SELECT * FROM :foo;
```

この例では、問い合わせはmy_tableテーブルに対して行われます。これが安全ではない場合があることに注意して下さい。変数の値はそのままコピーされるので、対応のとれていない引用符やバックスラッシュコマンドさえも含めることができます。挿入した場所で変数が展開された時に、確実に正しい意味になるようにしなければなりません。

値がSQLリテラルや識別子内で使用される場合、それが引用符付けされるように調整することがもっとも安全です。SQLリテラルとして変数値を引用符付けするためには、コロンの後に変数名を単一引用符で括って記述してください。SQL識別子として値を引用符付けするためには、コロン後に変数名を二重引用符で括っ

て記述してください。これらの式は正しく引用符と変数値内に埋め込まれた特殊文字を扱います。前の例は以下のように記述することでより安全になります。

```
testdb=> \set foo 'my_table'
testdb=> SELECT * FROM :"foo";
```

変数差し替えは、引用符付けされたSQLリテラルと識別子の中では行われません。したがって':foo'などの式は、変数の値から引用符付けしたリテラルを生成するようには動作しません。(値の中に埋め込まれた引用符を正しく取り扱えませんので、もし動作したとしたら安全ではありません。)

この機能の有効な利用方法の例は、ファイルの内容をテーブル列にコピーする場合も利用することができます。その際は、ファイルをまず変数に読み込み、引用符付けした文字列として変数名を差し替えます。

```
testdb=> \set content `cat my_file.txt`
testdb=> INSERT INTO my_table VALUES (:'content');
```

(my_file.txtにNULバイトが含まれている場合、これはうまく動作しないことに注意してください。psqlは変数値内に埋め込まれたNULバイトをサポートしません。)

コロン(:)もSQLコマンド内で正規に使用できますので、指定した変数が現在設定されていない場合、差し替え時の見かけの置換(:name、:'name'、:"name")は行われません。コロンをバックスラッシュでエスケープすれば、常に差し替えから保護することができます。

:{?name}特別構文は、その変数が存在しているかどうかに応じてTRUEかFALSEを返します。従って、コロンがバックスラッシュでエスケープされていない限り、常に置き換えられます。

変数用のコロン構文は、ECPGのような組み込みの問い合わせ言語用の標準SQLとして規定されています。配列の一部の切り出し、および型キャスト用のコロン構文はPostgreSQLの拡張であり、標準での使用方法と競合することがあります。SQLリテラルまたは識別子として変数の値をエスケープさせる引用符付きコロン構文はpsqlの拡張です。

プロンプト

psqlが発行するプロンプトは好みに応じてカスタマイズできます。PROMPT1、PROMPT2、PROMPT3という3つの変数はプロンプトの表示内容を示す文字列や特別なエスケープシーケンスを持ちます。プロンプト1はpsqlが新しいコマンドを受け付ける際に発行される通常のプロンプトです。プロンプト2は、例えばコマンドがセミコロンで終わっていない、または、引用符が閉じていないなど、コマンドの入力中にさらなる入力が期待される際に発行されます。プロンプト3はSQLのCOPY FROM STDINコマンドを実行中で、端末上で行の値の入力が必要な際に発行されます。

選択されたプロンプト変数の値はそのまま文字として表示されます。ただし、パーセント(%)が含まれる場合は例外です。この場合は、次の文字に従って、特定のテキストに置換されます。置換対象として定義されているのは次のものです。

%M

データベースサーバの(ドメイン名付きの)完全なホスト名です。その接続がUnixドメインソケットの場合は[local]となります。ただし、Unixドメインソケットがコンパイル時に設定したデフォルトの場所に存在しない場合は、[local:/dir/name]となります。

%m

最初のドット以降を省略したデータベースサーバのホスト名です。その接続がUnixドメインソケットの場合は[local]となります。

%>

データベースサーバが監視するポート番号です。

%n

データベースセッションユーザの名前です（この値の展開結果は、SET SESSION AUTHORIZATIONコマンドの実行によってデータベースセッション中に変わることがあります）。

%/

接続中のデータベース名です。

%~

デフォルトデータベースの場合に~（チルダ）が出力される点を除いて、%/と同じです。

%#

セッションユーザがデータベーススーパーユーザである場合は#、それ以外の場合は>となります（この値の展開結果は、SET SESSION AUTHORIZATIONコマンドの実行によってデータベースセッション中に変わることがあります）。

%p

現在接続しているバックエンドのプロセスIDです。

%R

プロンプト1の場合、通常は=ですが、条件ブロックの使われない部分では@、シングル行モードでは^、また、データベースとの接続が切れたセッションでは!になります（\connectが失敗した場合に発生します）。プロンプト2の場合、%Rは、なぜpsqlがさらなる入力を要求しているかによって決まる文字に置き換えられます。これは、単にコマンドがまだ終了していない場合は-ですが、/* ... */のコメントがまだ終了していない場合は*、引用符付きの文字列が終了していない場合は単一引用符、引用符付きの識別子が終了していない場合は二重引用符、ドル引用文字列が終了していない場合はドル記号、そして閉じられていない左括弧がある場合は(となります。プロンプト3の場合、%Rに対しては何も表示されません。

%x

トランザクションの状態です。トランザクションブロックの外にいる場合は空文字列に、トランザクションブロックの中にいる場合は*に、失敗したトランザクションブロックの中にいる場合は!に、（接続されていないなど）トランザクションの状態が不定の場合は?になります。

%l

現在の文の内部での行番号で、1から始まります。

%digits

指定された8進の数値コードの文字に置換されます。

%:name:

psqlの変数nameの値です。詳細は上記のVariablesを参照してください。

%`command`

通常の「逆引用符」による置き換えと同様で、commandの出力です。

%[... %]

プロンプトには端末制御文字を含めることができます。具体的には、色、背景、プロンプトテキストの様式の変更、端末ウィンドウのタイトルの変更などが指定できます。Readlineの行編集機能を適切に動作させるためには、印字されない制御文字を%[と%]で囲んで、不可視であることを明示しなければなりません。この記号の組み合わせはプロンプト内に複数記述することができます。以下に例を示します。

```
testdb=> \set PROMPT1 '%[%033[1;33;40m%]%n@%/%R%[%033[0m%]%# '
```

これにより、VT100互換のカラー端末では、太字フォントで(1;)、黒地に黄色の(33;40)プロンプトが表示されます。

%w

PROMPT1の直近の出力と同じ幅の空白です。これは、複数行の文が最初の行とそろっているが第2のプロンプトが見えないようにするためにPROMPT2の設定として使えます。

プロンプトにパーセント記号を入れる場合は、%%と記述してください。デフォルトのプロンプトは、プロンプト1と2が'%/%R%x%# '、プロンプト3が'>> 'です。

注記

この機能はtcshの機能を真似たものです。

コマンドライン編集

psqlは行内編集や繰り返し入力が簡便になるようにReadlineライブラリをサポートしています。コマンド履歴は、psqlの終了時に自動的に保存され、psqlの起動時に再読み込みされます。タブによる補完もサポートされていますが、SQLのパーサとしてコマンドを解釈して判断してくれるわけではありません。タブによる補完によって生成される問い合わせは他のSQLコマンド、例えばSET TRANSACTION ISOLATION LEVELと干渉することもあります。タブによる補完を何らかの事情により使用したくなければ、ホームディレクトリ内の.inputrcというファイルに以下のように書き込むことで無効にできます。

```
$if psql
set disable-completion on
$endif
```

（これはpsqlの機能ではなく、Readlineの機能です。詳細についてはReadlineのドキュメントを参照してください）。

環境

COLUMNS

\pset columnsがゼロの場合、wrapped書式の幅、および、幅の広い出力がページャを必要とするかどうかを決める幅を制御します。また自動拡張モードでは縦書式に切り替えるべきかどうかを制御します。

PGDATABASE
PGHOST
PGPORT
PGUSER

デフォルトの接続パラメータです（33.14を参照）。

PG_COLOR

診断メッセージで色を使うかどうかを指定します。可能な値はalways、auto、neverです。

PSQL_EDITOR
EDITOR
VISUAL

\eコマンド、\efコマンド、\evコマンドで使用されるエディタです。変数はこの順に検索され、設定された最初のものが使用されます。いずれも設定されていない場合、デフォルトでUnixシステムではviを、Windowsシステムではnotepad.exeを使用します。

PSQL_EDITOR_LINENUMBER_ARG

\e、\efまたは\evが行番号引数を付けて使用された場合、この変数は、ユーザのエディタに開始行番号を渡すために使用されるコマンドライン引数を指定します。Emacsまたはviのようなエディタでは、これはプラス(+)記号です。オプション名と行番号の間に空白文字が必要な場合は、変数の値の最後に空白文字を含めてください。以下に例を示します。

```
PSQL_EDITOR_LINENUMBER_ARG='+'
PSQL_EDITOR_LINENUMBER_ARG='--line '
```

Unixシステム上のデフォルトは+です。（デフォルトのエディタviに対応するものですが、他のよく使われる多くのエディタでも役に立ちます。）一方Windowsシステムではデフォルトはありません。

PSQL_HISTORY

コマンド履歴ファイルの場所を指定します。チルダ(~)展開が行われます。

PSQL_PAGER
PAGER

問い合わせ結果が画面に入り切らない場合、このコマンドによって結果をパイプします。一般的に指定される値は、more、またはlessです。ページャの使用を無効にするにはPSQL_PAGERやPAGERを空文字列に

するか、\psetコマンドのページャ関連のオプションを調整します。これらの変数は列挙した順で検査され、最初の設定されているものが使われます。いずれも設定されていない場合、デフォルトで大部分のプラットフォームではmoreが使われ、しかし、Cygwinではlessが使われます。

PSQLRC

ユーザの.psqlrcファイルの場所を指定します。チルダ(~)展開が行われます。

SHELL

\!コマンドが実行するコマンドです。

TMPDIR

一時ファイルを格納するディレクトリです。デフォルトは/tmpです。

このユーティリティは、他のほとんどのPostgreSQLユーティリティと同様、libpqでサポートされる環境変数を使用します(33.14を参照してください)。

ファイル

psqlrc and ~/.psqlrc

-Xオプションが渡されない場合、psqlは、データベースに接続した後、通常のコマンドを受け付け始める前に、システム全体用の開始ファイル(psqlrc)のコマンドを、続いてユーザ用の開始ファイル(~/.psqlrc)のコマンドを読み込み、実行しようとします。これらのファイルは、\setやSETコマンドを使用して、好みに応じたクライアントやサーバを設定するために使用することができます。

システム全体の開始ファイルはpsqlrcという名前で、インストレーションの「システム設定」ディレクトリの中で探されます。このディレクトリを特定するにはpg_config --sysconfdirを実行するのが最も確実です。デフォルトでは、PostgreSQLの実行ファイルを含むディレクトリからの相対パスで../etc/になります。このディレクトリの名前は環境変数PGSYSCONFDIRを使って明示的に設定することができます。

ユーザ用の開始ファイルは.psqlrcという名前で、実行しているユーザのホームディレクトリの中で探されます。Windowsではそのような概念がないので、個人用の開始ファイルは%APPDATA%\postgresql\psqlrc.confという名前になります。ユーザ用の開始ファイルは環境変数PSQLRCで明示的に設定することができます。

システム全体用の開始ファイルとユーザ用の開始ファイルに、例えば、~/.psqlrc-9.2や~/.psqlrc-9.2.5のように、ハイフン記号とPostgreSQLのメジャーリリース番号またはマイナーリリース番号をファイル名に付加することで、特定バージョンのpsql向けのファイルとすることができます。マッチするバージョンのファイルはバージョン指定のないファイルよりも優先して読み込まれます。

.psql_history

コマンドライン履歴はファイル~/.psql_history、Windowsの場合は%APPDATA%\postgresql\psql_historyに格納されます。

履歴ファイルの場所はpsql変数のHISTFILEまたは環境変数PSQL_HISTORYを介して明示的に設定することができます。

注釈

- psqlは、同じまたはより古いメジャーバージョンのサーバと稼働させることが最善です。特にバックスラッシュコマンドは、サーバがpsql自身のバージョンより新しいと失敗しやすくなります。\d系列のバックスラッシュコマンドは7.4までさかのぼるバージョンのサーバで動作するはずですが、psql自身よりもサーバが新しい場合は、必ずしもそうではありません。SQLコマンドの実行ならびに問い合わせ結果の表示といった一般的な機能はより新しいメジャーバージョンのサーバとでも動作するはずですが、すべての場合において保証することはできません。

  メジャーバージョンが異なる複数のサーバとの接続のためにpsqlを使用したいのであれば、psqlの最新版を使用することを勧めます。他の方法として、各メジャーバージョンのpsqlのコピーを保持し、確実にそれぞれのサーバに対応するバージョンを使用することができます。しかし実際には、この複雑さを追加することは必要ではないはずです。

- PostgreSQLの9.6より前では、-cオプションが-X (--no-psqlrc)を暗示しましたが、現在ではそうなっていません。

- PostgreSQLの8.4より前では、psqlで1文字のバックスラッシュコマンドの最初の引数をコマンドの直後に空白文字を挟むことなく記述できました。現在では何らかの空白が必要になっています。

Windowsユーザ向けの注意

psqlは「コンソールアプリケーション」としてコンパイルされます。Windowsのコンソールウィンドウは、システムの他の部分とは異なる符号化方式を使用しているので、psqlで8ビット文字を使用する時には特別な配慮が必要です。psqlは、コンソール用コードページとして問題があることを検出すると、起動時に警告を発します。コンソール用コードページを変更するためには、以下の2つが必要です。

- **cmd.exe /c chcp 1252**と入力して、コードページを設定します（1252はドイツ圏における適切なコードページです。システムに合わせて変更してください）。Cygwinを使用しているのであれば、このコマンドを/etc/profileに追加してください。

- コンソール用フォントをLucida Consoleに設定してください。ラスタフォントは、ANSIコードページでは正しく動作しないためです。

例

最初に、複数行にわたるコマンドの入力例を示します。プロンプトの変化に注意してください。

```
testdb=> CREATE TABLE my_table (
testdb(>  first integer not null default 0,
testdb(>  second text)
testdb-> ;
```

```
CREATE TABLE
```

さて、ここでテーブル定義を再度確認してみます。

```
testdb=> \d my_table
              Table "public.my_table"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 first  | integer |           | not null | 0
 second | text    |           |          |
```

次に、プロンプトをもっと面白いものに変更してみます。

```
testdb=> \set PROMPT1 '%n@%m %~%R%# '
peter@localhost testdb=>
```

テーブルにデータを入力したものと考えてください。データを見る場合は次のようにします。

```
peter@localhost testdb=> SELECT * FROM my_table;
 first | second
-------+--------
     1 | one
     2 | two
     3 | three
     4 | four
(4 rows)
```

\psetコマンドを使って、このテーブルの表示を違うタイプに変更することができます。

```
peter@localhost testdb=> \pset border 2
Border style is 2.
peter@localhost testdb=> SELECT * FROM my_table;
+-------+--------+
| first | second |
+-------+--------+
|     1 | one    |
|     2 | two    |
|     3 | three  |
|     4 | four   |
+-------+--------+
(4 rows)

peter@localhost testdb=> \pset border 0
Border style is 0.
peter@localhost testdb=> SELECT * FROM my_table;
first second
----- ------
```

```
    1 one
    2 two
    3 three
    4 four
(4 rows)

peter@localhost testdb=> \pset border 1
Border style is 1.
peter@localhost testdb=> \pset format csv
Output format is csv.
peter@localhost testdb=> \pset tuples_only
Tuples only is on.
peter@localhost testdb=> SELECT second, first FROM my_table;
one,1
two,2
three,3
four,4
peter@localhost testdb=> \pset format unaligned
Output format is unaligned.
peter@localhost testdb=> \pset fieldsep '\t'
Field separator is "    ".
peter@localhost testdb=> SELECT second, first FROM my_table;
one     1
two     2
three   3
four    4
```

その他の方法として、短縮されたコマンドを使用してみます。

```
peter@localhost testdb=> \a \t \x
Output format is aligned.
Tuples only is off.
Expanded display is on.
peter@localhost testdb=> SELECT * FROM my_table;
-[ RECORD 1 ]-
first  | 1
second | one
-[ RECORD 2 ]-
first  | 2
second | two
-[ RECORD 3 ]-
first  | 3
second | three
-[ RECORD 4 ]-
first  | 4
second | four
```

また、この出力書式オプションは\gを使って1つの問い合わせにだけ設定できます。

```
peter@localhost testdb=> SELECT * FROM my_table
peter@localhost testdb-> \g (format=aligned tuples_only=off expanded=on)
-[ RECORD 1 ]-
first  | 1
second | one
-[ RECORD 2 ]-
first  | 2
second | two
-[ RECORD 3 ]-
first  | 3
second | three
-[ RECORD 4 ]-
first  | 4
second | four
```

問い合わせの結果が適していれば、以下のように\crosstabviewコマンドを使用してクロス表形式で表示することができます。

```
testdb=> SELECT first, second, first > 2 AS gt2 FROM my_table;
 first | second | gt2
-------+--------+-----
     1 | one    | f
     2 | two    | f
     3 | three  | t
     4 | four   | t
(4 rows)

testdb=> \crosstabview first second
 first | one | two | three | four
-------+-----+-----+-------+------
     1 | f   |     |       |
     2 |     | f   |       |
     3 |     |     | t     |
     4 |     |     |       | t
(4 rows)
```

この2つ目の例では、掛け算の表を、行は逆順にソートし、列はそれとは別に昇順に示しています。

```
testdb=> SELECT t1.first as "A", t2.first+100 AS "B", t1.first*(t2.first+100) as "AxB",
testdb(> row_number() over(order by t2.first) AS ord
testdb(> FROM my_table t1 CROSS JOIN my_table t2 ORDER BY 1 DESC
testdb(> \crosstabview "A" "B" "AxB" ord
 A | 101 | 102 | 103 | 104
---+-----+-----+-----+-----
```

```
 4 | 404 | 408 | 412 | 416
 3 | 303 | 306 | 309 | 312
 2 | 202 | 204 | 206 | 208
 1 | 101 | 102 | 103 | 104
(4 rows)
```

reindexdb

reindexdb — PostgreSQLデータベースのインデックスを再作成する

概要

reindexdb [connection-option...] [option...] [**--schema** | **-S** schema] ... [**--table** | **-t** table] ... [**--index** | **-i** index] ... [dbname]

reindexdb [connection-option...] [option...] **--all** | **-a**

reindexdb [connection-option...] [option...] **--system** | **-s** [dbname]

説明

reindexdbは、PostgreSQLデータベース内のインデックスを再作成するユーティリティです。

reindexdbは、SQLコマンドREINDEXのラッパです。このユーティリティを使用しても他の方法でサーバにアクセスしても、データベースインデックスの再作成には実質的な違いはありません。

オプション

reindexdbは以下のコマンドライン引数を受け付けます。

-a
--all

すべてのデータベースのインデックスを再作成します。

--concurrently

CONCURRENTLYオプションを使います。REINDEXを参照してください。このオプションの注意がすべて詳しく説明されています。

[**-d**] dbname
[**--dbname=**]dbname

-a/--allが使用されていない場合に、インデックス再作成を行なうデータベースの名前を指定します。これが指定されていなければ、環境変数PGDATABASEからデータベース名が決まります。これも設定されていない場合は、接続時に指定したユーザ名が使用されます。dbnameは接続文字列でも構いません。その場合、接続文字列パラメータは衝突するコマンドラインオプションに優先します。

-e
--echo

reindexdbが生成しサーバに送信するコマンドを表示します。

-i index
--index=index

indexのみを再作成します。複数の-iスイッチを記述することで、複数のインデックスを再作成することができます。

-j njobs
--jobs=njobs

njobs個のコマンドを同時に実行することで、reindexコマンドを並列で実行します。このオプションは処理時間を短縮することもありますが、データベースサーバの負荷も増加します。

reindexdbはデータベースに対するnjobs個の接続を開くので、max_connectionsの設定は、これらの接続を許容するだけ十分に大きくしてください。

このオプションは--indexや--systemオプションと非互換であることに注意してください。

-q
--quiet

進行メッセージを表示しません。

-s
--system

データベースのシステムカタログのインデックスを再作成します。

-S schema
--schema=schema

schemaのみのインデックスを再作成します。複数の-Sスイッチを指定することで、複数のスキーマのインデックスを再作成できます。

-t table
--table=table

tableのインデックスのみを再作成します。複数の-tを記述することで、複数のテーブルのインデックスを再作成することができます。

-v
--verbose

処理中に詳細な情報を表示します。

-V
--version

reindexdbのバージョンを表示し、終了します。

-?
--help

reindexdbコマンドライン引数の使用方法を表示し、終了します。

また、reindexdbは、接続パラメータとして以下のコマンドライン引数を受け付けます。

-h host
--host=host

サーバが稼働しているマシンのホスト名を指定します。ホスト名がスラッシュから始まる場合、Unixドメインソケット用のディレクトリとして使用されます。

-p port
--port=port

サーバが接続を監視するTCPポートもしくはUnixドメインソケットファイルの拡張子を指定します。

-U username
--username=username

接続するユーザ名を指定します。

-w
--no-password

パスワードの入力を促しません。サーバがパスワード認証を必要とし、かつ、.pgpassファイルなどの他の方法が利用できない場合、接続試行は失敗します。バッチジョブやスクリプトなどパスワードを入力するユーザが存在しない場合にこのオプションは有用かもしれません。

-W
--password

データベースに接続する前に、reindexdbは強制的にパスワード入力を促します。

サーバがパスワード認証を要求する場合reindexdbは自動的にパスワード入力を促しますので、これが重要になることはありません。しかし、reindexdbは、サーバにパスワードが必要かどうかを判断するための接続試行を無駄に行います。こうした余計な接続試行を防ぐために-Wの入力が有意となる場合もあります。

--maintenance-db=dbname

-a/--allが使われている場合、どのデータベースを再インデックス付けしなければならないかを見つけ出すために接続するデータベースの名前を指定します。指定されなければpostgresデータベースが使用され、もし存在しなければtemplate1が使用されます。これは接続文字列でも構いません。その場合、接続文字列パラメータは衝突するコマンドラインオプションに優先します。また、データベース名自身以外の接続文字列パラメータは他のデータベースに接続する際に再利用されます。

環境

PGDATABASE
PGHOST
PGPORT
PGUSER

デフォルトの接続パラメータです。

PG_COLOR

診断メッセージで色を使うかどうかを指定します。可能な値はalways、auto、neverです。

また、このユーティリティは、他のほとんどのPostgreSQLユーティリティと同様、libpqでサポートされる環境変数を使用します（33.14を参照してください）。

診断

問題が発生した場合、考えられる原因とエラーメッセージについての説明はREINDEXとpsqlを参照してください。データベースサーバは、指定したホストで稼働している必要があります。また、libpqフロントエンドライブラリのデフォルトの設定や環境変数が適用されることに注意してください。

注釈

reindexdbの実行中にはPostgreSQLサーバに何度も接続しなければならないことがありますが、この場合その度にパスワードが必要です。そのような場合は~/.pgpassファイルを用意しておくと便利です。詳細は33.15を参照してください。

例

データベースtestのインデックスを再作成します。

```
$ reindexdb test
```

abcdというデータベース内のテーブルfooのインデックスとインデックスbarを再作成します。

```
$ reindexdb --table=foo --index=bar abcd
```

関連項目

REINDEX

vacuumdb

vacuumdb — PostgreSQLデータベースの不要領域の回収と解析を行う

概要

vacuumdb [connection-option...] [option...] [**--table** | **-t** table [(column [,...])]] ... [dbname]

vacuumdb [connection-option...] [option...] **--all** | **-a**

説明

vacuumdbは、PostgreSQLデータベースの不要領域のクリーンアップを行うユーティリティです。また、vacuumdbはPostgreSQLの問い合わせオプティマイザが使用する内部的な統計情報も生成します。

vacuumdbは、SQLコマンドVACUUMのラッパです。このユーティリティを使っても、これ以外の方法でサーバにアクセスしてバキュームや解析を行っても特に違いは生じません。

オプション

vacuumdbでは、下記のコマンドライン引数を指定できます。

-a
--all

全てのデータベースに対してバキュームを行います。

[-d] dbname
[--dbname=]dbname

-a(または--all)も指定されていない場合、不要領域のクリーンアップ、または、解析を行うデータベース名を指定します。データベース名が指定されていない場合は、データベース名は環境変数PGDATABASEから読み取られます。この変数も設定されていない場合は、接続時に指定したユーザ名が使用されます。dbnameは接続文字列に出来ます。その場合、接続文字列パラメータは競合するコマンドラインオプションを上書きします。

--disable-page-skipping

可視性マップの内容に基づいてページを飛ばすことのないようにします。

> **注記**
>
> このオプションはPostgreSQL 9.6以降が動作しているサーバでのみ利用可能です。

-e
--echo

vacuumdbが生成し、サーバに送るコマンドをエコー表示します。

-f
--full

「完全な(full)」クリーンアップを行います。

-F
--freeze

積極的にタプルを「凍結」します。

-j njobs
--jobs=njobs

njobs個のコマンドを同時に実行することで、vacuumまたはanalyzeコマンドを並列で実行します。このオプションは処理時間を短縮することもありますがデータベースサーバの負荷も増加します。

vacuumdbはデータベースに対するnjobs個の接続を開くので、max_connectionsの設定が、これらの接続を許容するだけ十分に大きくしてください。

このモードを-f(FULL)オプションと一緒に使うと、一部のシステムカタログが並列処理されてデッドロックのエラーを起こす場合があることに注意してください。

--min-mxid-age mxid_age

マルチトランザクションIDの年代が少なくともmxid_ageであるテーブルに対してのみ、バキュームもしくは解析コマンドを実行します。この設定は、マルチトランザクションIDの周回を防ぐためテーブルに優先順位を付けて処理するのに有用です(24.1.5.1を参照してください)。

このオプションの目的のため、リレーションのマルチトランザクションIDの年代は、主であるリレーションの年代と、存在するなら、関連するTOASTテーブルの年代のうち最大のものです。vacuumdbにより発行されたコマンドも、必要であればリレーションのTOASTテーブルを処理しますので、別々に分けて考える必要はないです。

> **注記**
>
> このオプションはPostgreSQL 9.6以降が動作しているサーバでのみ利用可能です。

--min-xid-age xid_age

トランザクションIDの年代が少なくともxid_ageであるテーブルに対してのみ、バキュームもしくは解析コマンドを実行します。この設定は、トランザクションIDの周回を防ぐためテーブルに優先順位を付けて処理するのに有用です(24.1.5を参照してください)。

このオプションの目的のため、リレーションのトランザクションIDの年代は、主であるリレーションの年代と、存在するなら、関連するTOASTテーブルの年代のうち最大のものです。vacuumdbにより発行されたコマンドも、必要であればリレーションのTOASTテーブルを処理しますので、別々に分けて考える必要はないです。

> **注記**
>
> このオプションはPostgreSQL 9.6以降が動作しているサーバでのみ利用可能です。

-P parallel_degree
--parallel=parallel_degree

並列バキュームの並列度を指定します。これによりバキュームが複数CPUを活用してインデックスを処理することが出来ます。VACUUMを参照してください。

> **注記**
>
> このオプションはPostgreSQL 13以降が動作しているサーバでのみ利用可能です。

-q
--quiet

進行メッセージを表示しません。

--skip-locked

処理のためにすぐにロックできないリレーションを飛ばします。

> **注記**
>
> このオプションはPostgreSQL 12以降が動作しているサーバでのみ利用可能です。

-t table [(column [,...])]
--table=table [(column [,...])]

tableのみをクリーンアップ/解析します。列名は--analyzeや--analyze-onlyオプションがある場合にのみ設定できます。複数の-tスイッチを記述することで複数のテーブルをバキュームすることができます。

> **ヒント**
>
> 列を指定する場合は、シェルから括弧をエスケープする必要があるでしょう（後述の例を参照してください）。

-v
--verbose

処理中に詳細な情報を表示します。

-V
--version

vacuumdbのバージョンを表示し、終了します。

-z
--analyze

オプティマイザが使用する、データベースの統計情報も算出します。

-Z
--analyze-only

オプティマイザにより使用される統計情報の計算のみを行います（バキュームを行いません）。

--analyze-in-stages

--analyze-onlyと同様、オプティマイザにより使用される統計情報の計算のみを行います（バキュームを行いません）。利用可能な統計情報をより速く生成するため、異なる設定を使って複数回(現在は3回)の解析を行います。

このオプションは、ダンプからリストアした、あるいはpg_upgradeを使って新しくデータを入れたデータベースを解析する時に便利です。このオプションでは、データベースを利用可能にするために、何らかの統計情報をできる限り速く作成しようとし、それから、引き続くステージで完全な統計情報を生成します。

-?
--help

vacuumdbのコマンドライン引数の使用方法を表示し、終了します。

vacuumdbには、以下に記す接続パラメータ用のコマンドライン引数も指定することもできます。

-h host
--host=host

サーバが稼働しているマシンのホスト名を指定します。ホスト名がスラッシュから始まる場合、Unixドメインソケット用のディレクトリとして使用されます。

-p port
--port=port

サーバが接続を監視するTCPポートもしくはUnixドメインソケットファイルの拡張子を指定します。

-U username
--username=username

接続するユーザ名を指定します。

-w
--no-password

パスワードの入力を促しません。サーバがパスワード認証を必要とし、かつ、.pgpassファイルなどの他の方法が利用できない場合、接続試行は失敗します。バッチジョブやスクリプトなどパスワードを入力するユーザが存在しない場合にこのオプションは有用かもしれません。

-W
--password

データベースに接続する前に、vacuumdbは強制的にパスワード入力を促します。

サーバがパスワード認証を要求する場合vacuumdbは自動的にパスワード入力を促しますので、これが重要になることはありません。しかし、vacuumdbは、サーバにパスワードが必要かどうかを判断するた

めの接続試行を無駄に行います。こうした余計な接続試行を防ぐために-Wの入力が有意となる場合もあります。

--maintenance-db=dbname

-a(または--all)が指定されている時、どのデータベースをバキュームしなければならないかを見つけ出すために接続するデータベースの名前を指定します。データベース名が指定されていなければpostgresデータベースが使用され、もし存在しなければtemplate1が使用されます。これは接続文字列に出来ます。その場合、接続文字列パラメータは競合するコマンドラインオプションを上書きします。また、データベース名以外の接続文字列パラメータは他のデータベースに接続する時に再利用されます。

環境

PGDATABASE
PGHOST
PGPORT
PGUSER

デフォルトの接続パラメータです。

PG_COLOR

診断メッセージで色を使うかどうかを指定します。可能な値はalways、auto、neverです。

また、このユーティリティは、他のほとんどのPostgreSQLユーティリティと同様、libpqでサポートされる環境変数を使用します(33.14を参照してください)。

診断

問題が発生した場合、考えられる原因とエラーメッセージについての説明はVACUUMとpsqlを参照してください。データベースサーバは、指定したホストで稼働している必要があります。また、libpqフロントエンドライブラリのデフォルトの設定や環境変数が適用されることに注意してください。

注釈

vacuumdbの実行中にはPostgreSQLサーバに何度も接続しなければならないことがありますが、この場合その度にパスワードが必要です。そのような場合は~/.pgpassファイルを用意しておくと便利です。詳細は33.15を参照してください。

例

testというデータベースをクリーンアップするには、下記のコマンドを実行します。

```
$ vacuumdb test
```

bigdbという名前のデータベースのクリーンアップとオプティマイザ用の解析を行う場合には、下記のコマンドを実行します。

```
$ vacuumdb --analyze bigdb
```

xyzzyという名前のデータベースのfooという1つのテーブルだけのクリーンアップと、そのテーブルのbarという1つの列にだけ対してオプティマイザ用の解析を行う場合には、下記のコマンドを実行します。

```
$ vacuumdb --analyze --verbose --table='foo(bar)' xyzzy
```

関連項目

VACUUM

PostgreSQLサーバアプリケーション

ここには、PostgreSQLサーバアプリケーションとサポートユーティリティに関するリファレンス情報があります。これらのコマンドは通常データベースサーバが稼働しているホスト上でのみ実行されます。他のユーティリティプログラムの一覧はPostgreSQLクライアントアプリケーションにあります。

目次

initdb

initdb — PostgreSQLのデータベースクラスタを新しく作成する

概要

```
initdb [option...] [ --pgdata | -D ] directory
```

説明

initdbはPostgreSQLのデータベースクラスタを新しく作成します。データベースクラスタとは、1つのサーバインスタンスで管理されるデータベースの集合です。

データベースクラスタの作成には、データベースのデータを保存するディレクトリの作成、共有カタログテーブル(特定のデータベースではなく、クラスタ全体に所属するテーブル)の生成、そしてtemplate1データベースとpostgresデータベースの作成といった作業が含まれます。後から新しいデータベースを作成する際は、template1データベースの全ての内容がコピーされます。(つまり、template1にインストールしたものすべてが、後に生成した各データベースに自動的にコピーされます。) postgresデータベースは、ユーザ、ユーティリティ、サードパーティ製アプリケーションのデフォルトデータベースとしての使用を意図したものです。

initdbは指定されたデータディレクトリを作成しようと試みますが、そのデータディレクトリの親ディレクトリの所有者がrootであるなど、権限がないことがあります。このような場合、まず、空のデータディレクトリをrootで作成し、chownを使ってそのディレクトリの所有権限をデータベースのユーザに変えてください。次にsuを使ってデータベースユーザとなり、initdbを実行します。

initdbは、サーバプロセスの所有者となるユーザによって実行されなければなりません。initdbによって作成されるファイルやディレクトリにサーバがアクセスできる必要があるからです。サーバをrootとして実行することはできませんので、rootでinitdbを実行してはいけません (実際には、実行しようとしても拒否されます)。

セキュリティ上の理由から、デフォルトではinitdbにより作られた新しいクラスタはクラスタの所有者だけがアクセスできます。--allow-group-accessにより、クラスタの所有者と同じグループのユーザがクラスタ内のファイルを読めるようになります。これは非特権ユーザとしてバックアップを実行するのに有用です。

initdbは、データベースクラスタのデフォルトのロケールと文字セット符号化方式を初期化します。文字セット符号化方式、照合順(LC_COLLATE)および文字セットクラス(LC_CTYPE、例えば、大文字、小文字、数字)は、データベース作成時、別々に設定することができます。initdbはtemplate1データベース用のこれらの値を決定します。この値が他のすべてのデータベースのデフォルトとして提供されます。

デフォルトの照合順や文字セットクラスを変更するためには、--lc-collateおよび--lc-ctypeオプションを使用してください。CやPOSIX以外の照合順には性能の劣化が伴います。これらの理由によりinitdb実行時にロケールを正しく選択することが重要です。

残りのロケール関連のものはサーバ起動時に変更することができます。--localeを使用して、照合順や文字セットクラスを含む、すべてのロケール関連のもののデフォルトを設定することも可能です。SHOW ALLを使用して、すべてのサーバのロケール値(lc_*)を表示することができます。詳細は23.1に記載しています。

デフォルトの符号化方式を変更するには、--encodingオプションを使用します。詳細は23.3に記載しています。

オプション

-A authmethod
--auth=authmethod

このオプションは、pg_hba.confにてローカルユーザ用に使用されるデフォルトの認証方法(host行とlocal行)を指定します。initdbは、非レプリケーションおよびレプリケーションの接続について、指定の認証方式を使うエントリをpg_hba.confに作成します。

システムのすべてのローカルユーザが信頼できるわけではない場合は、trustを使用しないでください。インストールを簡単にするためにtrustがデフォルトになっています。

--auth-host=authmethod

このオプションはpg_hba.confにてTCP/IP接続経由のローカルユーザ用に使用される認証方法(host行)を指定します。

--auth-local=authmethod

このオプションはpg_hba.confにてUnixドメインソケット接続経由のローカルユーザ用に使用される認証方法(local行)を指定します。

-D directory
--pgdata=directory

このオプションは、データベースクラスタを格納すべきディレクトリを指定します。initdbが必要とする情報はこれだけですが、環境変数PGDATAを設定しておけば、このオプションの指定を省略できます。この方法は、後に同じ変数を使用してデータベースサーバ(postgres)がデータベースディレクトリを参照できるので、便利です。

-E encoding
--encoding=encoding

テンプレートデータベースの符号化方式を選択します。これが今後作成される全てのデータベースについてのデフォルト符号化方式となりますが、作成時に上書きすることもできます。デフォルトは、ロケールから取得しますが、取得できなければSQL_ASCIIになります。PostgreSQLサーバによってサポートされる文字セットについては23.3.1で説明します。

-g
--allow-group-access

クラスタの所有者と同じグループのユーザが、initdbにより作られたクラスタファイルすべてを読むことを許可します。WindowsではPOSIX形式のグループパーミッションをサポートしませんので、このオプションは無視されます。

-k
--data-checksums

I/Oシステムによる破損検知を補助するために、データページにおいてチェックサムを使用します。これがないと警告もされません。チェックサムを有効にすると、認知できる程度の性能低下が発生する可能

性があります。設定した場合、すべてのデータベースにおいて、すべてのオブジェクトに対してチェックサムが計算されます。チェックサムの失敗はすべてpg_stat_databaseビューで報告されます。

--locale=locale

データベースクラスタ用のデフォルトのロケールを設定します。このオプションを指定しない場合は、initdbを実行している環境のロケールが継承されます。ロケールサポートについては23.1で説明します。

--lc-collate=locale
--lc-ctype=locale
--lc-messages=locale
--lc-monetary=locale
--lc-numeric=locale
--lc-time=locale

--localeと似ていますが、指定したカテゴリのロケールのみを設定します。

--no-locale

--locale=Cと同じです。

-N
--no-sync

デフォルトではinitdbはすべてのファイルが安全にディスクに書き出されるまで待機します。このオプションを使うとinitdbは待機せずに返るようになり、より高速になりますが、後でオペレーティングシステムがクラッシュした場合にデータディレクトリが破損状態になってしまう可能性があります。通常、このオプションは試験用では有用ですが、実用のインストレーションを作成する際に使用すべきではありません。

--pwfile=filename

initdbはデータベーススーパーユーザのパスワードをこのファイルから読み取ります。このファイルの最初の行がパスワードとして解釈されます。

-S
--sync-only

すべてのデータベースファイルを安全にディスクに書き出し、終了します。これは通常のinitdbの操作をまったく行いません。

-T config
--text-search-config=config

デフォルトの全文検索設定を設定します。詳細についてはdefault_text_search_configを参照してください。

-U username
--username=username

データベースのスーパーユーザのユーザ名を選択します。initdbを実行している実効ユーザの名前がデフォルトです。スーパーユーザの名前はあまり重要ではありませんが、慣習的に使われているpostgresを(オペレーティングシステムのユーザ名と異なっていても)使っても良いでしょう。

-W
--pwprompt

initdbにデータベースのスーパーユーザ権限を与えるためのパスワード入力のプロンプトを表示させます。パスワード認証を行うつもりがない場合は必要ありません。このオプションを指定しても、パスワードの設定を行わない限りパスワード認証は行われません。

-X directory
--waldir=directory

このオプションは先行書き込みログの格納ディレクトリを指定します。

--wal-segsize=size

*WAL*セグメントサイズをメガバイト単位で設定します。これはWALログの個々のファイルの大きさです。デフォルトの大きさは16メガバイトです。値は1から1024の間の2の冪でなければなりません。このオプションは初期化の際にのみ設定することができ、後で変更することはできません。

WALログのシッピングやアーカイブの粒度を制御するために、この大きさを調整することは有用でしょう。また、大量のWALのあるデータベースでは、ディレクトリ当たりのWALファイルの数だけでパフォーマンスや管理の問題となり得ます。WALファイルの大きさを増やせば、WALファイルの数は減るでしょう。

この他にも、使用頻度は下がりますが、下記のオプションが使用可能です。

-d
--debug

ブートストラップバックエンドからのデバッグ情報と、一般の利用者にはおそらく不要なその他の情報を出力します。ブートストラップバックエンドとはinitdbがカタログテーブルを作成する際に使用するプログラムです。このオプションはうんざりするようなログを大量に出力します。

-L directory

データベースクラスタを初期化する際に、initdbが参照すべき入力ファイルを指定します。これは通常必要ありません。明示的に指定する必要がある場合は、その時に指定するよう要求されます。

-n
--no-clean

デフォルトでは、initdbを実行中にエラーが発生し、データベースクラスタを完成できなかった場合に、そのエラーが発生する前に作成された全てのファイルを削除します。このオプションを指定すると、これらのファイルが削除しないで残されるので、デバッグの際にはとても便利です。

その他のオプションを以下に示します。

-V
--version

initdbのバージョンを表示し、終了します。

-?
--help

initdbのコマンドライン引数の使用方法を表示し、終了します。

環境

PGDATA

データベースクラスタを保存するディレクトリを指定します。-Dオプションを使用して上書きすることができます。

PG_COLOR

診断メッセージで色を使うかどうかを指定します。可能な値はalways、auto、neverです。

TZ

作成されるデータベースクラスタのデフォルトの時間帯を指定します。値は完全な時間帯の名前で指定することが望ましいです(8.5.3参照)。

また、このユーティリティは、他のほとんどのPostgreSQLユーティリティと同様、libpqでサポートされる環境変数を使用します(33.14を参照してください)。

注釈

initdbはpg_ctl initdb経由でも呼び出すことができます。

関連項目

pg_ctl, postgres

pg_archivecleanup

pg_archivecleanup — PostgreSQL WALアーカイブファイルを消去する

概要

pg_archivecleanup [option...] archivelocation oldestkeptwalfile

説明

pg_archivecleanupは、スタンバイサーバとして動作している際のWALファイルのアーカイブを消去するためのarchive_cleanup_commandとして使用されるように設計されています（26.2参照）。同時に、pg_archivecleanupはWALファイルのアーカイブを消去するためのスタンドアローンのプログラムとしても利用することができます。

pg_archivecleanupを使うためにスタンバイサーバを設定するには、以下をpostgresql.conf設定ファイルに記述します。

```
archive_cleanup_command = 'pg_archivecleanup archivelocation %r'
```

archivelocationは、WALセグメントファイルを削除するディレクトリです。

archive_cleanup_commandで使用される場合、論理的に%rより前のすべてのWALファイルはarchivelocationから削除されます。これによって、クラッシュ—リスタートの機能を維持するための、保持しなければならないファイル数を最小限にします。このパラメータの使用は、archivelocationがスタンバイサーバにおいて短期的な保存場所となっている場合には適切ですが、archivelocationを長期的なWALアーカイブ領域として使っている場合、または複数のスタンバイサーバが同一のアーカイブログの場所からリカバリしている場合には適切ではありません。

スタンドアローンプログラムとして使用される場合、oldestkeptwalfileより論理的に前のWALファイルは、すべてarchivelocationから削除されます。このモードでは、.partialまたは.backupのファイル名が指定されると、そのプレフィックス部だけがoldestkeptwalfileとして使用されます。この.backupのファイル名の処置により、エラーを起こすことなく特定のベースバックアップより前のすべてのWALファイルを削除することを可能にします。以下の例は、000000010000003700000010より古いすべてのファイルを削除する実例です。

```
pg_archivecleanup -d archive 000000010000003700000010.00000020.backup

pg_archivecleanup:  keep WAL file "archive/000000010000003700000010" and later
pg_archivecleanup:  removing file "archive/00000001000000370000000F"
pg_archivecleanup:  removing file "archive/00000001000000370000000E"
```

pg_archivecleanupは、archivelocationがサーバを実行しているユーザによって読み書き可能なディレクトリであるものと仮定しています。

オプション

pg_archivecleanupは、以下のコマンドライン引数を受け付けます。

`-d`

stderrに大量のデバッグログを出力します。

`-n`

削除されるファイルの名前をstdoutに出力します(実際には削除しません).

`-V`
`--version`

pg_archivecleanupのバージョンを表示して終了します。

`-x extension`

削除するファイルを決定する前にファイル名から取り除かれる拡張子を指定します。保存時に圧縮され、そのため圧縮プログラムにより拡張子を付けられたアーカイブを消去するのに特に役に立ちます。例: `-x.gz`

`-?`
`--help`

pg_archivecleanupのコマンドライン引数に関するヘルプを表示して終了します。

環境

環境変数PG_COLORは診断メッセージで色を使うかどうかを指定します。可能な値はalways、auto、neverです。

注釈

pg_archivecleanupは、PostgreSQL 8.0 とそれ以降において、スタンドアローンユーティリティとして動作するように設計されています。また、PostgreSQL 9.0 とそれ以降においてはアーカイブのクリーンナップコマンドとして動作するように設計されています。

pg_archivecleanupはC言語で書かれており、必要に応じて修正すべき部分が明確に示されているので、修正の容易なソースコードとなっています。

例

LinuxやUnixのシステムでは、以下のように使います。

```
archive_cleanup_command = 'pg_archivecleanup -d /mnt/standby/archive %r 2>>cleanup.log'
```

アーカイブディレクトリは物理的にはスタンバイサーバ上に配置されていますので、archive_commandはNFSを経由してアーカイブディレクトリにアクセスしますが、それらのファイルはスタンバイサーバにとってはローカルファイルです。この設定では、

- デバッグ出力をcleanup.logに生成します。
- アーカイブディレクトリの中から、不要になったファイルを削除します。

関連項目

pg_standby

pg_checksums

pg_checksums — PostgreSQLデータベースクラスタのデータチェックサムを有効化、無効化、あるいは検査する。

概要

pg_checksums [option...] [[-D | --pgdata] datadir]

説明

pg_checksumsはPostgreSQLクラスタのデータチェックサムの検査と有効化、無効化を行います。pg_checksumsを実行する前にサーバは正常停止されていなければなりません。チェックサムを検査する場合、終了ステータスはチェックサム誤りが無ければゼロで、チェックサム誤りが一つでも在ったなら非ゼロです。チェックサムの有効化、無効化をする場合、終了ステータスは操作に失敗したときに非ゼロになります。

チェックサムを検査する場合、クラスタ内の全てのファイルがスキャンされます。チェックサムの有効化ではクラスタ内のすべてのファイルがその場で書き換えられます。チェックサムの無効化ではファイルpg_controlだけ更新されます。

オプション

以下のコマンドラインオプションが使用できます。

-D directory
--pgdata=directory

データベースクラスタが格納されているディレクトリを指定します。

-c
--check

チェックサムを検査します。これは何も指定しなかった場合のデフォルトのモードです。

-d
--disable

チェックサムを無効化します。

-e
--enable

チェックサムを有効化します。

-f filenode
--filenode=filenode

指定したファイルノードfilenodeのリレーション内のチェックサムだけを検査します。

-N
--no-sync

pg_checksumsはデフォルトでは全てのファイルが安全にディスクに書かれるまで待ちます。このオプションは、pg_checksumsがこれを待たずに応答するようにします。より早いですが、その後のオペレーティングシステムのクラッシュで更新されたデータディレクトリの破損が残ることを意味します。一般に、このオプションはテストには有用ですが、本番導入むけには使うべきではありません。--checkを使う場合には、このオプションは効果がありません。

-P
--progress

進行報告を有効にします。これを有効にするとチェックサムの検査あるいは有効化で、進行報告が出力されます。

-v
--verbose

冗長な出力を有効にします。チェックした全ファイルの一覧を出力します。

-V
--version

pg_checksumsのバージョンを出力して終了します。

-?
--help

pg_checksumsのコマンドライン引数のヘルプを表示して終了します。

環境

PGDATA

データベースクラスタが格納されたディレクトリを指定します。これに対して-Dオプションで上書き指定できます。

PG_COLOR

対話的メッセージで色を使用するかを指定します。指定可能な値はalways、auto、neverです。

注釈

大きいクラスタでチェックサムを有効にするには、場合によっては長時間を要する可能性があります。この操作中にデータディレクトリに書き込みをするクラスタや他のプログラムを開始してはいけません。さもないとでデータ損失が起きるかもしれません。

リレーションファイルブロックの直接コピーを行うツール（例えばpg_rewind）でレプリケーションのセットアップを使う時のチェックサムの有効化や無効化は、操作が全ノードを通して一貫して行われない場合、不正なチェックサムという形でページ破損を引き起こすおそれがあります。したがって、レプリケーションのセットアップでチェックサムの有効化や無効化をするときには、一貫して切り替える前にすべてのクラスタを停止することを推奨します。全てのスタンバイを廃棄して、プライマリ上で操作を行い、最後にスタンバイを新たに再作成するのも安全です。

チェックサムの有効化や無効化をしている最中にpg_checksumsが中断されたり、killされたりした場合、クラスタのデータチェックサム設定は変更されないままとなり、pg_checksumsを同じ操作を再実行できます。

pg_controldata

pg_controldata — PostgreSQLデータベースクラスタの制御情報を表示する

概要

pg_controldata [option] [[--pgdata | -D] datadir]

説明

pg_controldataはカタログのバージョンなどinitdbの際に初期化された情報を表示します。また、WAL（ログ先行書き込み）およびチェックポイント処理に関する情報も表示します。この情報はクラスタ全体に関するものであり、特定のデータベースに関するものではありません。

このユーティリティの実行にはデータディレクトリへの読み取りアクセス権限が必要となるため、クラスタを初期化したユーザのみが実行できます。データディレクトリは、コマンドラインや環境変数PGDATAを使用して指定することができます。このユーティリティは、pg_controldataのバージョンを表示し終了する-Vおよび--versionオプションをサポートします。またサポートされる引数を出力する-?および--helpオプションもサポートします。

環境

PGDATA

デフォルトのデータディレクトリの場所です。

PG_COLOR

診断メッセージで色を使うかどうかを指定します。可能な値はalways、auto、neverです。

pg_ctl

pg_ctl — PostgreSQLサーバを初期化、起動、停止、制御する

概要

pg_ctl init[db] [-D datadir] [-s] [-o initdb-options]

pg_ctl start [-D datadir] [-l filename] [-W] [-t seconds] [-s] [-o options] [-p path] [-c]

pg_ctl stop [-D datadir] [-m s[mart] | f[ast] | i[mmediate]] [-W] [-t seconds] [-s]

pg_ctl restart [-D datadir] [-m s[mart] | f[ast] | i[mmediate]] [-W] [-t seconds] [-s] [-o options] [-c]

pg_ctl reload [-D datadir] [-s]

pg_ctl status [-D datadir]

pg_ctl promote [-D datadir] [-W] [-t seconds] [-s]

pg_ctl logrotate [-D datadir] [-s]

pg_ctl kill signal_name process_id

Microsoft Windows上ではさらに

pg_ctl register [-D datadir] [-N servicename] [-U username] [-P password] [-S a[uto] | d[emand]] [-e source] [-W] [-t seconds] [-s] [-o options]

pg_ctl unregister [-N servicename]

説明

pg_ctlはPostgreSQLデータベースクラスタの初期化、PostgreSQLのデータベースサーバ（postgres）を起動、停止、再起動する、あるいは稼働中のサーバの状態を表示するためのユーティリティです。サーバは手動で起動することも可能ですが、pg_ctlは、ログ出力のリダイレクトや、端末とプロセスグループの適切な分離などの作業を隠蔽してくれます。さらにシャットダウン制御のための便利なオプションも提供します。

initまたはinitdbモードはPostgreSQLの新しいデータベースクラスタ、つまり、単一のサーバインスタンスで管理されるデータベースの集合を作成します。このモードはinitdbコマンドを呼び出します。詳しくはinitdbを参照して下さい。

startモードは、新しいサーバを起動します。サーバはバックグラウンドで起動され、その標準入力は/dev/null（Windowsの場合はnul）に接続されます。Unix系のシステムではデフォルトで、サーバの標準出力と標準エラーはpg_ctlの標準出力に転送されます(標準エラー出力には転送されません)。pg_ctlの標準出力はファイルにリダイレクトするか、例えばrotatelogsなどのログローテーションプログラムのような別プロセスにパイプで渡すべきです。こうしないと、postgresはその出力を（バックグラウンドから）制御端末に書き出しますので、シェルのプロセスグループから切り離すことができません。Windowsではデフォルトで、サーバの標準出力と標準エラーは端末に送信されます。こうしたデフォルトの動作は-lを用いてサーバの出力をログファイルに追加するように変更することができます。-lまたは出力のリダイレクトのどちらかを使用することを勧めます。

stopモードは、指定されたデータディレクトリで稼働しているサーバを停止（シャットダウン）します。-mオプションでは、3つの異なる停止方式を選択できます。「smart」モードは、新しい接続を禁止してから、全ての接続しているクライアントが切断し、オンラインバックアップがあればそれが終了するまで待ちます。サーバがホットスタンバイ状態の場合、すべてのクライアント接続が切断された後にリカバリとストリーミングレプリケーションは終了します。「fast」モード（デフォルト）はクライアントが切断するまで待たず、かつ、実行中のオンラインバックアップを終了させます。全ての実行中のトランザクションをロールバックし、クライアントとの接続を強制的に切断した後、サーバを停止します。「immediate」モードは、クリーンアップ処理なしで、全てのサーバプロセスを即座に中断します。これを選択すると、次回のサーバ起動時にクラッシュリカバリサイクルが始まります。

restartモードは、実質的にはstopを実行して、その後、startを実行します。この時、postgresのコマンドラインオプションを変更、またはサーバの再起動なしには変更できない設定ファイルオプションを変更することができます。サーバ起動時に相対パスがコマンドラインから使用されていた場合、サーバ起動時と同じカレントディレクトリでpg_ctlを実行しなければ、restartが失敗する可能性があります。

reloadモードは、単にpostgresサーバプロセスにSIGHUPシグナルを送り、（postgresql.conf、pg_hba.confなどの）設定ファイルの再読み込みを実行させます。これにより、設定ファイルのオプションで完全なサーバ再起動を必要としないものについて、変更を反映させることができます。

statusモードは指定したデータディレクトリでサーバが稼働しているかどうかを確認します。稼働している場合はサーバのPIDと、サーバを起動する時に使われたコマンドラインオプションを表示します。サーバが稼働していない場合、pg_ctlは終了ステータス3を返します。アクセス可能なデータディレクトリが指定されていない場合、pg_ctlは終了ステータス4を返します。

promoteモードは、指定したデータディレクトリで実行中のスタンバイサーバに、スタンバイモードを終了し、読み書きの操作を開始するように指示します。

logrotateモードはサーバログファイルを回転します。外部のログ回転ツールでこのモードを使う方法の詳細については、24.3を参照してください。

killモードは指定したプロセスにシグナルを送信します。これは主に、組み込みのkillコマンドがないMicrosoft Windowsで有用です。サポートされているシグナル名の一覧を見るには--helpを使ってください。

registerモードはPostgreSQLサーバをMicrosoft Windows上のシステムサービスとして登録します。-Sオプションにより、「auto」（システムの起動時に自動的にサービスを開始する）と「demand」（要求に応じてサービスを開始する）のいずれかのサービス開始種別を選択できます。

unregisterモードによりMicrosoft Windows上のシステムサービスを登録解除することができます。これは過去にregisterコマンドによりなされた変更を元に戻します。

オプション

-c
--core-files

実現可能なプラットフォームにおいて、サーバクラッシュ時にcoreファイルを生成できるように関連するソフトリソース制限を上げます。障害が起きたサーバプロセスからスタックトレースを取得できますので、問題のデバッグや診断の際に有用です。

-D datadir
--pgdata=datadir

データベース設定ファイルのファイルシステム上の場所を指定します。このオプションが省略された場合、環境変数PGDATAが使われます。

-l filename
--log=filename

サーバログ出力をfilenameに追記します。そのファイルが存在しない場合は作成されます。umaskは077に設定されますので、他のユーザからのログファイルへのアクセスはデフォルトでは許可されません。

-m mode
--mode=mode

停止(シャットダウン)モードを指定します。modeはsmart、fast、immediate、もしくはこの3つのモード名の最初の1文字をとることができます。このオプションが省略された時はfastがデフォルトとなります。

-o options
--options=options

postgresコマンドに直接渡されるオプションを指定します。-oは複数回指定することができ、そこで指定されたすべてのオプションが渡されます。

optionsはそれが一つのグループとして渡されるようにするため、通常は単一引用符または二重引用符で括るべきです。

-o initdb-options
--options=initdb-options

initdbコマンドに直接渡されるオプションを指定します。-oは複数回指定することができ、そこで指定されたすべてのオプションが渡されます。

initdb-optionsはそれが一つのグループとして渡されるようにするため、通常は単一引用符または二重引用符で括るべきです。

-p path

postgresの実行プログラムの位置を指定します。デフォルトではpostgresの実行プログラムはpg_ctlと同じディレクトリにあるものとみなされます。また、このディレクトリに存在しなければ、構築時に指定したインストールディレクトリが使われます。このオプションは、何か異例なことをしていてpostgresが見つからないというようなエラーが出ない限り、使う必要はありません。

initモードでは、このオプションは同様にinitdb実行プログラムの場所を指定します。

-s
--silent

エラーメッセージのみを表示し、その他の情報を表示しません。

-t seconds
--timeout=seconds

操作が完了するまで待機する最大の秒数を指定します（オプション-wを参照）。デフォルトは環境変数PGCTLTIMEOUTの値で、それが設定されていなければ60秒です。

-V
--version

pg_ctlのバージョンを表示し、終了します。

-w
--wait

操作が完了するのを待ちます。これはstart、stop、restart、promote、registerのモードについてサポートされており、これらのモードについてのデフォルトです。

待機している場合、pg_ctlは繰り返しサーバのPIDファイルを確認し、確認と確認の間は少しの時間スリープします。起動は、サーバが接続を受け付ける準備ができたことをPIDファイルが示した時に、完了したとみなされます。停止は、サーバがPIDファイルを削除した時に、完了したとみなされます。pg_ctlは、起動もしくは停止が成功したかどうかに基づいて終了コードを返します。

操作がタイムアウト（オプション-t参照）までに完了しなかった場合、pg_ctlは非ゼロの終了コードで終了します。しかし、その操作はバックグラウンドで実行し続け、最終的に成功するかもしれないことに注意してください。

-W
--no-wait

操作が完了するのを待ちません。オプション-wの反対です。

待機が無効化されていると、要求された操作は実行されますが、それが成功したかどうかのフィードバックがなくなります。その場合、サーバのログファイルや外部のモニタリングシステムを使って操作の進行状況や成功したかどうかを確認する必要があるでしょう。

PostgreSQLの以前のリリースでは、stopモードを除き、これがデフォルトでした。

-?
--help

pg_ctlコマンドライン引数の説明を表示し、終了します。

指定されたオプションが有効ではあるが、指定の操作モードと関係ないものである場合、pg_ctlはそれを無視します。

Windows用オプション

-e source

Windowsのサービスとして実行する際に、イベントログへの出力用にpg_ctlが使用するイベントソースの名前です。デフォルトはPostgreSQLです。これはpg_ctl自体からのメッセージのみを制御することに注意してください。サーバが起動した後は、event_sourceで指定したイベントソースが使用されます。

サーバが起動の非常に早い段階のパラメータが設定されるより前に障害を起こした場合は、デフォルトのイベントソース名のPostgreSQLを使用するかもしれません。

-N servicename

登録するシステムサービスの名前です。この名前は、システム名としても表示名としても使用されます。デフォルトはPostgreSQLです。

-P password

サービスを開始するユーザ用のパスワードです。

-S start-type

システムサービスの起動種類です。start-typeはauto、demand、またはこれら2単語の先頭の文字のいずれかです。このオプションを省略した時はautoがデフォルトとなります。

-U username

サービスを起動するユーザのユーザ名です。ドメインユーザの場合はDOMAIN\username書式を使用してください。

環境

PGCTLTIMEOUT

起動または終了が完了するまでに待機する秒数のデフォルトの最大値です。設定されていない場合のデフォルトは60秒です。

PGDATA

デフォルトのデータディレクトリの場所です。

ほとんどのpg_ctlのモードはデータディレクトリの位置を知っている必要があるため、PGDATAが設定されていないときは-Dオプションが必須です。

また、pg_ctlは、他のほとんどのPostgreSQLユーティリティと同様、libpqでサポートされる環境変数を使用します(33.14を参照してください)。

サーバに影響を与える他の変数についてはpostgresを参照してください。

ファイル

postmaster.pid

pg_ctlは、データディレクトリのこのファイルを検査して、サーバが現在稼働中かどうかを決定します。

postmaster.opts

このファイルがデータディレクトリにあれば、pg_ctl(のrestartモード)は、-oで上書きされるものを除き、このファイルの内容をオプションとしてpostgresに渡します。また、このファイルの内容がstatusモードで表示されます。

例

サーバの起動

以下はサーバが接続を植え付けられるようになるまで待機する起動例です。

```
$ pg_ctl start
```

ポート5433を使いfsyncなしでサーバを起動します。

```
$ pg_ctl -o "-F -p 5433" start
```

サーバの停止

サーバを停止するためには以下を使用します。

```
$ pg_ctl stop
```

-mオプションによりどのようにバックエンドを停止させるかを制御できます。

```
$ pg_ctl stop -m smart
```

サーバの再起動

サーバの再起動は、サーバを停止してもう一度起動するのとほぼ同じですが、違うのは、pg_ctlがデフォルトでは以前起動していたインスタンスに渡されていたコマンドラインオプションを保存し再利用することです。下記はサーバを以前と同じオプションを使って再起動する方法です。

```
$ pg_ctl restart
```

しかし、-oを指定すれば、それによって以前のオプションが置換されます。以下はポート5433を使って再起動し、再起動後にfsyncを無効にする方法です。

```
$ pg_ctl -o "-F -p 5433" restart
```

サーバの状態表示

下記はpg_ctlからの状態の出力の例です。

```
$ pg_ctl status
```

```
pg_ctl: server is running (PID: 13718)
/usr/local/pgsql/bin/postgres "-D" "/usr/local/pgsql/data" "-p" "5433" "-B" "128"
```

2行目は再起動（restart）モードで呼び出されるコマンドラインです。

関連項目

initdb, postgres

pg_resetwal

pg_resetwal — PostgreSQLデータベースクラスタの先行書き込みログやその他の制御情報を初期化する

概要

```
pg_resetwal [ --force | -f ] [ --dry-run | -n ] [option...] [ --pgdata | -D ] datadir
```

説明

pg_resetwalは、先行書き込みログ（WAL）を消去し、さらにオプションでpg_controlファイル内に保存された制御情報の一部を初期化します。この機能は、これらのファイルが破損した場合に必要になることがあります。このような破損などが原因でサーバを起動できない場合の最後の手段としてのみ、この機能を使用してください。

このコマンドを実行すると、サーバが開始できるようになるはずです。ただし、不完全にコミットされたトランザクションが原因でデータベースのデータに矛盾が起こる可能性があることに注意してください。コマンドの実行後は、ただちにデータをダンプし、initdbを実行し、リロードすべきです。リロード後、矛盾がないか検査し、必要に応じて修復を行ってください。

このユーティリティの実行にはデータディレクトリへの読み込み/書き込みアクセス権限が必要となるため、サーバをインストールしたユーザのみが実行できます。安全のため、データディレクトリをコマンドラインで指定する必要があります。pg_resetwalは、環境変数PGDATAを使用しません。

pg_resetwalがpg_controlに対する有効なデータを判別できない場合、-f（force,強制）オプションを指定すれば強制的に処理を進めることができます。その場合、欠落したデータは無難な値で代用されます。ほとんどのフィールドでは適切な値が使用されますが、次のOID、次のトランザクションIDとエポック時間、マルチトランザクションIDとそのオフセット、WAL開始位置の値については、手動の操作が必要な場合があります。これらの値は下記で説明するオプションを使用して設定することができます。すべてに対して正しい値を決定できない場合でも-fを使用することができますが、この場合は回復したデータベースを通常よりさらに注意深く検査する必要があります。必ず、ただちにダンプおよびリロードを行ってください。決して、ダンプを行う前にデータ変更などの操作を行ってはなりません。そのような操作は、破損状態をさらに悪化させます。

オプション

-f
--force

上で説明したように、pg_controlに有効なデータが確認できない場合でも、強制的にpg_resetwalの処理を実行します。

-n
--dry-run

-n/--dry-runオプションを指定すると、pg_resetwalはpg_controlから再構築した値、および変更される値を出力して、何も変更せずに終了します。これは主にデバッグと目的としたツールですが、pg_resetwalを実際に進める前の検査としても有用な場合があります。

-V
--version

バージョン情報を表示して終了します。

-?
--help

ヘルプを表示して終了します。

以下のオプションはpg_resetwalがpg_controlを読んでも適切な値を決定できない場合にのみ必要になります。安全な値は以下で説明するようにして決定できます。数値を引数として取る値については、0xの接頭辞をつけることで16進数の値を指定できます。

-c xid,xid
--commit-timestamp-ids=xid,xid

コミットの時刻が取り出せる最古のトランザクションIDと最新のトランザクションIDを手作業で設定します。

コミット時刻が取り出せる最古のトランザクションIDとして安全な値（1番目の部分）はデータディレクトリの下のpg_commit_tsディレクトリの中で、数値的に最小のファイル名を探すことで決定できます。逆に、コミット時刻が取り出せる最新のトランザクションIDとして安全な値（2番目の部分）は同じディレクトリの中で、数値的に最大のファイル名を探すことで決定できます。ファイル名は16進数になっています。

-e xid_epoch
--epoch=xid_epoch

次のトランザクションIDのエポック時間を手作業で設定します。

pg_resetwalで設定されるフィールドを除き、トランザクションIDのエポック時間は実際にはデータベース内のどこにも格納されません。そのため、データベース自体だけを考えるのであれば、任意の値で動作するでしょう。Slony-IやSkytoolsなどのレプリケーションシステムが確実に正しく動作するように、この値を調整しなければならない可能性があります。その場合、適切な値は下流で複製されたデータベースの状態から得られるはずです。

-l walfile
--next-wal-file=walfile

次のWALセグメントファイル名を指定することでWAL開始位置を手動で設定します。

次のWALセグメントファイル名は、データディレクトリ以下のpg_walに現在存在するどのWALセグメントファイル名よりも大きくならなければなりません。この名前も16進数で、3つの部分に分かれています。最初の部分は「時系列ID」で、通常、この値は変更すべきではありません。例えば、pg_wal内で最大のエントリが000000010000003200000004Aである場合は、-l 000000010000003200000004B以上を使用してください。

デフォルト以外のWALセグメントサイズを使用しているときには、WALファイル名の番号はシステム関数やシステムビューで報告されるLSNとは異なるということに注意してください。このオプションはLSNではなくWALファイル名を引数に取ります。

> **注記**
>
> pg_resetwal自体はpg_wal内のファイルを参照し、最後の既存のファイル名より大きな値をデフォルトの-l設定として選択します。したがって、手作業による-lの調整は、オフラインアーカイブ内の項目などpg_walに現存しないWALセグメントファイルがあることに気づいた場合、または、pg_walの内容が完全に失われている場合にのみ必要とされます。

`-m mxid,mxid`
`--multixact-ids=mxid,mxid`

次のマルチトランザクションIDと最古のマルチトランザクションIDを手作業で設定します。

次のマルチトランザクションIDとして安全な値（1番目の部分）は、データディレクトリの下のpg_multixact/offsetsディレクトリの中で数値的に最大のファイル名を探し、1を加えてから65536(0x10000)を掛けることで決定できます。逆に、最古のマルチトランザクションIDとして安全な値（-mの2番目の部分）は、同じディレクトリの中で数値的に最小のファイル名を探し、65536を掛けることで決定できます。ファイル名は16進ですので、このための最も簡単なやり方は、オプション値を16進で指定し、ゼロを4つ追加することです。

`-o oid`
`--next-oid=oid`

次のOIDを手作業で設定します。

データベース内のOIDの最大値よりも大きな次のOIDを決定するには、上記のような簡単な方法はありません。しかし、幸いにも、次のOIDの設定を正しく取得することは、それほど重要ではありません。

`-O mxoff`
`--multixact-offset=mxoff`

次のマルチトランザクションオフセットを手作業で設定します。

安全な値は、データディレクトリの下のpg_multixact/membersディレクトリの中で数値的に最も大きなファイル名を探し、1を加えてから、52352(0xCC80)を掛けることで決定できます。ファイル名は16進数です。他のオプションのような0をつけるだけの簡単な計算方法はありません。

`--wal-segsize=wal_segment_size`

新たなWALセグメントサイズをメガバイトで設定します。値には1から1024（メガバイト）の2の冪乗を設定しなければなりません。詳しくはinitdbの同じオプションを参照してください。

> **注記**
>
> pg_resetwalが存在する最も新しいWALセグメントファイルを超えたWAL開始位置を設定するとき、一部のセグメントサイズ変更は前のWALファイル名の再使用をひき起こす可能性があります。あなたのアーカイブ戦略でWALファイル名のオーバーラップが問題を起こす場合には、このオプションと共にWAL開始位置を手動で設定する-lも使うことを推奨します。

`-x xid`
`--next-transaction-id=xid`

次のトランザクションIDを手作業で設定します。

安全な値は、データディレクトリの下のpg_xactディレクトリの中で数値的に最も大きなファイル名を探し、1を加えてから、1048576(0x100000)を掛けることで決定できます。ファイル名は16進数であることに注意して下さい。通常は、オプションの値も16進数で指定するのが最も簡単でしょう。例えば、0011がpg_xactで最も大きなエントリであれば、`-x 0x1200000`とすれば良いです（後ろにゼロを5つ付けると、正しく掛け算をしたことになります）。

環境

PG_COLOR

診断メッセージで色を使うかどうかを指定します。可能な値はalways、auto、neverです。

注釈

このコマンドは、サーバの稼動中に使用してはいけません。pg_resetwalは、データディレクトリにサーバのロックファイルがあると、実行されません。サーバがクラッシュした場合、ロックファイルがそのまま残される場合があります。その場合は、ロックファイルを削除すればpg_resetwalを実行することができます。しかし、そのようなことをする前に、まだ稼働中のサーバプロセスが一切ないことを慎重に確認してください。

pg_resetwalは同一のメジャーバージョンのサーバに対してのみ動作します。

関連項目

pg_controldata

pg_rewind

pg_rewind — PostgreSQLのデータディレクトリを、そこから派生した他のデータディレクトリと同期する

概要

```
pg_rewind [option...] { -D  | --target-pgdata } directory { --source-pgdata=directory | --source-server=connstr }
```

説明

pg_rewindは、PostgreSQLのクラスタのタイムラインが分岐した後、クラスタをその複製のクラスタに同期するためのツールです。典型的なシナリオとしては、フェイルオーバー後、新しいマスターに追従するスタンバイとして、古いマスターサーバをオンラインに戻す、というのがあります。

巻き戻し(rewind)が成功すれば、ターゲットデータディレクトリの状態はソースデータディレクトリのベースバックアップと類似したものになります。新しいベースバックアップを取ったり、rsyncのようなツールを使うのとは異なり、pg_rewindはクラスタ内の変更されていないリレーションブロックの比較やコピーを必要としません。既存のリレーションファイルのうちの変化のあったブロックだけがコピーされます。それ以外のすべてのファイルは、新しいリレーションファイルや設定ファイル、WALセグメントを含め、すべてのファイルがコピーされます。そのため、データベースが大きく、クラスタ間で変更されているブロックの割合が小さい場合には、巻き戻し操作は他の方法に比べて極めて高速になります。

pg_rewindはソースとターゲットクラスタ内のタイムラインヒストリーを調べ、それらがどの時点で異なるものになったのかを調べます。差異が発生した分岐点までずっと遡ることにより、ターゲットクラスタ内のpg_walディレクトリ内の分岐点に到達するWALを見つけようとします。変化の分岐点は、ターゲット側のタイムライン中、ソース側のタイムライン中、あるいはそれら共通の祖先の中に見つかる可能性が高いです。分岐点のあと間をおかずシャットダウンされたような典型的なフェイルオーバーのシナリオにおいては、これは特に問題になりません。しかし、分岐点の後にターゲットクラスタが長時間運用されていた場合には、古いWALファイルはもう存在しないかもしれません。この場合は、WALアーカイブから手動でpg_walディレクトリにコピーできます。あるいは、-cオプションを付けてpg_rewindを実行し、WALアーカイブから自動的に取り出すこともできます。pg_rewindの利用は、フェイルオーバーに留まりません。たとえば、スタンバイサーバが昇格してから書き込みトランザクションを実行し、再びスタンバイになるために巻き戻すこともできます。

pg_rewindを実行した後、データディレクトリが整合のとれた状態になるためにはWALリプレイの完了が必要です。ターゲットサーバは再起動すると、アーカイブリカバリに入り、分岐点の前の最後のチェックポイント以降にソースサーバで生成されたWALをすべてリプレイします。pg_rewindが実行された時にWALの中にソースサーバにないものがあり、pg_rewindのセッションではコピーできなかった場合は、ターゲットサーバが起動した時にWALを読む込めるようになっていなければなりません。recovery.signalファイルをターゲットデータディレクトリに置き、postgresql.confに適切なrestore_commandを設定することで、これを達成できます。

pg_rewindを使用するには、ターゲットサーバ上でpostgresql.confのwal_log_hintsオプションが有効になっているか、initdbでクラスタを初期化した時にデータチェックサムが有効になっていなければなりませ

ん。どちらもデフォルトではonではありません（無効です）。full_page_writesもon（有効）でなければなりませんが、これはデフォルトで有効です。

> **警告**
>
> 処理中にpg_rewindが失敗した場合、ターゲットのデータフォルダーはリカバリ可能な状態でない可能性があります。このような場合は、最新のバックアップを取ることをお勧めします。
>
> pg_rewindはソースから設定ファイルをそのままコピーするので、特にターゲットをソースのスタンバイとして再導入する場合には、ターゲットサーバを再起動する前にリカバリのために使われる設定を修正することが必要かもしれません。巻き戻し操作は終わったもののリカバリの設定をせずにサーバを再起動すると、ターゲットは再びプライマリから分岐するでしょう。
>
> pg_rewindは直接書き込めないファイルが見つかるとすぐに失敗します。たとえば、ソースサーバとターゲットサーバが読み取り専用のSSLキーと証明書に同じファイルマッピングを使用する場合に発生します。そのようなファイルがターゲットサーバ上に存在する場合、それらを削除してからpg_rewindを実行することをお勧めします。巻き戻しを行った後、それらのファイルの一部がソースからコピーされている可能性があります。その場合は、コピーされたデータを削除して、巻き戻し前に使用していたリンクのセットを元に戻す必要があります。

オプション

pg_rewindは以下のコマンドラインオプションを受け付けます。

`-D directory`
`--target-pgdata=directory`

このオプションは、同期するターゲットデータディレクトリを指定します。ターゲットサーバは、pg_rewindを実行する前に、正常にシャットダウンされていなければなりません。

`--source-pgdata=directory`

同期するソースサーバのデータディレクトリへのファイルシステム上のパスを指定します。このオプションを使用する場合は、ソースサーバは正常にシャットダウンされていなければなりません。

`--source-server=connstr`

ターゲットサーバに同期するソースPostgreSQLサーバに接続するlibpq接続文字列を指定します。接続は、ソースサーバ（詳しくは注釈を参照）でpg_rewindで使われる関数を実行する権限を持つロールまたはスーパーユーザロールでの通常の（レプリケーションでない）接続でなければなりません。このオプションはソースサーバが実行中であることとリカバリモードではないことを必要とします。

`-R`
`--write-recovery-conf`

出力ディレクトリで`standby.signal`を作成し、`postgresql.auto.conf`に接続設定を追加します。このオプションでは`--source-server`は必須です。

-n
--dry-run

ターゲットディレクトリを実際に更新する以外はすべてのことを行います。

-N
--no-sync

デフォルトでは、pg_rewindはファイルがすべて安全にディスクに書き込まれるのを待ちます。このオプションにより、pg_rewindは待つことなく戻るようになります。これは速いのですが、直後にオペレーティングシステムがクラッシュした場合、同期データディレクトリの破損が残るかもしれません。一般に、このオプションはテストするためには有用ですが、稼働用のインストレーションでは使うべきではありません。

-P
--progress

進行状況のレポートを有効にします。このオプションを有効にすると、データをソースクラスタからコピーする際のおおよその進行状況をレポートします。

-c
--restore-target-wal

WALファイルがpg_walディレクトリにもはや存在しない場合、ターゲットクラスタ設定内で定義されているrestore_commandを使ってWALファイルをWALアーカイブから取り出します。

--debug

主に開発者がpg_rewindをデバッグするのに役立つ冗長なデバッグ出力を印字します。

--no-ensure-shutdown

pg_rewindは、巻き戻す前にターゲットサーバが正常にシャットダウンされていることを要求します。デフォルトでは、ターゲットサーバが正常にシャットダウンされていなければ、pg_rewindはターゲットサーバをシングルユーザモードで起動し、まずクラッシュリカバリを完了して停止します。このオプションを渡すことで、pg_rewindはこれを飛ばして、サーバが正常にシャットダウンされていない場合にはすぐにエラーを発生します。その場合、ユーザが自身で状況を扱うことが期待されます。

-V
--version

バージョン情報を表示して終了します。

-?
--help

ヘルプを表示して終了します。

環境

--source-serverオプションを使用する場合、pg_rewindは libpqで利用できる環境変数を使用します(33.14を参照)。

環境変数PG_COLORは診断メッセージで色を使うかどうかを指定します。可能な値はalways、auto、neverです。

注釈

オンラインのクラスタをソースとして使用してpg_rewindを実行するとき、スーパーユーザの代わりにソースのクラスタ上でpg_rewindで使われる関数を実行できる権限を持ったロールを使うことができます。rewind_userという名前のこのようなロールの作り方を以下に示します。

```
CREATE USER rewind_user LOGIN;
GRANT EXECUTE ON function pg_catalog.pg_ls_dir(text, boolean, boolean) TO rewind_user;
GRANT EXECUTE ON function pg_catalog.pg_stat_file(text, boolean) TO rewind_user;
GRANT EXECUTE ON function pg_catalog.pg_read_binary_file(text) TO rewind_user;
GRANT EXECUTE ON function pg_catalog.pg_read_binary_file(text, bigint, bigint, boolean) TO
 rewind_user;
```

最近に昇格したオンラインのクラスタをソースとして使ってpg_rewindを実行するときには、コントロールファイルが最新のタイムライン情報を反映するように昇格後にCHECKPOINTを実行する必要があります。タイムライン情報はpg_rewindが指定されたソースクラスタを使ってターゲットクラスタを巻き戻しできるかどうか検査するのに使われます。

どうやって動くのか

基本的なアイデアは、ファイルシステムレベルの変更を、すべてをソースクラスタからターゲットクラスタにコピーする、というものです。

1. ソースクラスタのタイムライン履歴がターゲットクラスタから分岐した時点より前の最後のチェックポイントから始めて、ターゲットクラスタのWALログをスキャンします。各々のWALレコードについて、変更されたデータブロックを記録します。これにより、ソースクラスタが分岐した以降に、ターゲットクラスタで変更されたすべてのデータブロックのリストが作成されます。WALファイルの中にもう存在しないものがある場合は、失われたファイルをWALアーカイブで探すよう-cオプションを付けてpg_rewindを再実行してみます。

2. ファイルシステムへの直接アクセス（--source-pgdata）かSQL（--source-server）を使って、変更のあったすべてのブロックを、ソースクラスタからターゲットクラスタにコピーします。これでリレーションファイルは、ソースとターゲットのWALタイムラインが分岐した時点より前で最後に完了したチェックポイントの時点に加えて、分岐後にターゲットで変更されたブロックを含んだソースでの現在の状態に相当する状態になります。

3. 新しいリレーションファイルやWALセグメント、pg_xactや設定ファイルなどを含めて、それ以外のファイルをすべてソースクラスタからターゲットクラスタにコピーします。ベースバックアップと同様に、ディレクトリpg_dynshmem/、pg_notify/、pg_replslot/、pg_serial/、pg_snapshots/、pg_stat_tmp/、および、pg_subtrans/の内容は、ソースクラスタからコピーされるデータから省かれます。pgsql_tmpで始まるすべてのファイルやディレクトリは省かれます。ファイルbackup_label、tablespace_map、pg_internal.init、postmaster.opts、および、postmaster.pidも同様です。

4. フェイルオーバーで作成されたチェックポイントでWALリプレイを開始するためにbackup_labelファイルを作成し、動作中のソースから巻き戻す場合にはpg_current_wal_insert_lsn()の結果として定義される最小の整合のとれたLSNを、停止したソースから巻き戻す場合には最後のチェックポイントLSNをpg_controlファイルに設定します。

5. ターゲットが起動すると、PostgreSQLは必要なWALをすべてリプレイします。それにより、データディレクトリが整合のとれた状態になります。

pg_test_fsync

pg_test_fsync — PostgreSQLの最も高速なwal_sync_methodを決定する

概要

pg_test_fsync [option...]

説明

pg_test_fsyncは使用するシステムにおいて最速なwal_sync_methodは何かについて、合理的な見解を提供することを意図したものです。同時に識別されたI/O問題のイベントに診断情報を提供します。しかしpg_test_fsyncで示される差異が、実際のデータベーススループットにおいて差異として現れないことがあります。特に、多くのデータベースサーバは先行書き込みログにより速度制限されていないからです。pg_test_fsyncは各wal_sync_methodに対する平均ファイル同期処理時間をマイクロ秒単位で報告します。これはcommit_delayの値を最適化する時の情報としても使うことができます。

オプション

pg_test_fsyncは以下のコマンドラインオプションを受け付けます。

-f
--filename

テストデータを書き込むためのファイルの名前を指定します。このファイルはpg_walディレクトリがある、または格納する予定のファイルシステムと同じファイルシステムになければなりません。(pg_walにはWALファイルが含まれます。) デフォルトは現在のディレクトリ内のpg_test_fsync.outです。

-s
--secs-per-test

1テストあたりの秒数を指定します。テストあたりの時間を多くするほど、テストの精度が向上しますが、実行により時間がかかるようになります。デフォルトは、本プログラムがおよそ2分で完了することができる、5秒です。

-V
--version

pg_test_fsyncのバージョンを表示し、終了します。

-?
--help

pg_test_fsyncのコマンドライン引数の説明を表示し、終了します。

環境

環境変数PG_COLORは診断メッセージで色を使うかどうかを指定します。可能な値はalways、auto、neverです。

関連項目

postgres

pg_test_timing

pg_test_timing — 時間計測のオーバーヘッドを測定する

概要

```
pg_test_timing [option...]
```

説明

pg_test_timingはシステム上の時間計測のオーバーヘッドを測定し、またシステム時計が逆向きに進まないことを確認するためのツールです。時間のデータの収集に時間のかかるシステムでは、EXPLAIN ANALYZEの結果が不正確になることがあります。

オプション

pg_test_timingでは、以下のコマンド行オプションを指定できます。

-d *duration*
--duration=*duration*

テストの時間を秒単位で指定します。時間を長くすると、少しだけ結果が正確になり、またシステム時計が逆方向に進む問題を検出できる可能性が高くなります。デフォルトのテスト時間は3秒です。

-V
--version

pg_test_timingのバージョンを表示して終了します。

-?
--help

pg_test_timingのコマンドライン引数についてのヘルプを表示して終了します。

使用方法

結果の解釈

良い結果では、個々の時間計測のほとんど(90%以上)が1マイクロ秒未満になります。ループあたりの平均オーバーヘッド(per loop time)の値はさらに小さく、100ナノ秒未満になります。以下の例はIntel i7-860システムでTSC時計を使ったもので、非常に良い性能を示しています。

```
Testing timing overhead for 3 seconds.
Per loop time including overhead: 35.96 ns
Histogram of timing durations:
```

```
  < us   % of total      count
     1     96.40465   80435604
     2      3.59518    2999652
     4      0.00015        126
     8      0.00002         13
    16      0.00000          2
```

ループあたりの時間(per loop time)とヒストグラム(histogram)で時間の単位が異なることに注意して下さい。ループは数ナノ秒(ns)以内の精度を持つことができますが、個々の時間計測の呼び出しは1マイクロ秒(us)までの精度にしかできません。

エクゼキュータの時間計測オーバーヘッドの測定

問い合わせエクゼキュータがEXPLAIN ANALYZEを使って文を実行するとき、要約を表示する他に、個々のオペレーションについての時間計測もされます。次のpsqlプログラムで行を数えれば、システムのオーバーヘッドを調べることができます。

```
CREATE TABLE t AS SELECT * FROM generate_series(1,100000);
\timing
SELECT COUNT(*) FROM t;
EXPLAIN ANALYZE SELECT COUNT(*) FROM t;
```

測定に使ったi7-860システムでは、countの問い合わせを9.8ミリ秒で実行しましたが、EXPLAIN ANALYZEをつけたときは16.6ミリ秒かかりました。どちらの問い合わせもちょうど10万行を処理しています。この6.8ミリ秒の差は、行あたりの時間計測のオーバーヘッドが68ナノ秒であることを示しており、これはpg_test_timingによる推定値の約2倍です。この比較的小さな量のオーバーヘッドでも、countの文の完全な時間計測をすると70%長くかかりました。もっと本質的な問い合わせでは、時間計測のオーバーヘッドはあまり問題にならないでしょう。

時間の測定源の変更

一部の新しいLinuxシステムでは、時間データの収集に使う時計をいつでも変更することができます。2番目の例は、時間の測定源を、より遅いacpi_pmに変更することで、上の速い結果で使われたのと同じシステムでも、遅い結果が出るかもしれないことを示しています。

```
# cat /sys/devices/system/clocksource/clocksource0/available_clocksource
tsc hpet acpi_pm
# echo acpi_pm > /sys/devices/system/clocksource/clocksource0/current_clocksource
# pg_test_timing
Per loop time including overhead: 722.92 ns
Histogram of timing durations:
  < us   % of total      count
     1     27.84870    1155682
     2     72.05956    2990371
     4      0.07810       3241
     8      0.01357        563
```

```
    16      0.00007          3
```

この構成では、上のEXPLAIN ANALYZEの例の実行に115.9ミリ秒かかりました。つまり、時間計測に1061ナノ秒のオーバーヘッドとなりますが、やはりこのユーティリティによって直接測定したものに小さな数を掛けた程度の値です。この大きな時間計測のオーバーヘッドは、実際の問い合わせ自体は問い合わせに要した時間の本の一部を占めるだけで、大半はオーバーヘッドによって使われている、ということを示しています。この構成では、多くの時間計測オペレーションを含むEXPLAIN ANALYZEの合計は、時間計測のオーバーヘッドにより非常に大きな値になるでしょう。

FreeBSDでも時間の測定源を実行時に変更することができ、起動時に、選択されたタイマーに関する情報をログ出力します。

```
# dmesg | grep "Timecounter"
Timecounter "ACPI-fast" frequency 3579545 Hz quality 900
Timecounter "i8254" frequency 1193182 Hz quality 0
Timecounters tick every 10.000 msec
Timecounter "TSC" frequency 2531787134 Hz quality 800
# sysctl kern.timecounter.hardware=TSC
kern.timecounter.hardware: ACPI-fast -> TSC
```

他のシステムでは、起動時にのみ時間の測定源を設定することができます。古いLinuxシステムでは"clock"のカーネル設定が、この種の変更をする唯一の方法でした。比較的新しいLinuxでも、時計についての唯一のオプションが"jiffies"であるということもあります。jiffiesはLinuxの古いソフトウェア時計の実装で、十分に速い時間測定のハードウェアがあれば、以下の例のように良い精度を出すことができます。

```
$ cat /sys/devices/system/clocksource/clocksource0/available_clocksource
jiffies
$ dmesg | grep time.c
time.c: Using 3.579545 MHz WALL PM GTOD PIT/TSC timer.
time.c: Detected 2400.153 MHz processor.
$ pg_test_timing
Testing timing overhead for 3 seconds.
Per timing duration including loop overhead: 97.75 ns
Histogram of timing durations:
  < us   % of total      count
     1     90.23734   27694571
     2      9.75277    2993204
     4      0.00981       3010
     8      0.00007         22
    16      0.00000          1
    32      0.00000          1
```

時計のハードウェアと時間測定の正確さ

コンピュータでの正確な時間情報の収集は通常はハードウェアクロックを用いて行われ、その正確さのレベルは様々です。一部のハードウェアでは、オペレーティングシステムはシステムクロックの時間をほぼ直接

にプログラムに渡すことができます。システムクロックは、ある一定の時間ごとの単純な時間の割り込みを起こすチップによって提供されることもあります。いずれの場合でも、オペレーティングシステムのカーネルは、これらの詳細を隠して時間の測定源を提供します。しかし時間の測定源の正確さ、および結果を返す速さは、その基となっているハードウェアに依存して変わります。

時間を正確に保つことができないと、システムが不安定になることがあります。時間の測定源へのすべての変更について、非常に注意深く検査してください。オペレーティングシステムのデフォルトは、正確であることよりも信頼できることを重視して設定されているかも知れません。仮想マシンを使っている場合は、それと互換性があるとして推奨されている時間の測定源を調べてください。仮想マシンではタイマーをエミュレートするのに通常以上の困難があり、ベンダーによって推奨されるオペレーティングシステムごとの設定があることもあります。

タイムスタンプカウンタ(TSC)は、現行世代のCPU上で利用可能な最も正確な時間の測定源です。オペレーティングシステムがTSCクロックをサポートしていて、それが信頼出来るなら、システム時間をTSCで測ることが望ましいです。TSCが正確な時間測定源とならず、信頼できなくなる場合がいくつかあります。古いシステムでは、TSCの時計がCPUの温度に依存して変化するため、時間測定には利用できません。一部の古いマルチコアCPUでTSCを使うと、複数のコアの間で一貫しない時間が得られることがあります。この場合、時計が逆戻りすることがありますが、それがこのプログラムが検査対象としている問題の1つです。また最新のシステムでも、非常に積極的に電力節減を実施する設定の場合には、TSCの時間計測が不正確になることがあります。

新しいオペレーティングシステムには、既知のTSCの問題について検査し、問題が検出されると、遅いが、より安定した時間測定源を使うようにするものもあります。システムがTSC時計をサポートしているけれど、それがデフォルトになっていないとしたら、何らかの正しい理由によって無効にされたのかもしれません。一部のオペレーティングシステムでは、発生しうるすべての問題について正しく検知できないかも知れませんし、またTSCが正確でないとわかっている状況でもTSCを利用可能にするかも知れません。

高精度イベントタイマー(HPET)は、それが利用可能で、TSCが不正確なシステムにおいて、望ましいタイマーです。タイマーのチップ自体は、100ナノ秒単位の精度までプログラム可能ですが、システムクロックによっては、そこまでの正確さはないこともあります。

ACPI(Advanced Configuration and Power Interface)は電源制御（PM）タイマーを提供し、Linuxはこれをacpi_pmとして参照します。acpi_pmを利用した時計は、最高で300ナノ秒までの精度を提供します。

古いPCハードウェアで使われていたタイマーには、8254プログラマブルインターバルタイマー(PIT)、リアルタイムクロック(RTC)、APIC(Advanced Programmable Interrupt Controller)タイマー、サイクロン(Cyclone)などがあります。これらのタイマーは、ミリ秒の精度を目指したものです。

関連項目

EXPLAIN

pg_upgrade

pg_upgrade — PostgreSQLサーバインスタンスをアップグレードする

概要

```
pg_upgrade -b oldbindir -B newbindir -d oldconfigdir -D newconfigdir [option...]
```

説明

pg_upgrade(これまではpg_migratorと呼ばれていました)を使用することで、メジャーバージョンのアップグレード、例えば、9.5.8から9.6.4へ、10.7から11.2へのアップグレードで通常必要とされるデータのダンプ/リストアを行うことなく、PostgreSQLデータファイル内に格納されたデータをより最新のPostgreSQLメジャーバージョンに移行できます。これは、例えば9.6.2から9.6.3、10.1から10.2などマイナーバージョンのアップグレードでは必要ありません。

PostgreSQLのメジャーリリースでは通常、システムテーブルのレイアウトをよく変更する、多くの機能が追加されます。しかし内部データの格納書式はまれにしか変更されません。pg_upgradeはこの事実を使用して、 システムテーブルを新しく作成し、古いユーザデータファイルを単に再利用することで、高速なアップグレードを実施します。将来のメジャーリリースでついに古いデータ書式を読み取ることができなくなるようにデータ格納書式を変更した場合、pg_upgradeではこうしたアップグレードを扱うことができません。(コミュニティはこうした状況を防ごうと考えています。)

pg_upgradeは古いクラスタと新しいクラスタとの間で、例えばコンパイル時の設定に互換性があるかどうか、32ビットバイナリか64ビットバイナリかなど、バイナリ互換性があることを確実にするために最善を尽くします。任意の外部モジュールがバイナリ互換であることも重要ですが、これはpg_upgradeでは検査することができません。

pg_upgradeは8.4.X以降から現時点のPostgreSQLのメジャーリリース(スナップショット版やβリリースを含む)へのアップグレードをサポートします。

オプション

pg_upgradeは以下のコマンドライン引数を受け付けます。

`-b bindir`
`--old-bindir=bindir`

古いPostgreSQLの実行ファイル格納ディレクトリ。PGBINOLD環境変数

`-B bindir`
`--new-bindir=bindir`

新しいPostgreSQLの実行ファイル格納ディレクトリ。デフォルトはpg_upgradeのあるディレクトリ。PGBINNEW環境変数

`-c`
`--check`

クラスタの検査のみを行い、データの変更を行いません。

-d configdir
--old-datadir=configdir

古いクラスタの設定データディレクトリ。PGDATAOLD環境変数

-D configdir
--new-datadir=configdir

新しいクラスタの設定データディレクトリ。PGDATANEW環境変数

-j njobs
--jobs=njobs

使用する同時実行プロセスまたはスレッド数。

-k
--link

新しいクラスタにファイルをコピーするのではなく、ハードリンクを使用します。

-o options
--old-options options

古いpostgresコマンドに直接渡すオプションです。複数の起動オプションが追加されます。

-O options
--new-options options

新しいpostgresコマンドに直接渡すオプションです。複数の起動オプションが追加されます。

-p port
--old-port=port

古いクラスタのポート番号。PGPORTOLD環境変数

-P port
--new-port=port

新しいクラスタのポート番号。PGPORTNEW環境変数

-r
--retain

正常に完了した場合であってもSQLファイルとログファイルを保持します。

-s dir
--socketdir=dir

アップグレード中にpostmasterソケット用に利用するディレクトリ。デフォルトは現在の作業ディレクトリです。環境変数PGSOCKETDIR

-U username
--username=username

クラスタのインストールユーザの名称。PGUSER環境変数

-v
--verbose

冗長な内部ログを有効にします。

-V
--version

バージョン情報を表示し、終了します。

--clone

新しいクラスタにファイルをコピーする代わりに、効率的なファイルの複製(システムのいくつかでは「reflink」としても知られています)を使います。これは、データファイルをほぼ瞬間的にコピーし、古いクラスタに手をつけずに-k/--linkの速度の利点を与えることになるでしょう、

ファイルの複製はいくつかのオペレーティングシステムとファイルシステムでのみサポートされています。選択されたもののサポートされていなければ、pg_upgradeの実行はエラーになります。現在のところ、BtrfsとXFS(のreflinkサポート付きで作られたファイルシステム)のLinux(カーネル4.5以降)と、APFSのmacOSでサポートされています。

-?
--help

使用方法を表示し、終了します。

使用方法

pg_upgradeを用いたアップグレードを行う手順を示します。

1. **古いクラスタの移動(省略可能)**

   バージョンに関連したインストレーションディレクトリ(例えば/opt/PostgreSQL/13)を使用しているのであれば、古いクラスタを移動する必要はありません。グラフィカルインストーラはすべて、バージョンに関連したインストレーションディレクトリを使用します。

   インストレーションディレクトリがバージョンに関連していない(例えば/usr/local/pgsql)のであれば、新しいPostgreSQLのインストレーションに干渉しないように、現在のPostgreSQLインストレーションディレクトリを移動しなければなりません。一度現在のPostgreSQLサーバを停止させていれば、PostgreSQLのインストレーションの名前を変更しても安全です。古いディレクトリが/usr/local/pgsqlであれば、以下のようにディレクトリ名を変更します。

   ```
   mv /usr/local/pgsql /usr/local/pgsql.old
   ```

2. **ソースからのインストールの場合の新しいバージョンの構築**

   古いクラスタと互換性を持つようなconfigureオプションを付けて新しいPostgreSQLソースを構築してください。pg_upgradeはアップグレードを始める前にすべての設定が互換性を持っていることを確認するためにpg_controldataを検査します。

3. **新しいPostgreSQLのバイナリのインストール**

   新しいサーバのバイナリとサポートファイルをインストールしてください。pg_upgradeはデフォルトのインストレーションに含まれています。

   ソースからインストールする場合、独自の場所に新しいサーバをインストールしたければ、以下のようにprefixを使用してください。

   ```
   make prefix=/usr/local/pgsql.new install
   ```

4. **新しいPostgreSQLクラスタを初期化**

   initdbを使用して新しいクラスタを初期化してください。ここでも、古いクラスタに合うように互換性があるinitdbのオプションを使用してください。あらかじめ組み込まれたインストーラの多くは、この手順を自動的に実施します。新しいクラスタを起動する必要はありません。

5. **独自の共有オブジェクトファイルをインストール**

   例えば、pgcrypto.so、contribや何らかのその他のソースからインストールしたものなど、古いクラスタで使用していた独自の共有オブジェクトファイル(またはDLL)をすべて、新しいクラスタにインストールしてください。例えばCREATE EXTENSION pgcryptoなどのスキーマ定義はインストールしないでください。これらは古いクラスタからアップグレードされるためです。また、独自の全文検索ファイル(辞書、同義語、類語辞書、ストップワード)も新しいクラスタにコピーしなければなりません。

6. **認証の調整**

   pg_upgradeは古いサーバと新しいサーバに複数回接続します。このため、pg_hba.conf内でpeer認証に設定、あるいは、~/.pgpassファイル(33.15参照)を使用するようにした方が良いかもしれません。

7. **両サーバの停止**

   両サーバが停止していることを確実にしてください。例えばUnixでは以下を使用します。

   ```
   pg_ctl -D /opt/PostgreSQL/9.6 stop
   pg_ctl -D /opt/PostgreSQL/13 stop
   ```

   Windowsでは以下

   ```
   NET STOP postgresql-9.6
   NET STOP postgresql-13
   ```

   ストリーミングレプリケーションおよびログシッピングのスタンバイサーバは、後のステップまで実行中のまま残すことができます。

8. **スタンバイサーバのアップグレードの準備**

   スタンバイサーバをステップ 10の節で示した手順でアップグレードする場合は、pg_controldataを実行して、古いスタンバイサーバが、古いプライマリ及びスタンバイのクラスタに同期していることを確認してください。すべてのクラスタで「Latest checkpoint location」(最終チェックポイント位置)の値が一致することを確認してください。(古いスタンバイサーバが、古いプライマリより先にシャットダウンさ

れた場合、もしくは古いスタンバイサーバがまだ稼働中の場合は一致しないでしょう。) また、新しいプライマリのクラスタのpostgresql.confファイルでwal_levelをminimalに絶対に設定しないようにしてください。

9. **pg_upgradeの実行**

古いサーバのものではなく、常に新しいサーバのpg_upgradeバイナリを実行してください。pg_upgradeは古いクラスタおよび新しいクラスタのデータと実行形式ファイルの格納ディレクトリ(bin)の指定を要求します。また、ユーザやポート番号の指定や、デフォルト動作のコピーではなくデータファイルのリンクや複製を使用するかどうかを指定することができます。

リンクモードを使用する場合、アップグレードは非常に高速になり(ファイルのコピーがありません)、ディスク容量が少なくなりますが、アップグレード後に新しいクラスタを一度でも実行してしまうと、古いクラスタにアクセスすることができなくなります。リンクモードはまた、古いクラスタと新しいクラスタのデータディレクトリが同じファイルシステムにあることが必要です。(テーブル空間およびpg_walは異なるファイルシステムに置くことができます。) 複製モードは同じ速度とディスク容量の利点を提供しますが、新しいクラスタを一度実行しても、古いクラスタが使えなくなる訳ではありません。複製モードはまた、新旧のデータディレクトリが同じファイルシステム内にあることを要求します。このモードは特定のオペレーティングシステムのファイルシステム上でのみ利用可能です。

--jobsオプションにより複数のCPUコアを使用して、並行してファイルのコピー・リンク、データベーススキーマのダンプと再ロードを行うことができます。この値の最大値として検討を始める際には、CPUコア数またはテーブル空間数を勧めます。このオプションはマルチプロセッサを持つマシンで実行している複数のデータベースサーバをアップグレードする時間を大きく減らします。

Windowsユーザの場合、管理者アカウントでログオンしなければなりません。また、postgresユーザとしてシェルを起動し、適切なパスを設定してください。

```
RUNAS /USER:postgres "CMD.EXE"
SET PATH=%PATH%;C:\Program Files\PostgreSQL\13\bin;
```

そして、以下の例のように、引用符でくくったディレクトリを付けてpg_upgradeを実行してください。

```
pg_upgrade.exe
        --old-datadir "C:/Program Files/PostgreSQL/9.6/data"
        --new-datadir "C:/Program Files/PostgreSQL/13/data"
        --old-bindir "C:/Program Files/PostgreSQL/9.6/bin"
        --new-bindir "C:/Program Files/PostgreSQL/13/bin"
```

起動後、pg_upgradeは2つのクラスタに互換性があるかどうか検証し、その後、アップグレードを行います。検査のみを行うpg_upgrade --checkを使用することができます。この場合は古いサーバは稼動中であっても構いません。またpg_upgrade --checkは、アップグレード後に手作業で行わなければならない調整作業があればその概要を示します。リンクまたは複製モードを使用する予定であれば、モード固有の検査を有効にするために--checkを付けて--linkまたは--cloneオプションを使用すべきです。pg_upgradeは現在のディレクトリに対する書き込み権限を必要とします。

言うまでもありませんが、アップグレード中はクラスタにアクセスしてはいけません。意図しないクライアント接続を防ぐために、デフォルトではpg_upgradeは50432ポートでサーバを稼働します。古いクラ

スタと新しいクラスタが同時に稼動することはありませんので、両クラスタで同じポート番号を使用することができます。しかし実行中の古いクラスタを検査する時には、新旧で異なるポート番号でなければなりません。

データベーススキーマのリストア中にエラーが発生した場合、pg_upgradeは終了しますので、後述のステップ 16で示すように古いクラスタに戻さなければなりません。再びpg_upgradeを試すためには、pg_upgradeによるスキーマのリストアが成功するように古いクラスタを変更しなければなりません。問題がcontribモジュールであれば、そのモジュールがユーザデータを格納するために使用されていないことが前提ですが、古いクラスタからそのcontribモジュールをアンインストールし、アップグレードした後に新しいクラスタにそれをインストールする必要があるかもしれません。

10. **ストリーミングレプリケーションおよびログシッピングのスタンバイサーバのアップグレード**

リンクモードを使用した場合で、ストリーミングレプリケーション(26.2.5参照)またはログシッピング(26.2参照)のスタンバイサーバがある場合、以下の手順に従って、素早くアップグレードすることができます。スタンバイサーバでpg_upgradeは実行せず、代わりにプライマリでrsyncを実行することになります。どのサーバもまだ起動しないでください。

リンクモードを使用しなかった場合、rsyncの機能が使えない場合や使いたくない場合、あるいはもっと容易な方法を望む場合は、この節の手順を読み飛ばし、pg_upgradeが完了して新しいプライマリが起動したら、単純にスタンバイサーバを再作成してください。

a. **PostgreSQLの新しいバイナリをスタンバイサーバにインストール**

すべてのスタンバイサーバで新しいバイナリとサポートファイルがインストールされていることを確実にしてください。

b. **新しいスタンバイのデータディレクトリが存在しないことの確認**

新しいスタンバイのデータディレクトリが存在しないか空であることを確実にしてください。initdbが実行されている場合は、スタンバイサーバの新しいデータディレクトリを削除してください。

c. **カスタム共有オブジェクトファイルのインストール**

新しいプライマリのクラスタにインストールしたのと同じカスタム共有オブジェクトファイルを新しいスタンバイにインストールしてください。

d. **スタンバイサーバの停止**

スタンバイサーバがまだ実行中なら、上記の手順に従って停止してください。

e. **設定ファイルの保存**

スタンバイの設定ディレクトリの設定ファイルで保持する必要のあるもの、例えばpostgresql.conf(とそれがインクルードしているファイル)やpostgresql.auto.conf、pg_hba.confを保存してください。これらは次のステップで上書きあるいは削除されるからです。

f. **rsyncの実行**

リンクモードを使用する場合、rsyncを使用してスタンバイサーバを素早くアップグレードすることができます。これを実行するには、古いデータベースクラスタのディレクトリおよび新しいデータ

ベースクラスタのディレクトリより上位にあるプライマリサーバ上のディレクトリで、プライマリ上で各スタンバイサーバに対して以下を実行します。

```
rsync --archive --delete --hard-links --size-only --no-inc-recursive old_cluster
 new_cluster remote_dir
```

ここでold_clusterおよびnew_clusterはプライマリのカレントディレクトリからの相対パス、remote_dirはスタンバイサーバ上の古いクラスタと新しいクラスタのディレクトリの上位のディレクトリです。プライマリとスタンバイの指定したディレクトリより下のディレクトリ構造は一致しなければなりません。リモートディレクトリの指定の詳細についてはrsyncのマニュアルを参照してください。例を示します。

```
rsync --archive --delete --hard-links --size-only --no-inc-recursive /opt/PostgreSQL/9.5
 \
      /opt/PostgreSQL/9.6 standby.example.com:/opt/PostgreSQL
```

rsyncの--dry-runオプションを使うことで、そのコマンドが何をするか確認することができます。rsyncは少なくとも1つのスタンバイに対して、プライマリ上で実行しなければなりませんが、アップグレード済みのスタンバイがまだ起動していなければ、そのスタンバイ上でrsyncを実行して、他のスタンバイをアップグレードすることができます。

これが実行するのはpg_upgradeのリンクモードが作成したプライマリサーバ上の古いクラスタのファイルと新しいクラスタのファイルを接続するリンクについて記録することです。次にスタンバイの古いクラスタ内の一致するファイルを探し、スタンバイの新しいクラスタ内に対応するリンクを作成します。プライマリ上でリンクされなかったファイルは、プライマリからスタンバイにコピーされます。(それらは通常は小さいです。) これにより高速なアップグレードが可能になります。残念ながらrsyncは一時テーブルやログを取らないテーブルに関連したファイルを不要であるにも関わらずコピーします。これらのファイルが通常はスタンバイサーバに存在しないからです。

テーブル空間を使用している場合は、各テーブル空間のディレクトリについて、同様のrsyncコマンドを実行する必要があります。例を示します。

```
rsync --archive --delete --hard-links --size-only --no-inc-recursive /vol1/pg_tblsp/
PG_9.5_201510051 \
      /vol1/pg_tblsp/PG_9.6_201608131 standby.example.com:/vol1/pg_tblsp
```

pg_walをデータディレクトリの外側に移動している場合は、そのディレクトリについてもrsyncを実行しなければなりません。

g. **ストリーミングレプリケーションおよびログシッピングのスタンバイサーバの設定**

ログシッピングのためにサーバを設定します。(スタンバイはまだプライマリと同期されているので、pg_start_backup()とpg_stop_backup()を実行したり、ファイルシステムのバックアップを取得したりする必要はありません。)

11. **pg_hba.confの復旧**

    pg_hba.confを変更している場合は、元の認証設定に戻してください。また新しいクラスタにおけるこの他の設定ファイル、例えばpostgresql.conf(とそれがインクルードしているファイル)、postgresql.auto.confも古いクラスタに合わせるように調整する必要があります。

12. **新しいサーバの起動**

    ここで新しいサーバが安全に起動できます。それからrsyncしたすべてのスタンバイサーバを起動できます。

13. **移行後の処理**

    アップグレード後の処理が必要な場合、pg_upgradeはその終了時に警告を発します。また、管理者として実行しなければならないスクリプトファイルを生成します。このスクリプトファイルは、アップグレード後の処理を必要とするデータベースそれぞれに接続します。各スクリプトは以下を使用して実行しなければなりません。

    ```
    psql --username=postgres --file=script.sql postgres
    ```

    スクリプトは任意の順番で実行することができ、また、実行が終わったら削除することができます。

    > **注意**
    >
    > 一般的には、再構築用のスクリプトの実行が完了するより前に、再構築用のスクリプトで参照されるテーブルにアクセスすることは安全ではありません。これを行うと間違った結果が生じたり、性能が劣化することがあり得ます。再構築用のスクリプトで参照されないテーブルにはすぐにアクセスすることができます。

14. **統計情報**

    オプティマイザの統計情報はpg_upgradeにより転送されませんので、アップグレード後に統計情報を再生成するコマンドを実行するように指示されます。新しいクラスタに合わせるために接続パラメータを設定する必要があるかもしれません。

15. **古いクラスタの削除**

    アップグレード処理が満足いくものであれば、pg_upgradeの終了時に示されたスクリプトを使用して、古いクラスタのデータディレクトリを削除することができます。(古いデータディレクトリ内にユーザ定義のテーブル空間がある場合には、自動削除は不可能です。)(bin、shareなど)古いインストレーションディレクトリも削除できます。

16. **古いクラスタへの戻し**

    pg_upgradeを実行した後に古いクラスタに戻したい場合、いくつかの選択肢があります。

    - --checkオプションが使われた場合、古いクラスタは変更されません。再起動できます。
    - --linkオプションが使われなかった場合、古いクラスタは変更されません。再起動できます。

- --linkオプションが使われた場合、データファイルは古いクラスタと新しいクラスタは共有されるかもしれません。

  - リンク処理が始まる前にpg_upgradeが中断すれば、古いクラスタは変更されません。古いクラスタを再起動できます。

  - 新しいクラスタを開始しなかった場合、リンク処理が始まる時に$PGDATA/global/pg_controlに.old接尾辞が追加される点を除き、古いクラスタは変更されません。古いクラスタを再度使用するためには、$PGDATA/global/pg_controlから.old接尾辞を取り除きます。その後で古いクラスタを再起動できます。

  - 新しいクラスタを開始した場合、共有ファイルに書き込んでいますので古いクラスタを使うことは安全ではありません。この場合、古いクラスタをバックアップからリストアすることが必要でしょう。

注釈

pg_upgradeは現在の作業ディレクトリにスキーマのダンプ等、様々な作業ファイルを作成します。セキュリティのため、他のユーザがそのディレクトリを読んだり書いたりできないことを確実にしてください。

pg_upgradeは新旧のデータディレクトリで短命なpostmasterを起動します。このpostmasterの通信用の一時的なUnixのソケットファイルが、デフォルトでは現在の作業ディレクトリに作られます。状況に依っては、カレントディレクトリのパス名が有効なソケット名には長過ぎる場合もあるでしょう。その場合、-sオプションを使ってソケットファイルをより短いパス名のディレクトリに置くことができます。セキュリティのため、他のユーザがそのディレクトリを読んだり書いたりできないことを確実にしてください。(これはWindowsではサポートされていません。)

インストレーションに影響する場合、失敗、再構築、インデックス再作成はすべてpg_upgradeにより報告されます。テーブルおよびインデックスを再構築するアップグレード後処理スクリプトは自動的に生成されます。多くのクラスタのアップグレードを自動化することを考えているのであれば、すべてのクラスタのアップグレードにおいて同一のデータベーススキーマを持つクラスタが同じアップグレード後処理を必要とすることが分かるはずです。これはアップグレード後処理がデータ自体ではなくデータベーススキーマに基づいているためです。

展開試験を行うのであれば、古いクラスタのスキーマのみをコピーしたものを作成し、ダミーデータを挿入してから、アップグレードを行ってください。

pg_upgradeは次のOIDを参照するreg*システムデータ型を使うテーブル列を含むデータベースのアップグレードをサポートしません。

```
regcollation
regconfig
regdictionary
regnamespace
regoper
regoperator
regproc
regprocedure
```

(regclass、regrole、regtypeはアップグレード可能です。)

設定ファイルしか持たないディレクトリを使用するPostgreSQL 9.2よりも前のクラスタをアップグレードする場合、例えば-d /real-data-directory -o '-D /configuration-directory'のように、実際のデータディレクトリの場所をpg_upgradeに通知しなければならず、さらに設定用ディレクトリの場所をサーバに通知しなければなりません。

デフォルト以外のUnixドメインソケットディレクトリを使用する、または新しいクラスタのデフォルトとは異なるデフォルトを使用する9.1より前の古いサーバを使用している場合、古いサーバのソケット位置を指し示すようにPGHOSTを設定してください。(これはWindowsでは関係ありません。)

リンクモードを使用したいが、新しいクラスタを起動した時に古いクラスタを変更させたくない場合は、複製モードの使用を検討してください。もしそれが利用可能でないなら、古いクラスタのコピーを取得してからリンクモードでアップグレードを行ってください。有効な古いクラスタのコピーを取得するためには、サーバ稼動中にrsyncを使用して古いクラスタの変動があるかもしれないコピーを作成し、古いサーバを停止させた後に、rsync --checksumを再度実行して一貫性を保つために何らかの変更をコピーに反映させます。(rsyncはファイル更新時刻の粒度が1秒しかないので--checksumが必要です。) 例えば25.3.3で説明したpostmaster.pidなど、一部のファイルを除外したいと考えるかもしれません。スナップショットやコピーは同時、もしくはデータベースサーバが停止しているときに作らなければなりませんが、ファイルシステムがファイルシステムのスナップショットやコピーオンライトファイルコピーをサポートしているのなら、それを古いクラスタとテーブル空間のバックアップをするのに使うことができます。

関連項目

initdb, pg_ctl, pg_dump, postgres

pg_waldump

pg_waldump — PostgreSQLデータベースクラスタの先行書き込みログを可読性が高い表現で表示する

概要

pg_waldump [option...] [startseg [endseg]]

説明

pg_waldumpは先行書き出しログ（WAL）を表示します。主にデバッグや学習目的に有用です。

データディレクトリへの読み取り専用のアクセスが必要ですので、このユーティリティはサーバをインストールしたユーザによってのみ実行することができます。

オプション

以下のコマンドラインオプションは場所や出力書式を制御します。

startseg

指定したログセグメントファイルから読み取りを開始します。これは暗黙的に検索されるファイルのパスや使用するタイムラインを決定します。

endseg

指定したログセグメントファイルを読み取り後終了します。

-b
--bkp-details

バックアップブロックに関する詳細情報を出力します。

-e end
--end=end

ログストリームの終了点まで読み取る代わりに、指定したWAL位置で読み取りを終了します。

-f
--follow

有効なWALの終わりに到達した後、新しいWALが現れるのを1秒間に1回ポーリングし続けます。

-n limit
--limit=limit

指定レコード数を表示し、終了します。

-p path
--path=path

ログセグメントファイルを見つけ出すディレクトリ、あるいはログセグメントファイルが含まれるpg_walサブディレクトリが含まれるディレクトリを指定します。デフォルトではカレントディレクトリ、カレントディレクトリ内のpg_walサブディレクトリ、PGDATAのpg_walサブディレクトリから検索されます。

-q
--quiet

エラーを除いて、出力を表示しません。このオプションは、WALレコードの範囲の解析に成功したかを知りたいがレコードの内容を気にしない場合には有用でしょう。

-r rmgr
--rmgr=rmgr

指定されたリソースマネージャによって生成されたレコードのみを表示します。listが名称として渡された場合は、有効なリソースマネージャの一覧を表示します。

-s start
--start=start

読み取りを始めるWAL位置です。デフォルトでは、最も過去のファイルの中で見つかった最初の有効なログレコードから読み取りを始めます。

-t timeline
--timeline=timeline

ログレコードの読み取り先のタイムラインです。デフォルトでは、startsegが指定されている場合はstartseg内の値が使用されます。指定がない場合のデフォルトは1です。

-V
--version

pg_waldumpのバージョンを表示し終了します。

-x xid
--xid=xid

指定されたトランザクションIDで印付けられたレコードのみを表示します。

-z
--stats[=record]

個々のレコードの代わりに要約統計(レコードの数とサイズ、および全ページのイメージ)を表示します。オプションでrmgr毎の代わりにレコード毎の統計を生成します。

-?
--help

pg_waldumpのコマンドライン引数に関する説明を表示し、終了します。

環境

PGDATA

データディレクトリ。-pオプションも参照してください。

PG_COLOR

診断メッセージで色を使うかどうかを指定します。可能な値はalways、auto、neverです。

注釈

サーバが実行中の場合は間違った結果になることがあります。

指定されたタイムラインのみが表示されます（指定がなければデフォルトのみが表示）。他のタイムラインのレコードは無視されます。

pg_waldumpは拡張子.partialのWALファイルを読むことはできません。読む必要がある場合は、ファイル名から拡張子.partialを削除してください。

関連項目

29.5

postgres

postgres — PostgreSQLデータベースサーバ

概要

postgres [option...]

説明

postgresは、PostgreSQLのデータベースサーバです。クライアントアプリケーションがデータベースに接続するためには、稼働中のpostgresインスタンスに(ネットワークを介して、またはローカルで)接続する必要があります。その後、postgresは接続を取り扱うために別のサーバプロセスを開始します。

1つのpostgresインスタンスは常に1つのデータベースクラスタのデータを管理します。データベースクラスタとは、共通のファイルシステム領域(「データ領域」)に格納されているデータベースの集まりのことです。1つのシステム上で、同時に複数のpostgresインスタンスを実行することは可能ですが、それらは異なるデータ領域と、異なる接続ポート(下記参照)を使用する必要があります。postgresは、その起動時にデータ領域の場所を知らなければなりません。その場所を-Dオプションまたは環境変数PGDATAによって指定しなければなりません。デフォルト値はありません。通常、-DまたはPGDATAは、initdbで作成されたデータ領域ディレクトリを直接指し示します。他のファイルレイアウトを取ることもできます。19.2を参照してください。

デフォルトでは、postgresはフォアグラウンドで起動し、ログメッセージを標準エラー出力に出力します。実運用では、postgresアプリケーションはおそらくマシン起動時にバックグラウンドで起動されるはずです。

またpostgresはシングルユーザモードで呼び出すこともできます。このモードは、主にinitdbによる初期化処理の中で使用されます。これを、デバッグのためや障害からの復旧時に使用することもできます。実際には発生するはずのプロセス間通信やロック処理が発生しませんので、シングルユーザモードがサーバのデバッグに必ずしも適したものではないことに注意してください。シェルからシングルユーザモードで起動された場合、ユーザは問い合わせを入力できます。結果は画面上に表示されますが、その形式はエンドユーザ向けではなく開発者向けのものです。シングルユーザモードでは、セッションユーザはIDが1のユーザと設定され、暗黙的にスーパーユーザの権限がこのユーザに与えられます。このユーザは実際に存在していなくても構いません。このため、シングルユーザモードはシステムカタログの偶発的な損傷などから手動で復旧するために使用することができます。

オプション

postgresには、下記のコマンドライン引数を指定できます。オプションに関する詳細は第19章を参照してください。また、設定ファイルを使用することによって、これらのほとんどのオプションについて入力する手間を省くことができます。一部の(安全な)オプションは接続中のクライアントから設定することもできます。その設定方法はアプリケーションに依存し、また、そのセッションでのみ適用されます。たとえば、環境変数PGOPTIONSを設定すると、libpqに基づくクライアントはその文字列をサーバに渡します。渡された文字列はpostgresコマンドラインオプションとして解釈されます。

一般的な目的

-B nbuffers

サーバプロセスが使用する共有バッファの数を設定します。このパラメータのデフォルト値はinitdbにより自動的に決まります。このオプションを指定することは設定パラメータshared_buffersを設定することと同等です。

-c name=value

指定の実行時パラメータを設定します。PostgreSQLでサポートされる設定パラメータは第19章で説明します。実際には、他の多くのコマンドラインオプションはこのようなパラメータへの代入を簡略化したものです。-cは複数のパラメータを設定するために複数個使用することができます。

-C name

指定された名前の実行時パラメータの値を表示し、終了します、(詳細については上の-cを参照してください。) これは実行中のサーバに対して使用することができ、postgresql.confからの値やその起動時に与えられたパラメータにより変更された値を返します。クラスタが起動した時に与えられたパラメータは反映されません。

このオプションは、pg_ctlなど、サーバインスタンスと連携する他のプログラムが設定パラメータ値を問い合わせることを意図しています。ユーザ向けのアプリケーションでは代わりにSHOWやpg_settingsビューなどを使用すべきです。

-d debug-level

デバッグレベルを設定します。大きな値が設定されているほど、より多くのデバッグ情報がサーバのログに出力されます。値として設定できる範囲は、1から5までです。特定のセッションで-d 0を渡すこともできます。この設定により、親のpostgresプロセスのサーバログレベルがこのセッションに伝播されません。

-D datadir

ファイルシステム上のデータベース設定ファイルの場所を指定します。詳細は19.2を参照してください。

-e

デフォルトの日付データ形式を「ヨーロッパ式」、つまりDMYの順番にします。これにより、いくつかの日付出力形式でも、月の前に日が表示されるようになります。詳細は8.5を参照してください。

-F

性能向上のためにfsync呼び出しを無効とします。ただしこの場合、システムクラッシュ時にデータが損傷する危険性があります。このオプションの指定は設定パラメータfsyncを無効にすることと同一です。このオプションを使用する時は、事前に詳細な文書を一読してください。

-h hostname

postgresがクライアントアプリケーションとの接続を監視するホスト名、または、IPアドレスを指定します。この値には、カンマで区切ったアドレスの一覧、あるいは、全ての利用可能なインタフェースを監視

することを意味する*を指定することができます。空の値を指定すると、IPアドレスをまったく監視しないことを意味します。この場合、サーバへの接続には、Unixドメインソケットのみが使用されます。デフォルトではlocalhostのみを監視します。このオプションを指定することは設定パラメータlisten_addressesを設定することと同等です。

-i

リモートクライアントからのTCP/IP（インターネットドメイン）経由の接続を可能とします。このオプションを設定しない場合には、ローカルからの接続のみが使用可能となります。このオプションは、postgresql.conf内のlisten_addressesを、あるいは-hを*に設定することと同じ意味を持ちます。

このオプションはlisten_addressesの全ての機能を実現することができないため廃止が予定されています。通常は、直接listen_addressesを設定する方法をお勧めします。

-k directory

postgresがクライアントアプリケーションからの接続を監視するUnixドメインソケットのディレクトリを指定します。ディレクトリをカンマで区切ったリストを値とすることもできます。空の値を指定すると、Unixドメインソケットをまったく監視しません。この場合サーバに接続するためにはTCP/IPのみを使用することができます。デフォルトでは、通常/tmpとなっていますが、これはコンパイルに変更できます。このオプションを指定することは設定パラメータunix_socket_directoriesを設定することと同等です。

-l

SSLを使用して、安全な接続を行います。このオプションを使用するためには、PostgreSQLのコンパイル時にSSLを有効にする必要があります。SSLの使用に関する詳細は18.9を参照してください。

-N max-connections

このサーバが受け付けるクライアント接続数の最大値を設定します。このパラメータのデフォルト値はinitdbにより自動的に選択されます。このオプションを指定することは設定パラメータmax_connectionsを設定することと同じ意味を持ちます。

-o extra-options

extra-optionsで指定されたコマンドライン形式の引数は、このpostgresプロセスにより開始される全てのサーバプロセスに引き渡されます。

extra-options内の空白文字は、それがバックスラッシュ(\)でエスケープされていなければ、引数を区切るものと解釈されます。文字としてのバックスラッシュを記述するには、\\と書いてください。複数の引数は、-oを複数回使うことでも指定できます。

このオプションの使用は非推奨です。サーバプロセス用のすべてのコマンドラインオプションはpostgresコマンドラインで直接指定することができます。

-p port

クライアントアプリケーションからの接続をpostgresが監視するTCP/IPポート、またはローカルUnixドメインソケットファイルの拡張子を指定します。デフォルトでは、環境変数PGPORT、PGPORTが設定されていない場合はコンパイル時に設定された値（通常5432）が使用されます。デフォルトのポート以外を指定した場合には、コマンドラインオプション、またはPGPORTを使用して、全てのクライアントアプリケーションが同じポートを指定するようにしなければなりません。

-s

各コマンドの終了時に時間情報や他の統計情報を表示します。これはベンチマークやバッファ数の調整時の使用に適しています。

-S work-mem

ソート処理やハッシュテーブル処理で一時ディスクファイルに頼る前に使用する基本的なメモリ量を指定します。設定パラメータ19.4.1のwork_memの説明を参照してください。

-V
--version

postgresのバージョンを表示し、終了します。

--name=value

指定した実行時パラメータを設定します。-cの省略形式です。

--describe-config

このオプションは、サーバの内部設定変数、その説明、および、デフォルト値をタブ区切りのCOPY書式でダンプ出力します。これは、管理用ツールでの使用を主目的として設計されました。

-?
--help

postgresのコマンドライン引数に関する説明を表示し、終了します。

開発者向けオプション

ここで説明するオプションは、主にデバッグ用に、場合によっては深刻な障害を受けたデータベースの復旧を補助するために使われます。これらを運用状態のデータベースの設定として使用する理由はないはずです。これらのオプションを説明しておくのは、PostgreSQLシステム開発者が利用するためです。さらに、これらのオプションは、いずれも将来のリリースで予告なしに変更・廃止される可能性があります。

-f { s | i | o | b | t | n | m | h }

特定のスキャンと結合方式の使用を禁止します。sはシーケンシャルスキャン、iはインデックススキャンを無効にします。o、b、tはそれぞれ、インデックスオンリースキャン、ビットマップインデックススキャン、TIDスキャンを無効にします。nはネステッドループ結合、mはマージ結合、hはハッシュ結合を無効にします。

シーケンシャルスキャンとネステッドループ結合は、完全に無効にすることはできません。-fsオプションと-fnオプションは、単に「他の選択肢があるならば、これらの種類の計画を使わないようにする」ということをオプティマイザに指示するだけです。

-n

これはサーバプロセスが異常終了する問題をデバッグするためのオプションです。このような状態では、他のすべてのサーバプロセスに対し、終了し、共有メモリやセマフォを再初期化するように通知するのが、通常の戦略です。これはエラーが起きたサーバプロセスが、終了する前に、何かしら共有状態を破

損したかもしれないからです。このオプションは、postgresが共有データ構造を再初期化しないように指定します。知識のあるシステムプログラマであれば、その後にデバッガを使用して共有メモリやセマフォの状態を検証することができます。

-O

システムテーブルの構造の変更を可能にします。これはinitdbによって使われます。

-P

システムテーブルの読み込みの際にシステムインデックスを無視します。しかし、システムテーブルの更新の際にはインデックスを更新します。破損したシステムインデックスから復旧する場合に有用です。

-t pa[rser] | pl[anner] | e[xecutor]

それぞれの主要なシステムモジュールに関連する問い合わせに対し、時間に関する統計情報を表示します。このオプションは-sオプションと一緒に使うことはできません。

-T

これはサーバプロセスが異常終了する問題をデバッグするためのオプションです。このような状態では、他のすべてのサーバプロセスに対し、終了し、共有メモリやセマフォを再初期化するように通知するのが、通常の戦略です。これはエラーが起きたサーバプロセスが、終了する前に、何かしら共有状態を破損したかもしれないからです。このオプションは、postgresが他のすべてのサーバプロセスに対しSIGSTOPシグナルを送信して、終了させるのではなく停止させることを指定します。これによりシステムプログラマはすべてのサーバプロセスのコアダンプを手作業で収集することができます。

-v protocol

特定のセッションで使われるフロントエンド/バックエンドプロトコルのバージョン番号を指定します。このオプションは内部使用のみを目的としたものです。

-W seconds

新規サーバプロセスが起動するとき、認証手続きが行われた後、指定した秒数の遅延が発生します。サーバプロセスにデバッガを接続する時間を提供することが目的です。

シングルユーザモード用のオプション

以下のオプションはシングルユーザモードに対してのみ適用されます（以下のSingle-User Mode参照）。

--single

シングルユーザモードを選択します。これはコマンドラインで最初の引数でなければなりません。

database

アクセスするデータベースの名前を指定します。これはコマンドラインの最後の引数でなければなりません。省略時は、デフォルトでユーザ名になります。

-E

すべてのコマンドをその実行前に標準出力にエコー表示します。

`-j`

コマンド入力の終端文字として、単なる改行ではなく、セミコロンとそれに続く2つの改行を使用します。

`-r filename`

全てのサーバログ出力を`filename`に送ります。このオプションは、コマンド行オプションとして指定された場合のみ、受け付けられます。

環境

`PGCLIENTENCODING`

クライアントで使用される文字符号化方式のデフォルトです。(クライアントは個別に変更することができます。) また、この値は設定ファイルでも設定可能です。

`PGDATA`

デフォルトのデータディレクトリの場所です。

`PGDATESTYLE`

実行時パラメータDateStyleのデフォルト値です。(この環境変数の使用は廃止予定です。)

`PGPORT`

デフォルトのポート番号です。(設定ファイル内で設定するほうが好まれています。)

診断

semgetやshmgetについて言及するエラーメッセージは、おそらく、十分な共有メモリやセマフォを提供できるようにカーネルを再構築する必要があることを示します。詳細は18.4を参照してください。ただし、shared_buffersの値を小さくしてPostgreSQLの共有メモリ消費量を低減する、max_connectionsの値を小さくしてセマフォ消費量を低減するといった対策を行うことで、カーネルの再構築を先延ばしすることができるかもしれません。

他のサーバが既に実行中であることを示すエラーメッセージに対しては、例えば以下のコマンドを使用して、注意深く検査しなければなりません。

```
$ ps ax | grep postgres
```

もしくは

```
$ ps -ef | grep postgres
```

どちらを使用するかは、システムによって異なります。競合するサーバが稼働していないことが確実であれば、メッセージ内で示されたロックファイルを削除して、再度試してください。

ポートをバインドできなかったことを示すエラーメッセージの場合は、PostgreSQL以外のプロセスが既にそのポートを使用している可能性が考えられます。また、postgresを停止して、すぐに同じポートを使う設定で

再起動した場合、このエラーが出ることがあります。この場合、オペレーティングシステムがポートを閉鎖するまで数秒待ってから再挑戦してください。最後に、オペレーティングシステムが予約しているポート番号を指定した場合も、このエラーが発生することがあります。例えば、Unixの多くのバージョンでは1024より小さいポート番号は「信頼できる(trusted)」とみなされており、Unixのスーパーユーザ以外アクセスできません。

注釈

ユーティリティコマンドpg_ctlを使って、postgresサーバを安全かつ簡便に起動とシャットダウンすることができます。

postgresを終了させるためにSIGKILLを使用するのは可能な限り避けてください。SIGKILLで終了させると、postgresが終了前に保持しているシステムリソース(共有メモリやセマフォ)を解放しなくなることがあります。システムリソースが解放されていないと、新しくpostgresを起動する時に問題が起こる可能性があります。

postgresサーバを正常に終了させるために、SIGTERM、SIGINT、SIGQUITを使用することができます。SIGTERMは全てのクライアントが終了するのを待ってから終了させます。SIGINTは強制的に全てのクライアントとの接続を切断します。SIGQUITは適切なシャットダウンを行わないで、即時に終了させ、次回の起動時に復旧処理が行われます。

SIGHUPシグナルによりサーバの設定ファイルを再読み込みします。また、SIGHUPを個々のサーバプロセスに送信することも可能ですが、通常は意味がありません。

実行中の問い合わせを中止するには、そのコマンドを実行しているプロセスにSIGINTシグナルを送信してください。きれいにバックエンドプロセスを終了させるためには、対象プロセスにSIGTERMを送信してください。また、これらの2つと同じ操作を行うSQLから呼び出すことができるものについては9.27.2内のpg_cancel_backendおよびpg_terminate_backendを参照してください。

postgresサーバでは、通常のクリーンアップを行わずに下位のサーバプロセスを終了させるためにはSIGQUITを使用します。これはユーザが使用すべきではありません。また、サーバにSIGKILLシグナルを送信するのは好ましくありません。メインのpostgresはこれをクラッシュとして解釈するので、クラッシュからの標準的な復旧手続きの一環として、強制的に共通の親プロセスを持つpostgresプロセスを全て終了させます。

不具合

--オプションはFreeBSDやOpenBSDではうまく動きません。代わりに-cを使用してください。これは対象のオペレーティングシステムの不具合です。もし修正されなければ、将来のPostgreSQLリリースで回避策を提供する予定です。

シングルユーザモード

シングルユーザモードのサーバを起動するには、以下のようなコマンドを使用してください。

```
postgres --single -D /usr/local/pgsql/data other-options my_database
```

データベースディレクトリの正確なパスを-Dで指定します。パスを指定しない時は、必ず環境変数PGDATAを設定しておいてください。また、作業対象とするデータベースの名前も指定してください。

通常、シングルユーザモードのサーバでは、改行をコマンド入力の終わりとみなします。セミコロンについて、psqlにおけるような高度な機能はありません。コマンドが複数行にわたる場合は、改行を入力する前にバックスラッシュを入力しなければなりません（最終行を除く）。バックスラッシュとそれに続く改行は、いずれも入力コマンドから削除されます。これは文字列リテラルやコメントの中においても当てはまることに注意して下さい。

ただし、コマンドラインスイッチ-jを使用した場合は、1つの改行ではコマンド入力の終わりとはみなされず、セミコロン、改行、改行というシーケンスがコマンド入力の終わりとなります。つまり、セミコロンを入力し、そのすぐ後に続けて完全な空行を入力して下さい。このモードでは、バックスラッシュと改行の組み合わせは特別扱いされません。この場合も、文字列リテラルやコメント内に現れるこのようなシーケンスに対する高度な扱いはありません。

どちらの入力モードでも、コマンド入力終端の直前や一部でないセミコロンを入力した場合、それはコマンドを分割するものとしてみなされます。コマンド入力終端を入力すると、それまでに入力した複数の文が、1つのトランザクションとして実行されます。

セッションを終了するには、EOF（通常**Control**+**D**）を入力します。最後にコマンド入力終端を入力した後、何らかのテキストを入力している場合、EOFはコマンド入力の終端として扱われ、終了するにはもう一度EOFを入力する必要があります。

シングルユーザモードのサーバには、高度な行編集機能が用意されていないことに注意してください（例えばコマンド履歴機能はありません）。またシングルユーザモードは、自動チェックポイントやレプリケーションなどのバックグラウンド処理をまったく行いません。

例

デフォルト値を使用してpostgresをバックグラウンドで起動するには、以下を入力してください。

```
$ nohup postgres >logfile 2>&1 </dev/null &
```

例えば1234というポートを指定してpostgresを起動するには、以下を入力してください。

```
$ postgres -p 1234
```

psqlを使用して上のサーバに接続するには、以下のように-pオプションでこのポートを指定してください。

```
$ psql -p 1234
```

または環境変数PGPORTを設定してください。

```
$ export PGPORT=1234
$ psql
```

指定の実行時パラメータを以下のいずれかで設定することができます。

```
$ postgres -c work_mem=1234
$ postgres --work-mem=1234
```

どちらの形式でも、postgresql.conf内に設定されているwork_memの値を上書きします。コマンドラインでは、パラメータ内のアンダースコア(_)をアンダースコアとしてもダッシュ記号(-)としても記載できることに注意してください。おそらく短期間の実験という場合を除き、コマンドラインスイッチに依存するよりもpostgresql.conf内の設定を変更してパラメータを設定する方が実用的です。

関連項目

initdb, pg_ctl

postmaster

postmaster — PostgreSQLデータベースサーバ

概要

postmaster [*option*...]

説明

postmasterはpostgresの廃止予定の別名です。

関連項目

postgres

パート VII. 内部情報

ここでは、PostgreSQL開発者が使用できる情報を分類して説明します。

目次

第50章 PostgreSQL内部の概要

著者

本章は[sim98]として、ウィーン工科大学にてO.Univ.Prof.Dr. Georg GottlobとUniv.Ass.Mag. Katrin Seyr.の指導の下にStefan Simkovicsが書いた修士論文の一部が基になっています。

本章ではPostgreSQLのバックエンドの内部構造の概要を説明します。次からの節を読んだ後には、問い合わせがどのように処理されるかの概念がつかめているはずです。ここではPostgreSQLの内部操作の詳細な説明は行いません。記述するとしたら文章が膨大になってしまうからです。どちらかと言えば、バックエンドが問い合わせを受け取った時点からクライアントに結果を返す時点の間に引き起こる操作の一般的な流れを理解してもらうのが目的です。

50.1. 問い合わせの経路

ここでは、問い合わせが結果を得るためにたどる過程を簡単に説明します。

1. アプリケーションプログラムからPostgreSQLサーバに接続が確立されなくてはなりません。アプリケーションプログラムはサーバに問い合わせを送り、そしてサーバから送り返される結果を待ちます。

2. 構文解析過程で、アプリケーションプログラムから送られた問い合わせの構文が正しいかをチェックし、問い合わせツリーを作成します。

3. 書き換えシステムは構文解析過程で作られた問い合わせツリーを受け取り、問い合わせツリーに適用するための（システムカタログに格納されている）ルールを探します。そしてルール本体で与えられた変換を実行します。

   書き換えシステムの適用例の1つとしてビューの具現化が挙げられます。ビュー（すなわち仮想テーブル）に対して問い合わせがあると、書き換えシステムが代わってユーザの問い合わせを、ビュー定義で与えられた実テーブルにアクセスする問い合わせに書き換えます。

4. プランナ/オプティマイザは、（書き換えられた）問い合わせツリーを見て、エクゼキュータに渡すための問い合わせ計画を作ります。

   そのためにまず同じ結果をもたらす全ての可能な限りの経路を作ります。例えば、スキャンの対象となるリレーション上にインデックスがあるとすると2つの経路があります。1つは単純なシーケンシャルスキャンで、もう1つはインデックスを使ったスキャンです。次にそれぞれの経路を実行するためのコストが見積もられ、一番コストの小さい経路が選ばれます。一番コストの小さな経路は、エクゼキュータが実行できるように完全な計画に拡張されます。

5. エクゼキュータは再帰的に計画ツリー上を進み、計画で示されている方法で行を抽出します。エクゼキュータはリレーションをスキャンする間保存システムを利用してソートと結合を実行し、検索条件の評価を行い、最後に得られた行を返します。

これからの節では、PostgreSQLの内部制御とデータ構造をより良く理解するために、上に記載した事柄をさらに詳しく説明します。

50.2. 接続の確立

PostgreSQLは単純な「1プロセスに1ユーザ」のクライアント/サーバモデルによって実装されています。このモデルでは1つのサーバプロセスに対し厳密に1つのクライアントプロセスしか存在しません。いくつの接続が行われるか事前にわからないので、接続要求の度に新しいプロセスを作るマスタープロセスを使わなければなりません。このマスタープロセスはpostgresと呼ばれ、指定されたTCP/IPポートで入ってくる接続要求を監視します。接続要求を検出すると、postgresプロセスは新しいサーバプロセスを生み出します。このサーバのタスクはセマフォと共有メモリを活用してお互いに連絡を取り合い、同時にデータにアクセスしても整合性が保たれるようにします。

クライアントプロセスは第52章に記載されたPostgreSQLプロトコルを理解できるどんなプログラムでも構いません。多くのクライアントはlibpq C言語ライブラリに基づいていますが、Java JDBCドライバのようにいくつかの独立したプロトコル実装も存在します。

いったん接続が確立されると、クライアントプロセスはバックエンド（サーバ）に問い合わせを送ることができます。問い合わせは平文で送信されます。つまり、フロントエンド（クライアント）は構文解析を行いません。サーバは問い合わせの構文解析を行い、実行計画を作り、そして計画を実行し、抽出した行を確立された接続を通じてクライアントに返します。

50.3. 構文解析過程

構文解析過程は2つの部分から構成されています。

- gram.yとscan.lで定義されているパーサは、Unixのツールbisonとflexを使って構築されます。
- 変換プロセスは、パーサから返されたデータ構造の変更や追加を行います。

50.3.1. パーサ

パーサは、（平文のテキストとして渡される）問い合わせ文字列が正しい構文になっているかチェックしなければいけません。もし構文が正しい場合は構文解析ツリーが作られて返されます。正しくない場合はエラーが返されます。パーサと字句解析はUnixでよく知られたツールのbisonとflexを使用して実装されています。

字句解析はファイルscan.lで定義され、識別子やSQLキーワードなどの確認を担当します。検出された全てのキーワードや識別子に対しトークンが生成されパーサに渡されます。

パーサはファイルgram.yの中で定義され、文法ルールとルールが実行された時に実行されるアクションの組から構成されています。アクションのコード（実際はC言語コードです）は構文解析ツリーを作るのに使われます。

ファイルscan.lはプログラムflexを使ってCのソースファイルscan.cに変換されます。そしてgram.yはbisonを使ってgram.cに書き換えられます。これらの書き換えが終わると、パーサを作るために通常のCコンパイラが使えるようになります。生成されたCのファイルには絶対に変更を加えないでください。と言うのは次にflexもしくはbison が呼ばれた時に上書きされるからです。

> **注記**
>
> ここで言及した書き換えやコンパイルは通常PostgreSQLのソースと一緒に配布される*makefile*を使って自動的に行われます。

bisonまたはgram.yで定義される文法ルールの詳細は本稿では説明しきれません。flexやbisonについては本や資料がたくさん出ています。gram.yの文法の勉強を始める前にbisonの知識が必須となります。その知識なしではそこで何が起こっているのかを理解することは難しいでしょう。

50.3.2. 書き換えプロセス

構文解析過程ではSQLの構文構造に関する固定ルールのみを使って構文解析ツリーを作成します。システムカタログの参照を行わないので、要求されている操作の詳細な語義は理解しません。構文解析が終わった後に入力としてパーサから戻されたツリーを書き換えプロセスが引き受け、どのテーブル、関数、そして演算子が問い合わせによって参照されているのかの判断に必要な語義翻訳を行います。この情報を表すために作成されるデータ構造を問い合わせツリーと呼びます。

語義解釈と入力の構文解釈を切り分ける理由は、システムカタログの参照はトランザクション内でのみ行うことができますが、問い合わせ文字列を受け取ってすぐにトランザクションを開始することは好ましくないと考えられるからです。入力に対する構文解析過程ではトランザクション管理コマンド（BEGIN、ROLLBACKなど）を特定するだけで十分であるとともに、それ以上の分析を行わなくても正しい処理が実行されます。実際の問い合わせ（例えばSELECTもしくはUPDATE）に関わっていると言うことがわかっていて既にあるトランザクション内にいなければ新規トランザクションを開始することは問題ありません。これ以降に限り書き換えプロセスを起動することができます。

書き換えプロセスで作成された問い合わせツリーはほとんどの箇所で加工されていない構文解析ツリーに構造的には似ていますが、細部では数多くの相違が存在します。例えば、構文解析ツリーのFuncCallノードは構文的には関数呼び出しのように見える何かを表わしています。これは参照された名前が通常の関数になるか集約関数となるかによってFuncExprもしくはAggrefに書き換えられることがあります。さらに、列の実際のデータ型と式の結果についての情報が問い合わせツリーに書き加えられます。

50.4. PostgreSQLルールシステム

PostgreSQLには、ビューと理解の仕方でどうとも取れるビューの更新の仕様に対応する強力なルールシステムがあります。元々PostgreSQLのルールシステムは2つの実装で構成されていました。

- 初めの1つは行レベルの処理を使って動き、エクゼキュータの内部に深く実装されていました。個別の行がアクセスされる度にルールシステムが呼ばれていました。この実装は1995年、最後のBerkeley PostgreSQLプロジェクトの公式リリースがPostgres95へ移行する時に削除されました。
- ルールシステムの2つ目の実装は問い合わせ書き換えと呼ばれる手法です。書き換えシステムは構文解析過程とプランナ/オプティマイザの間にあるモジュールです。この手法は現在でも実装されています。

問い合わせの書き換えについては第40章にて詳しく論議されますので、ここでは取り扱いません。書き換えの入出力はともに問い合わせツリーである、つまり、ツリー内の表現の仕方や語義をどの程度詳しく判断するかには変更はない、ということを指摘するのに留めます。書き換えはマクロの拡張と捉えることもできます。

50.5. プランナ/オプティマイザ

プランナ/オプティマイザの役割は最適な実行計画を作ることです。ある与えられたSQL問い合わせは（それがある問い合わせツリーになるのですが）、同じ結果をもたらす、多くの異なった方法で実際には実行できます。もしもコンピュータの演算として可能であれば、問い合わせオプティマイザは可能な実行計画をすべて検証し、実行するとした場合に一番早く結果をもたらすと想定される実行計画を選択します。

> **注記**
>
> 場合によっては、問い合わせがどう実行されるか、可能性のある全ての手段を検証するため、膨大な時間とメモリ空間を消費する可能性があります。特に数多くの結合操作に問い合わせが関わった時です。相応な（必ずしも最適ではありませんが）問い合わせ計画を、相応な時間内で決定するためPostgreSQLは結合の数が閾値を越えた場合、遺伝的問い合わせオプティマイザ（第59章参照）を使用します（geqo_thresholdを参照ください）。

このプランナの検索手順は、実際には経路という名前のデータ構造を使用します。経路とは、プランナが決定を行うために必要な情報のみに切り詰めた単なる計画の表現です。最も安価である経路が決定された後、全てが揃った計画ツリーが作成されてエクゼキュータに渡されます。これはつまり、要求されている実行計画はエクゼキュータがそれを実行するために十分な詳しい内容を所有していることを表しています。本節の残りでは、経路と計画の違いについて無視します。

50.5.1. 実行可能な計画の生成

プランナ/オプティマイザは、問い合わせの中で使用される個々のリレーション（テーブル）をスキャンするための計画を生成することから始めます。各リレーション上で利用できるインデックスにより実行可能な計画が決まります。リレーションをシーケンシャルスキャンする可能性は常にありますので、シーケンシャルスキャンを使用する計画は常に作成されます。リレーション上にインデックス（例えばB-treeインデックス）が定義され、問い合わせには`relation.attribute OPR constant`という条件があるとしましょう。もし`relation.attribute`がB-treeインデックスのキーと一致し、OPRがインデックスの演算子クラスに列挙されている演算子の1つであれば、リレーションをスキャンするためにB-treeインデックスを使用する別の計画が作られます。さらに他のインデックスが存在し、問い合わせの中で条件がインデックスのキーに一致した場合、なおその上に計画が検討されます。インデックススキャン計画は、問い合わせの（もし存在すれば）`ORDER BY`句に一致するソート順、もしくはマージ結合に便利なソート順を所有するインデックスに対して生成されます（以下を参照してください）。

問い合わせが2つ以上のリレーションの結合を必要とすると、リレーションを結合する計画は、単一のリレーションをスキャンするために全ての実行可能な計画が探し出された後に検討されます。3つの実行可能な結合戦略を示します。

- ネステッドループ結合：左側のリレーションの中で見つけられた行ごとに右側のリレーションが1回スキャンされます。この戦略は実装が簡単ですが、時間がかかる場合があります（とは言っても右側のリレーションがインデックススキャンによってスキャン可能であればよい戦略になります。右側のインデックススキャンのキーとして左側のリレーションの現在の行の値を使用することができます。）

- マージ結合：結合を開始する前に、それぞれのリレーションを結合属性でソートします。そして、2つのリレーションを並行してスキャンし、一致する行を結合行の形にまとめます。それぞれのリレーションがたった

1回しかスキャンされなくて済むのでこの結合は魅力的です。要求されるソートは、明示的なソート段階、または、結合キー上のインデックスを使用して適切な順序でリレーションをスキャンすることにより行われます。

- ハッシュ結合: 右側のリレーションがハッシュキーとして結合属性を用いて初めにスキャンされ、ハッシュテーブルに読み込まれます。次に左側のリレーションがスキャンされ、見つかったそれぞれの行に相応しい値が、右側のリレーションの行を探し出すためのハッシュキーとして使われます。

問い合わせが3つ以上のリレーションを含む場合、それぞれ2つの入力を持つ結合段階のツリーによって最終結果を構築しなければなりません。プランナは最も低コストな計画を見つけ出すために、あり得る異なった結合順序を検証します。

問い合わせがgeqo_thresholdより少ないリレーションを使用する場合、最適な結合シーケンスを見つけ出すため、完璧に近い検索が行われます。プランナはWHERE条件での対応する結合句が存在する(すなわち、 `where rel1.attr1=rel2.attr2`のような制約に対して)、あらゆる2つのリレーション間の結合を優先的に考慮します。結合句のない結合ペアは他に選択のない場合に考慮されます。つまり、ある特定のリレーションが他のどんなリレーションに対しても有効な結合句を持たない場合です。すべての有効な計画はプランナが考慮したすべての結合ペアに対し生成され、最も安価な(と評価された)ものが選択されます。

geqo_thresholdを上回ると、考慮された結合シーケンスは第59章に記載されているように経験則で決定されます。そうでない時、処理は変わりません。

最終的な計画ツリーは基になっているリレーションのシーケンシャルもしくはインデックススキャン、そして必要に応じてネステッドループ、マージ、またはハッシュ結合のノード、さらにはソートまたは集約関数計算ノードのような必要とされる補助の手順から構成されます。これらほとんどの計画ノード型は*選択*(特定の論理演算条件に合致しない行を破棄すること)および*射影*(与えられた列の値に基づき派生した列の集合を計算すること、つまり必要なところでスカラ式の評価をすること)を行う追加的能力を持っています。プランナの1つの責任は、WHERE句から選択条件を付加して計画ツリーの最も適切なノードに対し必要とされる出力式を計算することです。

50.6. エクゼキュータ

*エクゼキュータ*は、プランナ/オプティマイザで作成された計画を受け取り、必要な行の集合を抽出するために再帰的に処理します。これは本質的に要求引き寄せ型(demand-pull)パイプライン機能です。計画ノードが呼ばれる度にもう1つの行を引き渡すか、行を引き渡したことの報告を行わなければなりません。

具体的な例を提供する目的で頂点のノードがMergeJoinノードである場合を想定しましょう。いかなるマージも実行される前に(それぞれの副計画から1つずつ)2つの行を取ってこなくてはいけません。ですからエクゼキュータは副計画を処理するために自分自身を再帰的に呼び出します(lefttreeに付随する副計画から開始します)。新しい頂点のノード(左の副計画の頂点のノード)はSortノードであるとしましょう。ここでもノード自体が処理される前に入力行を取ってこなくてはいけません。Sortの子ノードは実際のテーブルの読み取りを表現しているSeqScanノードのこともあり得ます。このノードの処理はエクゼキュータにテーブルから行を抽出させ、呼び出しているノードに渡し戻させます。Sortノードはソート対象の全てのノードを取得するために子ノードを繰り返し呼び出します。入力がなくなった時(子ノードが行ではなくNULLを返してきた時)Sortコードがソートを実行して最終的に最初の出力行を返すことができるようになります。つまりソート順における最初の結果です。後での要求に答えるためソート順に引き渡すことができるように残っている行は保存されます。

MergeJoinノードは同じようにしてその右副計画から最初の行を要求します。そこで2つの行が結合できるかどうか比較されます。もし結合できる場合には呼び出し側に結合された行が返されます。次の呼び出しの時に、もしくは入力された現在の組み合わせが結合できない場合はすぐに、あるテーブルあるいはそれ以外のテーブル（比較の結果に依存して）の次の行に進んで、さらに一致があるかどうか検証されます。最終的にはある副計画もしくは他の計画が使いきられ、MergeJoinノードがこれ以上の結合行を生成できないという意味のNULLを返すことになります。

複雑な問い合わせは多くの階層となった計画ノードに関わるかもしれませんが、概略的な取り扱い方は同じです。それぞれのノードは呼び出される度に次の出力行を計算して返します。それぞれのノードは同時にプランナによって割り当てられたいかなる選択式や射影式でも適用する責任があります。

エクゼキュータ機構は全ての4つの基本的なSQL型を検証するために用いられます。4つのSQL型とはSELECT、INSERT、UPDATE、そしてDELETEです。SELECTでは、最上位階層のエクゼキュータコードは問い合わせ計画によって返されるそれぞれの行をクライアントへ送り返すだけでよいことになっています。INSERTでは、INSERTで指定された対象となるテーブルに返された行が挿入されます。これはModifyTableと呼ばれる特別な最上位階層の計画ノードの中で行われます。（ある単純なINSERT ... VALUESコマンドは1つのResultノード（これは、結果としての行を1つだけ計算するに留まります）とその上の挿入を実行するためのModifyTableからなる単純な計画ツリーを生成します。しかし、INSERT ... SELECTはエクゼキュータ機構ができ得る限りの能力を発揮することを要求する場合があります）。UPDATEではプランナはすべての更新された列の値を含んだ行の演算結果と*TID*（タプルID、または行ID）を準備します。このデータはModifyTableノードに入力され、ノードでは新しく更新された行の作成と削除された古い行に印を付けるためこの情報を利用します。DELETEでは計画から実際に返されるただ1つの列はTIDで、ModifyTableノードは単に各対象行を尋ね当てて削除の印を付けるためにこのTIDを使用します。

第51章 システムカタログ

システムカタログとは、リレーショナルデータベース管理システムがテーブルや列の情報などのスキーマメタデータと内部的な情報を格納する場所です。PostgreSQLのシステムカタログは通常のテーブルです。テーブルを削除したり再作成したり、列の追加および値の挿入や更新をすることは可能ですが、これらの操作でデータベースシステムを台なしにしてしまう可能性もあります。通常手作業でシステムカタログを変更してはいけません。その代わりとしてSQLコマンドを使用します（例えばCREATE DATABASEによりpg_databaseカタログに1行挿入し、ディスク上にデータベースを実際に作成します）。しかし特に難易度の高い操作の時などの例外がありますが、それらの多くは時間と共にSQLコマンドとして利用可能となっており、それゆえシステムカタログを直接操作する必要は無くなってきています。

51.1. 概要

表 51.1にシステムカタログを列挙します。以降システムカタログについてより詳細を説明します。

ほとんどのシステムカタログはデータベースを作成する時にテンプレートデータベースからコピーされ、以降はデータベースに固有のものになります。ごく一部のカタログがデータベースクラスタ内の全てのデータベースにわたって物理的に共有されます。これらについては、それぞれのカタログで説明します。

表51.1 システムカタログ

カタログ名	用途
pg_aggregate	集約関数
pg_am	リレーションアクセスメソッド
pg_amop	アクセスメソッド演算子
pg_amproc	アクセスメソッドサポート関数
pg_attrdef	列デフォルト値
pg_attribute	テーブル列（「属性」）
pg_authid	認証識別子（ロール）
pg_auth_members	認証識別子メンバーシップ関係
pg_cast	キャスト（データ型変換）
pg_class	テーブル、インデックス、シーケンス、ビュー（「リレーション」）
pg_collation	照合順序（ロケール情報）
pg_constraint	検査制約、一意性制約、主キー制約、外部キー制約
pg_conversion	エンコード方式変換情報
pg_database	このデータベースクラスタにあるデータベース
pg_db_role_setting	ロール毎およびデータベース毎の設定
pg_default_acl	オブジェクト種類のデフォルト権限
pg_depend	データベースオブジェクト間の依存関係
pg_description	データベースオブジェクトの説明やコメント

カタログ名	用途
pg_enum	列挙型のラベルや値の定義
pg_event_trigger	イベントトリガ
pg_extension	インストールされた拡張
pg_foreign_data_wrapper	外部データラッパの定義
pg_foreign_server	外部サーバの定義
pg_foreign_table	追加の外部テーブル情報
pg_index	追加インデックス情報
pg_inherits	テーブル継承階層
pg_init_privs	オブジェクトの初期権限
pg_language	関数記述言語
pg_largeobject	ラージオブジェクト用のデータページ
pg_largeobject_metadata	ラージオブジェクトのメタデータ
pg_namespace	スキーマ
pg_opclass	アクセスメソッド演算子クラス
pg_operator	演算子
pg_opfamily	アクセスメソッド演算子族
pg_partitioned_table	テーブルのパーティションキーについての情報
pg_policy	行単位セキュリティポリシー
pg_proc	関数とプロシージャ
pg_publication	論理レプリケーションのパブリケーション
pg_publication_rel	リレーションとパブリケーションの対応
pg_range	範囲型の情報
pg_replication_origin	登録されたレプリケーション起点
pg_rewrite	問い合わせ書き換えルール
pg_seclabel	データベースオブジェクト上のセキュリティラベル
pg_sequence	シーケンスについての情報
pg_shdepend	共有オブジェクトの依存関係
pg_shdescription	共有オブジェクトに対するコメント
pg_shseclabel	共有データベースオブジェクトのセキュリティラベル
pg_statistic	プランナ統計情報
pg_statistic_ext	プランナ拡張統計情報(定義)
pg_statistic_ext_data	プランナ拡張統計情報(構築統計情報)
pg_subscription	論理レプリケーションのサブスクリプション
pg_subscription_rel	サブスクリプションについてのリレーションの状態

カタログ名	用途
pg_tablespace	データベースクラスタ内のテーブル空間
pg_transform	変換（データ型を手続き言語に変換）
pg_trigger	トリガ
pg_ts_config	全文検索設定
pg_ts_config_map	全文検索設定のトークン写像
pg_ts_dict	全文検索辞書
pg_ts_parser	全文検索パーサ
pg_ts_template	全文検索テンプレート
pg_type	データ型
pg_user_mapping	外部サーバへのユーザのマッピング

51.2. pg_aggregate

pg_aggregateカタログには集約関数の情報が格納されています。集約関数とは、値の集合（多くの場合は問い合わせ条件に該当する各行の1つの列）にある操作を行い、それらすべての値の演算の結果得られる単一の値を返します。集約関数の代表的なものはsum、countそしてmaxです。pg_aggregate内の各項目は、pg_proc内の項目の拡張です。pg_procの項目には、集約の名前、入出力データ型および通常の関数と類似したその他の情報が含まれます。

表51.2 pg_aggregateの列

列 型 説明
aggfnoid regproc（参照先 pg_proc.oid） 集約関数のpg_proc OID
aggkind char 集約関数の種類: n「通常の」集約関数、o「順序集合の」集約関数、h「仮想集合の」集約関数
aggnumdirectargs int2 順序集合や仮想集合の集約関数では、（集約されていない）複数の引数は、可変長配列として1個の引数と見なします。引数が数がpronargsと同じ場合、最終的な直接引数同様、集約された引数として、集約関数の引数は、可変または可変長配列で記述しなければなりません。通常の集約関数は引数を取りません。
aggtransfn regproc（参照先 pg_proc.oid） 遷移関数
aggfinalfn regproc（参照先 pg_proc.oid） 最終関数（ない時はゼロ）
aggcombinefn regproc（参照先 pg_proc.oid） 結合関数（ない時はゼロ）
aggserialfn regproc（参照先 pg_proc.oid） 直列化関数（ない時はゼロ）

列 型 説明
aggdeserialfn regproc (参照先 pg_proc.oid) 逆直列化関数(ない時はゼロ)
aggmtransfn regproc (参照先 pg_proc.oid) 移動集約モードの順方向遷移関数(ない時はゼロ)
aggminvtransfn regproc (参照先 pg_proc.oid) 移動集約モードの逆遷移関数(ない時はゼロ)
aggmfinalfn regproc (参照先 pg_proc.oid) 移動集約モードの最終関数(ない時はゼロ)
aggfinalextra bool aggfinalfnに追加の仮引数を渡す場合はTrue
aggmfinalextra bool aggmfinalfnに追加の仮引数を渡す場合はTrue
aggfinalmodify char aggfinalfnが遷移状態値を変更するかどうか。読み出し専用ならr、aggfinalfnの後でaggtransfnを適用できなければs、その値に書き込むならw。
aggmfinalmodify char aggmfinalfn用であることを除き、aggfinalmodifyと同様。
aggsortop oid (参照先 pg_operator.oid) 関連するソート演算子(ない時はゼロ)
aggtranstype oid (参照先 pg_type.oid) 集約関数の内部遷移(状態)データのデータ型
aggtransspace int4 遷移状態データの推定平均サイズ(バイト)、またはデフォルトの推定値であるゼロ
aggmtranstype oid (参照先 pg_type.oid) 移動集約モードの、集約関数の内部遷移(状態)データのデータ型(ない時はゼロ)
aggmtransspace int4 移動集約モードの、遷移状態データの推定平均サイズ(バイト)、またはデフォルトの推定値であるゼロ
agginitval text 遷移状態の初期値。外部文字列表現での初期値を含んだテキストフィールド。フィールドがNULL値の場合、推移状態はNULL値で始まります。
aggminitval text 移動集約モードの、遷移状態の初期値。外部に文字列表記された初期値を含むテキストフィールド。このフィールドがnullの場合、遷移状態の値はnullから始まります。

新しい集約関数はCREATE AGGREGATEコマンドで登録されます。集約関数の書き方や遷移関数の説明などの詳細は37.12を参照してください。

51.3. pg_am

pg_amカタログにはリレーションアクセスメソッドの情報が格納されます。システムがサポートするアクセスメソッド毎に1つの行が存在します。今の所、テーブルとインデックスのみがアクセスメソッドを持ちます。テーブルとインデックスアクセスメソッドの要件は第60章と第61章で詳しく説明されています。

表51.3 pg_amの列

列 型 説明
oid oid 行識別子
amname name アクセスメソッド名
amhandler regproc (参照先 pg_proc.oid) アクセスメソッドについての情報の提供に責任を持つハンドラ関数のOID
amtype char t = テーブル(マテリアライズドビューを含む)、i = インデックス。

> **注記**
>
> PostgreSQL9.6以前では、pg_amにはインデックスアクセスメソッドのプロパティを表す多くの追加列が含まれていました。そのデータは今ではCコードレベルで直接参照できるのみです。しかし、pg_index_column_has_property()関数と関連する関数群が、インデックスアクセスメソッドのプロパティを検査するためのSQLクエリを許容するために追加されました。表 9.68を参照してください。

51.4. pg_amop

pg_amopカタログにはアクセスメソッド演算子の集合に関連付けられた演算子の情報が格納されています。演算子族のメンバである演算子毎に1つの行が存在します。演算子族のメンバは検索演算子または順序付け演算子のいずれかになることができます。演算子は1つ以上の演算子族に現れますが、演算子族の中では検索でも順序付けでも複数現れることはありません。(ほとんどありませんが、ある演算子が検索目的と順序付け目的の両方で使用されることが許されます。)

表51.4 pg_amopの列

列 型 説明
oid oid 行識別子
amopfamily oid (参照先 pg_opfamily.oid) この項目用の演算子族
amoplefttype oid (参照先 pg_type.oid) 演算子の左辺側のデータ型
amoprighttype oid (参照先 pg_type.oid) 演算子の右辺側のデータ型

列 型 説明
amopstrategy int2 演算子の戦略番号
amoppurpose char 演算子の目的。検索用ならばs、順序付け用であればo
amopopr oid (参照先 pg_operator.oid) 演算子のOID
amopmethod oid (参照先 pg_am.oid) 演算子族用のインデックスアクセスメソッド
amopsortfamily oid (参照先 pg_opfamily.oid) 順序付け用の演算子の場合、この項目のソートが従うB-tree演算子族。検索用演算子であればゼロ

「検索」用演算子の項目は、この演算子族のインデックスをWHERE indexed_column operator constantを満たすすべての行を見つけるための検索に使用できることを示します。いうまでもありませんが、こうした演算子は論理型を返さなければならず、また左辺の入力型はインデックス列のデータ型に一致しなければなりません。

「順序付け」用演算子の項目は、この演算子族のインデックスをORDER BY indexed_column operator constantで表される順序で行を返すためのスキャンに使用できることを示します。こうした演算子の左辺の入力型はインデックス列のデータ型に一致しなければならないことは同じですが、任意のソート可能なデータ型を返すことができます。ORDER BYの正確な意味は、この演算子の結果型用のB-tree演算子族を参照する必要があるamopsortfamily列により指定されます。

> **注記**
>
> 現在、順序付け演算子のソート順は参照される演算子族のデフォルト、つまりASC NULLS LASTであると仮定されています。これは将来、ソートオプションを明示的に指定する追加列を加えることで緩和されるかもしれません。

項目のamopmethodは、項目を含む演算子族のopfmethodに一致しなくてはいけません。(ここでのamopmethodは、性能上の理由からカタログ構造を意図的に非正規化したものも含みます。) また、amoplefttypeとamoprighttypeは、参照されているpg_operator項目のoprleftとoprrightに一致しなくてはいけません。

51.5. pg_amproc

pg_amprocカタログには、アクセスメソッド演算子族に関連付けられたサポート関数の情報が格納されます。演算子族に含まれるサポート関数毎に1つの行が存在します。

表51.5 pg_amprocの列

列 型 説明
oid oid

列 型 説明
行識別子
amprocfamily oid（参照先 pg_opfamily.oid） この項目用の演算子族
amproclefttype oid（参照先 pg_type.oid） 関連付けられた演算子の左辺のデータ型
amprocrighttype oid（参照先 pg_type.oid） 関連付けられた演算子の右辺データ型
amprocnum int2 サポート関数番号
amproc regproc（参照先 pg_proc.oid） 関数のOID

amproclefttypeとamprocrighttype属性の通常の解釈は、それらが、特定のサポート関数がサポートする演算子の左と右の入力型を識別することです。いくつかのアクセスメソッドに対して、これらはサポート関数自身の入力データ型に一致することもあります。また、そうでないものもあります。インデックスに対して「デフォルト」サポート関数の概念があります。これはamproclefttypeとamprocrighttypeの両方が、インデックス演算クラスのopcintypeに等しい、という概念です。

51.6. pg_attrdef

pg_attrdefシステムカタログは列のデフォルト値を格納します。列の主要な情報はpg_attributeに格納されています。デフォルト値が明示的に設定された列のみここに項目を持ちます。

表51.6 pg_attrdefの列

列 型 説明
oid oid 行識別子
adrelid oid（参照先 pg_class.oid） この列が属するテーブル
adnum int2（参照先 pg_attribute.attnum） 列番号
adbin pg_node_tree nodeToString()表現の列デフォルト値。pg_get_expr(adbin, adrelid)を使ってSQL式に変換します。

51.7. pg_attribute

pg_attributeカタログにはテーブルの列情報が格納されます。データベース内のすべてのテーブルの各列に対し必ず1つのpg_attribute行があります。(また、インデックスとpg_classに項目を持つすべてのオブジェクトに対しての属性記述があります。)

属性という表現は列と同等の意味で、歴史的背景からそのように呼ばれています。

表51.7 pg_attributeの列

列 型 説明
attrelid oid (参照先 pg_class.oid) この列が属するテーブル
attname name 列名
atttypid oid (参照先 pg_type.oid) この列のデータ型
attstattarget int4 attstattargetはANALYZEによるこの列に対する蓄積された統計情報をどの程度詳しく管理するかを規定します。値がゼロの場合は統計情報を収集しません。負の値の場合は、システムのデフォルトの統計目標を使用すべきであるということです。正の値が厳密に意味するところはデータ型に依存します。スカラデータ型に対してattstattargetは収集する「最も一般的な値」の目標となる数であり、また作成する度数分布ビンの目標数でもあります。
attlen int2 この列の型のpg_type.typlenのコピー
attnum int2 列番号。通常の列には1から始まる番号付けがなされます。ctidのようなシステムによる列には(任意の)負の番号が付きます。
attndims int4 列が配列型の場合は次元数を表現し、そうでない時はゼロです。(現在配列の次元数は強制されていませんのでゼロ以外のどのような値であっても「これは配列である」ということを意味します。)
attcacheoff int4 格納時は必ず-1ですが、メモリ内の行記述子に読み込まれた場合は、行内での属性オフセットをキャッシュするために更新される可能性があります。
atttypmod int4 atttypmodは、テーブル作成時に与えられた型固有のデータ(たとえばvarchar列の最大長)を記録します。これは型固有の入力関数や長さ強制関数に渡されます。atttypmodを必要としない型では、通常、この値は-1です。
attbyval bool この列の型のpg_type.typbyvalのコピー
attstorage char 通常、この列の型のpg_type.typstorageのコピー。TOAST可能なデータ型では、格納ポリシーを制御するために列の作成後に変更することができます。
attalign char この列の型のpg_type.typalignのコピー
attnotnull bool

列 型 説明
非NULL制約を表します。
atthasdef bool この列にはデフォルト値あるいは生成式があります。その場合、実際に値を定義するpg_attrdefカタログ内に対応する項目があります。(これがデフォルトなのか生成式なのかはattgeneratedをチェックします。)
atthasmissing bool この列は、行から列の値が完全に失われている場合に使われる値を持ちます。これは、行が作られた後で非不安定(non-volatile)なDEFAULT値を持つ列が追加される際に起こります。実際に使われる値はattmissingval列に格納されています。
attidentity char 0バイト('')ならこれはIDENTITY列ではありません。IDENTITY列では、GENERATED ALWAYSならa、GENERATED BY DEFAULTならdになります。
attgenerated char ゼロバイト('')なら、生成列ではありません。さもなければs= storedです。(将来他の値が追加されるかも知れません。)
attisdropped bool この列は既に削除されていて有効ではありません。削除された列は物理的にはまだテーブル上に存在していますが、パーサによって無視されるためSQLでアクセスすることができません。
attislocal bool この列はリレーション内でローカルに定義されます。列がローカルに定義されると同時に継承される場合もあることに注意してください。
attinhcount int4 この列が持つ直接の祖先の数です。祖先を持っている列の削除や名前は変更はできません。
attcollation oid (参照先 pg_collation.oid) 列で定義された照合順序。列が照合順序の設定ができないデータ型の場合はゼロ
attacl aclitem[] この列に特定して付与された場合における、列レベルのアクセス権限
attoptions text[] 「keyword=value」文字列のような、属性レベルのオプション
attfdwoptions text[] 「keyword=value」文字列のような、外部データラッパオプションの属性レベル
attmissingval anyarray この列は、行から列の値が完全に失われている場合に使われる値を持つ一要素配列を持ちます。これは、行が作られた後で非不安定(non-volatile)なDEFAULT値を持つ列が追加される際に起こります。この値はatthasmissingが真のときだけ使用されます。値がなければその列はNULLになります。

削除された列のpg_attribute項目では、atttypidはゼロにリセットされます。しかしattlenとpg_typeからコピーされた他のフィールドは、有効なままです。この動作は、削除された列のデータ型が後になって削除されて、pg_type行が存在しないような状況の場合に必要です。attlenと他のフィールドは、テーブル内の行の内容を解釈するために使用されます。

51.8. pg_authid

pg_authidカタログはデータベース認証識別子(ロール)の情報を保持します。ロールは「ユーザ」と「グループ」の概念を包括しています。ユーザは本質的にrolcanloginフラグセットを持ったロールです。どのようなロール(rolcanloginを持っている、持っていないに関わらず)も他のロールをメンバとして持っていても構いません。pg_auth_membersを参照してください。

このカタログはパスワードを含んでいるため、第三者が内容を読むことができないようにしなければいけません。pg_rolesは、pg_authidのビューで、パスワードのフィールドは空白となっていますので内容を読み取ることができます。

第21章でユーザと権限管理に関するより詳細について説明します。

ユーザの本人確認はクラスタ全体にわたる情報ですので、pg_authidはクラスタのすべてのデータベースで共有されます。データベース毎ではなく、クラスタ毎にたった1つだけpg_authidが存在します。

表51.8 pg_authidの列

列 型 説明
oid oid 行識別子
rolname name ロール名
rolsuper bool ロールはスーパーユーザの権限を持っています
rolinherit bool ロールは自動的にメンバとして属するロールの権限を継承します
rolcreaterole bool ロールはロールを作成することができます
rolcreatedb bool ロールはデータベースを作成することができます
rolcanlogin bool ロールはログインすることができます。つまりロールはセッションを始める認証の識別子となることができます。
rolreplication bool ロールはレプリケーション用のロールです。レプリケーションロールは、レプリケーション接続を開始すること、およびレプリケーションスロットを作成および削除することができます。
rolbypassrls bool すべての行単位セキュリティポリシーを無視するロール。詳しくは5.8を参照してください。
rolconnlimit int4 ログイン可能なロールでは、これはロールが確立できる同時実行接続数を設定します。-1は制限無しを意味します。
rolpassword text (おそらく暗号化された)パスワード。無い場合はNULLです。書式は使用される暗号化の形式に依存します。
rolvaliduntil timestamptz パスワード有効期限(パスワード認証でのみ使用)。NULLの場合には満了時間はありません。

MD5で暗号化されたパスワードでは、rolpassword列は文字列md5で始まり、それに32文字の16進MD5ハッシュ値が続きます。MD5ハッシュは、ユーザのパスワードとユーザ名を繋げたものに対して生成されます。例えばjoeのパスワードがxyzzyなら、PostgreSQLはxyzzyjoeのMD5ハッシュを格納します。

パスワードがSCRAM-SHA-256で暗号化される場合、次の書式になります。

```
SCRAM-SHA-256$<iteration count>:<salt>$<StoredKey>:<ServerKey>
```

ここで、salt、StoredKey、ServerKeyはBase64の符号化書式に従います。この書式はRFC 5803で指定されているものと同じです。

これらのいずれの書式にも従っていないパスワードは、暗号化されていないものとみなされます。

51.9. pg_auth_members

pg_auth_membersカタログはロール間のメンバシップ関係を示しています。循環していなければ、どのような関係でも許可されています。

ユーザの同一性はクラスタ間で保たれる必要があるため、pg_auth_membersはクラスタ間のすべてのデータベースで共有されています。pg_auth_membersのコピーはデータベースごとではなく、各クラスタにひとつだけ持っています。

表51.9 pg_auth_membersの列

列 型 説明
roleid oid（参照先 pg_authid.oid） メンバを持っているロールのID
member oid（参照先 pg_authid.oid） roleidのメンバであるロールのID
grantor oid（参照先 pg_authid.oid） このメンバシップを与えたロールのID
admin_option bool もしmemberがroleidのメンバシップを他に与えることができる場合は真

51.10. pg_cast

pg_castカタログにはデータ型変換パスが格納されます。ここには、組み込みのパスとユーザ定義のパスが存在します。

pg_castは、システムがどのように動作するかわかっているような、あらゆる型変換を表しているわけではないということに注意してください。いくつかの一般的な規則から推測できないような型変換についてのみ表しています。例えば、ドメインとその基本の型は明示的にpg_cast内で表されていません。他の重要な例外は「自動I/O変換キャスト」です。これらのキャストは、text型やほかの文字列型から変換したりされたりする

のにデータ型自身のI/O関数を用いていますが、これらのキャストは明示的にpg_cast内において表されていません。

表51.10 pg_castの列

列 型 説明
oid oid 行識別子
castsource oid（参照先 pg_type.oid） 変換元データ型のOID
casttarget oid（参照先 pg_type.oid） 対象データ型のOID
castfunc oid（参照先 pg_proc.oid） このキャストを実行するために使用する関数のOID。キャストメソッドが関数を必要としない場合はゼロが格納されます。
castcontext char キャストがどの文脈で呼び出し可能かを示します。eは明示のキャストとしてのみ起動されることを意味します（CAST、::構文を使用します）。aは、対象となる列を明示的に特定するだけでなく暗黙的にも特定することを意味します。iは他の場合と同様に演算式内で暗黙的であることを意味します。
castmethod char どのようにキャストが実行されるかを示します。fはcastfuncフィールド内で示される関数が使用されていることを意味します。iは入出力関数が使用されていることを示します。bは型がバイナリを強制しているため、変換が必要ないことを意味します。

pg_cast内に挙げられているキャスト関数は、第1番目の引数の型として、キャスト元の型をいつも取らなければいけません。また、キャスト関数は、結果の型としてキャスト先の型を返します。キャスト関数は3つまで引数を持つことができます。もし存在するなら、2番目の引数はinteger型でなくてはなりません。この引数はキャスト先の型に関連付けられた型修飾子を受け取ります。2番目の引数がない場合は、-1です。3番目の引数は、もし存在する場合は、boolean型でなくてはなりません。この引数は、もしキャストが明示的なキャストであればtrueを受け取り、そうでない場合はfalseを受け取ります。

もし関連のある関数が複数の引数を持つ場合、キャストの元と先で型が同じであるpg_cast項目を作成することが妥当です。このような項目は、「length coercion functions」を表現します。「length coercion functions」は型の値を特定の型の修飾子の値に適するように修正します。

pg_cast項目が異なるキャスト元とキャスト先の型を持っていて、かつ関数が複数の引数を持つ時は、1つの型から別の型への変換し、かつ、1つの手順で長さの修正を適用することを意味します。このような項目が利用できない時は、型修飾子を使用した型の修正は2つの手順が必要です。1つはデータ型の間での変換で、2つ目は修飾子を適用することです。

51.11. pg_class

pg_classカタログはテーブルと、その他に列を持つもの、あるいはテーブルに似た全てのものを目録にしています。その中にはインデックス（pg_indexも参照）、シーケンス（pg_sequenceも参照）、ビュー、マテリアライズドビュー、複合型およびTOASTテーブルが含まれます。relkindを参照してください。これより以降、「リ

レーション」と記されている場合はこれらすべてのオブジェクトを意味しています。すべての列が全てのリレーションの型に該当するとは限りません。

表51.11 pg_classの列

列 型 説明
oid oid 行識別子
relname name テーブル、インデックス、ビューなどの名前
relnamespace oid (参照先 pg_namespace.oid) このリレーションを持つ名前空間のOID
reltype oid (参照先 pg_type.oid) もし何らかの(pg_type項目を持たないインデックスではゼロ)が存在した場合このテーブルの行の型に対応するデータ型のOID
reloftype oid (参照先 pg_type.oid) 型付けされたテーブルでは背後にある複合型のOID。その他のリレーションではゼロ。
relowner oid (参照先 pg_authid.oid) リレーションの所有者
relam oid (参照先 pg_am.oid) これがテーブルあるいはインデックスの場合は、それに使われている(ヒープ、B-tree、ハッシュなどの)アクセスメソッド
relfilenode oid このリレーションのディスク上のファイルの名前です。ゼロはディスク上のファイル名が低レベルな状態で決定される「マップ付けされた」リレーションであることを意味します。
reltablespace oid (参照先 pg_tablespace.oid) このリレーションが保存されているテーブル空間。もしゼロならば、このデータベースのデフォルトテーブル空間を意味します。(リレーションがディスク上のファイルになくても、問題はありません。)
relpages int4 このテーブルのディスク上におけるページ単位(BLCKSZ)の表現サイズ。これはプランナで使用される単なる推測値です。VACUUM、ANALYZEおよびCREATE INDEXコマンドなどの一部のDDLコマンドで更新されます。
reltuples float4 テーブル内の生きている行数。これはプランナで使用される単なる推測値です。VACUUM、ANALYZE、CREATE INDEXなどの一部のDDLコマンドで更新されます。
relallvisible int4 テーブル内の可視マップ内で全て可視とマークされているページ数。これはプランナで使用される単なる見積です。これはVACUUM、ANALYZEさらにCREATE INDEXといったいくつかのDDLコマンドで更新されます。
reltoastrelid oid (参照先 pg_class.oid) このテーブルに関連しているTOASTテーブルのOID。何もない場合はゼロです。TOASTテーブルは「行に収まらない」大きい属性を副テーブルに格納します。
relhasindex bool テーブルであり、かつ、インデックスを持つ(あるいはつい最近まで持っていた)時は真。

列 型 説明
relisshared bool クラスタ内の全てのデータベースにわたってこのテーブルが共有されている場合は真。(pg_databaseのような)ある特定のシステムカタログのみ共有されます。
relpersistence char pは永続テーブル、uはログを取らないテーブル、tは一時テーブルを表します。
relkind char rは通常のテーブル、iはインデックス、Sはシーケンス、tはTOASTテーブル、vはビュー、mはマテリアライズドビュー、cは複合型、fは外部テーブル、pはパーティションテーブル、Iはpartitioned indexを表します。
relnatts int2 リレーションにあるユーザ列数(システム列は含みません)。pg_attributeにこれに対応する数多くの項目があるはずです。pg_attribute.attnumも参照してください。
relchecks int2 テーブル上のCHECK制約の数。pg_constraintカタログを参照してください。
relhasrules bool もしテーブルにルールがある(あるいは以前あった)場合に真。pg_rewriteカタログを参照してください。
relhastriggers bool もしテーブルにトリガがある(あるいは以前あった)場合に真。pg_triggerカタログを参照してください。
relhassubclass bool もしテーブルあるいはインデックスが子テーブルに継承されている(または以前に継承されていた)場合は真。
relrowsecurity bool 行単位セキュリティが有効なテーブルでは真。pg_policyカタログを参照。
relforcerowsecurity bool 行単位セキュリティが(有効にされているとして)テーブルの所有者にも適用されるなら真。pg_policyカタログを参照。
relispopulated bool リレーションにデータが投入されている場合に真(マテリアライズドビュー以外のすべてのリレーションでは真です。)
relreplident char 行に「replica identity」フォームを使った列。d デフォルト(もしあれば主キー)、n 無し、f 全ての列、i インデックスと indisreplidentのセット(使用されていたインデックスが削除されていた場合は、無し、と同様)
relispartition bool テーブルあるいはインデックスがパーティションなら真
relrewrite oid (参照先 pg_class.oid) テーブルの書き換えが必要なDDL操作中に書き込みが行われる新しいリレーションでは、これは元のリレーションのOIDを持ちます。この状態は内部的にのみ可視です。このフィールドはユーザから見えるリレーションでは0以外を持つべきではありません。
relfrozenxid xid この値より以前のトランザクションIDはすべて、このテーブルで永続的な(「凍結された」)トランザクションIDに置き換えられています。これは、このテーブルに対して、トランザクションID周回を防ぎ、かつ、pg_xactを縮小させることを目的としたバキュームを行うかどうかを追跡するために使用されます。リレーションがテーブルではない場合は0(InvalidTransactionId)です。

列 型 説明
relminmxid xid このテーブル内のトランザクションIDによって置換される前のすべてのマルチトランザクションID。これは、マルチトランザクションIDのID周回を防ぐ、またはpg_multixactを縮小させるために、テーブルをバキュームする必要があるかどうかを追跡するために使用されます。リレーションがテーブルではない場合はゼロ(InvalidMultiXactId)です。
relacl aclitem[] アクセス権限。詳細は5.7を参照してください。
reloptions text[] 「keyword=value」文字列のような、アクセスメソッド特有のオプション。
relpartbound pg_node_tree テーブルがパーティションの場合(relispartition参照)のパーティション境界の内部表現。

pg_class内の複数の論理型フラグは、ゆっくりと保守されます。正しい状態にあるときに真であることが保証されていますが、その条件が真でなくなった時即座に偽に再設定されないかもしれません。例えばrelhasindexはCREATE INDEXで設定されますが、DROP INDEXでは決して初期化されません。代わりにVACUUMがそのテーブルにインデックスがないことを判定した場合にrelhasindexを初期化します。この調整により競合状態を防止し、同時実行性が向上します。

51.12. pg_collation

pg_collationカタログは利用可能、SQL名とオペレーティングシステムのロケールカテゴリとの基本的な対応付けを行う照合順序を記述します。詳細は 23.2を参照してください。

表51.12 pg_collationの列

列 型 説明
oid oid 行識別子
collname name 照合順序の名前(名前空間およびエンコード方式で一意)
collnamespace oid (参照先 pg_namespace.oid) この照合順序を含む名前空間のOID
collowner oid (参照先 pg_authid.oid) 照合順序の所有者
collprovider char 照合順序の提供者。dはデータベースのデフォルト、cはlibc、iはICU。
collisdeterministic bool 照合順序は決定論的か?
collencoding int4 この照合順序を適用することができるエンコード方式。任意のエンコード方式で動作する場合は-1
collcollate name

列 型 説明
この照合順序オブジェクト用のLC_COLLATE
collctype name この照合順序オブジェクト用のLC_CTYPE
collversion text この照合順序に対する提供者固有のバージョンです。これは照合順序が作成された時に記録され、データの破壊につながりかねない照合順序定義の変更を検知するために使用時に検査されます。

このカタログの一意キーは(collname, collnamespace)だけではなく(collname,collencoding, collnamespace)です。PostgreSQLは通常、collencodingが現在のデータベースのエンコード方式または-1と一致しない照合順序をすべて無視します。また、collencoding = -1を持つ項目と名前が一致する新しい項目の作成は許されません。したがって照合順序を識別するためには、カタログの定義に従った一意ではない場合であっても、限定されたSQL名称(schema.name)を使用することで十分です。このようにカタログを定義した理由は、クラスタの初期化時にinitdb がシステムで利用可能なすべてのロケール用の項目でこのカタログにデータを投入するためです。その為、今後そのクラスタで使用される可能性があるすべてのエンコード方式のエントリーを保持できるようにしなければなりません。

後でtemplate0から複製されるデータベースのエンコード方式と一致するかもしれないので、template0データベースのデータベースのエンコード方式と一致しないものの照合順を作成することが有用になるかもしれません。現在これは手作業で行う必要があります。

51.13. pg_constraint

pg_constraintカタログはテーブル上の検査制約、主キー制約、一意性制約、外部キー制約、排他制約を格納します（列制約は特別扱いされていません。全ての列制約は何らかのテーブル制約と同等です。）非NULL制約はここではなく、pg_attributeカタログで示されます。

(CREATE CONSTRAINT TRIGGERで作成される)ユーザ定義の制約トリガもこのテーブルの項目の元になります。

ドメイン上の検査制約もここに格納されます。

表51.13 pg_constraintの列

列 型 説明
oid oid 行識別子
conname name 制約名（一意である必要はありません）。
connamespace oid（参照先 pg_namespace.oid） この制約を含む名前空間のOID。
contype char c = 検査制約、f = 外部キー制約、p = 主キー制約、u = 一意性制約、t = 制約トリガ, x = 排他制約
condeferrable bool

列 型 説明
制約は遅延可能かどうか?
condeferred bool 制約はデフォルトで遅延可能かどうか?
convalidated bool 制約が検証されているか?現時点では外部キーとチェック制約の場合のみ偽になる可能性があります。
conrelid oid (参照先 pg_class.oid) この制約が存在しているテーブルです。テーブル制約でなければ0です。
contypid oid (参照先 pg_type.oid) この制約が存在しているドメインです。ドメイン制約でなければ0です。
conindid oid (参照先 pg_class.oid) 一意性制約、主キー制約、外部キー制約、排他制約の場合、この制約をサポートするインデックス。さもなくばゼロ。
conparentid oid (参照先 pg_constraint.oid) パーティション内の制約なら、親パーティションテーブルの該当制約、そうでなければ0。
confrelid oid (参照先 pg_class.oid) 外部キーであれば、参照されるテーブルです。そうでなければ0です。
confupdtype char 外部キー更新アクションコード: a = no action, r = restrict, c = cascade, n = set null, d = set default
confdeltype char 外部キー削除アクションコード: a = no action, r = restrict, c = cascade, n = set null, d = set default
confmatchtype char 外部キーの一致型: f = full, p = partial, s = simple
conislocal bool この制約はリレーションでローカルに定義されています。制約はローカルに定義されていて同時に継承されます。
coninhcount int4 この制約がもつ直系の先祖の数。先祖の数がゼロではない制約は削除や改名はできません。
connoinherit bool この制約はリレーションのためにローカルで定義されます。これは非継承制約です。
conkey int2[] (参照先 pg_attribute.attnum) テーブル制約(外部キーを含みますが制約トリガは含みません)であれば、その制約によって制約される列のリスト
confkey int2[] (参照先 pg_attribute.attnum) 外部キーであれば、参照される列のリスト
conpfeqop oid[] (参照先 pg_operator.oid) 外部キーであれば、PK = FKの比較のための同値演算子のリスト
conppeqop oid[] (参照先 pg_operator.oid) 外部キーであれば、PK = PKの比較のための同値演算子のリスト
conffeqop oid[] (参照先 pg_operator.oid) 外部キーであれば、FK = FKの比較のための同値演算子のリスト
conexclop oid[] (参照先 pg_operator.oid)

列 型 説明
排他制約の場合、列単位の排他演算子のリスト。
conbin pg_node_tree チェック制約なら式の内部表現。(pg_get_constraintdef()を使ってチェック制約の定義を取り出すことをお勧めします。)

排他制約の場合、単純な列参照である制約要素でのみconkeyが有用です。その他の場合、conkeyはゼロであり、関連するインデックスは制約される式を調査して見つけなければなりません。(したがってインデックスではconkeyはpg_index.indkeyの内容と同じものを持ちます。)

> **注記**
>
> pg_class.relchecksはそれぞれのリレーションに対してこのテーブルで検出された検査制約の項目数と一致しなければなりません。

51.14. pg_conversion

pg_conversionカタログはエンコード方式変換関数を記述します。詳細はCREATE CONVERSIONを参照してください。

表51.14 pg_conversionの列

列 型 説明
oid oid 行識別子
conname name 変換名(名前空間内で一意)
connamespace oid (参照先 pg_namespace.oid) この変換を含む名前空間のOID。
conowner oid (参照先 pg_authid.oid) 変換の所有者
conforencoding int4 エンコード元のID
contoencoding int4 エンコード先のID
conproc regproc (参照先 pg_proc.oid) 変換プロシージャ
condefault bool これがデフォルト変換である場合は真

51.15. pg_database

pg_databaseカタログには使用可能なデータベースの情報が格納されます。データベースはCREATE DATABASEコマンドで作成されます。いくつかのパラメータの詳細については第22章を参照してください。

ほとんどのシステムカタログとは異なり、pg_databaseはクラスタにおける全てのデータベースにわたって共有されます。データベース毎に1つではなく、クラスタ毎にたった1つだけpg_databaseのコピーが存在します。

表51.15 pg_databaseの列

列 型 説明
oid oid 行識別子
datname name データベース名
datdba oid（参照先 pg_authid.oid） データベースの所有者。通常はそのデータベースの作成者
encoding int4 このデータベースの文字エンコード方式。（pg_encoding_to_char()で、この番号からエンコード方式名称に変換できます。）
datcollate name このデータベースのLC_COLLATE
datctype name このデータベースのLC_CTYPE
datistemplate bool trueの場合、このデータベースはどのユーザでもCREATEDBを使って複製することができます。falseの場合、スーパーユーザまたはデータベースの所有者だけが複製することができます。
datallowconn bool 偽の時はこのデータベースには誰も接続できません。これはtemplate0データベースが変更されることを防ぐために使用されます。
datconnlimit int4 このデータベースに対する同時のコネクションの最大数を設定します。-1は無制限を意味します。
datlastsysoid oid データベース最終のシステムOID。pg_dumpでは特に有用です。
datfrozenxid xid このデータベースの中で、この値よりも前のトランザクションIDは、永続的な（「凍結された」）トランザクションIDを持つように変更されています。これは、このデータベースに対して、トランザクションID周回を防ぎ、かつ、pg_xactを縮小させることを目的としたバキュームを行うかどうかを追跡するために使用されます。これはテーブル毎のpg_class.relfrozenxid値の最小値になります。
datminmxid xid

列 型 説明
このデータベース内のトランザクションIDで置換される前のすべてのマルチトランザクションID。これは、トランザクションIDの周回問題を防ぐ、またはpg_multixactを縮小させるためにデータベースをバキュームする必要があるかどうかを追跡するために使用されます。これはテーブル毎のpg_class.relminmxidの最小値です。
dattablespace oid（参照先 pg_tablespace.oid） データベース用のデフォルトテーブル空間。このデータベース内でpg_class.reltablespaceがゼロであるすべてのテーブルは、このテーブル空間に格納されます。特に、共有されていないすべてのシステムカタログはこのテーブル空間にあります。
datacl aclitem[] アクセス権限。詳細は5.7を参照してください。

51.16. pg_db_role_setting

pg_db_role_settingカタログはロールとデータベースの組み合わせ毎に、実行時設定変数に設定されるデフォルト値を記録します。

ほとんどのカタログとは異なり、pg_db_role_settingはクラスタのすべてのデータベースにまたがって共有されます。つまりクラスタにはpg_db_role_settingのコピーは1つしかありません。データベース毎に1つではありません。

表51.16 pg_db_role_settingの列

列 型 説明
setdatabase oid（参照先 pg_database.oid） この設定が適用されるデータベースのOID。データベース固有でなければゼロです。
setrole oid（参照先 pg_authid.oid） この設定が適用されるロールのOID。ロール固有でなければゼロです。
setconfig text[] 実行時設定パラメータのデフォルト。

51.17. pg_default_acl

pg_default_aclカタログには、新規に作成されたオブジェクトに割り当てられた初期権限が格納されます。

表51.17 pg_default_aclの列

列 型 説明
oid oid 行識別子
defaclrole oid（参照先 pg_authid.oid）

列 型 説明
この項目に関連するロールのOID。
defaclnamespace oid (参照先 pg_namespace.oid) この項目に関連する名前空間のOID。何もない場合はゼロです。
defaclobjtype char この項目のオブジェクト種類。r = リレーション(テーブル、ビュー)、S = シーケンス、f = 関数、T = 型、n = スキーマ
defaclacl aclitem[] この種類のオブジェクトが作成時に保持しなければならないアクセス権限。

pg_default_aclの項目は、指示されたユーザに属するオブジェクトに割り当てられる初期権限を示します。現在2種類の項目があります。defaclnamespace = 0を持つ「大域的」な項目と特定のスキーマを参照する「スキーマ単位」の項目です。大域的な項目が存在する場合、その種類のオブジェクトの通常の組み込まれたデフォルト権限を上書きします。もしスキーマ単位の項目があれば、それは大域的な権限または組み込まれたデフォルト権限に追加される権限を表します。

他のカタログ内のACL項目がNULLの場合、その時のpg_default_acl内のものではではなくそのオブジェクトの組み込まれたデフォルト権限を表すものが取られます。pg_default_aclはオブジェクトの生成時のみに考慮されます。

51.18. pg_depend

pg_dependカタログは、データベースオブジェクト間の依存関係を記録します。この情報によってDROPコマンドが、他のどのオブジェクトをDROP CASCADEで削除する必要があるか、また、DROP RESTRICTで削除を防止するかの場合を判断します。

pg_shdependも参照してください。これはデータベースクラスタ間で共有されるオブジェクトの依存関係に対する似たような機能を持っています。

表51.18 pg_dependの列

列 型 説明
classid oid (参照先 pg_class.oid) 依存するオブジェクトを含んだシステムカタログのOID
objid oid (いずれかのOID列) 特定の依存するオブジェクトのOID
objsubid int4 テーブル列の場合、これは列番号です(objidとclassidはテーブル自身を参照します)。他のすべての種類のオブジェクトでは、このフィールドはゼロです。
refclassid oid (参照先 pg_class.oid) 参照されるオブジェクトが存在するシステムカタログのOID
refobjid oid (いずれかのOID列)

列 型 説明
特定の参照されるオブジェクトのOID
refobjsubid int4 テーブル列の場合、これは列番号です（refobjidとrefclassidはテーブル自身を参照します）。他のすべての種類のオブジェクトでは、このフィールドはゼロです。
deptype char この依存関係の特定のセマンティクスを定義するコード（後述）。

すべての場合において、pg_dependエントリは依存するオブジェクトも削除しない限り、参照されるオブジェクトを削除できないことを示します。もっとも、deptypeによって指定される以下のようないくつかのオプションもあります。

DEPENDENCY_NORMAL (n)

個別に作成されたオブジェクト間の通常の関係です。依存するオブジェクトは参照されるオブジェクトに影響を与えずに削除できます。参照されるオブジェクトはCASCADEを指定することによってのみ削除することができます。この場合は依存するオブジェクトも削除されます。例：テーブルの列はそのデータ型に対して通常の依存関係を持ちます。

DEPENDENCY_AUTO (a)

依存するオブジェクトは参照されるオブジェクトから独立して削除することができます。そして、参照されるオブジェクトが削除される時は（RESTRICTもしくはCASCADEモードに関わりなく）依存するオブジェクトも自動的に削除されなければなりません。例：テーブル上の名前付き制約はテーブル上に自動設定されているため、テーブルが削除されるとなくなります。

DEPENDENCY_INTERNAL (i)

依存するオブジェクトは参照されるオブジェクトの作成時に作成されたもので、実際には内部実装の一部に過ぎません。依存するオブジェクトに対してDROPコマンドを直接的に実行できません（その代わりに、参照されるオブジェクトに対してDROPを実行するように指示されます）。参照されるオブジェクトにDROPを実行すると、CASCADEが指定されているかどうかに関わらず、依存するオブジェクトも削除されます。削除されるオブジェクトへの依存関係で依存しているオブジェクトを削除しなければならない場合、その削除は参照されるオブジェクトの削除に変換されます。ですから依存しているオブジェクトのNORMALとAUTO依存関係は、参照されるオブジェクトの依存関係に非常に似通った振る舞いをします。例：ビューのON SELECTルールがビューに依存して内部的に作られ、ビューが存在する限り削除されることを防ぎます。ルールの依存関係（たとえばそれが参照するテーブル）はビューの依存関係であるかのように振る舞います。

DEPENDENCY_PARTITION_PRI (P)
DEPENDENCY_PARTITION_SEC (S)

依存するオブジェクトは参照されるオブジェクトの生成の一環で作成され、実際にはこれは内部的な実装の一部に過ぎません。しかし、INTERNALとは違って複数の参照されるオブジェクトが存在します。参照されているオブジェクトの少なくとも1つが削除されない限り、依存するオブジェクトは削除されてはいけません。もし参照されているオブジェクトの一つが削除されたら、CASCADEが指定されているかどうかに関わらず、依存しているオブジェクトは削除されるべきです。また、INTERNALとは違って、依存オブ

ジェクトが依存しているオブジェクトを削除してもパーティション参照オブジェクトを自動的に削除することにはなりません。ですからその削除処理によって他の経路でこれらのオブジェクトの少なくとも1つに連鎖波及しない限り、削除は拒否されます。(たいていの場合、依存するオブジェクトはすべての非パーティション依存関係を、少なくとも1つのパーティション参照オブジェクトと共有するので、この制限によって連鎖削除をブロックすることにはなりません。) エラーメッセージで優先的に主パーティションが使われることを除くと、主および二次パーティション依存関係は同じように振る舞います。よって、パーティション依存オブジェクトは一つの主パーティション依存関係と1つ以上の二次パーティション依存関係を持つはずです。パーティション依存関係は、オブジェクトが通常持っている依存関係に加えて作成されるのであり、それを置き換えるものではないことに注意してください。これによってATTACH/DETACH PARTITION操作が簡単になります。パーティション依存関係は追加されるか削除されるかのどちらかになります。例:子パーティション化インデックスは、それが作成されているパーティションテーブルと親パーティション化インデックスの両方にパーティション依存します。ですから、このどちらかが削除されると削除されますが、それ以外の場合には削除されません。親インデックスへの依存関係は主なので、 ユーザが子パーティション化インデックスを削除しようとすると、エラーメッセージは(テーブルではなく)親インデックスを削除するように示唆します。

DEPENDENCY_EXTENSION (e)

依存するオブジェクトは参照されるオブジェクトの拡張のメンバです(pg_extension参照)。依存するオブジェクトは参照されるオブジェクトに対するDROP EXTENSION経由でのみ削除することができます。機能的にはこの種類の依存関係はINTERNAL依存関係と同様に動作しますが、明確さとpg_dumpを単純化するために別々に保持されます。

DEPENDENCY_AUTO_EXTENSION (x)

依存するオブジェクトは参照されるオブジェクトの拡張のメンバではありません(そしてそれゆえpg_dumpによって無視されません)が、拡張なしに機能することが出来ず、拡張自体が削除される時に自動的に削除されるでしょう。依存するオブジェクトは、同様にそれ自身で削除されるかもしれません。機能的にはこの種類の依存関係はAUTO依存関係と同様に動作しますが、明確さとpg_dumpを単純化するために別々に保持されます。

DEPENDENCY_PIN (p)

依存するオブジェクトはありません。この種類の項目は、参照されるオブジェクトにシステム自体が依存し、したがってオブジェクトを絶対に削除してはならないことを示します。この種類の項目はinitdbの実行時にのみ作成されます。依存するオブジェクト列にはゼロが含まれます。

将来的に、他の依存関係のオプションが必要になる可能性があります。

2つのオブジェクトが複数のpg_dependエントリでリンクされていることは十分ありえます。たとえば子パーティション化インデックスは、パーティションテーブルに対してパーティション型依存関係を持ち、更にインデックスが貼ってあるテーブルの列に自動依存関係を持ちます。この種の状況は、複数の依存関係セマンティクスの和で表現されます。自動削除の条件をこの依存関係の一つが満たすならば依存するオブジェクトはCASCADEなしに削除できます。逆に、どのオブジェクトが一緒に削除されなければならないかに関するすべての依存関係の制限は満足されなければなりません。

51.19. pg_description

各データベースオブジェクトに対して付けられたオプションの補足説明（コメント）はpg_descriptionカタログに格納されます。補足説明はCOMMENTコマンドで編集でき、psqlの\dコマンドで表示できます。多くの組み込み型のシステムオブジェクトの説明は、pg_descriptionの最初の部分で提供されています。

pg_shdescriptionも参照してください。こちらは、データベースクラスタに渡って共有されるオブジェクトに関する説明について、同様の機能を行います。

表51.19 pg_descriptionの列

列 型 説明
objoid oid（いずれかのOID列） この補足説明が属するオブジェクトのOID
classoid oid（参照先 pg_class.oid） このオブジェクトが現れるシステムカタログのOID
objsubid int4 テーブル列についてのコメントの場合、これは列の（objoidもしくはclassoidはテーブル自身を参照します）列番号です。他のすべての種類のオブジェクトでは、このフィールドはゼロです。
description text このオブジェクトの説明となる任意のテキスト

51.20. pg_enum

pg_enumカタログは、各列挙型についてその値とラベルを示す項目を含みます。ある与えられた列挙値の内部表現は、実際にはpg_enum内の関連付けられた行のOIDです。

表51.20 pg_enumの列

列 型 説明
oid oid 行識別子
enumtypid oid（参照先 pg_type.oid） この列挙値を所有しているpg_type項目のOID
enumsortorder float4 列挙型におけるこの列挙値のソート位置
enumlabel name この列挙値のテキストラベル

pg_enum行のOIDは次のような特別な規則に従います。列挙型のソート順序と同じ順序で並んだ偶数のOIDが保証されています。つまり、2つの偶数のOIDが同じ列挙型に属する場合、OIDの小さい方がより小さいenumsortorder値を持たなければなりません。奇数のOID値はソート順序に関連を持ってはなりません。この規則により列挙の比較処理は多くの一般的な場合でカタログの検索を防ぐことができます。列挙型の作成および変更を行う処理は、可能であれば偶数のOIDを列挙値に割り当てようとします。

列挙型を作成する時、そのメンバには1..nのソート順位置が割り当てられます。しかし後で追加されたメンバには負もしくはenumsortorderの小数値が与えられる可能性があります。これらの値の要件は、各列挙型の中で正しく順序付けされ、かつ一意であることだけです。

51.21. pg_event_trigger

pg_event_triggerカタログはイベントトリガを格納します。詳細は第39章を参照してください。

表51.21 pg_event_triggerの列

列 型 説明
oid oid 行識別子
evtname name トリガ名（一意でなければなりません）
evtevent name このトリガが発行する対象のイベントを識別します。
evtowner oid（参照先 pg_authid.oid） イベントトリガの所有者
evtfoid oid（参照先 pg_proc.oid） 呼び出される関数
evtenabled char どのsession_replication_role モードでこのイベントトリガを発行するかを制御します。Oでは、トリガは「origin」および「local」モードで発行します。Dでは、トリガは無効です。R では、トリガは「replica」モードで発行します。Aでは、トリガは常に発行します。
evttags text[] このトリガを発行するコマンドタグです。NULLの場合、このトリガの発行はコマンドタグに基づいて制限されていません。

51.22. pg_extension

pg_extensionカタログにはインストールされた拡張に関する情報が格納されます。拡張の詳細については37.17を参照してください。

表51.22 pg_extensionの列

列 型 説明
oid oid 行識別子
extname name 拡張の名前
extowner oid（参照先 pg_authid.oid）

列 型 説明
拡張の所有者
extnamespace oid（参照先 pg_namespace.oid） 拡張が提供するオブジェクトを含むスキーマ
extrelocatable bool 拡張が他のスキーマに再配置可能である場合真
extversion text 拡張のバージョン名
extconfig oid[]（参照先 pg_class.oid） 拡張の設定テーブルのregclass OIDの配列。なければNULL
extcondition text[] 拡張の設定テーブル用のWHERE句フィルタ条件の配列。なければNULL

ほとんどの「名前空間」に関する列を持つカタログと異なり、extnamespaceは、拡張がそのスキーマに属することを意図したものではありません。拡張の名前は決してスキーマで修飾されません。extnamespaceは、拡張のオブジェクトのすべて、あるいは、ほとんどを含むスキーマを示します。extrelocatableが真の場合、このスキーマは拡張に属するすべてのスキーマ修飾可能なオブジェクトを含まなければなりません。

51.23. pg_foreign_data_wrapper

pg_foreign_data_wrapperカタログは外部データラッパの定義を保存します。外部データラッパは外部サーバにあるデータにアクセスするための機構です。

表51.23 pg_foreign_data_wrapperの列

列 型 説明
oid oid 行識別子
fdwname name 外部データラッパの名前
fdwowner oid（参照先 pg_authid.oid） 外部データラッパの所有者
fdwhandler oid（参照先 pg_proc.oid） 外部データラッパに対する実行関数の提供に責任を持つハンドラ関数への参照。ハンドラ関数がない場合はゼロになります。
fdwvalidator oid（参照先 pg_proc.oid） 外部サーバや外部データラッパを使用するユーザマップと同様に外部データラッパに対して与えられたオプションの正当性を検査する有効性検証関数。有効性検証関数がない場合はゼロになります。
fdwacl aclitem[] アクセス権限。詳細は5.7を参照してください。

列 型 説明
fdwoptions text[] 外部データラッパの「keyword=value」のような特定のオプション。

51.24. pg_foreign_server

pg_foreign_serverカタログは外部サーバの定義を保存します。外部サーバはリモートサーバなど外部データの源を記述します。外部サーバは外部データラッパを介してアクセスされます。

表51.24 pg_foreign_serverの列

列 型 説明
oid oid 行識別子
srvname name 外部サーバの名前
srvowner oid (参照先 pg_authid.oid) 外部サーバの所有者
srvfdw oid (参照先 pg_foreign_data_wrapper.oid) 外部サーバの外部データラッパのOID
srvtype text サーバの型 (オプション)
srvversion text サーバのバージョン (オプション)
srvacl aclitem[] アクセス権限。詳細は5.7を参照してください。
srvoptions text[] 外部サーバの「keyword=value」のような特定のオプション。

51.25. pg_foreign_table

pg_foreign_tableカタログには、外部テーブルに関する補助情報が含まれます。外部テーブルは主にpg_classの項目により表されます。pg_foreign_tableの項目には、外部テーブルに属する情報のみに関する情報が含まれ、他の種類のリレーションは含まれません。

表51.25 pg_foreign_tableの列

列 型 説明
ftrelid oid (参照先 pg_class.oid)

列 型 説明
この外部テーブルに対するpg_class項目のOID
ftserver oid (参照先 pg_foreign_server.oid) この外部テーブルに対する外部サーバOID
ftoptions text[] 「keyword=value」文字列のような、外部テーブルのオプション

51.26. pg_index

pg_indexカタログはインデックス情報の一部を保持します。その他のほとんどの情報はpg_classにあります。

表51.26 pg_indexの列

列 型 説明
indexrelid oid (参照先 pg_class.oid) このインデックスに対するpg_class項目のOID
indrelid oid (参照先 pg_class.oid) このインデックスが使われるテーブルに対するpg_class項目のOID
indnatts int2 インデックス内の列数(pg_class.relnattsの複製)
indnkeyatts int2 格納されているだけで、インデックスのセマンティクスに寄与していない*included columns*を含まないインデックス内の*key columns*の数。
indisunique bool 真の場合は一意性インデックス
indisprimary bool 真の場合このインデックスはテーブルの主キーを表します (この値が真の時indisuniqueは常に真でなければなりません)。
indisexclusion bool 真の場合、このインデックスは排他制約をサポートします。
indimmediate bool 真の場合、一意性検査が挿入時即座に強制されます。(indisuniqueが真でなければ無関係です。)
indisclustered bool 真の場合、前回このインデックスを元にテーブルはクラスタ化されました。
indisvalid bool 真の場合、現在このインデックスは問い合わせに対して有効です。偽は、インデックスが不完全かもしれないことを意味します。INSERT/UPDATE操作による変更が行われているはずで、問い合わせに使用するには安全ではありません。一意性インデックスであれば、一意性も保証されません。
indcheckxmin bool

列 型 説明
真の場合、pg_index行のxminがTransactionXminイベント境界値を下回るまで、問い合わせはインデックスを使用してはいけません。なぜなら、テーブルは互換性の無い行と共に破壊されたHOTチェインを含み、それらが可視であるかもしれないからです。
indisready bool 真の場合、インデックスは挿入に対する準備ができています。偽の場合はインデックスはINSERT/UPDATE操作により無視されなければならないことを意味します。
indislive bool 偽の場合、インデックスの削除処理が進行中であり、このためすべての目的において（HOT安全性の決定を含む）無視しなければなりません。
indisreplident bool 真の場合、このインデックスはALTER TABLE ... REPLICA IDENTITY USING INDEX ...を用いて「replica identity」が選択されます。
indkey int2vector（参照先 pg_attribute.attnum） このインデックスがどのテーブル列をインデックスとしているかを示すindnatts配列の値です。例えば、1 3は1番目と3番目のテーブル列がインデックスキーとなっていることを示します。キー列は、（INCLUDE句で指定した）非キー列の前に来ます。この配列でゼロとなっているのは対応するインデックスの属性が単純な列参照ではなくテーブル列に渡った演算式であることを示します。
indcollation oidvector（参照先 pg_collation.oid） インデックスキー（indnkeyattsの値）内の各列に関してここにはインデックスで使用される照合順序のOIDが含まれます。照合できないデータ型の列ではゼロが入ります。
indclass oidvector（参照先 pg_opclass.oid） インデックスキー（indnkeyattsの値）内のそれぞれの列に対して、使用する演算子クラスのOIDを保持します。pg_opclassを参照してください。
indoption int2vector 列毎のフラグビットを格納するindnkeyatts値の配列です。ビットの意味はインデックスのアクセスメソッドによって定義されています。
indexprs pg_node_tree 単純な列参照でないインデックス属性の（nodeToString()表現による）演算式ツリー。indkeyがゼロの各エントリについて1つの要素があるリストになっています。すべてのインデックス属性が単純な参照ならNULLとなります。
indpred pg_node_tree 部分インデックス属性の（nodeToString()表現による）演算式ツリー。部分インデックスでなければNULL。

51.27. pg_inherits

pg_inheritsカタログはテーブルとインデックスの継承階層の情報を記録します。データベース内の、それぞれの直接の親子テーブルあるいはインデックス関係に対して1つの記述があります（直接ではない継承は、記述の連鎖から決定されます）。

表51.27 pg_inheritsの列

列 型 説明
inhrelid oid (参照先 pg_class.oid) 子テーブルあるいはインデックスのOID。
inhparent oid (参照先 pg_class.oid) 親テーブルあるいはインデックスのOID。
inhseqno int4 子テーブルの直接の親が複数あるとき(多重継承)、この数は継承列を整える順序を導きます。1から数えます。 インデックスは多重継承できません。宣言的パーティショニングを使用する際にしか継承できないからです。

51.28. pg_init_privs

pg_init_privsカタログは、システム内のオブジェクトの初期権限についての情報を記録します。データベース内の初期権限のセットがデフォルトでない(NULLでない)オブジェクトごとに1つの記述があります。

オブジェクトは、システムが(initdbによって)初期化された時、またはオブジェクトがCREATE EXTENSIONの実行中に作成され、拡張スクリプトがGRANTコマンドを使用して初期権限をシステムにセットする時に初期権限を持つことができます。システムは、拡張スクリプトの実行中に権限の記録を自動的に処理することや、 拡張作成者が権限を記録させるためにスクリプトの中でGRANTとREVOKEステートメントの使用のみを必要とすることに注意してください。privtype列は、初期権限がinitdbによって設定されたか、もしくはCREATE EXTENSIONコマンド実行中に設定されたかを表示します。

initdbによって設定された初期権限を持つオブジェクトは、privtypeが'i'で、CREATE EXTENSIONによって設定された初期権限を持つオブジェクトは、privtypeが'e'になります。

表51.28 pg_init_privsの列

列 型 説明
objoid oid (いずれかのOID列) 特定のオブジェクトのOID
classoid oid (参照先 pg_class.oid) オブジェクトが存在するシステムカタログのOID
objsubid int4 テーブル列においては、列番号です(objoidとclassoidはテーブル自身を参照します)。その他すべてのオブジェクト型においては、この列はゼロです。
privtype char オブジェクトの初期権限の型を設定しているコード。テキストを参照してください。
initprivs aclitem[] 初期アクセス権限。詳細は5.7を参照してください。

51.29. pg_language

pg_languageカタログはユーザ定義関数やストアドプロシージャを作成することができる言語を登録します。言語ハンドラの詳細はCREATE LANGUAGEと第41章を参照してください。

表51.29 pg_languageの列

列 型 説明
oid oid 行識別子
lanname name 言語名称
lanowner oid (参照先 pg_authid.oid) 言語の所有者
lanispl bool (SQLのような)内蔵言語ではfalseで、ユーザ定義言語ではtrueです。現在、pg_dumpではどの言語がダンプされる必要があるかを特定するためにこれを利用していますが、近い将来に異なるメカニズムによって置き換わる可能性があります。
lanpltrusted bool 信頼できる言語の場合はtrueです。信頼できる言語とは、通常のSQL実行環境の外側にある、いかなる言語へのアクセス許可も付与されていないと信用できる言語です。スーパーユーザのみが信頼されない言語で関数を作成することができます。
lanplcallfoid oid (参照先 pg_proc.oid) 非内蔵言語用の、言語ハンドラを参照します。これは、この言語で記述されたすべての関数を実行するための責任を持つ特別な関数です。
laninline oid (参照先 pg_proc.oid) これは「インライン」匿名コードブロック(DOブロック)の実行に責任を持つ関数を参照します。インラインブロックをサポートしない場合はゼロ。
lanvalidator oid (参照先 pg_proc.oid) これは、新しい関数が作成された時に構文や有効性の検査を引き受ける言語有効性検査関数を参照します。有効性検査関数がない場合はゼロになります。
lanacl aclitem[] アクセス権限。詳細は5.7を参照してください。

51.30. pg_largeobject

pg_largeobjectカタログは「ラージオブジェクト」を構築するデータを保持します。ラージオブジェクトは作成された時に割り当てられたOIDで識別されます。それぞれのラージオブジェクトはpg_largeobjectの行に都合良く格納されるのに十分に足る小さなセグメント、もしくは「ページ」に分割されます。ページごとのデータ量は(現在BLCKSZ/4あるいは典型的に2キロバイトの)LOBLKSIZEとして定義されます。

PostgreSQL 9.0より前までは、ラージオブジェクトに関連した権限構造はありませんでした。その結果pg_largeobjectは可読性が高いもので、システム内のすべてのラージオブジェクトのOIDを入手するた

めに使用することができました。これはもはや当てはまりません。ラージオブジェクトのOIDのリストを入手するためにはpg_largeobject_metadataを使用してください。

表51.30 pg_largeobjectの列

列 型 説明
loid oid（参照先 pg_largeobject_metadata.oid） このページを含んだラージオブジェクトの識別子
pageno int4 ラージオブジェクト内の（ゼロから数えた）このページのページ番号
data bytea ラージオブジェクト内に保存された実データ。LOBLKSIZEバイトを絶対上回りません。たぶんそれより小さいでしょう。

pg_largeobjectのそれぞれの行はオブジェクト内のバイトオフセット（pageno * LOBLKSIZE）から始まるラージオブジェクトの1ページ分のデータを保持します。ページが見つからなかったり、たとえオブジェクトの最後のページでない場合でもLOBLKSIZEより小さくてもよいといった、あちこちに散らばって保存されてもよいような実装になっています。ラージオブジェクトの中で見つからない部分はゼロとして読み込まれます。

51.31. pg_largeobject_metadata

pg_largeobject_metadataはラージオブジェクトに関連したメタデータを保持します。実際のラージオブジェクトデータはpg_largeobjectに格納されます。

表51.31 pg_largeobject_metadataの列

列 型 説明
oid oid 行識別子
lomowner oid（参照先 pg_authid.oid） ラージオブジェクトの所有者
lomacl aclitem[] アクセス権限。詳細は5.7を参照してください。

51.32. pg_namespace

pg_namespaceカタログは名前空間を保存します。名前空間はSQLスキーマの裏にある構造です。それぞれの名前空間は、リレーション、型などの集合を、名前が競合することなく、個別に持ちます。

表51.32 pg_namespaceの列

列 型 説明
oid oid

列 型 説明
行識別子
nspname name 名前空間の名前
nspowner oid（参照先 pg_authid.oid） 名前空間の所有者
nspacl aclitem[] アクセス権限。詳細は5.7を参照してください。

51.33. pg_opclass

pg_opclassカタログはインデックスアクセスメソッド演算子クラスを定義します。それぞれの演算子クラスは特定のデータ型のインデックス列のセマンティクスと特定のインデックスアクセスメソッドを定義します。演算子クラスは、ある特定の演算子族は特定のインデックス可能な列データの型に対して適用できる、ということを本質的に特定します。インデックス付けされた列を実際に使用可能な演算子族の演算子群は、その列のデータ型を左辺の入力として受け付けます。

演算子クラスについては37.16に詳細に説明されています。

表51.33 pg_opclassの列

列 型 説明
oid oid 行識別子
opcmethod oid（参照先 pg_am.oid） 対象のインデックスアクセスメソッド演算子クラス
opcname name この演算子クラスの名前
opcnamespace oid（参照先 pg_namespace.oid） この演算子クラスの名前空間
opcowner oid（参照先 pg_authid.oid） 演算子クラスの所有者
opcfamily oid（参照先 pg_opfamily.oid） 演算子クラスを含んでいる演算子族
opcintype oid（参照先 pg_type.oid） 演算子クラスがインデックスを作成するデータ型
opcdefault bool 演算子クラスがopcintypeのデフォルトである場合は真
opckeytype oid（参照先 pg_type.oid） インデックス内に格納されているデータ型。opcintypeと同じ場合はゼロ

演算子クラスのopcmethodは、演算子クラスが含んでいる演算子族のopfmethodに一致しなければいけません。また、任意のopcmethodとopcintypeの組み合わせに対してopcdefaultが真となるようなpg_opclass行が複数存在してはいけません。

51.34. pg_operator

pg_operatorカタログは演算子の情報を保存します。CREATE OPERATORと37.14を参照してください。

表51.34 pg_operatorの列

列 型 説明
oid oid 行識別子
oprname name 演算子名
oprnamespace oid（参照先 pg_namespace.oid） この演算子を含む名前空間のOID
oprowner oid（参照先 pg_authid.oid） 演算子の所有者
oprkind char b = 挿入辞（「両側」）、l = 接頭辞（「左側」）、r = 接尾辞（「右側」）
oprcanmerge bool この演算子はマージ結合をサポートします。
oprcanhash bool この演算子はハッシュ結合をサポートします。
oprleft oid（参照先 pg_type.oid） 左辺オペランドの型
oprright oid（参照先 pg_type.oid） 右辺オペランドの型
oprresult oid（参照先 pg_type.oid） 結果の型
oprcom oid（参照先 pg_operator.oid） もし存在すればこの演算子の交代演算子
oprnegate oid（参照先 pg_operator.oid） もし存在すればこの演算子の否定子
oprcode regproc（参照先 pg_proc.oid） この演算子を実装する関数
oprrest regproc（参照先 pg_proc.oid） この演算子の制約選択評価関数
oprjoin regproc（参照先 pg_proc.oid）

列 型 説明
この演算子の結合選択評価関数

未使用の列にはゼロが入ります。例えば、接頭辞演算子ではoprleftはゼロです。

51.35. pg_opfamily

pg_opfamilyカタログは演算子族を定義します。それぞれの演算子族は、演算子とサポートルーチン(特定のインデックスアクセスメソッドのために特化されたセマンティクスを実装するような関連付けられたもの)を集めたものです。さらに、演算子族内の演算子はすべて、アクセスメソッドにより特定される方法において「互換性」があります。演算子族の概念は、データ型を跨る演算子がインデックスで使用されることを許可し、さらにアクセスメソッドのセマンティクスの知識を使用することについて理由付けすることも許可します。

演算子族については37.16で詳しく説明します。

表51.35 pg_opfamilyの列

列 型 説明
oid oid 行識別子
opfmethod oid (参照先 pg_am.oid) 演算子族用のインデックスアクセスメソッド
opfname name 演算子族の名称
opfnamespace oid (参照先 pg_namespace.oid) 演算子族の名前空間
opfowner oid (参照先 pg_authid.oid) 演算子族の所有者

演算子族を定義している情報の大部分が、pg_opfamily行にあるわけではなく、pg_amopやpg_amprocやpg_opclass行にあります。

51.36. pg_partitioned_table

カタログpg_partitioned_tableはテーブルがどのようにパーティションに分けられているかに関する情報を格納します。

表51.36 pg_partitioned_tableの列

列 型 説明
partrelid oid (参照先 pg_class.oid) このパーティションテーブルのpg_classのエントリのOID

列 型 説明
partstrat char パーティショニング戦略。hならハッシュパーティションテーブル、lならリストパーティションテーブル、rなら範囲パーティションテーブル。
partnatts int2 パーティションキーの列の数
partdefid oid (参照先 pg_class.oid) このパーティションのデフォルトパーティションのpg_classエントリのOID。このパーティションテーブルにデフォルトパーティションがなければ0。
partattrs int2vector (参照先 pg_attribute.attnum) これはpartnatts値の配列で、どのテーブル列がパーティションキーの一部となっているかを示します。例えば、値が1 3であれば、テーブルの1番目と3番目の列がパーティションキーを構成することを意味します。この配列がゼロの場合は、対応するパーティションキー列が式であって、単なる列参照ではないことを示します。
partclass oidvector (参照先 pg_opclass.oid) これは、パーティションキーの各列について、使用する演算子クラスのOIDが入ります。詳細についてはpg_opclassを参照してください。
partcollation oidvector (参照先 pg_collation.oid) これは、パーティションキーの各列について、パーティショニングで使用する照合のOIDが入ります。列が照合できないデータ型の場合はゼロが入ります。
partexprs pg_node_tree 単純な列参照ではないパーティションキー列についての(nodeToString()形式での)式ツリーです。partattrsがゼロの各エントリについて1つの要素があるリストになっています。すべてのパーティションキー列が単純な参照ならNULLとなります。

51.37. pg_policy

カタログpg_policyはテーブルの行単位セキュリティのポリシーを格納します。ポリシーには、それが適用されるコマンドの種類(すべてのコマンドのこともあります)、それが適用されるロール、セキュリティバリアの制約として、そのテーブルを含む問い合わせに追加される式、そしてテーブルに新しいレコードを追加しようとする問い合わせのためにWITH CHECKオプションとして追加される式が含まれます。

表51.37 pg_policyの列

列 型 説明
oid oid 行識別子
polname name ポリシーの名前
polrelid oid (参照先 pg_class.oid) ポリシーが適用されるテーブル
polcmd char

列 型 説明
ポリシーが適用されるコマンドの種類：r SELECT、a INSERT、w UPDATE、d DELETE、* すべて
polpermissive bool 許容(permissive)ポリシーか、制限(restrictive)ポリシーか
polroles oid[]（参照先 pg_authid.oid） ポリシーが適用されるロール
polqual pg_node_tree テーブルを使用する問い合わせにセキュリティバリアの制約として追加される式のツリー
polwithcheck pg_node_tree テーブルに行を追加する問い合わせにWITH CHECKの制約として追加される式のツリー

注記

pg_policyに格納されるポリシーは、そのテーブルにpg_class.relrowsecurityが設定されている場合にのみ適用されます。

51.38. pg_proc

pg_procカタログは関数、プロシージャ、集約関数あるいはWINDOW関数（これらをまとめてルーチンとも言います）に関する情報を格納します。CREATE FUNCTION、CREATE PROCEDUREと37.3を参照してください。

prokindがそのエントリが集約関数であることを示しているなら、pg_aggregateに一致する行があるはずです。

表51.38 pg_procの列

列 型 説明
oid oid 行識別子
proname name 関数名
pronamespace oid（参照先 pg_namespace.oid） この関数を含む名前空間のOID
proowner oid（参照先 pg_authid.oid） 関数の所有者
prolang oid（参照先 pg_language.oid） この関数の実装言語または呼び出しインタフェース
procost float4 推定実行コスト(cpu_operator_cost単位です)。proretsetの場合は、返される行毎のコストになります。

列 型 説明
prorows float4 結果の推定行数(proretsetでなければゼロになります)
provariadic oid (参照先 pg_type.oid) 可変配列パラメータの要素のデータ型。関数が可変パラメータを持っていない場合はゼロになります。
prosupport regproc (参照先 pg_proc.oid) この関数に対する任意のプランナサポート関数 (37.11参照)
prokind char fなら通常の関数、pならプロシージャ、aなら集約関数、wならWINDOW関数
prosecdef bool セキュリティ定義の関数(すなわち「setuid」関数)
proleakproof bool この関数には副作用がありません。引数に関する情報が戻り値以外から伝わることがありません。引数の値に依存するエラーを発生する可能性がある関数はすべてリークプルーフ関数ではありません。
proisstrict bool 関数は呼び出し引数がNULLの場合にはNULLを返します。その場合、関数は実際にはまったく呼び出されません。「厳密」ではない関数はNULL値入力を取り扱えるようにしなければいけません。
proretset bool 集合(すなわち指定されたデータ型の複数の値)を返す関数
provolatile char provolatileは、関数の結果が入力引数のみで決定されるか、または外部要素に影響されるかを示します。iは「immutable(不変)」関数を表し、同じ入力に対し常に同じ結果をもたらします。sは「stable(安定)」関数を表し、(固定入力に対する)結果はスキャン内で変わりません。vは「volatile(不安定)」関数を表し、どのような場合にも結果は異なる可能性があります。(また、副作用を持つ関数にvを使用することで、その関数に対する呼び出しが最適化で消されないようにできます。)
proparallel char proparallelは関数が並列モードにて安全に実行できるかを示します。sは、制限なしに並列モードにて実行することが安全である関数を表します。rは、並列モードにて実行可能な関数を表しますが、実行は並列グループリーダに制限されます。並列ワーカプロセスはこれらの関数を呼び出すことができません。uは、並列モードにて安全ではない関数を表します。このような関数が存在すると、直列的な実行プランが強制されます。
pronargs int2 入力の引数の数
pronargdefaults int2 デフォルト値を持つ引数の数
prorettype oid (参照先 pg_type.oid) 戻り値のデータ型
proargtypes oidvector (参照先 pg_type.oid) 関数の引数のデータ型を格納した配列。これは入力引数(INOUTとVARIADICも含みます)のみを含んでいて、関数の呼び出しシグネチャを表現します。
proallargtypes oid[] (参照先 pg_type.oid)

列 型 説明
関数の引数のデータ型を格納した配列。これは(OUTとINOUT引数を含んだ)全ての引数を含みます。しかし、すべての引数がINであった場合は、この列はNULLになります。歴史的な理由からproargtypesは0から番号が振られていますが、添字は1から始まっていることに注意してください。
proargmodes char[] 関数の引数のモードを格納した配列。以下のようにエンコードされています。IN引数に対してはi、OUT引数に対してはo、INOUT引数に対してはb、VARIADIC引数に対してはv、TABLE引数に対してはt。もしすべての引数がIN引数であった場合は、この列はNULLです。添字はproargtypesではなくproallargtypesの位置に対応していることに注意してください。
proargnames text[] 関数の引数名を格納する配列。名前のない引数は、配列内では空文字列で設定されます。もしすべての引数に名前がない場合は、この列はNULLです。添字はproargtypesではなくproallargtypesの位置に対応していることに注意してください。
proargdefaults pg_node_tree デフォルト値のための(nodeToString()表現の)演算式ツリー。これはpronargdefaultsの要素のリストで、最後のN個の入力引数と対応しています(つまり最後のN proargtypesの位置ということです)。もし引数にデフォルト値がない場合は、この列はNULLになります。
protrftypes oid[] 変換が適用されるデータ型のOID
prosrc text 関数の起動方法を関数ハンドラに伝えます。実装言語や呼び出し規約に依存して、使用する言語用の関数の実際のソースコード、リンクシンボル、ファイル名などになります。
probin text 関数の起動方法についての追加情報。同じように解釈は言語に依存します。
proconfig text[] 実行時の設定変数に対する関数のローカル設定
proacl aclitem[] アクセス権限。詳細は5.7を参照してください。

コンパイル言語で作成された、組み込みおよび動的にロードされる関数では、prosrcは関数のC言語名(リンクシンボル)を持ちます。他の種類の言語はすべて、prosrcは関数のソーステキストを持ちます。probinは動的にロードされるC関数に対してその関数を保有する共有ライブラリファイルの名前を与える以外には使用されていません。

51.39. pg_publication

カタログpg_publicationには、データベース内に作成されたすべてのパブリケーションが含まれます。パブリケーションについての詳細は30.1を参照してください。

表51.39 pg_publicationの列

列 型 説明
oid oid

列 型 説明
行識別子
pubname name パブリケーションの名前
pubowner oid（参照先 pg_authid.oid） パブリケーションの所有者
puballtables bool 真の場合、このパブリケーションは、将来作成されるテーブルを含め、データベース内の全テーブルを自動的に含みます。
pubinsert bool 真の場合、パブリケーション内のテーブルに対するINSERT操作は複製されます。
pubupdate bool 真の場合、パブリケーション内のテーブルに対するUPDATE操作は複製されます。
pubdelete bool 真の場合、パブリケーション内のテーブルに対するDELETE操作は複製されます。
pubtruncate bool 真の場合、パブリケーション内のテーブルに対するTRUNCATE操作は複製されます。
pubviaroot bool 真の場合、自分自身ではなく、パブリケーションが言及しているパーティションの最上位の祖先の識別子とスキーマを使って、リーフパーティションに対する操作が複製されます。

51.40. pg_publication_rel

カタログpg_publication_relにはデータベース内のリレーションとパブリケーションのマッピングが含まれます。これは多対多のマッピングです。この情報のよりユーザフレンドリなビューについては51.78を参照してください。

表51.40 pg_publication_relの列

列 型 説明
oid oid 行識別子
prpubid oid（参照先 pg_publication.oid） パブリケーションの参照
prrelid oid（参照先 pg_class.oid） リレーションの参照

51.41. pg_range

pg_rangeカタログは、範囲型についての情報を保存します。これはpg_type内の型のエントリに追加されます。

表51.41 pg_rangeの列

列 型 説明
rngtypid oid (参照先 pg_type.oid) 範囲型のOID
rngsubtype oid (参照先 pg_type.oid) この範囲型の要素型(派生元型)のOID
rngcollation oid (参照先 pg_collation.oid) 範囲比較のために使用される照合のOID。何もない場合はゼロです。
rngsubopc oid (参照先 pg_opclass.oid) 範囲比較のために使用される派生元型の演算子クラスのOID
rngcanonical regproc (参照先 pg_proc.oid) 範囲型を標準型に変換する関数のOID。何もない場合はゼロです。
rngsubdiff regproc (参照先 pg_proc.oid) 2つの要素値の間の違いをdouble precisionとして返す関数のOID、なければゼロ

rngsubopc (および、要素型が照合可能である場合はrngcollation)は 範囲型で使用されるソートの順番を決定します。rngcanonicalは要素型が離散的である場合に使用されます。rngsubdiffは省略可能ですが、範囲型に対するGiSTインデックスの性能を向上するためには提供しなければなりません。

51.42. pg_replication_origin

pg_replication_originカタログは、作成されたすべてのレプリケーション起点を含んでいます。レプリケーション起点についての詳細は第49章を参照してください。

ほとんどのシステムカタログとは異なり、pg_replication_originはクラスタ内の全データベースで共有されます。つまりクラスタごとにpg_pg_replication_originの実体は1つだけ存在し、データベースごとに1つではありません。

表51.42 pg_replication_originの列

列 型 説明
roident oid クラスタ全体で一意なレプリケーション起点の識別子。システムから除かれてはいけません。
roname text レプリケーション起点のユーザ定義の外部名

51.43. pg_rewrite

pg_rewriteカタログはテーブルとビューに対する書き換えルールを保存します。

表51.43 pg_rewriteの列

列 型 説明
oid oid 行識別子
rulename name ルール名
ev_class oid (参照先 pg_class.oid) ルールを適用するテーブル
ev_type char ルールを適用するイベントの型: 1 = SELECT、2 = UPDATE、3 = INSERT、4 = DELETE
ev_enabled char ルールがどのsession_replication_roleモードで発行されるかを制御します。O = ルールは「origin」および「local」モードで発行。D = ルールは無効。R = ルールは「replica」モードで発行。A = ルールは常に発行。
is_instead bool ルールがINSTEADルールの場合は真
ev_qual pg_node_tree ルールの制約条件に対する(nodeToString()表現による)演算式ツリー
ev_action pg_node_tree ルールのアクションに対する(nodeToString()表現による)問い合わせツリー

注記

テーブルがこのカタログ内のルールを持つ場合、pg_class.relhasrulesは真でなければなりません。

51.44. pg_seclabel

pg_seclabelカタログにはデータベースオブジェクト上のセキュリティラベルが格納されます。セキュリティラベルはSECURITY LABELコマンドを用いて操作することができます。セキュリティラベルを閲覧するより簡単な方法については51.83を参照してください。

pg_shseclabelを参照してください。これは、データベースクラスタ間で共有されたデータベースオブジェクトにおけるセキュリティラベルのための類似した機能を提供します。

表51.44 pg_seclabelの列

列 型 説明
objoid oid (いずれかのOID列)

列 型 説明
このセキュリティラベルが関係するオブジェクトのOIDです。
classoid oid（参照先 pg_class.oid） このオブジェクトが現れるシステムカタログのOID
objsubid int4 テーブル列上のセキュリティラベルでは、これは列番号です（objoidおよびclassoidはテーブル自身を参照します）。他のすべての種類のオブジェクトでは、この列はゼロです。
provider text このラベルに関連付いたラベルプロバイダです。
label text このオブジェクトに適用されるセキュリティラベルです。

51.45. pg_sequence

カタログpg_sequenceにはシーケンスに関する情報が含まれます。名前やスキーマなどシーケンスに関する情報の一部はpg_classにあります。

表51.45 pg_sequenceの列

列 型 説明
seqrelid oid（参照先 pg_class.oid） このシーケンスのpg_classのエントリのOID
seqtypid oid（参照先 pg_type.oid） シーケンスのデータ型
seqstart int8 シーケンスの開始値
seqincrement int8 シーケンスの増分値
seqmax int8 シーケンスの最大値
seqmin int8 シーケンスの最小値
seqcache int8 シーケンスのキャッシュサイズ
seqcycle bool シーケンスが周回するかどうか

51.46. pg_shdepend

pg_shdependカタログは、データベースオブジェクトとロールのような共有オブジェクト間のリレーション依存関係を保持します。この情報はPostgreSQLが依存関係を削除しようとする前に、これらのオブジェクトを参照されないようにすることを保証することを許可します。

pg_dependも参照してください。pg_dependは単一のデータベース内のオブジェクトに関する依存関係について同じような機能を実行します。

多くのシステムカタログと異なりpg_shdependはクラスタの全てのデータベースに共有されています。データベース毎ではなく、クラスタ毎にただ1つのpg_shdependのコピーがあります。

表51.46 pg_shdependの列

列 型 説明
dbid oid (参照先 pg_database.oid) 依存するオブジェクトが格納されたデータベースのOID。共有オブジェクトではゼロ
classid oid (参照先 pg_class.oid) 依存するオブジェクトを含んだシステムカタログのOID
objid oid (いずれかのOID列) 特定の依存するオブジェクトのOID
objsubid int4 テーブル列の場合、これは列番号です(objidとclassidはテーブル自身を参照します)。他のすべての種類のオブジェクトでは、この列はゼロになります。
refclassid oid (参照先 pg_class.oid) 参照されるオブジェクトが入っているシステムカタログのOID(共有カタログである必要があります)
refobjid oid (いずれかのOID列) 特定の参照されるオブジェクトのOID
deptype char この依存関係の特定のセマンティクスを定義するコード(後述)。

すべての場合において、pg_shdepend項目は依存するオブジェクトも削除しない限り、参照されるオブジェクトを削除できないことを示します。もっとも、deptypeによって指定される以下のようないくつかのオプションもあります。

SHARED_DEPENDENCY_OWNER (o)

参照されるオブジェクト(ロールである必要があります)が依存するオブジェクトの所有者です。

SHARED_DEPENDENCY_ACL (a)

参照されたオブジェクト(ロールである必要があります)が、依存するオブジェクトのACL(アクセス制御リスト。権限リストのこと)内で述べられています。(所有者はSHARED_DEPENDENCY_OWNER項目を持つため、SHARED_DEPENDENCY_ACL項目は、オブジェクトの所有者に対して作成されません。)

SHARED_DEPENDENCY_POLICY (r)

参照されたオブジェクト(ロールである必要があります)が、依存するポリシーオブジェクトのターゲットとして述べられています。

SHARED_DEPENDENCY_PIN (p)

依存するオブジェクトはありません。この種類の項目は、システム自体が参照されるオブジェクトに依存している記号です。よってオブジェクトは削除されてはいけません。この種類の項目はinitdbによってのみ作成されます。依存するオブジェクトの列はゼロを含んでいます。

他の依存関係のオプションが将来必要になる可能性があります。現状の定義は、参照されるオブジェクトとしてロールのみをサポートしていることに特に注意してください。

51.47. pg_shdescription

pg_shdescriptionには共有データベースオブジェクトに対する補足説明(コメント)を格納します。補足説明はCOMMENTコマンドを使用して編集でき、psqlの\dコマンドを使用して閲覧することができます。

pg_descriptionも参照してください。こちらは、単一データベース内のオブジェクトに関する説明について、同様の機能を行います。

他のシステムカタログと異なり、pg_shdescriptionはクラスタ内のすべてのデータベースに渡って共有されます。データベース毎に存在するのではなく、1つのクラスタにpg_shdescriptionが1つのみ存在します。

表51.47 pg_shdescriptionの列

列 型 説明
objoid oid (いずれかのOID列) この補足説明が属するオブジェクトのOID
classoid oid (参照先 pg_class.oid) このオブジェクトが現れるシステムカタログのOID
description text このオブジェクトの説明となる任意のテキスト

51.48. pg_shseclabel

pg_shseclabelカタログは、共有データベースオブジェクト上のセキュリティラベルを保存します。セキュリティラベルはSECURITY LABELコマンドで操作されます。セキュリティラベルを見る簡単な方法は、51.83を参照してください。

pg_seclabelも参照してください。これは、単一データベース内のオブジェクトを含むセキュリティラベルのための類似した機能を提供します。

多くのシステムカタログと違い、pg_shseclabelはクラスタ内の全てのデータベース間で共有されます。pg_shseclabelは、データベースごとではなく、クラスタごとに1つのみ存在します。

表51.48 pg_shseclabelの列

列 型 説明
objoid oid (いずれかのOID列) このセキュリティラベルが関係するオブジェクトのOIDです。
classoid oid (参照先 pg_class.oid) このオブジェクトが現れるシステムカタログのOID
provider text このラベルに関連付いたラベルプロバイダです。
label text このオブジェクトに適用されるセキュリティラベルです。

51.49. pg_statistic

pg_statisticカタログはデータベースの内容に関する統計データを保存します。項目はANALYZEで作成され、後に問い合わせプランナで使用されます。最新のものと思ってもすべての統計データは本質的に大雑把なものであることに注意してください。

通常は、解析されるテーブル列毎に、stainherit = falseを持つ1つの項目が存在します。テーブルが継承された子を持つ場合、stainherit = trueを持つ2つ目の項目が作成されます。この行は継承ツリー全体に渡る列の統計情報、つまり、SELECT column FROM table*で確認できるデータに対する統計情報を表します。一方でstainherit = falseの行はSELECT column FROM ONLY tableの結果を表します。

pg_statisticはインデックス式の値についての統計データも格納します。これらはあたかも値が実際のデータ列であるかのように表現されます。特にstarelidはインデックスを参照します。これは元のテーブル列の項目に対して冗長となるので、普通の式を持たないインデックス列では項目は作成されません。現在インデックス式用の項目は常にstainherit = falseを持ちます。

異なる種類のデータに対しては違った種類の統計が相応しいことからpg_statisticはどのような情報を保存するか深く推定しないように設計されています。(例えばNULLであるような)極端に一般的な統計のみpg_statisticの特定の列に入ります。その他すべてはスロット列の内の1つのコード番号でその内容が識別される相関している列のグループである「スロット」に保存されます。src/include/catalog/pg_statistic.hを参照してください。

pg_statisticはテーブル内容に関する統計情報と言えども秘密の情報とみなされますので、一般のユーザが読み取り可能であってはいけません。(給与列の最高額と最低額などは誰もが興味をそそられる良い例ですよね。) pg_statsは一般のユーザが読み取り可能なpg_statisticに対するビューで、既存のユーザが読んでも差し支えないテーブルの情報のみを開示しています。

表51.49 pg_statisticの列

列 型 説明
starelid oid (参照先 pg_class.oid)

列 型 説明
記述された列が属するテーブルもしくはインデックス
staattnum int2 (参照先 pg_attribute.attnum) 記述された列数
stainherit bool 真の場合、統計情報には指定されたテーブルの値だけではなく、継承関係の子の列が含まれます。
stanullfrac float4 NULL値である列項目の割合
stawidth int4 非NULL項目の平均保存幅(バイト単位)
stadistinct float4 列内で非NULL個別値を持つデータ数。ゼロより大きい値は実際の個別値の数です。ゼロより小さい値はテーブル内の行数に対する乗数を負にしたものです。例えば、約80%の値が非NULLで、それぞれの非NULL値が平均して2回ほど出現する列はstadistinct = -0.4であると表現されます。ゼロは個別値の数を特定できない場合です。
stakindN int2 pg_statistic行のN番目の「スロット」に保存されている統計情報の種類を示すコード番号。
staopN oid (参照先 pg_operator.oid) N番目の「スロット」に保存されている統計情報を引き出すために使われる演算子。例えば、度数分布スロットはデータの並び換えの順序を定義する<演算子を示します。
stacollN oid (参照先 pg_collation.oid) N番目の「スロット」に格納された統計情報を派生させるために使われる照合順序。たとえば、照合可能な列のヒストグラムスロットはそのデータをソート順を定義する照合順を表示します。ゼロなら照合可能ではないデータです。
stanumbersN float4[] N番目の「スロット」に対する適切な種類の数値統計情報、もしくはスロットの種類に数値が含まれない時はNULLです。
stavaluesN anyarray N番目の「スロット」に対する適切な種類の列データの値、もしくはスロットの種類にデータ値が何も保存されていない場合はNULL。それぞれの配列要素の値は実際には特定された列のデータ型、もしくは配列要素の型といったような関連のある型になります。ですからanyarrayとする以外に列型を定義することはできません。

51.50. pg_statistic_ext

カタログpg_statistic_extはプランナの拡張統計情報の定義を保持します。このカタログの各行はCREATE STATISTICSで作成された統計オブジェクトに対応します。

表51.50 pg_statistic_extの列

列 型 説明
oid oid 行識別子
stxrelid oid (参照先 pg_class.oid)

列 型 説明
このオブジェクトが記述する列を含むテーブル
stxname name 統計オブジェクトの名前
stxnamespace oid（参照先 pg_namespace.oid） この統計オブジェクトを含む名前空間のOID
stxowner oid（参照先 pg_authid.oid） 統計オブジェクトの所有者
stxstattarget int4 stxstattargetはANALYZEによるこの列に対する蓄積された統計情報をどの程度詳しく管理するかを規定します。値がゼロの場合は統計情報を収集しません。負の値の場合は、参照する列の統計情報の収集目標の最大値があればそれを使い、なければシステムのデフォルトの統計目標を使用すべきであるということです。正の値のstxstattargetは、収集する「最も一般的な値」の目標となる数を決定します。
stxkeys int2vector（参照先 pg_attribute.attnum） 属性番号の配列で、どのテーブル列が統計オブジェクトに含まれるかを示します。例えば、値が1 3なら、テーブルの1番目と3番目の列が含まれるということになります。
stxkind char[] 有効にされた統計種別のコードが入る配列です。有効な値は、N個別統計を表すdと、関数従属統計を表すfと最も共通した値(MCV)を表すmです。

CREATE STATISTICSの実行中にpg_statistic_extエントリはすべて満たされますが、実際の統計データ値は計算されません。あとで実行されるANALYZEコマンドが必要な値を計算し、pg_statistic_ext_dataカタログのエントリに投入します。

51.51. pg_statistic_ext_data

カタログpg_statistic_ext_dataは、pg_statistic_extで定義されたプランナの拡張統計情報のデータを保持します。このカタログの個々の行はCREATE STATISTICSで作成された統計情報オブジェクトに関連します。

pg_statistic同様、pg_statistic_ext_dataはテーブル内容が秘密の情報とみなされますので、一般のユーザが読み取り可能であってはいけません。（列の値の最も共通した組み合わせは誰もが興味をそそられる良い例ですよね。）pg_stats_extは一般のユーザが読み取り可能な（pg_statistic_extと結合後の）pg_statistic_ext_dataに対するビューで、現在のユーザが読んでも差し支えないテーブルと列の情報のみを開示しています。

表51.51 pg_statistic_ext_dataの列

列 型 説明
stxoid oid（参照先 pg_statistic_ext.oid） このデータの定義を含む拡張統計情報オブジェクト
stxdndistinct pg_ndistinct

列 型 説明
pg_ndistinct型にシリアライズされたN個別値の数
stxddependencies pg_dependencies pg_dependencies型にシリアライズされた関数従属統計
stxdmcv pg_mcv_list pg_mcv_list型にシリアライズされたMCV(最も共通の値)リスト統計情報

51.52. pg_subscription

カタログpg_subscriptionには、存在するすべての論理レプリケーションのサブスクリプションが入ります。論理レプリケーションについての詳細な情報は第30章を参照してください。

ほとんどのシステムカタログとは異なり、pg_subscriptionはクラスタ内の全データベースで共有されます。つまりクラスタごとにpg_subscriptionの実体は1つだけ存在し、データベースごとに1つではありません。

列subconninfoには平文のパスワードが含まれる可能性があるため、一般ユーザによるアクセス権は取り消されています。

表51.52 pg_subscriptionの列

列 型 説明
oid oid 行識別子
subdbid oid (参照先 pg_database.oid) サブスクリプションが存在するデータベースのOID
subname name サブスクリプションの名前
subowner oid (参照先 pg_authid.oid) サブスクリプションの所有者
subenabled bool 真の場合、サブスクリプションは有効でレプリケーションが行われています。
subconninfo text 上流のデータベースへの接続文字列
subslotname name 上流のデータベースのレプリケーションスロットの名前。(ローカルレプリケーションのオリジン名としても使われます。) NULLはNONEを表します。
subsynccommit text サブスクリプションワーカのsynchronous_commitの設定値が入ります。
subpublications text[] サブスクライブされるパブリケーション名の配列です。パブリッシャーのサーバのパブリケーションを参照します。パブリケーションについての詳細は30.1を参照してください。

51.53. pg_subscription_rel

カタログpg_subscription_relには各サブスクリプションで複製される各リレーションの状態が入ります。これは多対多のマッピングです。

このカタログにはCREATE SUBSCRIPTIONあるいはALTER SUBSCRIPTION ... REFRESH PUBLICATIONを実行した後でサブスクリプションに知られることになったテーブルのみが含まれます。

表51.53 pg_subscription_relの列

列 型 説明
srsubid oid (参照先 pg_subscription.oid) サブスクリプションの参照
srrelid oid (参照先 pg_class.oid) リレーションの参照
srsubstate char 状態コードで、iは初期化、dはデータのコピー中、sは同期済み、rは準備完了(通常のレプリケーション)を表します。
srsublsn pg_lsn sあるいはrの状態なら、同期の調停で使われる状態変更のリモートLSNです。それ以外の場合はNULLです。

51.54. pg_tablespace

pg_tablespaceカタログは利用できるテーブル空間についての情報を格納します。テーブルは、ディスクの配置を管理できるようにするために特定のテーブル空間に格納することができます。

システムカタログの大部分とは違って、pg_tablespaceは、すべてのクラスタのデータベース間で共有されます。(データベース毎ではなく)クラスタ毎に、pg_tablespaceのコピーが1つだけ存在します。

表51.54 pg_tablespaceの列

列 型 説明
oid oid 行識別子
spcname name テーブル空間名
spcowner oid (参照先 pg_authid.oid) テーブル空間の所有者。たいていはテーブル空間を作成したユーザ
spcacl aclitem[] アクセス権限。詳細は5.7を参照してください。
spcoptions text[] 「keyword=value」文字列のようなテーブル空間レベルのオプション。

51.55. pg_transform

カタログpg_transformは変換についての情報を格納します。変換はデータ型を手続き言語に適合させるための機構です。詳しくはCREATE TRANSFORMを参照してください。

表51.55 pg_transformの列

列 型 説明
oid oid 行識別子
trftype oid (参照先 pg_type.oid) この変換の対象のデータ型のOID
trflang oid (参照先 pg_language.oid) この変換の対象の言語のOID
trffromsql regproc (参照先 pg_proc.oid) データ型を手続き言語への入力(例えば関数のパラメータ)に変換する時に使う関数のOID。この操作がサポートされない場合はゼロが格納されます。
trftosql regproc (参照先 pg_proc.oid) 手続き言語からの出力(例えば戻り値)をデータ型に変換する時に使う関数のOID。この操作がサポートされないときはゼロが格納されます。

51.56. pg_trigger

pg_triggerカタログはテーブルおよびビュー上のトリガを保存します。CREATE TRIGGERを参照してください。

表51.56 pg_triggerの列

列 型 説明
oid oid 行識別子
tgrelid oid (参照先 pg_class.oid) トリガのかかっているテーブル
tgparentid oid (参照先 pg_trigger.oid) このトリガが複製された親のトリガです。複製されていなければゼロです。パーティションが作成されたか、あるいはパーティションテーブルにアタッチされたときに起こります。
tgname name トリガ名(同一テーブル内で一意である必要があります)
tgfoid oid (参照先 pg_proc.oid) 呼び出される関数
tgtype int2

列 型 説明
トリガ発行条件を指定するビットマスク
tgenabled char どのsession_replication_roleモードでトリガが発行されるかを制御します。O = 「起点」モードと「ローカル」モードでトリガを発行します, D = トリガは無効です, R = 「replica」モードでトリガを発行します, A = 常にトリガを発行します.
tgisinternal bool トリガが(通常tgconstraintにより識別される制約を強制するために)内部的に生成される場合は真。
tgconstrrelid oid (参照先 pg_class.oid) 参照整合性制約で参照されるテーブル
tgconstrindid oid (参照先 pg_class.oid) 一意性、主キー、参照整合性制約や排他制約をサポートするインデックス
tgconstraint oid (参照先 pg_constraint.oid) 存在する場合は、トリガに関連するpg_constraintの項目
tgdeferrable bool トリガが遅延可能である場合は真
tginitdeferred bool トリガの初期状態が遅延可能と宣言されていれば真
tgnargs int2 トリガ関数に渡される引数の数
tgattr int2vector (参照先 pg_attribute.attnum) トリガが列固有であれば列番号。さもなくば空の配列
tgargs bytea トリガに渡される引数文字列で、それぞれヌル文字で終結
tgqual pg_node_tree トリガのWHEN条件に関する(nodeToString()表現による)式ツリー。存在しなければNULL
tgoldtable name OLD TABLEに対するREFERENCING句の名前、なければNULL
tgnewtable name NEW TABLEに対するREFERENCING句の名前、なければNULL

現在、列固有のトリガ処理はUPDATEイベントのみでサポートされていますので、tgattrはこの種類のイベントにのみ関連します。tgtypeにはこの他のイベント用のビットが含まれているかもしれませんが、これらはtgattrの値とは関係ないテーブル全体のものであると仮定されます。

> **注記**
>
> tgconstraintがゼロではないとき、tgconstrrelid、tgconstrindid、tgdeferrable、tginitdeferredは参照されるpg_constraint項目と共に冗長となっています。しかし遅延不可能なトリガを遅延可能な制約に関連付けさせることが可能です。外部キー制約では一部を遅延可能、一部を遅延不可能なトリガを持つことができます。

> **注記**
>
> pg_class.relhastriggersは、リレーションがこのカタログ内にトリガを持っている場合は真とならなければなりません。

51.57. pg_ts_config

pg_ts_configカタログは、テキスト検索の設定を表す項目を含みます。設定は、特定のテキスト検索パーサと、それぞれのパーサの出力トークン型のために使用される辞書の一覧を指定します。パーサはpg_ts_config項目内に示されていますが、トークンと辞書の対応付けは、pg_ts_config_map内の補助項目内に定義されています。

PostgreSQLのテキスト検索機能については第12章で詳しく説明します。

表51.57 pg_ts_configの列

列 型 説明
oid oid 行識別子
cfgname name テキスト検索設定の名称
cfgnamespace oid (参照先 pg_namespace.oid) この設定を含む名前空間のOID
cfgowner oid (参照先 pg_authid.oid) この設定の所有者
cfgparser oid (参照先 pg_ts_parser.oid) この設定のためのテキスト検索パーサのOID

51.58. pg_ts_config_map

pg_ts_config_mapカタログは、どのテキスト検索辞書を参照するべきかを示す項目を含みます。さらに、それぞれのテキスト検索設定のパーサの出力トークンをどの順番で参照すべきかを示す項目を含みます。

PostgreSQLのテキスト検索機能については第12章で詳しく説明します。

表51.58 pg_ts_config_mapの列

列 型 説明
mapcfg oid (参照先 pg_ts_config.oid) このマップ項目を所有するpg_ts_config項目のOID
maptokentype int4

列 型 説明
設定のパーサにより発行されるトークンの種類
mapseqno int4 この項目を参照する順番(小さいmapseqnoが先です)
mapdict oid (参照先 pg_ts_dict.oid) 参照するテキスト検索辞書のOID

51.59. pg_ts_dict

pg_ts_dictカタログは、テキスト検索辞書を定義する項目を含みます。辞書は、必要な実装関数すべてを指定するテキスト検索のテンプレートに依存します。辞書自身は、テンプレートによりサポートされている、ユーザが設定可能なパラメータ値を提供します。ここでは、辞書が特権のないユーザにより作成されることを許可します。パラメータは、dictinitoptionテキスト文字列で指定されます。その書式と意味はテンプレートにより変化します。

PostgreSQLのテキスト検索機能については第12章で詳しく説明します。

表51.59 pg_ts_dictの列

列 型 説明
oid oid 行識別子
dictname name テキスト検索辞書の名称
dictnamespace oid (参照先 pg_namespace.oid) この辞書を含む名前空間のOID
dictowner oid (参照先 pg_authid.oid) 辞書の所有者
dicttemplate oid (参照先 pg_ts_template.oid) 辞書のためのテキスト検索テンプレートのOID
dictinitoption text テンプレートのための初期化オプション文字列

51.60. pg_ts_parser

pg_ts_parserカタログはテキスト検索パーサを定義する項目を含みます。パーサは、入力テキストを語彙素に分割することとトークン型を語彙素に割り当てることに責任を持ちます。パーサはC言語レベルの関数で実装されていなくてはいけないため、新規のパーサの作成はデータベースのスーパーユーザに制限されています。

PostgreSQLのテキスト検索機能については第12章で詳しく説明します。

表51.60 pg_ts_parserの列

列 型 説明
oid oid 行識別子
prsname name テキスト検索パーサの名称
prsnamespace oid（参照先 pg_namespace.oid） このパーサを含む名前空間のOID
prsstart regproc（参照先 pg_proc.oid） パーサ起動関数のOID
prstoken regproc（参照先 pg_proc.oid） パーサの次のトークン関数のOID
prsend regproc（参照先 pg_proc.oid） パーサの終了関数のOID
prsheadline regproc（参照先 pg_proc.oid） パーサの見出し関数のOID
prslextype regproc（参照先 pg_proc.oid） パーサの字句型関数のOID

51.61. pg_ts_template

pg_ts_templateカタログはテキスト検索テンプレートを定義する項目を含みます。テンプレートはテキスト検索辞書クラスの骨格を実装したものです。テンプレートはC言語レベルの関数で実装されなくてはいけないため、新規のテンプレートの作成はデータベースのスーパーユーザに制限されています。

PostgreSQLのテキスト検索機能については第12章で詳しく説明します。

表51.61 pg_ts_templateの列

列 型 説明
oid oid 行識別子
tmplname name テキスト検索テンプレートの名称
tmplnamespace oid（参照先 pg_namespace.oid） このテンプレートを含む名前空間のOID
tmplinit regproc（参照先 pg_proc.oid） テンプレートの初期化関数のOID
tmpllexize regproc（参照先 pg_proc.oid） テンプレートの字句関数のOID

51.62. pg_type

pg_typeカタログはデータ型の情報を保存します。基本型と列挙型（スカラ型）はCREATE TYPEで作成され、ドメインはCREATE DOMAINで作成されます。複合型がテーブルの行構成を表すためデータベースの個々のテーブルに対して自動的に作成されます。複合型をCREATE TYPE ASで作成することもできます。

表51.62 pg_typeの列

列 型 説明
oid oid 行識別子
typname name データ型名
typnamespace oid（参照先 pg_namespace.oid） この型を含む名前空間のOID
typowner oid（参照先 pg_authid.oid） 型の所有者
typlen int2 固定長型では、typlenは型の内部表現内でのバイト数です。しかし、可変長型ではtyplenは負です。-1は「varlena」型（最初の4バイトにデータ長を含むもの）を意味し、-2はヌル終端のC言語の文字列を示します。
typbyval bool typbyvalは内部関数がこの型の値を値渡しか、参照渡しかを決定します。typlenが1、2、4バイト長（もしくはDatumが8バイトのマシン上では8バイト長）以外であれば、typbyvalを偽にする必要があります。可変長型は必ず参照渡しになります。typbyvalは長さが値渡し可能でも偽になり得ることに注意してください。
typtype char typtypeでは、bは基本型、cは複合型（例えばテーブルの行の型）、dは派生型（ドメインなど）、eは列挙型、pは疑似型、rは範囲型です。typrelidおよびtypbasetypeも参照してください。
typcategory char typcategoryは、パーサがどの暗黙のキャストが「選択」されるべきか決定するのに使用されるデータ型の任意の分類です。表 51.63を参照してください。
typispreferred bool 型がtypcategory内で選択されたキャスト対象である場合に真です。
typisdefined bool 型が定義されると真、ここが未定義型に対する予備の場所である時は偽。typisdefinedが偽の場合、型名と名前空間とOID以外は信頼すべきでありません。
typdelim char 配列入力の構文解析をする際にこの型の2つの値を分離する文字。区切り文字は配列データ型ではなく配列要素データ型に関連付けられることに注意してください。
typrelid oid（参照先 pg_class.oid） もしこれが複合型（typtypeを参照）であれば、この列は関連するテーブルを定義するpg_class項目を指します。（独立の複合型の場合、pg_class項目は実際にはテーブルを表しませんが、いずれにしても型のpg_attribute項目をリンクするために必要です。）複合型でない場合はゼロです。

列 型 説明
typelem oid（参照先 pg_type.oid） typelemがゼロでない場合、これはpg_typeの別の列を特定します。現在の型は、typelem型の値を生成する配列のように、配列要素を持てるようになります。「本当の」の配列型は可変長（typlen = -1）ですが、例えばnameとpointのように、いくつかの固定長（typlen > 0）型は同時に非ゼロのtypelemを持つことができます。もし固定長型がtypelemを持つ場合、その内部表現は他のデータを持たないtypelemデータ型の数個の値でなければなりません。可変長配列型には配列サブルーチンで定義されたヘッダを持ちます。
typarray oid（参照先 pg_type.oid） typarrayがゼロでない場合、typarrayはpg_type内のもうひとつの行を特定します。もうひとつの行は、この型を要素として持っている「本当」の配列型です。
typinput regproc（参照先 pg_proc.oid） 入力変換関数（テキスト形式）
typoutput regproc（参照先 pg_proc.oid） 出力変換関数（テキスト形式）
typreceive regproc（参照先 pg_proc.oid） 入力変換関数（バイナリ形式）、なければゼロ
typsend regproc（参照先 pg_proc.oid） 出力変換関数（バイナリ形式）、なければゼロ
typmodin regproc（参照先 pg_proc.oid） 型修飾子の入力関数。型が修飾子をサポートしていない場合はゼロ
typmodout regproc（参照先 pg_proc.oid） 型修飾子の出力関数。標準書式を使用する場合はゼロ
typanalyze regproc（参照先 pg_proc.oid） 独自のANALYZE関数。標準関数を使用する場合はゼロ
typalign char typalignはこの型の値を格納する際に必要な整列です。ディスク上での格納だけでなく、PostgreSQL内部の値の表現にも適用されます。ディスク上の完全な行の表現のように、複数の値が隣接して格納される際には、指定された境界で始まるように、この型のデータの前にパディングが挿入されます。アライメントの参照先は、連続しているデータ中の先頭のデータの開始位置です。使用可能な値は以下の通りです。 • c = char整列（すなわち、整列は必要ありません）。 • s = short整列（多くのマシンでは2バイトになります）。 • i = int整列（多くのマシンでは4バイトになります）。 • d = double整列（多くのマシンでは8バイトになりますが、必ずしもすべてがそうであるとは限りません）。
typstorage char typstorageは、varlena型（typlen = -1であるもの）において、その型がトーストされる予定であるか、この型においてアトリビュートに対するデフォルトの戦略が何であるかを示します。可能な値は以下です。 • p (plain)：値は常にplainで格納されなければなりません（非varlena型は常にこの値を使います）。

列 型 説明
• e：値は「従属的」リレーションに格納できます（リレーションがあるとき。リレーションに関してはpg_class.reltoastrelidを参照してください）。 • m（main）：値は圧縮してインラインで格納できます。 • x（extended）：値は圧縮することもできますし、圧縮した上で更に従属的リレーションに移動することもできます。 トースト可能な型に対してはxが通常の選択です。m値も、どうしても必要なら従属的格納に移動できることに注意してください（eとx値は、まず最初に移動します）。
typnotnull bool typnotnullは型に対し非NULL制約を表します。ドメインでのみ使用されます。
typbasetype oid（参照先 pg_type.oid） もしこれがドメイン（typtypeを参照）であれば、typbasetypeはこれが基づいている型を指定します。ドメインでない場合はゼロです。
typtypmod int4 ドメインはtyptypmodを使用して、基本型に適用されるtypmodを記録します（基本型がtypmodを使用しない場合は-1）。この型がドメインでない場合は-1です。
typndims int4 typndimsは配列であるドメインの配列の次元数です（つまり、typbasetypeは配列型です。）。配列型のドメインでない場合はゼロです。
typcollation oid（参照先 pg_collation.oid） typcollationは型の照合順序を指定します。型が照合順序をサポートしない場合、ゼロになります。照合順序をサポートする基本型はここでゼロ以外の値を持ちます。典型的にはDEFAULT_COLLATION_OIDです。照合順序の設定可能な型全体のドメインは、そのドメインで照合順序が指定されていれば、基本型とは異なる照合順序OIDを持つことができます。
typdefaultbin pg_node_tree typdefaultbinがNULLでない場合、これは型のデフォルト式のnodeToString()表現です。ドメインでのみ使用されます。
typdefault text 関連するデフォルト値を持たない型であればtypdefaultはNULLです。typdefaultbinがNULLでない場合、typdefaultは、typdefaultbinによって表される人間が見てわかる形式のデフォルト式を含む必要があります。typdefaultbinがNULLでtypdefaultがNULLでない場合、typdefaultは型のデフォルト値の外部表現です。これは、定数を生成するために型の入力変換処理に渡されることがあります。
typacl aclitem[] アクセス権限。詳細は5.7を参照してください。

注記

固定長型のシステムテーブルでは、pg_typeで定義されているサイズとアライメントと、コンパイラがテーブル行を表現する構造体の中で列を格納する方法とで合意が取れていることが重要です。

表 51.63はシステムで定義されたtypcategoryの値の一覧です。今後この一覧に追加されるものは同様に大文字のASCII文字になります。他のすべてのASCII文字はユーザ定義のカテゴリのために予約されています。

表51.63 typcategoryのコード

コード	カテゴリ
A	配列型
B	論理値型
C	複合型
D	日付時刻型
E	列挙型
G	幾何学型
I	ネットワークアドレス型
N	数値型
P	仮想型
R	範囲型
S	文字列型
T	時間間隔型
U	ユーザ定義型
V	ビット列型
X	unknown型

51.63. pg_user_mapping

pg_user_mappingカタログはローカルのユーザから遠隔のユーザへのマッピングを保持します。一般ユーザからのこのカタログへのアクセスは制限されています。代わりにpg_user_mappingsを使用してください。

表51.64 pg_user_mappingの列

列 型 説明
oid oid 行識別子
umuser oid (参照先 pg_authid.oid) マッピングされているローカルのロールのOID。ユーザマッピングが公開されている場合は0になります。
umserver oid (参照先 pg_foreign_server.oid) マッピングを保持する外部サーバのOID
umoptions text[] 「keyword=value」文字列のようなユーザマッピングの特定のオプション

51.64. システムビュー

システムカタログに加えPostgreSQLは数多くの組み込みビューを提供しています。システムビューはいくつかの一般的に使用されるシステムカタログに対する問い合わせに手近にアクセスできるようにします。他のビューはサーバ状態内部へのアクセスを提供します。

情報スキーマ(第36章)はシステムビューと重複する、もう一方のビューの集合を提供しています。ここで説明しているビューはPostgreSQL特有のものであるのに対し、情報スキーマはSQL標準であることから、もし情報スキーマが必要とする情報をすべて提供してくれるのであれば情報スキーマを使用する方が良いでしょう。

表 51.65は、ここで説明しているシステムビューの一覧です。それぞれのビューのさらに詳細な説明は、これより後に述べられています。統計情報の結果にアクセスするためのいくつかの追加のビューがあります。それらは表 27.2で説明されています。

注意書きがない限り、ここでのすべてのビューは読み取り専用です。

表51.65 システムビュー

ビュー名	目的
pg_available_extensions	利用可能な拡張
pg_available_extension_versions	利用可能な拡張のバージョン
pg_config	コンパイル時の設定パラメータ
pg_cursors	開いているカーソル
pg_file_settings	設定ファイルの内容の要約
pg_group	データベースのユーザのグループ
pg_hba_file_rules	クライアント認証の設定ファイルの内容の要約
pg_indexes	インデックス
pg_locks	現在保持されている、または待っているロック
pg_matviews	マテリアライズドビュー
pg_policies	ポリシー
pg_prepared_statements	準備済みの文
pg_prepared_xacts	準備済みのトランザクション
pg_publication_tables	パブリケーションとそれに関連するテーブル
pg_replication_origin_status	レプリケーションの進捗を含めたレプリケーション起点に関する情報
pg_replication_slots	レプリケーションスロットの情報
pg_roles	データベースロール
pg_rules	ルール
pg_seclabels	セキュリティラベル
pg_sequences	シーケンス

ビュー名	目的
pg_settings	パラメータ設定
pg_shadow	データベースのユーザ
pg_shmem_allocations	shared memory allocations
pg_stats	プランナの統計
pg_stats_ext	プランナの拡張統計情報
pg_tables	テーブル
pg_timezone_abbrevs	時間帯省略形
pg_timezone_names	時間帯名
pg_user	データベースのユーザ
pg_user_mappings	ユーザマッピング
pg_views	ビュー

51.65. pg_available_extensions

pg_available_extensionsビューはインストレーションで利用可能な拡張を列挙します。現在インストールされている拡張を表すpg_extensionカタログも参照してください。

表51.66 pg_available_extensionsの列

列 型 説明
name name 拡張名
default_version text デフォルトのバージョン名称。何も指定がなければNULL
installed_version text 現在インストールされている拡張のバージョン。インストールされていない場合はNULL
comment text 拡張の制御ファイルからのコメント文字列

pg_available_extensionsビューは読み取り専用です。

51.66. pg_available_extension_versions

pg_available_extension_versionsビューはインストレーションで利用可能な特定の拡張のバージョンを列挙します。現在インストールされている拡張を表すpg_extensionカタログも参照してください。

表51.67 pg_available_extension_versionsの列

列 型 説明
name name 拡張名
version text バージョン名称
installed bool 現在このバージョンの拡張がインストールされている場合に真
superuser bool スーパユーザがこの拡張をインストールできるなら真です(ただし、trustedを見てください)。
trusted bool 適切な権限を持つ非スーパユーザがこの拡張をインストールできるなら真です。
relocatable bool 拡張が他のスキーマに再配置可能である場合真
schema name 拡張がインストールされなければならないスキーマの名前。一部の再配置またはすべての再配置を行うことができる場合はNULL
requires name[] 前もって必要な拡張の名前。なければNULL
comment text 拡張の制御ファイルからのコメント文字列

pg_available_extension_versionsビューは読み取り専用です。

51.67. pg_config

pg_configビューは、現在インストールされているPostgreSQLのバージョンのコンパイル時設定パラメータを表示します。例えば、PostgreSQLとインタフェースしたいソフトウェアパッケージによって、要求されるヘッダファイルとライブラリを探す手助けとなるために使用されることが意図されます。PostgreSQLクライアントアプリケーションであるpg_configと同様な基本的な情報を提供します。

デフォルトではpg_configビューはスーパユーザだけが読み取りできます。

表51.68 pg_configの列

列 型 説明
name text パラメータ名
setting text パラメータ値

51.68. pg_cursors

pg_cursorsビューは現在利用可能なカーソルを列挙します。以下のようにカーソルは複数の方法で定義可能です。

- SQLからDECLARE文経由。
- 52.2.3で説明する、フロントエンド/バックエンドプロトコルからBindメッセージ経由。
- 46.1で説明する、サーバプログラミングインタフェース(SPI)経由。

pg_cursorsビューは、上のいずれかの方法で作成されたカーソルを表示します。カーソルは、WITH HOLDと宣言されていない限り、それを定義したトランザクション期間しか存在しません。したがって、保持不可能なカーソルは、作成元トランザクションが終わるまでの間のみ、このビューに現れます。

> **注記**
>
> 手続き言語など、一部のPostgreSQLの要素を実装するために内部的にカーソルが使用されています。したがって、pg_cursorsにはユーザが明示的に作成していないカーソルも含まれる可能性があります。

表51.69 pg_cursorsの列

列 型 説明
name text カーソルの名前
statement text カーソル宣言の際に投稿された逐語的問い合わせ文字列
is_holdable bool 保持可能カーソル(つまりカーソルを宣言したトランザクションがコミットされた後でもアクセス可能なカーソル)であればtrueです。さもなくばfalseです。
is_binary bool カーソルがBINARYで宣言されていたらtrue、さもなくばfalse。
is_scrollable bool カーソルがスクロール可能(順序通り以外の方法に行を取り出すことが可能)であればtrue、さもなくばfalse。
creation_time timestamptz カーソルが宣言された時間。

pg_cursorsビューは読み取り専用です。

51.69. pg_file_settings

ビューpg_file_settingsはサーバの設定ファイルの内容の要約を提供します。ファイル内にある各「name = value」のエントリについて、このビューの1行が存在し、その値が正しく適用可能かどうかの注釈が含まれ

ます。ファイル内の構文エラーなど「name = value」のエントリと関係のない問題についての行がさらに存在することもあります。

設定ファイルについて予定している変更が動作するかどうかの確認や、以前のエラーの調査分析をする際にこのビューは役立ちます。このビューはファイルの現在の内容についてレポートするのであって、サーバが最後に適用した内容ではないことに注意してください。(後者を知るには、通常はpg_settingsビューで十分でしょう。)

デフォルトで、pg_file_settingsビューはスーパーユーザのみが参照可能です。

表51.70 pg_file_settingsの列

列 型 説明
sourcefile text 設定ファイルの完全なパス名
sourceline int4 設定ファイル内のエントリの行番号
seqno int4 エントリが処理される順序(1..n)
name text 設定パラメータ名
setting text パラメータに代入される値
applied bool 値が正しく適用可能なら真
error text NULLでないときは、このエントリが適用できない理由についてのエラーメッセージ

設定ファイルに構文エラーや不正なパラメータ名がある場合、サーバはファイル内の設定をまったく適用せず、すべてのappliedフィールドは偽になります。このような場合は、errorフィールドが非NULLで問題を示唆する行が1行以上あるでしょう。それ以外の場合は、個々の設定は可能であれば適用されます。個々の設定が適用できない場合(例えば、不正な値、サーバの起動後は設定が変更できないなど)はerrorフィールドに適切なメッセージがあります。エントリのappliedが偽になる別の理由は、同じパラメータがそれより後のエントリで上書きされている場合です。この場合はエラーとはみなされませんので、errorフィールドには何も表示されません。

実行時パラメータを変更する様々な方法について、詳しくは19.1を参照してください。

51.70. pg_group

pg_groupビューは下位互換のために存在しています。バージョン8.1以前のPostgreSQLのカタログを模擬しています。このビューは、rolcanloginとしてマークされていない、すべてのロールの名前とメンバを保持しています。これはグループとして使用されているロールの集合と似ています。

表51.71 pg_groupの列

列 型 説明
groname name（参照先 pg_authid.rolname） グループの名前
grosysid oid（参照先 pg_authid.oid） グループのID
grolist oid[]（参照先 pg_authid.oid） このグループのロールIDを含む配列

51.71. pg_hba_file_rules

ビューpg_hba_file_rulesはクライアント認証の設定ファイルpg_hba.confの内容の要約を提供します。設定ファイル内の空でない、コメントでもない各行について、このビュー内に行が1つあり、ルールが正しく適用できたかどうかを示す注記が入ります。

このビューは、認証の設定ファイルについて計画している変更が動作するかどうかを確認する、あるいは以前の失敗について分析するのに役立つでしょう。このビューはサーバが最後に読み込んだものではなく、ファイルの現在の内容について報告することに注意してください。

デフォルトでは、スーパーユーザのみがpg_hba_file_rulesビューを読み取ることができます。

表51.72 pg_hba_file_rulesの列

列 型 説明
line_number int4 pg_hba.conf内でのこのルールの行番号
type text 接続の種別
database text[] このルールが適用されるデータベース名のリスト
user_name text[] このルールが適用されるユーザ名とグループ名のリスト
address text ホスト名、IPアドレス、あるいはall、samehost、samenetのいずれか。ローカル接続の場合はNULL。
netmask text IPアドレスマスク。当てはまらない場合はNULL。
auth_method text 認証方法
options text[] 認証方法について指定されたオプション（あれば）

列 型 説明
error text NULLでないなら、この行がなぜ処理できなかったかを示すエラーメッセージ

不正なエントリに対応する行は、通常はline_numberフィールドとerrorフィールドにのみ値が入ります。

クライアント認証設定の詳細については第20章を参照してください。

51.72. pg_indexes

pg_indexesビューはデータベース内のそれぞれのインデックスについて有用な情報を提供します。

表51.73 pg_indexesの列

列 型 説明
schemaname name (参照先 pg_namespace.nspname) テーブルとインデックスを含むスキーマの名前
tablename name (参照先 pg_class.relname) インデックスのついているテーブルの名前
indexname name (参照先 pg_class.relname) インデックスの名前
tablespace name (参照先 pg_tablespace.spcname) インデックスを含むテーブル空間の名前(データベースのデフォルトはNULL)
indexdef text インデックス定義(再作成用CREATE INDEXコマンド)

51.73. pg_locks

pg_locksビューはデータベースサーバ内でアクティブなプロセスによって保持されたロックに関する情報へのアクセスを提供します。ロックに関するより詳細な説明は第13章を参照してください。

pg_locksにはロック対象となる進行中のオブジェクト、要求されたロックモード、および関連するプロセス毎に1つの行を持ちます。ですから、もし複数のプロセスが同じロック対象オブジェクトに対してロックを保持していたりロックを待機している場合には、同じロック対象オブジェクトが何度も出現することがあります。しかし現在ロックされていないオブジェクトはまったく現れません。

ロック対象オブジェクトには異なる型がいくつか存在します。リレーション全体(例:テーブル)、リレーションの個別のページ、リレーションの個別のタプル、トランザクションID(仮想と永続の両方のID)、一般的なデータベースオブジェクト(これはpg_descriptionやpg_dependと同様にクラスOIDとオブジェクトOIDで識別されます)。さらに、リレーションを拡張する権利は、pg_database.datfrozenxidを更新する権利と同様に、別のロック対象オブジェクトとして表現されます。また「勧告的」ロックはユーザ定義の意味を持つ複数から形成されるかもしれません。

表51.74 pg_locksの列

列 型 説明
locktype text ロックオブジェクトのタイプです。relation、extend、frozenid、page、tuple、transactionid、virtualxid、spectoken、object、userlock、advisoryのどれかです（表 27.11も見てください）。
database oid（参照先 pg_database.oid） ロック対象が存在しているデータベースのOID。対象が共有オブジェクトの場合はゼロ。対象がトランザクションIDである場合はNULL。
relation oid（参照先 pg_class.oid） ロックの対象となるリレーションのOID。対象がリレーションではない場合かリレーションの一部である場合はNULL。
page int4 ロックの対象となるリレーション内のページ番号。対象がタプルもしくはリレーションページではない場合はNULL。
tuple int2 ページ内のロックの対象となっているタプル番号。対象がタプルではない場合はNULL。
virtualxid text ロックの対象となるトランザクションの仮想ID。対象が仮想トランザクションIDではない場合はNULL。
transactionid xid ロックの対象となるトランザクションのID。対象がトランザクションIDではない場合はNULL。
classid oid（参照先 pg_class.oid） ロックの対象を含むシステムカタログのOID。対象が一般的なデータベースオブジェクトではない場合はNULL。
objid oid（いずれかのOID列） システムカタログ内のロックの対象のOID。対象が一般的なデータベースオブジェクトでない場合はNULL。
objsubid int2 ロック対象の列番号（classidとobjidはテーブル自身を参照します）、その他の一般的なデータベースオブジェクトではゼロ、一般的ではないデータベースオブジェクトではNULLです。
virtualtransaction text ロックを保持、もしくは待っている仮想トランザクションID。
pid int4 ロックを保持、もしくは待っているサーバプロセスのプロセスID。ただしプリペアードトランザクションによりロックが保持されている場合はNULL。
mode text このプロセスで保持または要求するロックモードの名称。（13.3.1 and 13.2.3参照）
granted bool ロックが保持されている場合は真、ロックが待ち状態の場合は偽
fastpath bool ファストパス経由でロックが獲得されている場合は真、メインロックテーブル経由で獲得されている場合は偽。

指定されたプロセスにより保持されているロックを表す行内ではgrantedは真です。偽の場合はこのロックを獲得するため現在プロセスが待機中であることを示しています。つまり、同じロック対象のオブジェクトに対して何らかの他のプロセスが競合するロックを保持、もしくは待機していることを意味します。待機中のプロセ

スはその別のプロセスがロックを解放するまで活動を控えます。(もしくはデッドロック状態が検出されることになります)。単一プロセスでは一度に多くても1つのロックを獲得するために待機します。

トランザクションの実行中は常に、サーバプロセスはその仮想トランザクションID上に排他的ロックをかけます。もしある永続IDがトランザクションに割り当てられる(普通はトランザクションがデータベースの状態を変化させるときのみに発生します)と、トランザクションは終了するまで永続トランザクションIDに対して排他ロックを保持します。あるトランザクションが他のトランザクションを特定して終了まで待機しなければならないと判断した場合、他とみなしたトランザクションのIDに対し共有ロックを獲得するように試み、目的を達します。(仮想IDであるか永続IDであるかは、その状況によります)。これは、他とみなしたトランザクションが完了し、そしてロックを解放した場合のみ成功します。

タプルはロック対象のオブジェクト種類ですが、行レベルロックについての情報はメモリではなく、ディスクに保存されます。よって行レベルロックは通常、このビューには現れません。もしプロセスが行レベルロックの待ち状態である場合は、その行ロックを保持している永続トランザクションIDを待つ状態で、そのトランザクションはビューに現れます。

勧告的ロックは、単一のbigint値、または、2つの整数値をキーとして獲得することができます。bigintの場合は、その上位半分がclassid列内に表示され、残りの下位半分はobjid列内に表示されます。また、objsubidは1です。元のbigint値を(classid::bigint << 32) | objid::bigintという式で再構成することができます。整数値キーでは、最初のキーがclassid列に、2番目のキーがobjid列に表示され、objsubidは2です。キーの実際の意味はユーザに任されています。勧告的ロックはデータベースに対して局所的ですので、勧告的ロックではdatabase列が意味を持ちます。

pg_locksは現行のデータベースに関連するロックのみならず、データベースクラスタ内のすべてのロックに関する全体的なビューを提供します。relation列はロックされたリレーションを識別するためにpg_class.oidと結合することができますが、これは現行のデータベース内のリレーション(database列が現行のデータベースのOIDまたはゼロとなっているもの)に対してのみ正常に動作します。

それぞれのロックを保持もしくは待機しているセッションのさらなる情報を入手するためpg_stat_activityビューのpid列とpid列を結合することができます。例えば、このような感じです。

```
SELECT * FROM pg_locks pl LEFT JOIN pg_stat_activity psa
    ON pl.pid = psa.pid;
```

また、プリペアードトランザクションを使用している場合には、ロックを保持しているプリペアードトランザクションに関してより多くの情報を得るため、virtualtransaction列は、pg_prepared_xactsビューのtransaction列と結合することができます。(プリペアードトランザクションはロックを待つことはありませんが、実行時に獲得したロックを保持し続けます。) 例えば、このような感じです。

```
SELECT * FROM pg_locks pl LEFT JOIN pg_prepared_xacts ppx
    ON pl.virtualtransaction = '-1/' || ppx.transaction;
```

pg_locksビューとそれ自身の結合によって、どのプロセスが他のどのプロセスをブロックしているかの情報を入手することが可能ですが、同時に詳細な正しい情報を得ることは非常に困難です。このようなクエリはどのロックモードが他のものと衝突しているかについての知見を書き出すべきです。さらに悪いことに、pg_locksビューは、ロック待ちキューにてどのプロセスが他のどのプロセスに先行しているかの情報を提供しない、またはどのプロセスが他のクライアントセッションのために動作している並列ワーカプロセスかの情報を提供しません。待機しているプロセスが、どのプロセスにブロックされているかを識別するためにより良い方法は、pg_blocking_pids()関数(表 9.63を参照してください)を使用することです。

pg_locksビューは、異なるシステムにおける、通常のロックマネージャと述語ロックマネージャの両方からのデータを表示します。さらに通常のロックマネージャではロックを通常ロックと近道ロックに細分化します。このデータが完全に一貫性があることは保証されません。ビューが問い合わせられると、近道ロック（fastpath = trueが真）は、ロックマネージャ全体の状態を凍結することなく、各バックエンドからひとつひとつ収集されます。このため情報収集期間中にロックが獲得されたり解放されたりされる可能性があります。しかし、これらのロックはその時点で存在する他のロックと競合することがないことが分かっていることに注意してください。近道ロックについてすべてのバックエンドを問い合わせた後、通常のロックマネージャの残りは1つの単位としてロックされ、残りすべてのロックの一貫性があるスナップショットを原子的な処理で収集します。ロックマネージャのロックを解除した後、述語ロックマネージャは同様にロックされ、すべての述語ロックを原子的な処理で収集します。このように、近道ロックという例外がありますが、各ロックマネージャは一貫性をもった結果セットを生成します。しかし、両方のロックマネージャを同時にロックしませんので、通常のロックマネージャを問い合わせた後と述語ロックマネージャを問い合わせる前の間にロックが獲得されたり解放されたりされる可能性があります。

このビューが頻繁にアクセスされている場合は、通常もしくは述語ロックマネージャをロックするとデータベースのパフォーマンスに影響があります。ロックマネージャからデータを取得するために、ロックは必要最低限の時間だけ保持されますが、パフォーマンスに影響がある可能性が全くないわけではありません。

51.74. pg_matviews

pg_matviewsビューは、データベース内のマテリアライズドビューそれぞれに関する有用な情報へのアクセスを提供します。

表51.75 pg_matviewsの列

列 型 説明
schemaname name（参照先 pg_namespace.nspname） マテリアライズドビューを含むスキーマの名前
matviewname name（参照先 pg_class.relname） マテリアライズドビューの名前
matviewowner name（参照先 pg_authid.rolname） マテリアライズドビューの所有者の名前
tablespace name（参照先 pg_tablespace.spcname） マテリアライズドビューを含むテーブル空間の名前（データベースのデフォルトであればNULL）
hasindexes bool マテリアライズドビューがインデックスを持つ（または最近まで持っていた）場合に真
ispopulated bool マテリアライズドビューが現在データ投入されている場合に真
definition text マテリアライズドビューの定義（再構成されたSELECT問い合わせ）

51.75. pg_policies

ビューpg_policiesはデータベース内の行単位セキュリティのポリシーについて便利な情報へのアクセスを提供します。

表51.76 pg_policiesの列

列 型 説明
schemaname name（参照先 pg_namespace.nspname） ポリシーが適用されているテーブルがあるスキーマの名前
tablename name（参照先 pg_class.relname） ポリシーが適用されているテーブルの名前
policyname name（参照先 pg_policy.polname） ポリシーの名前
permissive text 許容(permissive)ポリシーか、制限(restrictive)ポリシーか
roles name[] このポリシーが適用されるロール
cmd text ポリシーが適用されるコマンドの種類
qual text このポリシーが適用される問い合わせにセキュリティバリアの制約として追加される式
with_check text このテーブルに行を追加する問い合わせにWITH CHECKの制約として追加される式

51.76. pg_prepared_statements

pg_prepared_statementsビューは現在のセッションで利用可能な準備済み文をすべて表示します。準備済み文についての詳細はPREPAREを参照してください。

pg_prepared_statementsには、1つの準備済み文に対して一行が存在します。新しい準備済み文が作成されると行が追加され、準備済み文が解放される（例えばDEALLOCATEを使用）と行が削除されます。

表51.77 pg_prepared_statementsの列

列 型 説明
name text 準備済み文の識別子
statement text この準備済み文を作成するためにクライアントが送付した問い合わせ文字列。SQL経由で作成された準備済み文では、これはクライアントが送信したPREPARE文です。フロントエンド/バックエンドプロトコル経由で作成された準備済み文では、これは準備済み文自身のテキストです。
prepare_time timestamptz 準備済み文が作成された時間

列 型 説明
parameter_types regtype[] regtype配列形式の準備済み文が想定しているパラメータ型。配列要素に対応するOIDは、regtypeからoidへのキャストを行うことで取り出すことができます。
from_sql bool 準備済み文がPREPARE SQLコマンド経由で作成された場合はtrue、フロントエンド/バックエンドプロトコル経由で文が準備された場合はfalse

pg_prepared_statementsビューは読み取り専用です。

51.77. pg_prepared_xacts

pg_prepared_xactsビューは、現状で2相コミットのためにプリペアードトランザクションについての情報を表示します（詳細はPREPARE TRANSACTIONを参照してください）。

pg_prepared_xactsは、プリペアードトランザクション毎に1つの行を含みます。この項目はトランザクションがコミットもしくはロールバックされたときに削除されます。

表51.78 pg_prepared_xactsの列

列 型 説明
transaction xid プリペアードトランザクションに対する数値のトランザクション識別子
gid text トランザクションに割り当てられたグローバルのトランザクション識別子
prepared timestamptz トランザクションがコミットのために準備された時間
owner name（参照先 pg_authid.rolname） トランザクションを実行したユーザ名
database name（参照先 pg_database.datname） トランザクションを実行したデータベース名

pg_prepared_xactsビューにアクセスすると、内部のトランザクション管理データ構造が一時的にロックされます。そして表示用にコピーが作成されます。これは、必要以上に長く通常の操作をブロックさせずに、ビューが一貫性のある結果を生成することを保証します。このビューが頻繁にアクセスされると、データベースの性能になんらかの影響を及ぼします。

51.78. pg_publication_tables

ビューpg_publication_tablesはパブリケーションとそれに含まれるテーブルの間のマッピングに関する情報を提供します。その元となるカタログpg_publication_relとは異なり、このビューはFOR ALL TABLESで定

義されたパブリケーションを展開するため、そのようなパブリケーションについては対象となる各テーブルについて1行があります。

表51.79 pg_publication_tablesの列

列 型 説明
pubname name（参照先 pg_publication.pubname） パブリケーションの名前
schemaname name（参照先 pg_namespace.nspname） テーブルがあるスキーマの名前
tablename name（参照先 pg_class.relname） テーブルの名前

51.79. pg_replication_origin_status

pg_replication_origin_statusビューには、ある起点の再生の進捗についての情報が含まれます。レプリケーション起点についての詳細は第49章を参照してください。

表51.80 pg_replication_origin_statusの列

列 型 説明
local_id oid（参照先 pg_replication_origin.roident） 内部ノード識別子
external_id text（参照先 pg_replication_origin.roname） 外部ノード識別子
remote_lsn pg_lsn そのデータまで複製されたことを示す起点ノードのLSN
local_lsn pg_lsn そのremote_lsnが複製されたことを示す、このノードのLSN。非同期コミットを使用している場合に、データをディスクに書き出す前にコミットレコードをフラッシュするために使用されます。

51.80. pg_replication_slots

pg_replication_slotsは、現在存在するデータベースクラスタとその状態、全てのレプリケーションスロットの一覧を提供します。

レプリケーションスロットに関する詳細は、26.2.6と第48章を参照してください。

表51.81 pg_replication_slotsの列

列 型 説明
slot_name name

列 型 説明
クラスタ間で一意なレプリケーションスロットの識別子
plugin name 出力プラグインに使用されている論理スロットまたは物理スロットの場合はnull、を含む共有オブジェクトの基底名。
slot_type text スロットのタイプ。physicalまたはlogical。
datoid oid（参照先 pg_database.oid） このスロットと関連しているデータベースのOID、またはnull。論理スロットだけがデータベースと関連を持つことができます。
database name（参照先 pg_database.datname） このスロットと関連しているデータベース名、またはnull。論理スロットだけがデータベースと関連を持つことができます。
temporary bool これが一時レプリケーションスロットの場合真になります。一時スロットはディスクに保存されず、エラーのとき、またはセッションが終了したときには自動的に削除されます。
active bool このスロットが現在アクティブで使用されている場合、真。
active_pid int4 このスロットが現在アクティブで使用されている場合は、スロットを使用しているセッションのプロセスID。アクティブでなければNULL。
xmin xid このスロットがデータベースとの接続を必要としている最も古いトランザクション。VACUUM は後でトランザクションによって削除されたタプルを除去できません。
catalog_xmin xid このスロットがデータベースとの接続を必要としている、システムカタログに影響する最も古いトランザクション。VACUUMは後でトランザクションによって削除されたカタログのタプルを除去できません。
restart_lsn pg_lsn 消費者のスロットによって必要とされており、LSNが現在のLSNからmax_slot_wal_keep_size以上遅れていない限り、チェックポイント中に自動的に削除されない最古のアドレス（LSN）です。このスロットのLSNが保存されていなければNULLです。
confirmed_flush_lsn pg_lsn 利用者がデータの受信を確認できている論理スロットのアドレス（LSN））。これより古いデータは、もはや有効ではありません。物理スロットの場合はNULL。
wal_status text このスロットが報告するWALファイルの入手可能性。可能な値は以下です。 • reservedなら、報告されたファイルはmax_wal_size内であることを意味します。 • extendedなら、報告されたファイルはmax_wal_sizeを越えているものの、レプリケーションスロットあるいはwal_keep_sizeによって、まだ保存されていることを意味します。 • unreservedは、スロットはもはや必要なWALファイルを保存しておらず、その一部は次のチェックポイントで削除予定であることを意味します。この状態はreservedあるいはextendedを返すことがあります。 • lostは、必要なWALファイルの一部が削除されており、このスロットはもはや利用可能ではないことを意味します。

列 型 説明
最後の2つの状態は、max_slot_wal_keep_sizeが非負の場合にのみ起こります。restart_lsnがNULLなら、このフィールドはNULLです。
safe_wal_size int8 「ロスト」状態に陥る危険性のないスロットにおいて、WALに書き込むことのできるバイト数です。失われたスロットに対して、あるいはmax_slot_wal_keep_sizeが-1ならNULLです。

51.81. pg_roles

pg_rolesビューはデータベースのロールに関する情報を提供します。これは単に一般に公開されているpg_authidのビューですが、パスワード列が空白になっています。

表51.82 pg_rolesの列

列 型 説明
rolname name ロール名
rolsuper bool ロールはスーパーユーザの権限を持っています
rolinherit bool ロールは自動的にメンバとして属するロールの権限を継承します
rolcreaterole bool ロールはロールを作成することができます
rolcreatedb bool ロールはデータベースを作成することができます
rolcanlogin bool ロールはログインすることができます。つまりロールはセッションを始める認証の識別子となることができます
rolreplication bool ロールはレプリケーション用のロールです。レプリケーションロールは、レプリケーション接続を開始すること、およびレプリケーションスロットを作成および削除することができます。
rolconnlimit int4 ログイン可能なロールでは、これはロールが確立できる同時実行接続数を設定します。-1は制限無しを意味します。
rolpassword text パスワードでありません（常に********のように読まれます）
rolvaliduntil timestamptz パスワード有効期限（パスワード認証でのみ使用）。NULLの場合には満了時間はありません。
rolbypassrls bool すべての行単位セキュリティポリシーを無視するロール。詳しくは5.8を参照してください。
rolconfig text[]

列 型 説明
実行時設定変数に関するロール固有のデフォルト
oid oid（参照先 pg_authid.oid） ロールのID

51.82. pg_rules

pg_rulesビューは問い合わせ書き換えルールについての有用な情報へのアクセスを提供します。

表51.83 pg_rulesの列

列 型 説明
schemaname name（参照先 pg_namespace.nspname） テーブルがあるスキーマの名前
tablename name（参照先 pg_class.relname） ルールの対象のテーブル名
rulename name（参照先 pg_rewrite.rulename） ルール名
definition text ルール定義（再構築された生成コマンド）

pg_rulesビューは、ビューおよびマテリアライズドビューに対するON SELECTルールを除外します。これらはpg_viewsおよびpg_matviewsにあります。

51.83. pg_seclabels

pg_seclabelsビューはセキュリティラベルに関する情報を提供します。これはpg_seclabelカタログをより問い合わせし易くしたものです。

表51.84 pg_seclabelsの列

列 型 説明
objoid oid（いずれかのOID列） このセキュリティラベルが関係するオブジェクトのOIDです。
classoid oid（参照先 pg_class.oid） このオブジェクトが現れるシステムカタログのOID
objsubid int4 テーブル列上のセキュリティラベルでは、これは列番号です（objoidおよびclassoidはテーブル自身を参照します）。他のすべての種類のオブジェクトでは、この列はゼロです。

列 型 説明
objtype text このラベルが適用されるオブジェクトの種類のテキスト表現
objnamespace oid（参照先 pg_namespace.oid） もし適用可能であればこのオブジェクト用の名前空間のOID。さもなければNULL
objname text このラベルが適用されるオブジェクト名称のテキスト表現
provider text（参照先 pg_seclabel.provider） このラベルに関連付いたラベルプロバイダです。
label text（参照先 pg_seclabel.label） このオブジェクトに適用されるセキュリティラベルです。

51.84. pg_sequences

ビューpg_sequencesはデータベース内の各シーケンスについての有用な情報へのアクセスを提供します。

表51.85 pg_sequencesの列

列 型 説明
schemaname name（参照先 pg_namespace.nspname） シーケンスがあるスキーマの名前
sequencename name（参照先 pg_class.relname） シーケンスの名前
sequenceowner name（参照先 pg_authid.rolname） シーケンスの所有者の名前
data_type regtype（参照先 pg_type.oid） シーケンスのデータ型
start_value int8 シーケンスの開始値
min_value int8 シーケンスの最小値
max_value int8 シーケンスの最大値
increment_by int8 シーケンスの増分値
cycle bool シーケンスが周回するかどうか
cache_size int8 シーケンスのキャッシュサイズ

列 型 説明
last_value int8 ディスクに書き込まれた最後のシーケンス値。キャッシュが使用されている場合、この値はシーケンスから最後に取り出された値より大きくなることがあります。シーケンスからまだ読み取られていないときはNULLになります。また、現在のユーザがシーケンスについてUSAGEあるいはSELECT権限がない場合も値はNULLになります。

51.85. pg_settings

pg_settingsビューはサーバの実行時パラメータへのアクセスを提供します。基本的にSHOWとSETコマンドの代わりとなるインタフェースです。同時に最大・最小値などのようにSHOWコマンドでは直接入手できないそれぞれのパラメータのいくつかの実状にアクセスする機能を提供します。

表51.86 pg_settingsの列

列 型 説明
name text 実行時設定パラメータ名
setting text パラメータの現在値
unit text 暗黙的なパラメータの単位
category text パラメータの論理グループ
short_desc text パラメータの簡潔な説明
extra_desc text 追加で、より詳細なパラメータについての説明
context text パラメータ値を設定するために必要な文脈（後述）
vartype text パラメータの型（bool、enum、integer、realもしくはstring）
source text 現在のパラメータ値のソース
min_val text 容認されている最小のパラメータ値（数値でない場合はNULL）
max_val text 容認されている最大のパラメータ値（数値でない場合はNULL）
enumvals text[] 許可された列挙パラメータの値(列挙型ではない場合はNULL)

列 型 説明
boot_val text パラメータが設定されていなかったとした場合に仮定されるサーバ起動時のパラメータ値
reset_val text 現状のセッションにおいてRESETによって戻されるパラメータの値
sourcefile text 現状の値が設定されている設定ファイル(設定ファイル以外のソースから設定された値の場合、スーパーユーザでもpg_read_all_settingsのメンバーでもないユーザから検査された時はNULLです)。設定ファイル内でinclude指示子を使用する時に役に立ちます。
sourceline int4 現状の値が設定されている設定ファイル内の行番号(設定ファイル以外のソースから設定された値の場合、スーパーユーザでもpg_read_all_settingsのメンバーでもないユーザから検査された時はNULLです。)
pending_restart bool 値が設定ファイル内で変更されたが再起動が必要という場合はtrue、それ以外の場合はfalse。

contextが取り得る値は複数あります。この設定の変更の困難さを軽くするために、以下に示します。

internal

これらの設定は直接変更できません。これらは内部で決定された値を反映するものです。一部は異なる設定オプションでサーバを再構築する、または、initdbに与えるオプションを変更することで調整することができます。

postmaster

これらの設定はサーバ起動時にのみ適用することができます。このため何かを変更するためにはサーバを再起動しなければなりません。これらの設定用の値は通常postgresql.confファイル内に格納されている、あるいは、サーバを起動する際のコマンドラインから渡されます。当然ながら、より低い種類のcontextを持つ設定もサーバ起動時に設定することができます。

sighup

これらの設定は、サーバを再起動することなくpostgresql.conf内を変更することで行うことができます。postgresql.confを再度読み込み、変更を適用させるためには、postmasterにSIGHUPシグナルを送信してください。すべての子プロセスが新しい値を選択するように、postmasterは同時に子プロセスにSIGHUPシグナルを転送します。

superuser-backend

これらの設定は、サーバを再起動することなくpostgresql.conf内を変更することで行うことができます。また、接続要求パケットの中で特定のセッション向けに設定することもできます(例えばlibpqのPGOPTIONS環境変数)が、これは接続ユーザがスーパーユーザの場合に限られます。しかし、これらの設定はセッションが開始してから、そのセッションの中で変更することはできません。postgresql.conf内でそれらを変更した場合は、postgresql.confを再度読み込ませるために、postmasterにSIGHUPシグナルを送信してください。新しい値はその後で始まったセッションにのみ影響を与えます。

backend

これらの設定は、サーバを再起動することなくpostgresql.conf内を変更することで行うことができます。また、接続要求パケットの中で特定のセッション向けに設定することもできます（例えばlibpqのPGOPTIONS環境変数）。どのユーザでも、自分のセッション向けにそのような変更をすることができます。しかし、これらの設定はセッションが開始してから、そのセッションの中で変更することはできません。postgresql.conf内でそれらを変更した場合は、postgresql.confを再度読み込ませるために、postmasterにSIGHUPシグナルを送信してください。新しい値はその後で始まったセッションにのみ影響を与えます。

superuser

これらの設定はpostgresql.conf、または、セッションの中でSETコマンドを使用することで設定することができます。しかしSET経由で変更できるのはスーパーユーザのみです。postgresql.conf内の変更は、セッション独自の値がSETで設定されていない場合にのみ、既存のセッションに影響を与えます。

user

これらの設定はpostgresql.conf、または、セッションの中でSETコマンドを使用することで設定することができます。任意のユーザが自身のセッション独自の値を変更することが許されています。postgresql.conf内の変更は、セッション独自の値がSETで設定されていない場合にのみ、既存のセッションに影響を与えます。

これらのパラメータを変更する各種方法に関する情報については19.1を参照してください。

pg_settingsビューには挿入も削除もできませんが、更新することは可能です。pg_settings行に適用されるUPDATEは名前付きのパラメータに対してSETコマンドを実行することと同値です。変更は現在のセッションで使用されている値にのみ有効です。もしも後に中止されるトランザクション内でUPDATEが発行されると、トランザクションがロールバックされた時点でUPDATEコマンドは効力を失います。排他制御中のトランザクションがひとたびコミットされると、その効果は他のUPDATEもしくはSETコマンドで上書きされない限りセッションの完了まで保たれます。

51.86. pg_shadow

pg_shadowビューは下位互換のために存在しています。バージョン8.1以前のPostgreSQLに存在していたカタログを模擬します。pg_authid内でrolcanloginのマークがついた全てのロールの属性を保持します。

名前の由来は、このテーブルがパスワードを含むため、一般的には読めないことから来ています。pg_userは、pg_shadowのビューですが、パスワードの列が空白となっているため一般に読むことが可能です。

表51.87 pg_shadowの列

列 型 説明
usename name（参照先 pg_authid.rolname） ユーザ名
usesysid oid（参照先 pg_authid.oid） ユーザID

列 型 説明
usecreatedb bool ユーザはデータベースを作成可能です。
usesuper bool ユーザはスーパーユーザです。
userepl bool ユーザはストリーミングレプリケーションを開始することができ、システムをバックアップモードにしたり、戻したりできます。
usebypassrls bool ユーザはすべての行単位セキュリティポリシーを無視します。詳しくは5.8を参照してください。
passwd text (おそらく暗号化された)パスワード。存在しない場合はNULLです。暗号化されたパスワードの格納方法についてはpg_authidを参照してください。
valuntil timestamptz パスワード有効期限(パスワード認証でのみ使用)
useconfig text[] 実行時設定変数のセッションデフォルト

51.87. pg_shmem_allocations

pg_shmem_allocationsビューは、サーバの主共有メモリセグメントによるメモリの獲得状況を表示します。これはpostgres自身が獲得したメモリと、37.10.10で詳細を説明している機構を使って拡張が獲得したメモリの両方が含まれます。

このビューは動的共有メモリ基盤を使って獲得したメモリは含まれないことに注意してください。

表51.88 pg_shmem_allocationsの列

列 型 説明
name text 共有メモリ獲得の名前です。NULLなら未使用のメモリで、無名の獲得なら<anonymous>です。
off int8 この獲得が開始する位置です。NULLなら無名の獲得あるいは未使用のメモリです。
size int8 獲得サイズです。
allocated_size int8 パディングを含む獲得サイズです。無名の獲得では、パディングに関する情報はありません。ですからsizeとallocated_sizeは常に同じです。パディングは未使用メモリでは意味がありません。ですからそのような列でも同じになります。

無名の獲得は、ShmemInitStruct()あるいはShmemInitHash()ではなく、ShmemAlloc()で直接行われたものです。

デフォルトではpg_shmem_allocationsはスーパユーザだけが読み取りできます。

51.88. pg_stats

pg_statsビューはpg_statisticカタログの情報にアクセスするためのビューです。このビューは、ユーザが読み込み権限を持つテーブルに一致するpg_statisticの行に対してのみアクセスを許可しています。よって、このビューに対して一般に読み込みを許可しても安全です。

pg_statsも、その基礎となっているカタログよりも、より読みやすい書式で情報を提供するように設計されています。しかし、これは、もしpg_statisticに対して新しいスロット型が定義されるたびに、スキーマが拡張されなくてはならない、という犠牲を払っています。

表51.89 pg_statsの列

列 型 説明
schemaname name（参照先 pg_namespace.nspname） テーブルがあるスキーマの名前
tablename name（参照先 pg_class.relname） テーブルの名前
attname name（参照先 pg_attribute.attname） この行が記述する列名
inherited bool 真の場合、この行には指定されたテーブルの値だけではなく、継承関係の子の列が含まれます。
null_frac float4 NULLとなっている列項目の割合
avg_width int4 列項目のバイト単位による平均幅
n_distinct float4 ゼロより大きい値は列内の個別値の推定数です。ゼロより小さければ行数で個別値を割算した数字の負数です。（テーブルが肥大するにつれ個別値の増大があり得るとANALYZEが判断した場合に負変換形式が使われます。正変換形式は列の取り得る値が固定数を持つと思われる場合に使用されます）。例えば-1は個別値の数が行数と等しいような、一意な列を表します。
most_common_vals anyarray 列の中の最も共通した値のリストです（他の値よりもより共通している値がない場合はNULLです）。
most_common_freqs float4[] 最も一般的な値の出現頻度のリストで、つまり行の総数で出現数を割算した数字です（most_common_valsがNULLの時はNULLです）。
histogram_bounds anyarray 列の値を満遍なく似たような数でグループに分配した値のリストです。most_common_valsの値がもし存在すればこの度数分布計算は行われません。（列データ型が<演算子を所有しない場合、もしくはmost_common_valsが全体の構成要素アカウントをリストしている場合、この列はNULLです）。
correlation float4 物理的な[訳注：ディスク上の]行の並び順と論理的な列の値の並び順に関する統計的相関です。この値は-1から+1の範囲です。値が-1もしくは+1の近辺にある時、ディスクにランダムアクセスする必要が少なくなるためこの列に対してのインデッ

列 型 説明
クススキャンは0近辺にある場合に比較して安価であると推定されます。(列データ型に<演算子がない場合、この列はNULLです)。
most_common_elems anyarray 列の値の中で最もよく出現する非NULLの要素値のリストです。(スカラ型の場合はNULLです。)
most_common_elem_freqs float4[] 最も一般的な要素値の出現頻度のリストで、与えられた値の少なくとも1つのインスタンスを含む行の断片です。2つもしくは3つの追加の値が1つの要素ごとの出現頻度に続きます。最小で最大の要素ごとの出現頻度があります。さらにオプションとしてNULL要素の出現頻度もあります。(most_common_elemsがNULLの時はNULLです。)
elem_count_histogram float4[] 列の値でNULLではない要素値の個別数のヒストグラム。これは個別のNULLではない平均値が後に続きます。(スカラ型の場合はNULLです。)

配列の最大項目数はALTER TABLE SET STATISTICSコマンドで列ごとに設定されるか、もしくはdefault_statistics_target実行時パラメータで包括的に設定されるかのいずれかです。

51.89. pg_stats_ext

pg_stats_extはpg_statistic_extとpg_statistic_ext_dataカタログに格納されている情報へのアクセスを提供します。このビューは、ユーザが読み込み権限を持つテーブルに一致するpg_statistic_extとpg_statistic_ext_dataの行に対してのみアクセスを許可しています。よって、このビューに対して一般に読み込みを許可しても安全です。

pg_stats_extも、その基礎となっているカタログよりも、より読みやすい書式で情報を提供するように設計されています。しかし、これはpg_statistic_extに対して新しいスロット型が定義されるたびに、スキーマが拡張されなくてはならない、という犠牲を払っています。

表51.90 pg_stats_extの列

列 型 説明
schemaname name (参照先 pg_namespace.nspname) テーブルがあるスキーマの名前
tablename name (参照先 pg_class.relname) テーブルの名前
statistics_schemaname name (参照先 pg_namespace.nspname) 拡張統計情報を含むスキーマの名前
statistics_name name (参照先 pg_statistic_ext.stxname) 拡張統計情報の名前
statistics_owner name (参照先 pg_authid.rolname) 拡張統計情報所有者
attnames name[] (参照先 pg_attribute.attname) 拡張統計情報が定義された列名

列 型 説明
kinds char[] このレコードに対して有効になった拡張統計情報の型
n_distinct pg_ndistinct 列値の組み合わせに対するN個別統計カウント。ゼロよりも大きければ、その組み合わせに対する個別の値の数の見積で、ゼロよりも小さければ、個別の値の数の見積を符号反転し行数で割ったものです。(負の値の形式は、ANALYZEがテーブルが大きくなるにつれ個別の値の数も大きくなると判断した場合に使用されます。正の値の形式は、可能な値の数が定まった数になると思われる時に使用されます。) たとえば-1は、列のユニークな組み合わせに対し異なる組み合わせの数が行数と同じであることを示しています。
dependencies pg_dependencies 関数従属性統計情報
most_common_vals text[] 列における最も共通した値の組み合わせのリスト。(他の組み合わせよりも共通した組み合わせが見つからない場合はNULL)
most_common_val_nulls bool[] 最も共通した値の組み合わせに対するNULLフラグのリスト。(most_common_valsがNULLならNULL。)
most_common_freqs float8[] 最も共通した組み合わせの発生頻度のリスト。つまり、発生数を合計行数で割ったもの。(most_common_valsがNULLならNULL。)
most_common_base_freqs float8[] 最も共通した組み合わせの発生頻度の基底のリスト。つまり値ごとの頻度の積。(most_common_valsがNULLの時はNULLです)。

ALTER TABLE SET STATISTICSコマンドを使って配列フィールドの最大のエントリ数を列単位に制御できます。あるいはdefault_statistics_target実行時パラメータを設定して一括で制御できます。

51.90. pg_tables

pg_tablesビューはデータベース内のそれぞれのテーブルに関する有用な情報へのアクセスを提供します。

表51.91 pg_tablesの列

列 型 説明
schemaname name (参照先 pg_namespace.nspname) テーブルがあるスキーマの名前
tablename name (参照先 pg_class.relname) テーブルの名前
tableowner name (参照先 pg_authid.rolname) テーブルの所有者
tablespace name (参照先 pg_tablespace.spcname) テーブルを含むテーブル空間の名前(データベースのデフォルトの場合はNULL)

列 型 説明
hasindexes bool (参照先 pg_class.relhasindex) テーブルがインデックスを持っている(もしくは最近まで持っていた)なら真
hasrules bool (参照先 pg_class.relhasrules) テーブルにルールがある(もしくは以前あった)時は真
hastriggers bool (参照先 pg_class.relhastriggers) テーブルにトリガがある(もしくは以前あった)時は真
rowsecurity bool (参照先 pg_class.relrowsecurity) テーブルの行セキュリティが有効なら真

51.91. pg_timezone_abbrevs

pg_timezone_abbrevsビューは、現在日付時間の入力処理で認識されている、時間帯省略形のリストを提供します。このビューの内容は、timezone_abbreviations実行時パラメータが変更された時に変わります。

表51.92 pg_timezone_abbrevsの列

列 型 説明
abbrev text 時間帯省略形
utc_offset interval UTCからのオフセット(正はグリニッジより西側を意味する)
is_dst bool 夏時間省略形の場合は真

多くのタイムゾーンの省略形は、UTCからの固定されたオフセットで表現されている一方で、いくつかのものは歴史的にオフセット値が変化しています(詳細はB.4を参照してください)。このような場合には、それらの現在の意味を表示します。

51.92. pg_timezone_names

pg_timezone_namesは、SET TIMEZONEで認識される時間帯名称の一覧を提供します。ここには、その関連付けされた省略形、UTCオフセット、夏時間状況などが含まれます。(PostgreSQLは技術的には、うるう秒を扱いませんので、UTCを使用しません。) pg_timezone_abbrevsで示した省略形とは異なり、名前の多くが夏時間変換規則を意味しています。したがって、関連する情報はローカルなDST境界によって異なります。表示される情報は、現在のCURRENT_TIMESTAMPに基づいて計算されたものです。

表51.93 pg_timezone_namesの列

列 型 説明
name text

列 型 説明
時間帯名
abbrev text 時間帯省略形
utc_offset interval UTCからのオフセット(正はグリニッジより西側を意味する)
is_dst bool 現在夏時間である場合に真

51.93. pg_user

pg_userビューはデータベースユーザに関する情報へのアクセスを提供します。これはパスワードフィールドを隠蔽したpg_shadowを公に読めるようにしたビューです。

表51.94 pg_userの列

列 型 説明
usename name ユーザ名
usesysid oid ユーザID
usecreatedb bool ユーザはデータベースを作成可能です。
usesuper bool ユーザはスーパーユーザです。
userepl bool ユーザはストリーミングレプリケーションを開始することができ、システムをバックアップモードにしたり、戻したりできます。
usebypassrls bool ユーザはすべての行単位セキュリティポリシーを無視します。詳しくは5.8を参照してください。
passwd text パスワードでありません(常に********のように読まれます)
valuntil timestamptz パスワード有効期限(パスワード認証でのみ使用)
useconfig text[] 実行時設定変数のセッションデフォルト

51.94. pg_user_mappings

pg_user_mappingsビューはユーザマッピングについての情報へのアクセスを提供します。これはユーザが使用する権利を持っていないオプションフィールドを取り除いた、基本的には公開されていて読み取り可能なpg_user_mappingのビューです。

表51.95 pg_user_mappingsの列

列 型 説明
umid oid（参照先 pg_user_mapping.oid） ユーザマッピングのOID
srvid oid（参照先 pg_foreign_server.oid） マッピングを保持する外部サーバのOID
srvname name（参照先 pg_foreign_server.srvname） 外部サーバの名前
umuser oid（参照先 pg_authid.oid） マッピングされているローカルのロールのOID。ユーザマッピングが公開されている場合は0になります。
usename name マッピングされているローカルユーザの名前
umoptions text[] 「keyword=value」文字列のようなユーザマッピングの特定のオプション

ユーザマッピングオプションとして格納されたパスワード情報を保護するために、umoptions列は以下に該当しない限りはnullとして読み込みます。

- 現在のユーザはマッピングされているユーザであり、サーバを所有しているか、サーバ上にUSAGE権限を持っている
- 現在のユーザはサーバ所有者であり、PUBLICとしてマッピングされている
- 現在のユーザはスーパーユーザである

51.95. pg_views

pg_viewsビューはデータベース内のそれぞれのビューに関する有用な情報へのアクセスを提供します。

表51.96 pg_viewsの列

列 型 説明
schemaname name（参照先 pg_namespace.nspname） ビューを持つスキーマ名
viewname name（参照先 pg_class.relname） ビュー名
viewowner name（参照先 pg_authid.rolname） ビューの所有者

列 型 説明
definition text ビュー定義(再構築されたSELECT問い合わせ)

第52章 フロントエンド/バックエンドプロトコル

PostgreSQLはフロントエンドとバックエンド（クライアントとサーバ）の通信にメッセージベースのプロトコルを使用します。このプロトコルはTCP/IPに加え、Unixドメインソケットをサポートします。ポート番号5432は、このプロトコルをサポートするサーバ用のTCPポートとしてIANAに登録されています。しかし、実際には任意の非特権ポート番号を使用することができます。

この文書はPostgreSQL 7.4以降で実装されたプロトコル3.0バージョンについて記載します。以前のプロトコルバージョンについての説明は、PostgreSQLの以前のリリースの文書を参照してください。単一のサーバが複数のプロトコルバージョンをサポートできます。初めの開始要求メッセージは、サーバに対し、クライアントが使用する予定のプロトコルバージョンを通知します。クライアントが要求したメジャーバージョンをサーバがサポートしない場合は、接続は拒否されます。（これはたとえば、クライアントが本稿執筆時点で存在しないプロトコルバージョン4.0を要求した際に起るでしょう。）クライアントが要求したマイナーバージョンをサーバがサポートしない場合は（たとえばクライアントがバージョン3.1を要求し、サーバが3.0しかサポートしていないようなときです）、サーバは接続を拒否しても良いですし、サポートする最も大きなマイナープロトコルバージョンを含むNegotiateProtocolVersionメッセージを返しても構いません。クライアントは、指定されたプロトコルバージョンで接続を継続するか、あるいは接続を切断するかのどちらかを選択できます。

複数のクライアントにサービスを効率的に提供するために、サーバは各クライアント毎に新規の「バックエンド」プロセスを起動します。現在の実装では、サーバに接続が届いたことを検知すると即座に新しい子プロセスが生成されます。しかし、これはプロトコルに対して透過的です。プロトコルという意味では、「バックエンド」と「サーバ」という用語は相互交換可能です。同様に「フロントエンド」と「クライアント」も相互交換可能です。

52.1. 概要

このプロトコルでは、接続開始と通常操作で段階が分かれています。接続開始段階で、フロントエンドはサーバへの接続を開き、サーバの義務を履行できるよう自身を証明します。（これは使用する認証方法に応じて、単一のメッセージになったり、複数のメッセージになったりします。）すべてうまく行けば、サーバはフロントエンドに状態情報を送信し、最終的に通常操作段階に入ります。初期の開始要求メッセージを除いて、プロトコルのこの部分はサーバによって駆動されます。

通常操作の間、フロントエンドは問い合わせやその他のコマンドをバックエンドに送信し、バックエンドは問い合わせ結果やその他の応答を返送します。（NOTIFYのように）バックエンドから依頼されずにメッセージが送信されるまれな場合がありますが、セッションのこの部分のほとんどはフロントエンドの要求によって駆動されます。

セッションの終了は通常フロントエンドが選択することですが、特定の場合はバックエンドによって強制される可能性があります。どちらの場合でも、バックエンドが接続を閉ざす時、終了前に実行中の（未完の）トランザクションをすべてロールバックします。

通常操作中は、SQLコマンドを2つのサブプロトコルのうちのいずれかによって実行することができます。「簡易問い合わせ」プロトコルでは、フロントエンドはテキストで問い合わせ文字列を単に送信し、バックエンドによって解析され、即実行されます。「拡張問い合わせ」プロトコルでは、問い合わせの処理は、解析、パラ

メータ値の結び付け、そして実行という複数の段階に分離されます。これは複雑性が加わりますが、柔軟性と性能という点で利点が生まれます。

通常操作には、さらに、COPYのような特殊な操作向けのサブプロトコルがあります。

52.1.1. メッセージ処理の概要

すべての通信はメッセージストリームを介します。メッセージの先頭バイトはメッセージ種類を識別するもの、次の4バイトはメッセージの残りの長さを指定するものです（この長さにはメッセージ種類バイトは含まれませんが、自身を含んで数えられます）。残りのメッセージの内容は、メッセージ種類で決まります。歴史的な理由のため、一番初めにクライアントから送信されるメッセージ（開始メッセージ）にはメッセージ種類バイトはありません。

メッセージストリームの同期ずれを防ぐために、サーバとクライアントの両方は、通常、メッセージの内容を処理し始める前に、（バイト数を使用して）メッセージ全体をバッファ内に読み込みます。これにより、その内容を処理する時にエラーが検出された場合に、簡単に復旧することができます。（メッセージをバッファするために十分なメモリがない場合のような）極限状況では、受信側はメッセージの読み取りを再開する前にどれだけの量の入力を飛ばすかどうかを決定するために、バイト数を活用することができます。

反対に、サーバとクライアントの両方は、不完全なメッセージを決して送信しないように注意しなければなりません。これは通常、送信する前にバッファ内のメッセージ全体を整列させることで行われます。メッセージの送受信の途中で通信エラーが発生した場合、メッセージ境界の同期を復旧できる望みはほとんどありませんので、実用的な唯一の応答は通信を中断することです。

52.1.2. 拡張問い合わせの概要

拡張問い合わせプロトコルでは、SQLコマンドの実行は複数の段階に分割されます。段階間で保持される状態は、プリペアド文とポータルの2種類のオブジェクトで表現されます。プリペアド文は、テキスト問い合わせ文字列の解析、意味解析を表現します。プリペアド文は実行準備が整ったことを示すものではありません。パラメータの特定の値が欠落しているかもしれないからです。ポータルは、すべてのパラメータ値が設定され、実行準備が整った、あるいは、既に一部実行された文を表現します。（SELECT文では、ポータルは開いているカーソルと等価です。しかし、カーソルはSELECT以外の文を扱いませんので、ここでは異なる用語を使用するよう選択しました。）

実行サイクル全体は、テキストの問い合わせ文字列からプリペアド文を生成する解析段階、プリペアド文と必要なパラメータ値によりポータルを作成するバインド段階、ポータルの問い合わせを実行する実行段階からなります。行を返す問い合わせ（SELECT、SHOWなど）の場合、操作を完了させるために複数の実行段階が必要とすることができるように、実行段階に限定された行数のみを取り出すよう指示することができます。

バックエンドは複数のプリペアド文とポータルの経過を追うことができます（しかし、1つのセッション内でのみ存在可能です。複数のセッションで共有することはできません）。存在するプリペアド文とポータルは、作成時に割り当てられた名前で参照されます。さらに、「無名の」プリペアド文とポータルも存在します。これらは名前付きのオブジェクトとほとんど同じ動きをしますが、問い合わせを一回だけ実行し、その後に破棄する場合に備えて、これらに対する操作は最適化されています。一方、名前付きオブジェクトの操作は複数回の使用を想定して最適化されています。

52.1.3. 書式と書式コード

特定のデータ型のデータはいくつかの異なる*書式*で転送することができます。PostgreSQL 7.4の時点でサポートしている書式は「テキスト」と「バイナリ」のみですが、プロトコルは将来の拡張に備えて準備をしています。任意の値の必要な書式は*書式コード*で指定されます。クライアントは、転送されるパラメータ値それぞれに書式コードを指定することも、問い合わせ結果の列それぞれに書式コードを指定することもできます。テキストは書式コード0、バイナリは書式コード1です。他の書式コードは将来の定義用に予約されています。

値のテキスト表現は、特定のデータ型の入出力変換関数で生成され、受け付けられる何らかの文字列です。転送時の表現では、ヌル終端文字はありません。フロントエンドでC言語文字列として処理したい場合は、必ず受信した値にヌル終端文字を追加しなければなりません。(ついでですが、テキスト書式ではヌルを埋め込むことはできません。)

整数用のバイナリ表現はネットワークバイト順(先頭に最上位バイト)を使用します。他のデータ型のバイナリ表現については、文書もしくはソースコードを参照してください。複雑なデータ型のバイナリ表現はサーバのバージョンによって異なる可能性があることに注意してください。通常、テキスト書式がより移植性が高い選択肢です。

52.2. メッセージの流れ

本節では、メッセージの流れと各メッセージ種類のセマンティクスを説明します。(各メッセージの正確な表現の詳細については52.7で説明します。) 開始、問い合わせ、関数呼び出し、COPY、終了といった接続状態に応じて、複数の異なるサブプロトコルがあります。また、開始段階の後の任意の時点で発生する可能性がある、非同期操作(通知応答やコマンドのキャンセルを含む)用の特別な準備もあります。

52.2.1. 開始

セッションを開始するために、フロントエンドはサーバへの接続を開き、開始メッセージを送信します。このメッセージには、ユーザ名と接続を希望するデータベース名が含まれます。これはまた、使用する特定のプロトコルバージョンを識別します。(オプションとして、開始メッセージに、実行時パラメータの追加設定を含めることもできます。) サーバはその後、この情報と設定ファイル(pg_hba.confなど)の内容を使用して、接続が暫定的に受け付けられるかどうか、そして(もしあれば)どのような追加認証が必要かを決定します。

サーバはその後、適切な認証要求メッセージを送信します。フロントエンドはこれに適切な認証応答メッセージ(パスワードなど)で答えなければなりません。GSSAPI、SSPI、SASLを除くすべての認証方式では、多くても1つの要求と1つの応答が存在します。認証方式の中には、フロントエンドからの応答をまったく必要としないものもあり、その場合、認証要求も発生しません。GSSAPI、SSPI、SASLでは認証を完了するために複数のパケットの交換が必要となるかもしれません。

認証サイクルは、サーバによって接続要求を拒絶する(ErrorResponse)か、あるいはAuthenticationOkを送信することで終わります。

この段階でサーバから送信される可能性があるメッセージを以下に示します。

ErrorResponse

接続試行が拒絶されました。サーバはその後即座に接続を閉ざします。

AuthenticationOk

認証情報の交換が正常に完了しました。

AuthenticationKerberosV5

フロントエンドはここでサーバとのKerberos V5認証ダイアログ(ここでは説明しません。Kerberos仕様の一部)に参加する必要があります。これが成功すれば、サーバはAuthenticationOk応答を行います。失敗すれば、ErrorResponse応答を行います。これはもはやサポートされていません。

AuthenticationCleartextPassword

フロントエンドはここで平文形式のパスワードを含むPasswordMessageを送信する必要があります。これが正しいパスワードであればサーバはAuthenticationOk応答を行います。さもなくば、ErrorResponse応答を行います。

AuthenticationMD5Password

フロントエンドはここでMD5で暗号化したパスワード(とユーザ名)を再度AuthenticationMD5Passwordメッセージで指定されたランダムな4バイトのソルトを使用して暗号化したものを含むPasswordMessageを送信する必要があります。これが正しいパスワードであればサーバはAuthenticationOk応答を行います。さもなくば、ErrorResponse応答を行います。実際のPasswordMessageを`concat('md5', md5(concat(md5(concat(password, username)), random-salt)))`というSQLで計算することができます。(`md5()`関数がその結果を16進数表記で返すことに注意してください。)

AuthenticationSCMCredential

この応答はSCM資格証明メッセージをサポートするプラットフォーム上のローカルなUnixドメイン接続でのみあり得ます。フロントエンドはSCM資格証明メッセージを発行し、その後単一のデータバイトを送信する必要があります。(データバイトの内容には意味はありません。これはサーバが資格証明メッセージの受信にどれだけ待機すればよいのかを確実にするためだけに使用されます。) 資格証明が受け付け可能であれば、サーバはAuthenticationOk応答を行います。さもなくば、ErrorResponse応答を行います。(この種類のメッセージは9.1より前のサーバでのみ発行されます。最終的にはプロトコル仕様から削除されるかもしれません。)

AuthenticationGSS

ここでフロントエンドはGSSAPIの調停を開始しなければなりません。これに対する応答におけるGSSAPIデータストリームの最初の部分で、フロントエンドはGSSResponseを送信します。さらにメッセージが必要となる場合、サーバはAuthenticationGSSContinueで応答します。

AuthenticationSSPI

ここでフロントエンドはSSPI調停を開始しなければなりません。これに対する応答におけるSSPIデータストリームの最初の部分で、フロントエンドはGSSResponseを送信します。さらにメッセージが必要となる場合、サーバはAuthenticationGSSContinueで応答します。

AuthenticationGSSContinue

このメッセージには、GSSAPIまたはSSPI調停の直前の段階(AuthenticationGSS、AuthenticationSSPIまたは前回のAuthenticationGSSContinue)についての応答データが含まれます。このメッセージ内のGSSAPIまたはSSPIデータが認証を完結させるため、更なる追加データが必要

であることを指示している場合、フロントエンドは他のGSSResponseとしてデータを送信しなければなりません。このメッセージでGSSAPIまたはSSPI認証が完了すれば、次にサーバはAuthenticationOkを送信して認証が成功したことを示すか、あるいはErrorResponseを送信して失敗したことを示します。

AuthenticationSASL

ここでフロントエンドはメッセージ内に列挙されているSASL機構の1つを使ってSASL調停を開始しなければなりません。これに応答するSASLデータストリームの最初の部分で、フロントエンドはSASLInitialResponseと選択した機構の名前を送信します。さらにメッセージが必要な場合、サーバはAuthenticationSASLContinueで応答します。詳細については52.3を参照してください。

AuthenticationSASLContinue

このメッセージには、SASL調停における直前の段階（AuthenticationSASLまたは以前のAuthenticationSASLContinue）のチャレンジデータが含まれます。フロントエンドはSASLResponseメッセージで応答しなければなりません。

AuthenticationSASLFinal

機構固有のクライアント用の追加データを伴ってSASL認証が完了します。サーバは次に認証成功を示すAuthenticationOkを送信するか、あるいは失敗を示すErrorResponseを送信します。このメッセージはSASLの機構が完了時にサーバからクライアントに送信する追加データを指定しているときにのみ送信されます。

NegotiateProtocolVersion

サーバはクライアントが要求したマイナープロトコルバージョンをサポートしませんが、それ以前のバージョンをサポートします。このメッセージは、サポートしている最も高いマイナーバージョンを示します。このメッセージは、クライアントがサポートされないプロトコルオプション（つまり_pq_.で始まる）をスタートアップパケットの中で指定した場合にも送られます。このメッセージの後に、ErrorResponseか、認証が成功あるいは失敗したことを示すメッセージが続きます。

サーバが要求した認証方式をフロントエンドがサポートしていない場合、フロントエンドは即座に接続を閉ざします。

AuthenticationOkを受け取った後、フロントエンドはさらにサーバからのメッセージを待機する必要があります。この段階で、バックエンドプロセスが起動し、このフロントエンドは単なる関心を有する第三者となります。開始試行が失敗（ErrorResponse）するか、サーバが要求されたマイナープロトコルバージョンを拒否する（NegotiateProtocolVersion）可能性がまだありますが、通常、バックエンドは何らかのParameterStatusメッセージ、BackendKeyData、そして最後にReadyForQueryを送信します。

この段階の期間中、バックエンドは開始メッセージで与えられた任意の実行時パラメータの追加設定を適用しようとします。成功した場合は、これらの値はセッションのデフォルトになります。エラーが発生した場合はErrorResponseを行い、終了します。

この段階でバックエンドから送信される可能性があるメッセージを以下に示します。

BackendKeyData

このメッセージは、フロントエンドがキャンセル要求を後で送信できるようにしたい場合に保存しなければならない、秘密キーデータを用意します。フロントエンドはこのメッセージに応答してはいけませんが、ReadyForQueryメッセージの監視を続けなくてはなりません。

ParameterStatus

このメッセージは、フロントエンドに現在（初期）のclient_encodingやDateStyleなどのバックエンドパラメータの設定情報を通知します。フロントエンドはこのメッセージを無視しても、将来の使用に備えてその設定を記録しても構いません。詳細は52.2.6を参照してください。フロントエンドはこのメッセージに応答してはいけませんが、ReadyForQueryメッセージの監視を続けなくてはなりません。

ReadyForQuery

開始処理が完了しました。フロントエンドはここでコマンドを発行することができます。

ErrorResponse

開始処理が失敗しました。接続はこのメッセージの送信後に閉ざされます。

NoticeResponse

警告メッセージが発行されました。フロントエンドはこのメッセージを表示し、ReadyForQueryもしくはErrorResponseメッセージの監視を続けなければなりません。

ReadyForQueryメッセージは各コマンドサイクルの後にバックエンドが発行するものと同じものです。フロントエンドのコーディングにおいて必要であれば、ReadyForQueryをコマンドサイクルの開始とみなしても構いませんし、ReadyForQueryを開始段階とその後の各コマンドサイクルの終端とみなしても構いません。

52.2.2. 簡易問い合わせ

フロントエンドがQueryメッセージをバックエンドに送信することで、簡易問い合わせサイクルが開始されます。このメッセージには、テキスト文字列で表現されたSQLコマンド（またはコマンド）が含まれます。そうすると、バックエンドは、問い合わせコマンド文字列の内容に応じて1つ以上の応答を送信し、最終的にReadyForQueryを応答します。ReadyForQueryは、新しいコマンドを安全に送信できることをフロントエンドに知らせます。（実際には、フロントエンドが他のコマンドを発行する前にReadyForQueryを待機することは不要です。しかし、フロントエンドは、前のコマンドが失敗し、発行済みの後のコマンドが成功した場合に何が起きるかを了解する責任を持たなければなりません。）

バックエンドから送信される可能性がある応答メッセージを以下に示します。

CommandComplete

SQLコマンドが正常に終了しました。

CopyInResponse

バックエンドがフロントエンドからのデータをテーブルにコピーする準備ができました。52.2.5を参照してください。

CopyOutResponse

バックエンドがデータをテーブルからフロントエンドにコピーする準備ができました。52.2.5を参照してください。

RowDescription

SELECTやFETCHなどの問い合わせの応答の行がまさに返されようとしていることを示します。このメッセージには、行の列レイアウトに関する説明が含まれます。このメッセージの後に、フロントエンドに返される各行に対するDataRowメッセージが続きます。

DataRow

SELECTやFETCHなどの問い合わせで返される行の集合の1つです。

EmptyQueryResponse

空の問い合わせ文字列が検知されました。

ErrorResponse

エラーが起こりました。

ReadyForQuery

問い合わせ文字列の処理が終了しました。問い合わせ文字列は複数のSQLコマンドが含まれる場合があるため、このことを通知するために分離したメッセージが送出されます。(CommandCompleteは文字列全体ではなく、1つのSQLコマンドの処理の終了を明らかにします。) 処理が成功またはエラーで終了したかどうかにかかわらずReadyForQueryは常に送出されます。

NoticeResponse

問い合わせに関して警告メッセージが発行されました。警告メッセージは他の応答に対する追加のメッセージです。したがって、バックエンドはそのコマンドの処理を続行します。

SELECT問い合わせ(あるいは、EXPLAINやSHOWなどの行集合を返す他の問い合わせ)に対する応答は、通常、RowDescription、0個以上のDataRowメッセージ、そしてその後のCommandCompleteから構成されます。フロントエンドへのCOPYもしくはフロントエンドからのCOPYは52.2.5で説明する特別なプロトコルを呼び出します。他の種類の問い合わせは通常CommandCompleteメッセージのみを生成します。

問い合わせ文字列には(セミコロンで区切られた)複数の問い合わせが含まれることがありますので、バックエンドが問い合わせ文字列の処理を完了する前に、こうした応答シーケンスが複数発生する可能性があります。ReadyForQueryは、文字列全体が処理され、バックエンドが新しい問い合わせ文字列を受け付ける準備が整った時点で発行されます。

完全に空の(空白文字以外の文字がない)問い合わせ文字列を受け取った場合、その応答は、EmptyQueryResponse、続いて、ReadyForQueryとなります。

エラーが発生した場合、ErrorResponse、続いて、ReadyForQueryが発行されます。その問い合わせ文字列に対する以降の処理は(複数の問い合わせが残っていたとしても)すべて、ErrorResponseによって中断されます。これは、個々の問い合わせで生成されるメッセージの並びの途中で発生する可能性があることに注意してください。

簡易問い合わせモードでは、読み出される値の書式は常にテキストです。ただし、与えられたコマンドがBINARYオプション付きで宣言されたカーソルからのFETCHであった場合は例外です。この場合は、読み出される値はバイナリ書式になります。RowDescriptionメッセージ内で与えられる書式コードは、どの書式が使用されているかを通知します。

他の種類のメッセージの受信を待機している時、フロントエンドは常にErrorResponseとNoticeResponseメッセージを受け取る準備ができていなければなりません。また、外部イベントのためにバックエンドが生成する可能性があるメッセージの扱いについては52.2.6を参照してください。

メッセージの正しい並びを前提としてコーディングするのではなく、任意のメッセージ種類を、そのメッセージが意味を持つ任意の時点で受け付ける状態マシン形式でフロントエンドのコーディングを行うことを推奨します。

52.2.2.1. 簡易問い合わせでの複文

簡易Queryメッセージが二つ以上の(セミコロンで区切られた)SQL文を含むとき、振る舞いを変えるように明示的なトランザクション制御コマンドが含まれていない限り、これらの文は単一トランザクションで実行されます。例えばメッセージが以下を含む場合、

```
INSERT INTO mytable VALUES(1);
SELECT 1/0;
INSERT INTO mytable VALUES(2);
```

SELECTでのゼロ除算エラーは最初のINSERTのロールバックを強制します。さらに、メッセージの実行が最初のエラー時点で中止されるので、二番目のINSERTは全く試みられません。

代わりにメッセージが以下を含んでいる場合、

```
BEGIN;
INSERT INTO mytable VALUES(1);
COMMIT;
INSERT INTO mytable VALUES(2);
SELECT 1/0;
```

最初のINSERTは明示的なCOMMITコマンドによりコミットされます。二番目のINSERTとSELECTは、やはり単一トランザクションとして処理されます。そのためゼロ除算エラーが二番目のINSERTをロールバックしますが、最初のINSERTはロールバックされません。

この振る舞いは、暗黙トランザクションブロックで複文Queryメッセージ内の文を実行することで、その中に明示的なトランザクションブロックがある場合を除き、発現します。暗黙トランザクションブロックと通常のトラザクションブロックとの主な違いは、暗黙ブロックは自動的にQueryメッセージの最後にて、エラーが無いなら暗黙のコミット、エラーがあるなら暗黙のロールバックで、自動的に閉じられることです。これは(トランザクションブロック内に無いときの)文の単体実行に対して生じる暗黙のコミットあるいはロールバックに似ています。

何らか手前のメッセージでのBEGINの結果として、セッションが既にトランザクションブロック内である場合、Queryメッセージは、含まれるのが単一文でもいくつかの文でも、単にそのトランザクションを継続します。しかしながら、Queryメッセージが既存トランザクションブロックを閉じるCOMMITやROLLBACKを含む場合、続く全ての文は暗黙トランザクションブロックで実行されます。逆に言えば、複文QueryメッセージでBEGINが現れたなら、このQueryメッセージ内または後のメッセージのいずれかにあらわれる明示的なCOMMITやROLLBACKでのみ終了する、通常のトランザクションブロックが開始されます。BEGINが暗黙トランザクションブロックとして実行されたいくつかの文に続く場合、これらの文が直ちにコミットされることはありません。結果として、これらは遡って新たな通常のトランザクションブロックに含められます。

暗黙トランザクションブロック内に現れたCOMMITやROLLBACKは通常通り実行され、暗黙ブロックを閉じますが、手前のBEGIN無しのCOMMITやROLLBACKは誤りであるかもしれないので警告が発行されます。さらに文が続く場合、それらに対して新たな暗黙トランザクションブロックが開始されます。

エラー時の自動ブロッククローズの振る舞いと競合するので暗黙トランザクションブロックでセーブポイントは使えません。

現状のいかなるトランザクション制御コマンドでも、Queryメッセージの実行は最初のエラー時点で打ち切られることに留意してください。例を示します。

```
BEGIN;
SELECT 1/0;
ROLLBACK;
```

上記が単一Queryメッセージにあるとして、ゼロ除算エラーの後にROLLBACKに達することがないため、このセッションは失敗した通常のトランザクション内のままとなります。このセッションを通常の状態に回復させるには別のROLLBACKが必要となります。

その他の注意すべき振る舞いは、初期の字句および構文解析が少しも実行されない段階で問い合わせ文字列全体に対して行われることです。従って、後ろの分にある（スペルミスしたキーワードなどの）単純なエラーは全ての文の実行を妨げることがあります。暗黙トランザクションブロックとして起きたとき、いずれにせよ全ての文はロールバックされるので、これは通常はユーザに見えません。しかしながら、複文問い合わせの中で複数のトランザクションを実行しようとするとき、この挙動が明らかになることがあります。例えば、タイプミスで先の例を以下のようにします。

```
BEGIN;
INSERT INTO mytable VALUES(1);
COMMIT;
INSERT INTO mytable VALUES(2);
SELCT 1/0;
```

そうすると、含まれる文は一つも実行されず、最初のINSERTがコミットされないという違いが明らかになります。エラーは、ミススペルしたテーブルやカラム名など、語彙の解析かその後に検出され、コマンドの効力はありません。

52.2.3. 拡張問い合わせ

拡張問い合わせプロトコルは、上述の簡易問い合わせプロトコルを複数段階に分解します。予備段階の結果は複数回再利用できますので、効率が上がります。さらに、問い合わせ文字列に直接埋め込むのではなく、データ値をパラメータとして分離して提供できる機能など、利用できる追加機能があります。

拡張プロトコルでは、フロントエンドはまず、テキストの問い合わせ文字列とオプションとしてパラメータプレースホルダのデータ型情報やプリペアド文のオブジェクトの宛先名（空文字列は無名のプリペアド文を選択）を含む、Parseメッセージを送信します。この応答はParseCompleteまたはErrorResponseです。パラメータデータ型はそのOIDで指定することができます。指定がなければ、パーサは型指定のないリテラル文字列定数に対する方法と同じ方法でデータ型を推定します。

> **注記**
>
> パラメータデータ型をゼロと設定する、または、問い合わせ文字列内で使用されているパラメータ記号($n)の数より短くパラメータ型のOIDの配列を作成することで、指定しないまま残すことができます。他にも、パラメータの型をvoid(つまりvoid仮想型のOID)と指定するという特別な場合があります。これは、パラメータ記号を、実際のOUTパラメータである関数パラメータとして使用することができることを意味します。通常では、voidパラメータが使用される文脈はありませんが、関数パラメータリストにこうしたパラメータ記号があると、実質的には無視されます。例えば、$3と$4がvoid型を持つと指定された場合、foo($1,$2,$3,$4)といった関数呼び出しは、2つのINと2つのOUT引数を持つ関数に一致します。

> **注記**
>
> Parseメッセージ内の問い合わせ文字列には、複数のSQL文を含めることはできません。さもないと、構文エラーが報告されます。この制限は簡易問い合わせプロトコルにはありませんが、複数のコマンドを持つプリペアド文やポータルを許すと、プロトコルが複雑になり過ぎるため、拡張プロトコルではこの制限があります。

作成に成功すると、名前付きプリペアド文オブジェクトは明示的に破棄されない限り現在のセッションが終わるまで残ります。無名のプリペアド文オブジェクトは、次に無名のプリペアド文を宛先に指定したParse文が発行されるまでの間のみに残ります。(単なるQueryメッセージでも無名のプリペアド文オブジェクトは破壊されることに注意してください。) 名前付きプリペアド文は、他のParseメッセージで再定義する前に明示的に閉じなければなりません。しかし、これは無名のプリペアド文では必要ありません。名前付きプリペアド文はまた、SQLコマンドレベルでPREPAREとEXECUTEを使用して作成しアクセスすることができます。

プリペアド文が存在すると、Bindメッセージを使用してそれを実行可能状態にすることができます。Bindメッセージは、元となるプリペアド文(空文字列は無名のプリペアド文を表します)の名前、宛先となるポータル(空文字列は無名ポータルを表します)の名前、およびプリペアド文内のパラメータプレースホルダに使用する値を与えます。与えられたパラメータ集合はプリペアド文で必要とするものと一致しなければなりません。(Parseメッセージ内でvoidパラメータを1つでも宣言した場合、Bindメッセージではそれらには NULLを渡します。) また、Bindは問い合わせで返されるデータに使用する書式を指定します。書式は全体に対して指定することも、列単位で指定することも可能です。応答はBindCompleteもしくはErrorResponseです。

> **注記**
>
> テキスト出力とバイナリ出力との選択は、含まれるSQLコマンドに関係なく、Bindで与えられた書式コードで決定されます。拡張問い合わせプロトコルを使用する場合、カーソル宣言のBINARY属性は役に立ちません。

典型的に問い合わせ計画はBindメッセージが処理される時に作成されます。プリペアド文がパラメータを持たない場合、または繰り返し実行される場合、サーバは作成した計画を保管し、その後の同じプリペアド文に対するBindメッセージの際に再利用する可能性があります。しかし、作成できる汎用的な計画が提供された特定のパラメータ値に依存する計画より効率が大して劣化しないことが分かった場合のみ、このように動作します。プロトコルに注目している限り、これは透過的に行われます。

作成に成功すると、名前付きポータルオブジェクトは明示的に破棄されない限り現在のセッションが終わるまで残ります。無名ポータルは、トランザクションの終わり、もしくは、次に無名ポータルを宛先に指定したBind文が発行されるまでの間のみに残ります。(単なるQueryメッセージでも無名ポータルは破壊されることに注意してください。) 名前付きポータルは、他のBindメッセージで再定義する前に明示的に閉じなければなりません。しかし、これは無名ポータルでは必要ありません。名前付きポータルはまた、SQLコマンドレベルでDECLARE CURSORとFETCHを使用して作成しアクセスすることができます。

ポータルが存在すると、Executeメッセージを使用してそれを実行することができます。Executeメッセージは、ポータル名(空文字列は無名ポータルを表します)と結果行数の最大値(ゼロは「fetch all rows」を意味します)を指定します。結果行数は、ポータルが行集合を返すコマンドを含む場合のみ意味があります。その他の場合では、コマンドは常に終わりまで実行され、行数は無視されます。Executeで起こり得る応答は、ExecuteではReadyForQueryやRowDescriptionが発行されない点を除き、上述の簡易問い合わせプロトコル経由で発行された問い合わせの場合と同じです。

Executeがポータルの実行を完了する前に(非ゼロの結果行数に達したために)終了した場合、PortalSuspendedを送信します。このメッセージの出現は、フロントエンドに操作を完了させるためには同一のポータルに対して、別のExecuteを発行しなければならないことを通知します。元となったSQLコマンドが完了したことを示すCommandCompleteメッセージはポータルが完了するまで送信されません。したがって、Execute段階は常にCommandComplete、EmptyQueryResponse(空の問い合わせ文字列からポータルが作成された場合)、ErrorResponse、またはPortalSuspendedの中の、正確にどれかが出現することによって常に終了します。

拡張問い合わせメッセージの一連の流れのそれぞれの終了時、フロントエンドはSyncメッセージを発行しなければなりません。このパラメータのないメッセージにより、もしBEGIN/COMMITトランザクションブロックの内部でなければ、バックエンドは現在のトランザクションを閉ざします(「閉ざす」とは、エラーがなければコミット、エラーがあればロールバックすることを意味します)。そして、ReadyForQuery応答が発行されます。Syncの目的は、エラーからの復旧用の再同期を行う時点を提供することです。拡張問い合わせメッセージの処理中にエラーが検出されると、バックエンドはErrorResponseを発行し、Syncが届くまでメッセージを読み、それを破棄します。その後、ReadyForQueryを発行し、通常のメッセージ処理に戻ります。(しかし、Sync処理中にエラーが検出された場合に処理が飛ばされないことに注意してください。これにより、各Syncに対してReadyForQueryが1つのみであることを保証します。)

> **注記**
>
> Syncによって、BEGINで開かれたトランザクションブロックが閉ざされることはありません。ReadyForQueryメッセージにはトランザクションの状態情報が含まれていますので、この状況を検出することができます。

これらの基本的な必要操作に加え、拡張問い合わせプロトコルで使用することができる、複数の省略可能な操作があります。

Describeメッセージ(ポータルの亜種)は、既存のポータルの名前(もしくは、無名ポータル用の空文字)を指定します。応答は、実行中のポータルで返される予定の行を記述するRowDescriptionメッセージです。ポータルが行を返す問い合わせを含まない場合はNoDataメッセージです。指定したポータルが存在しない場合はErrorResponseです。

Describeメッセージ(ステートメントの亜種)は、既存のプリペアド文の名前(もしくは無名のプリペアド文用の空文字)を指定します。応答は、文で必要とされるパラメータを記述するParameterDescriptionメッセー

ジ、続いて、文が実行された場合に返される予定の行を記述するRowDescriptionメッセージ（もしくは文が行を返さない場合のNoDataメッセージ）です。指定したプリペアド文が存在しない場合はErrorResponseが発行されます。Bindがまだ発行されていませんので、返される列の書式はまだバックエンドでは不明であることに注意してください。RowDescriptionメッセージ内の書式コードフィールドはこの場合はゼロになります。

> **ヒント**
>
> ほとんどの状況では、フロントエンドはExecuteを発行する前に、返ってくる結果を解釈する方法を確実に判断できるように、Describeもしくはその亜種を実行すべきです。

Closeメッセージは、既存のプリペアド文、もしくはポータルを閉ざし、リソースを解放します。存在しない文やポータルに対してCloseを発行してもエラーになりません。応答は通常CloseCompleteですが、リソースの解放に何らかの問題が発生した場合はErrorResponseになります。プリペアド文を閉じると、そこから構築され、開いたポータルが暗黙的に閉ざされることに注意してください。

Flushメッセージにより特定の出力が生成されることはありません。しかし、バックエンドに対して、出力バッファ内で待機しているデータを強制的に配送させます。フロントエンドが他のコマンドを発行する前にコマンドの結果を検証したい場合に、FlushはSync以外の拡張問い合わせコマンドの後に送信される必要があります。Flushを行わないと、バックエンドで返されるメッセージは、ネットワークオーバーヘッドを最小化する、最小限のパケット数にまとめられます。

> **注記**
>
> 簡易問い合わせメッセージは、おおよそ、無名のプリペアド文とポータルオブジェクトを使用したパラメータなしのParse、Bind、ポータル用Describe、Execute、Close、Syncの流れと同一です。違いは、問い合わせ文字列内に複数のSQL文を受け付けられ、bind/describe/executeという流れがそれぞれが成功すれば自動的に行われる点です。他の違いとして、ParseCompleteやBindComplete、CloseComplete、NoDataメッセージが返されない点があります。

52.2.4. 関数呼び出し

関数呼び出しサブプロトコルにより、クライアントはデータベースのpg_procシステムカタログに存在する任意の関数を直接呼び出す要求を行うことができます。クライアントはその関数を実行する権限を持たなければなりません。

> **注記**
>
> 関数呼び出しサブプロトコルは、おそらく新しく作成するコードでは使用すべきではない古い機能です。同様の結果は、`SELECT function($1, ...)`を行うプリペアド文を設定することで得ることができます。そして、この関数呼び出しサイクルをBind/Executeで置き換えることができます。

関数呼び出しサイクルはフロントエンドがFunctionCallメッセージをバックエンドに送ることで起動されます。バックエンドは1つまたは複数の応答メッセージを関数呼び出しの結果に基づいて送り、最終的にReadyForQueryメッセージを送出します。ReadyForQueryはフロントエンドに対し新規の問い合わせまたは関数呼び出しを行っても安全確実であることを通知します。

バックエンドから送信される可能性がある応答メッセージを以下に示します。

ErrorResponse

エラーが起こりました。

FunctionCallResponse

関数呼び出しが完了し、メッセージで与えられた結果が返されました。(関数呼び出しプロトコルは単一のスカラ結果のみを扱うことができます。行型や結果集合を扱うことはできません。)

ReadyForQuery

関数呼び出しの処理が終了しました。処理が成功またはエラーで終了したかどうかにかかわらずReadyForQueryは常に送出されます。

NoticeResponse

関数呼び出しに関して警告メッセージが出されました。警告メッセージは他の応答に対する追加のメッセージです。したがって、バックエンドはそのコマンドの処理を続行します。

52.2.5. COPY操作

COPYコマンドにより、サーバとの間で高速な大量データ転送を行うことができます。コピーインとコピーアウト操作はそれぞれ接続を別のサブプロトコルに切り替えます。これは操作が完了するまで残ります。

コピーインモード(サーバへのデータ転送)は、バックエンドがCOPY FROM STDIN SQL文を実行した時に起動されます。バックエンドはフロントエンドにCopyInResponseメッセージを送信します。フロントエンドはその後、ゼロ個以上のCopyDataメッセージを送信し、入力データのストリームを形成します。(このメッセージの境界は行の境界に何かしら合わせる必要ありませんが、往々にしてそれが合理的な選択となります。) フロントエンドは、CopyDoneメッセージ(正常に終了させる)、もしくは、CopyFailメッセージ(COPY SQL文をエラーで失敗させる)を送信することで、コピーインモードを終了させることができます。そして、バックエンドは、COPYが始まる前の、簡易もしくは拡張問い合わせプロトコルを使用するコマンド処理モードに戻ります。そして次に、CommandComplete(成功時)またはErrorResponse(失敗時)のどちらかを送信します。

コピーインモードの期間中にバックエンドがエラー(CopyFailメッセージの受信を含む)を検知すると、バックエンドはErrorResponseメッセージを発行します。拡張問い合わせメッセージ経由でCOPYコマンドが発行された場合、バックエンドはSyncメッセージを受け取るまでフロントエンドのメッセージを破棄するようになります。Syncメッセージを受け取ると、ReadyForQueryを発行し、通常処理に戻ります。簡易問い合わせメッセージでCOPYが発行された場合、メッセージの残りは破棄され、ReadyForQueryメッセージを発行します。どちらの場合でも、その後にフロントエンドによって発行されたCopyData、CopyDone、CopyFailは単に削除されます。

バックエンドは、コピーインモード期間中、FlushとSyncメッセージを無視します。その他の種類の非コピーメッセージを受け取ると、エラーになり、上述のようにコピーイン状態を中断します (クライアントライブラリがExecuteメッセージの後に、実行したコマンドがCOPY FROM STDINかどうかを検査することなく、常にFlushまたはSyncを送信できる、という便利さのためにFlushとSyncは例外です)。

コピーアウトモード(サーバからのデータ転送)は、バックエンドがCOPY TO STDOUT SQL文を実行した時に起動します。バックエンドはCopyOutResponseメッセージをフロントエンドに送信し、その後、ゼロ個以上のCopyDataメッセージ(常に行ごとに1つ)を、そして、CopyDoneを送信します。その後、バックエンド

はCOPYが始まる前のコマンド処理モードに戻り、CommandCompleteを送信します。フロントエンドは（接続の切断やCancel要求の発行は例外ですが）転送を中断することはできません。しかし、不要なCopyDataとCopyDoneメッセージを破棄することは可能です。

コピーアウトモード期間中バックエンドはエラーを検知すると、ErrorResponseメッセージを発行し、通常処理に戻ります。フロントエンドはErrorResponseの受信をコピーアウトモードの終了として扱うべきです。

NoticeResponseおよびParameterStatusメッセージがCopyDataメッセージ間に散在することがあります。フロントエンドはこのような場合も扱わなければなりません。また、他の種類の非同期メッセージ（52.2.6を参照）も同様に準備すべきです。さもなくば、CopyDataまたはCopyDone以外の種類のメッセージがコピーアウトモードの終了として扱われてしまう可能性があります。

他にも、サーバへ、およびサーバからの高速な一括データ転送を行うことができるコピーボースというコピーに関連したモードがあります。コピーボースモードは、walsenderモードのバックエンドがSTART_REPLICATION文を実行した時に初期化されます。バックエンドはCopyBothResponseメッセージをフロントエンドに送信します。この後バックエンドとフロントエンドの両方が、接続が終了するまでの間にCopyDataメッセージを送信できるようになります。同様に、サーバがCopyDoneメッセージを送信した場合、接続はコピーインモードとなり、サーバはそれ以上のCopyDataメッセージを送信できません。両方の側がCopyDoneメッセージを送信した後、コピーモードは終了し、バックエンドはコマンド処理モードに戻ります。コピーボースモード中にバックエンドが検出したエラーのイベントにおいては、バックエンドはErrorResponseメッセージを発行し、Syncメッセージの受信までフロントエンドのメッセージを破棄し、その後ReadyForQueryを発行して通常の処理に戻ります。フロントエンドは両方向のコピーを終了するように、ErrorResponse受理の処理をするべきです。この場合CopyDoneを送信するべきではありません。コピーボースモードにおけるサブプロトコル転送の詳細は52.4を参照してください。

CopyInResponse、CopyOutResponse、CopyBothResponseメッセージには、フロントエンドに1行当たりの列数と各列で使用する書式コードを通知するためのフィールドが含まれています。（現在の実装では、COPY操作で与えられるすべての列は同一の書式を使用します。しかし、メッセージ設計においては、これを前提としていません。）

52.2.6. 非同期操作

バックエンドが、フロントエンドのコマンドストリームで特に依頼されていないメッセージを送信する場合が複数あります。フロントエンドは、問い合わせ作業を行っていない時であっても常に、これらのメッセージを扱う準備をしなければなりません。少なくとも、問い合わせの応答の読み込みを始める前にこれらを検査すべきです。

外部の活動によって、NoticeResponseメッセージが生成されることがあり得ます。例えば、データベース管理者が「高速」データベース停止コマンドを実行した場合、バックエンドは接続を閉ざす前にこれを通知するためにNoticeResponseを送信します。したがって、たとえ接続が名目上待機状態であったとしても、フロントエンドは常にNoticeResponseメッセージを受け付け、表示する準備をすべきです。

ParameterStatusメッセージは、任意のパラメータの実際の値が変更され、それをバックエンドがフロントエンドに通知するべきであるとみなした場合は常に生成されます。ほとんどの場合、これはフロントエンドによるSET SQLコマンド実行に対する応答の中で起こります。これは実質的には同期していますが、管理者が設定ファイルを変更し、SIGHUPシグナルをサーバに送った場合にも、パラメータ状態の変更が発生することがあります。また、SETコマンドがロールバックされた場合、現在の有効値を報告するために適切なParameterStatusメッセージが生成されます。

現時点では、ParameterStatusを生成するパラメータ群は固定されています。それらはserver_version、server_encoding、client_encoding、application_name、is_superuser、session_authorization、DateStyle、IntervalStyle、TimeZone、integer_datetimes、およびstandard_conforming_stringsです。(8.0より前までのリリースでは、server_encoding、TimeZoneおよびinteger_datetimesは送信されませんでした。8.1より前までのリリースでは、standard_conforming_stringsは送信されませんでした。8.4より前のリリースでは、IntervalStyleは送信されませんでした。9.0より前のリリースでは、application_nameは送信されませんでした。) server_version、server_encodingおよびinteger_datetimesは仮想的なパラメータであり、起動後に変更することができないことに注意してください。これは今後変更される、あるいは設定変更可能になる可能性があります。したがって、フロントエンドは未知または注目していないParameterStatusを単に無視すべきです。

フロントエンドがLISTENコマンドを発行した場合、同じチャネル名に対しNOTIFYコマンドが実行された時にバックエンドはNotificationResponseメッセージ(NoticeResponseと間違えないように!)を送出します。

> **注記**
>
> 現在、NotificationResponseをトランザクションの外部でのみ送信することができます。このため、これはコマンド応答の流れの途中では起こりませんが、ReadyForQueryの直前に発生する可能性があります。しかし、これを前提にフロントエンドのロジックを設計することは避けてください。プロトコル内の任意の時点でNotificationResponseを受け付けられるようにすることを勧めます。

52.2.7. 処理中のリクエストの取り消し

問い合わせの処理中に、フロントエンドが問い合わせを取り消す可能性があります。取り消し要求は、効率を高めるために、接続を開いたバックエンドに対して直接送信されません。その問い合わせを処理中のバックエンドが、フロントエンドからの新しい入力があるかどうかを定期的に確認することは好ましくありません。取り消し要求はたいていの場合、頻繁には起こらないので、通常の状態においての負担を避けるため、多少扱いにくくなっています。

取り消し要求を出す場合、フロントエンドは通常の新規接続の時に送出されるStartupMessageメッセージではなくCancelRequestメッセージをサーバに送り、新規接続を開始します。サーバはこの要求を処理し、接続を切断します。セキュリティ上の理由から、取り消し要求メッセージに対し直接の回答はありません。

CancelRequestメッセージは、接続開始段階でフロントエンドに送られたものと同一の鍵データ(PIDと秘密鍵)を含んでいない場合は無視されます。現在バックエンドが実行中の処理に対するPIDと秘密鍵が要求と一致した場合、その現在の問い合わせ処理は中断されます。(現状では、これは、その問い合わせを処理しているバックエンドプロセスに対し特別なシグナルを送ることで実装されています。)

この取り消しシグナルは何の効果も生まないことがあります。例えば、バックエンドが問い合わせの処理を完了した後に届いた場合、効果がありません。もし取り消し処理が有効であれば、エラーメッセージとともに、現在のコマンドは終了されます。

セキュリティと効率上の理由による上述の実装の結果、フロントエンドは取り消し要求が成功したかどうかを直接判断することはできません。フロントエンドはバックエンドからの問い合わせの回答を待ち続けなければいけません。取り消しを要求することは単に現在の問い合わせを早めに終わらせ、成功ではなくエラーメッセージを出して不成功とする可能性を単に高める程度のものです。

取り消し要求は、通常のフロントエンドとバックエンドの通信接続を通してではなく新規のサーバとの接続に送られるため、取り消される問い合わせを実行したフロントエンドだけでなく、任意のプロセスによっても送信することができます。このことはマルチプロセスアプリケーションを作るに当たって柔軟性を提供します。同時に、権限のない者が問い合わせを取り消そうとするといったセキュリティ上のリスクも持ち込みます。このセキュリティ上のリスクは、取り消し要求内に動的に生成される秘密キーを供給することを必須とすることで回避されます。

52.2.8. 終了

通常の洗練された終了手順はフロントエンドがTerminateメッセージを出し、すぐに接続を閉じることです。このメッセージを受け取るとすぐにバックエンドは接続を閉じ終了します。

まれに(管理者によるデータベース停止コマンドなど)、バックエンドはフロントエンドからの要求なしに切断することがあります。こうした場合、バックエンドは、接続を閉ざす前に切断理由を通知するエラーまたは警報メッセージの送信を試みます。

他にも、どちらかの側のコアダンプ、通信リンクの消失、メッセージ境界の同期の消失など各種失敗によって終了する状況があります。フロントエンドかバックエンドいずれかが予期しない接続の中断を検知した場合、後始末を行い終了しなければいけません。フロントエンドはもし自身が終了を望まない場合、サーバに再交信し新規のバックエンドを立ち上げる選択権を持っています。解釈できないメッセージ種類を受け取った時、おそらくメッセージ境界の同期が消失したことを示しますので、接続を閉ざすことを勧めます。

通常の終了、異常な終了のどちらの場合でも、開いているトランザクションはすべてコミットされずにロールバックされます。しかし、フロントエンドがSELECT以外の問い合わせを処理中に切断した場合、バックエンドはおそらく切断に気付く前にその問い合わせを完了させてしまうでしょう。その問い合わせがトランザクションブロック(BEGIN ... COMMITの並び)外部であった場合、切断に気付く前にその結果はコミットされる可能性があります。

52.2.9. SSLセッション暗号化

PostgreSQLがSSLサポート付きで構築された場合、フロントエンドとバックエンド間の通信をSSLで暗号化することができます。攻撃者がセッショントラフィックをキャプチャできるような環境における通信を安全にすることができます。SSLを使用したPostgreSQLセッションの暗号化に関する詳細は18.9を参照してください。

SSL暗号化接続を開始するには、フロントエンドはまず、StartupMessageではなくSSLRequestを送信します。その後サーバはそれぞれSSLの実行を行うか行わないかを示すSかNかを持つ1バイトの応答を返します。フロントエンドはその応答に満足できなければ、この時点で接続を切断することができます。Sの後に継続するのであれば、サーバと間でSSL起動ハンドシェーク(ここではSSLの仕様に関しては説明しません)を行います。これが成功した場合、続いて通常のStartupMessageの送信を行います。この場合、StartupMessageおよびその後のデータはSSLにより暗号化されます。Nの後に、通常のStartupMessageを送信することで暗号化なしで進みます。

また、フロントエンドはサーバからのSSLRequestに対するErrorMessage応答を取り扱うための用意もすべきです。これは、PostgreSQLにSSLサポートが追加される前のサーバの場合のみで発生します。(現在ではこうしたサーバは非常に古いものといえ、ほとんど存在しません。) この場合接続を切断しなければなりませんが、フロントエンドはSSL要求なしで新しく接続を開き、処理を進めることもできます。

最初のSSLRequestはCancelRequestメッセージを送信するために開いた接続で使用することもできます。

プロトコル自体には、サーバにSSL暗号化を強制する方法は用意されていませんが、管理者は認証検査の一方法として、暗号化されていないセッションを拒否するようにサーバを設定することができます。

52.2.10. GSSAPIセッション暗号化

PostgreSQLがGSSAPIサポートを有効にして構築されていれば、GSSAPIを使ってフロントエンド/バックエンド通信を暗号化できます。これにより、攻撃者がセッションのやり取りを読み取れるかもしれない環境における通信のセキュリティが提供されます。PostgreSQLでの通信をGSSAPIで暗号化するための情報に関しては、18.10をご覧ください。

GSSAPI暗号化接続を開始するには、フロントエンドは最初にStartupMessageではなく、GSSENCRequestメッセージを送ります。次にサーバは、それぞれGSSAPI暗号化を希望する、しないを意味するGまたはNを含む1バイトを送信します。このレスポンスに満足できなければ、この時点でフロントエンドは接続を打ち切って構いません。Gの受信後継続するには、RFC2744あるいは同様の文書で議論されているGSSAPI Cバインディングを使い、ループの中でgss_init_sec_context()を呼び出してGSSAPIを初期化し、結果をサーバに送信し、空の入力を受け取ることから始めて、サーバが出力を返さなくなるまでサーバからの出力を受け取ります。gss_init_sec_context()の結果をサーバに送る際には、ネットワークバイトオーダーで4バイトの整数にメッセージ長を先頭に付与します。これに成功したら、gss_wrap()を使って通常のStartupMessageと後続のすべてのメッセージを暗号化します。実際に暗号化した送信データの前に、gss_wrap()の結果をネットワークバイトオーダーで4バイトの整数にしたものを付与します。サーバは16kB未満のクライアントからの暗号化パケットだけを受け付けることに注意してください。クライアントはgss_wrap_size_limit()を使って暗号化前のメッセージの大きさがこの制限に収まるかどうかを確認し、それより大きなメッセージは複数のgss_wrap()呼び出しに分解すべきです。典型的なセグメントは暗号化前で8kBのデータで、暗号化後のパケットは8kBより少し大きくなりますが、最大長の16kB以内には問題なく収まります。サーバは16kBよりも大きな暗号化パケットをクライアントに送らないものと期待することができます。Nの後継続するには、通常のStartupMessageを送信し、暗号化せずに続けてください。

また、フロントエンドはサーバからのGSSENCRequestへのErrorMessage応答に備えるべきです。これはサーバがPostgreSQLへのGSSAPI暗号化サポートを追加する以前だったときにのみ発生します。この場合は接続を切断しなければなりませんが、フロントエンドは新しい接続を開いてGSSAPI暗号化を要求せずに進めることを選択するかもしれません。上述の長さ制限により、ErrorMessageがサーバからの適切な長さを持つ正しい応答と混同されることはありません。

最初のGSSENCRequestは、CancelRequestメッセージを送信するために開いている接続でも利用できます。

プロトコル自身はサーバにGSSAPI暗号化を強制する方法を提供していませんが、管理者は認証チェックの副次的効果として暗号化されていないセッションをサーバが拒否するように設定できます。

52.3. SASL認証

*SASL*は接続指向のプロトコルでの認証のためのフレームワークです。現時点ではPostgreSQLは2つのSASLの認証機構、SCRAM-SHA-256とSCRAM-SHA-256-PLUSを実装しています。将来には他の機構が追加されるかもしれません。以下の手順は、SASLの認証が一般的にどのように行われるかを示したものですが、次の副節ではSCRAM-SHA-256とSCRAM-SHA-256-PLUSにおけるより詳細について説明します。

SASL認証のメッセージフロー

1. SASL認証の交換を開始するため、サーバはAuthenticationSASLメッセージを送信します。これにはサーバが受け付けることができるSASLの認証機構を、サーバにとって望ましいものから順に並べたリストが含まれます。

2. クライアントはリストからサポートされる機構を1つ選択し、サーバにSASLInitialResponseメッセージを送信します。このメッセージには選択された機構の名前が含まれ、また選択した機構がInitial Client Response（最初のクライアントの応答）を使用するなら、オプションでそれも含まれます。

3. サーバのチャレンジメッセージおよびクライアントのレスポンスメッセージが1つ以上続きます。サーバのチャレンジはそれぞれがAuthenticationSASLContinueメッセージで送信され、それにクライアントからのレスポンスがSASLResponseメッセージで続きます。メッセージの詳細は機構に固有のものです。

4. 最後に、認証の交換が成功裏に終了すると、サーバはAuthenticationSASLFinalメッセージを送信し、その直後にAuthenticationOkメッセージを送信します。AuthenticationSASLFinalにはサーバからクライアントへの追加のデータが含まれ、その内容は選択した認証機構毎に異なります。完了時に送信する追加データを認証機構が使用していない場合、AuthenticationSASLFinalメッセージは送信されません。

エラーが発生したときは、サーバは認証を任意の段階で終了してErrorMessageを送信することができます。

52.3.1. SCRAM-SHA-256 認証

今のところ実装されているSASL機構はSCRAM-SHA-256とチャンネルバインディングを伴う変種のSCRAM-SHA-256-PLUSです。詳細はRFC 7677およびRFC 5802で説明されています。

PostgreSQLでSCRAM-SHA-256を使用する場合、クライアントが`client-first-message`で送信するユーザ名をサーバは無視します。その代わりに、開始メッセージで送信済みのユーザ名が使用されます。SCRAMはユーザ名としてUTF-8の使用を指示していますが、PostgreSQLは複数の文字符号化方式をサポートするため、PostgreSQLのユーザ名をUTF-8で表現できないかもしれません。

SCRAMの仕様ではパスワードもUTF-8であり、*SASLprep*アルゴリズムで処理されることが規定されています。しかしPostgreSQLではパスワードにUTF-8が使用されることを必須としていません。ユーザのパスワードが設定されたとき、実際に使用された符号化方式に関わらず、それがUTF-8であるかのようにSASLprepで処理されます。しかし、それが正当なUTF-8バイト列でない場合、あるいはSASLprepが禁止するUTF-8バイト列を含む場合、エラーを発生させるのではなく、SASLprep処理のない生のパスワードが使用されます。これにより、パスワードがUTF-8であればそれを正規化できる一方で、UTF-8以外のパスワードを使用することもでき、またシステムもパスワードがどの符号化であるかを知る必要もありません。

SSLをサポートするPostgreSQLビルドでチャンネルバインディングがサポートされます。チャンネルバインディングを伴うSCRAMに対するSASL機構名はSCRAM-SHA-256-PLUSです。PostgreSQLで使われるチャンネルバインディングのタイプは`tls-server-end-point`です。

チャンネルバインディングを伴わないSCRAMではサーバは、送信されるパスワードハッシュの中でユーザに応じたパスワードと混合してクライアントに送る乱数を選びます。これはパスワードハッシュが後のセッションで再送信されて認証に成功してしまうことを防止しますが、真のサーバとクライアントの間の偽サーバがサーバのランダム値を中継して認証に成功してしまうことを防止しません。

チャンネルバインディングを伴うSCRAMはこのような中間者攻撃をサーバ証明書のシグネチャを送信されるパスワードハッシュと混合することで防止します。偽サーバは真のサーバの証明書を再送信できますが、その証明書に一致する秘密鍵にアクセスできず、それゆえ所有者であることを証明できず、結果としてSSL接続は失敗します。

例

1. サーバはAuthenticationSASLメッセージを送信します。それにはサーバが受け付けることができるSASL認証機構のリストが含まれます。サーバがSSLサポート有でビルドされていれば、これはSCRAM-SHA-256-PLUSとSCRAM-SHA-256になり、そうでなければ後者のみとなります。

2. クライアントはSASLInitialResponseメッセージを送信することで応答します。これは選択した機構、すなわちSCRAM-SHA-256またはSCRAM-SHA-256-PLUSを示します。(クライアントは何れかの機構を自由に選びますが、より良いセキュリティのためサポートされているならチャンネルバインディングを伴うものを選ぶべきです。) Initial Clientの応答フィールドでは、メッセージにSCRAMのclient-first-messageが含まれます。client-first-messageにはクライアントが選んだチャンネルバインディングのタイプも含まれます。

3. サーバがAuthenticationSASLContinueメッセージを送信します。その内容はSCRAMのserver-first-messageです。

4. クライアントがSASLResponseメッセージを送信します。その内容はSCRAMのclient-final-messageです。

5. サーバがSCRAMのserver-final-messageを含むAuthenticationSASLFinalメッセージを送信し、その直後にAuthenticationOkメッセージを送信します。

52.4. ストリーミングレプリケーションプロトコル

ストリーミングレプリケーションを初期化するために、フロントエンドは開始メッセージにてreplicationパラメータを送信します。ブール値のtrue(またはon、yes、1)がバックエンドに対して、SQL文ではなく小規模なレプリケーションコマンド群を発行できるようになる、物理レプリケーションのwalsenderモードに入るように伝えます。

replicationパラメータに対する値としてdatabaseを渡すことは、バックエンドにロジカルレプリケーションのwalsendeモードに入ることを指示します。ロジカルレプリケーションwalsenderモードでは、以下に示すレプリケーションコマンドを通常のSQLコマンドと同様に実行できます。

物理レプリケーション、ロジカルレプリケーションいずれかのwalsenderモードでは、簡易問い合わせプロトコルのみ使用できます。

レプリケーションコマンドをテストするために、replicationオプションを含む接続文字列を使用して、psqlまたは他のlibpqによるレプリケーション接続を作成できます。例を示します。

```
psql "dbname=postgres replication=database" -c "IDENTIFY_SYSTEM;"
```

しかし、物理的レプリケーションのためにpg_receivewalを使用し、論理的レプリケーションのためpg_recvlogicalを使用すれば、もっと有用なことが多いです。

log_replication_commandsが有効であるとき、サーバログにレプリケーションコマンドが記録されます。

レプリケーションモードで受け付けられるコマンドは以下の通りです。

IDENTIFY_SYSTEM

サーバに自身を識別することを要求します。サーバは以下の4つのフィールドを持つ単一行の結果セットをもって応答します。

systemid (text)

クラスタを識別する一意なシステム識別子です。これを使用してスタンバイを初期化するために使用するベースバックアップが同じクラスタに由来していることを検査することができます。

timeline (int4)

現在のタイムラインIDです。同様にスタンバイがマスタと一貫性を持つことを検査するために使用されます。

xlogpos (text)

現在のWALの吐き出し位置です。ストリーミングを開始できる先行書き込みログの既知の位置を得る際に有用です。

dbname (text)

接続したデータベース名またはNULLです。

SHOW name

実行時パラメータの現在の設定を送信するようサーバに要求します。これはSQLコマンドSHOWと同等です。

name

実行時パラメータの名前です。利用できるパラメータは第19章に記述されています。

TIMELINE_HISTORY tli

tliのタイムラインのため、サーバにタイムライン履歴ファイルの送付を要求します。サーバは2列単一行の結果セットを返します。

filename (text)

タイムライン履歴ファイル名、例えば00000002.history

content (bytea)

タイムライン履歴ファイルの内容

CREATE_REPLICATION_SLOT slot_name [TEMPORARY] { PHYSICAL [RESERVE_WAL] | LOGICAL output_plugin [EXPORT_SNAPSHOT | NOEXPORT_SNAPSHOT | USE_SNAPSHOT] }

物理的または論理的レプリケーションスロットを作成します。レプリケーションスロットの詳細は26.2.6を参照。

slot_name

作成するスロット名。有効なレプリケーションスロット名でなければならない。(26.2.6.1を参照)。

output_plugin

ロジカルデコーディングに使用される出力プラグイン名。(48.6を参照)。

TEMPORARY

このレプリケーションスロットが一時スロットであることを指定します。一時スロットはディスクに保存されず、エラー発生時またはセッション終了時に自動で機に削除されます。

RESERVE_WAL

この物理的レプリケーションスロットが直ちにWALを予約することを指定します。そうでなければ、WALはストリーミングレプリケーションクライアントからの接続に応じてだけ予約されます。

EXPORT_SNAPSHOT
NOEXPORT_SNAPSHOT
USE_SNAPSHOT

論理スロットの初期化時に作成されたスナップショットの処理について決定します。デフォルトのEXPORT_SNAPSHOTはスナップショットが他のセッションで利用できるようエクスポートします。このオプションはトランザクションの内側で使用することはできません。USE_SNAPSHOTはこのコマンドを実行している現在のトランザクションでスナップショットを利用します。このオプションはトランザクション内で使用しなければならず、CREATE_REPLICATION_SLOTがそのトランザクション内で実行される最初のコマンドでなければなりません。最後に、NOEXPORT_SNAPSHOTは論理デコーディングで通常通りにスナップショットを使用するだけで、他には何もしません。

このコマンドへの応答として、サーバは以下のフィールドを含む1行の結果集合を送信します。

slot_name (text)

新しく作成されたレプリケーションスロットの名前です。

consistent_point (text)

スロットが一貫性のある状態になった時点のWAL位置です。これが、このスロット上でストリーミングを開始できる最も早い場所となります。

snapshot_name (text)

このコマンドでエクスポートされるスナップショットの識別子です。スナップショットは、この接続上で新しいコマンドが実行されるか、レプリケーション接続が閉じられるまで有効です。作成されたのが物理スロットの場合はNULLになります。

output_plugin (text)

新しく作成されたレプリケーションスロットが使用する出力プラグインの名前です。作成されたのが物理スロットの場合はNULLになります。

START_REPLICATION [SLOT slot_name] [PHYSICAL] XXX/XXX [TIMELINE tli]

サーバに対して、WALのストリーミングをXXX/XXX WAL時点から開始するよう指示します。TIMELINEオプションが指定された場合、ストリーミングはtliのタイムラインから開始されます。そうでなければ、サーバの現在のタイムラインが選択されます。サーバが、例えば、要求されたWALの断片がすでに回収されているなど、エラーを返すことがありえます。成功時サーバはCopyBothResponseメッセージで応答し、フロントエンドに対するWALストリームを開始します。

slot_nameを経由してスロット名が提供された場合、それはレプリケーションの進行として更新されます。それによってサーバは、どのWALセグメントがまだスタンバイに必要か、hot_standby_feedbackのトランザクションはどれか、を感知します。

最新ではなくて、サーバの過去のタイムラインをクライアントが要求した場合、サーバは要求された開始時点から他のタイムラインに切り替えるまでの、全てのWALストリームを送付します。クライアントが旧タイムラインの終点のストリームを要求した場合、サーバはCOPYモードに入らずにCommandCompleteをすぐに応答します。

最新でないタイムラインの全てのWALストリームを送付した後、サーバはCOPYモードを出ることによりストリームを終了します。クライアントもCOPYモードを出ることにより承認した場合、サーバは2列単一行の結果セットを送付し、サーバにある次のタイムラインを示します。最初の列は次のタイムラインID(int8型)であり、次の列は切り替えたWALの位置(text型)です。通常切り替えた位置はWALストリームの終点ですが、昇格する前に再実行されなかった旧タイムラインからWALを送付するというまれな場合もあります。最後に、サーバは2つのCommandCompleteメッセージ(一方はCopyDataを終了し、もう一方はSTART_REPLICATION自体を終了する)を送付し、新規のコマンドを受理できるようになります。

WALデータはCopyDataメッセージ群として送信されます。(これにより他の情報を混在させることができます。具体的にはサーバはストリーム開始後に失敗が起きた場合にErrorResponseメッセージを送信することができます。) サーバからクライアントへの各CopyDataメッセージのペイロード、は以下の書式のどれかを含みます。

XLogData (B)

Byte1('w')

メッセージをWALデータとして識別します。

Int64

このメッセージ内のWALの開始点。

Int64

サーバ上の現在のWAL終了点。

Int64

転送時点でのサーバのシステム時刻。2000年1月1日午前0時からのマイクロ秒。

Byten

WALデータストリームの断片。

単一のWALレコードが2つのXLogDataメッセージに分かれることはありません。しかしWALレコードがWALページ境界を跨る場合、継続レコードを用いてすでに分割されていますので、ページ境界で分割することができます。言い換えると、先頭の主WALレコードとその継続レコードは、別のXLogDataメッセージとして分かれることがありえます。

プライマリキープアライブメッセージ(B)

Byte1('k')

このメッセージを送信元キープアライブとして識別します。

Int64

サーバ上の現在のWAL終端。

Int64

転送時点でのサーバのシステム時刻。2000年1月1日午前0時からのマイクロ秒。

Byte1

タイムアウトによる切断を避けるため、クライアントがこのメッセージに即時に応答するべき方法の1つ。0またはその他

以下のメッセージ書式の1つ（およびCopyDataメッセージのペイロード中のもの）を使用して、受理プロセスは送信者にいつでも応答できます。

スタンバイ状態の更新(F)

Byte1('r')

メッセージを受信側の状態更新として識別します。

Int64

スタンバイにおいて受信しディスクに書き込まれた最終WALバイト+1の場所。

Int64

スタンバイにおいてディスクに吐き出された最終WALバイト+1の場所。

Int64

スタンバイにおいて適用された最終WALバイト+1の場所。

Int64

転送時点でのクライアントのシステム時刻。2000年1月1日午前0時からのマイクロ秒。

Byte1

値が1の場合、このメッセージにすぐ応答するように、クライアントはサーバへ要求します。この方法は、接続がまだ保持されているか検査するために、サーバへのピング送信として使用できます。

ホットスタンバイフィードバックメッセージ(F)

Byte1('h')

メッセージをホットスタンバイのフィードバックメッセージとして識別します。

Int64

転送時点でのクライアントのシステム時刻。2000年1月1日午前0時からのマイクロ秒

Int32

スタンバイの現在のグローバルのxminですが、すべてのレプリケーションスロットのcatalog_xminは除きます。この値と次のcatalog_xminがいずれも0なら、この接続ではホットスタンバイのフィードバックはもう送信されないという通知として扱われます。後でゼロでないメッセージによりフィードバック機構を再開することができます。

Int32

スタンバイのグローバルのxmin xidのエポックです。

Int32

スタンバイのすべてのレプリケーションスロットのcatalog_xminの最小値です。スタンバイ上にcatalog_xminが存在しない、あるいはホットスタンバイのフィードバックが無効化されている場合は0に設定します。

Int32

スタンバイのcatalog_xmin xidのエポックです。

START_REPLICATION SLOT slot_name LOGICAL XXX/XXX [(option_name [option_value] [, ...])]

サーバに対して、XXX/XXXWAL時点から、論理的レプリケーションのWALストリームを開始するよう指示します。例えば、要求されたWALがすでに回収された場合、サーバはエラーを返します。サーバが、例えば、要求されたWALセクションがすでに回収されている場合、エラーを返すことがありえます。成功時サーバはCopyBothResponseメッセージで応答し、フロントエンドに対するWALストリームを開始します。

CopyBothResponse内部のメッセージは、2つのCommandCompleteメッセージを含めてSTART_REPLICATION ... PHYSICALの記述と同じ書式です。

選択されたスロットに関連した出力プラグインは、出力ストリームの処理に使用されます。

SLOT slot_name

ストリームを変更したスロット名。このパラメータは必須であり、LOGICALモードにおいてCREATE_REPLICATION_SLOTによって作成された、実在する論理的レプリケーションスロットに対応しなければなりません。

XXX/XXX

ストリームを開始するWAL時点。

option_name

レプリケーションスロットのロジカルデコーディング出力プラグインに渡すオプション名。

option_value

オプションの値。文字列定数の形式。

DROP_REPLICATION_SLOT slot_name [WAIT]

レプリケーションスロットを削除し、サーバ側で準備した資源を解放します。このスロットが、walsenderが接続しているデータベース以外のデータベースで作成された論理スロットの場合、このコマンドは失敗します。

slot_name

削除するスロット名。

WAIT

このオプションを使用すると、スロットが使用中の時に、スロットの使用が終わるまでコマンドを待機させます。デフォルトの動作ではエラーを発生させます。

BASE_BACKUP [LABEL 'label'] [PROGRESS] [FAST] [WAL] [NOWAIT] [MAX_RATE rate] [TABLESPACE_MAP] [NOVERIFY_CHECKSUMS] [MANIFEST manifest_option] [MANIFEST_CHECKSUMS checksum_algorithm]

サーバにベースバックアップのストリーミングを始めるよう指示します。システムはバックアップが開始される前に自動的にバックアップモードになり、バックアップが完了した時に取り出されます。以下のオプションを受け付けることができます。

LABEL 'label'

バックアップのラベルを設定します。指定がない場合、base backupというバックアップラベルが使用されます。ラベルについての引用符付け規則は、standard_conforming_stringsを有効にした場合の標準SQLの文字列の規則と同じです。

PROGRESS

進行状況の報告を生成するために必要な情報を要求します。これは、ストリームが完了するまでにどのくらいかかるかを計算するために使用することができる、各テーブル空間のヘッダ内の概算容量を返送します。これは、転送を始める前のすべてのファイルサイズを1度数え上げることで計算されます。これ自体が性能に与える悪影響があるかもしれません。特に最初のデータがストリームされるまでにより多くの時間がかかる可能性があります。データベースファイルはバックアップの間変更される可能性がありますので、容量は概算に過ぎず、概算時と実ファイルを送信するまでの間に増減される可能性があります。

FAST

高速チェックポイントを要求します。

WAL

バックアップ内に必要なWALセグメントを含めます。ベースディレクトリtarファイルのpg_walディレクトリにある、バックアップの開始から終了までのすべてのファイルが含まれます。

NOWAIT

デフォルトでは、バックアップは必要な最終WALセグメントがアーカイブされるまで待機します。ログアーカイブが有効でない場合は警告が発せられます。NOWAITにより、必要なログが利用できるようになったことを確認することをクライアント側の責任として、この待機や警告が無効になります。

MAX_RATE *rate*

サーバからクライアントへ転送する単位時間当たりの最大データ容量を制限します（絞ります）。予期される単位はkB/s（キロバイト/秒）です。このオプションが指定された場合、値はゼロまたは32 kB以上1 GB以下でなければなりません。ゼロが渡されるかオプションが指定されない場合、転送の制約は課されません。

TABLESPACE_MAP

ディレクトリpg_tblspcにあるシンボリックリンクに関する情報をtablespace_mapという名前のファイルに含めます。テーブル空間マップファイルには、ディレクトリpg_tblspc/に存在する各シンボリックリンクの名前とそのシンボリックリンクのフルパスが含まれています。

NOVERIFY_CHECKSUMS

デフォルトでは、チェックサムが有効である場合、ベースバックアップ中にチェックサムが検証されます。NOVERIFY_CHECKSUMSを指定すると、この検証を無効にします。

MANIFEST *manifest_option*

このオプションをyesまたはforce-encodeの値で設定すると、バックアップマニフェストが作成され、バックアップとともに送信されます。マニフェストは、含まれる可能性のあるWALファイルを除きバックアップ内に存在するすべてのファイルのリストです。また、サイズ、最終更新時刻、オプションでファイル毎のチェックサムも格納します。force-encodeの値は、全てのファイル名を強制的に16進数でエンコーディングします。それ以外の場合、このタイプのエンコードは、名前が非UTF-8オクテットシーケンスであるファイルに対してのみ実行されます。force-encodeは、主にテスト目的で使用されており、バックアップマニュフェストを読み取るクライアントがこのケースを処理できることを確認するために使用されます。以前のリリースとの互換性を保つため、デフォルトは、MANIFEST 'no'です。

MANIFEST_CHECKSUMS *checksum_algorithm*

バックアップマニュフェストに含まれている各ファイルに適用するチェックサムアルゴリズムを指定します。現在使用可能なアルゴリズムは、NONE、CRC32C、SHA224, SHA256、SHA384、SHA512です。デフォルトはCRC32Cです。

バックアップを開始する時、サーバはまず2つの通常の結果セットを送信し、続けて1つ以上のCopyResponse結果を送信します。

最初の通常の結果セットには、1行2列という形でバックアップの開始位置が含まれます。最初の列にはXLogRecPtr書式の開始位置が、2番目の列には対応するタイムラインIDが含まれます。

2番目の通常の結果セットには各テーブル空間に付き1行を持ちます。この行のフィールドは以下の通りです。

spcoid (oid)

テーブル空間のOIDです。ベースディレクトリの場合はNULLです。

spclocation (text)

テーブル空間ディレクトリのフルパスです。ベースディレクトリの場合はNULLです。

size (int8)

進行状況の報告が要求された場合は、テーブル空間の概算容量です(キロバイト(1024バイト)単位)。要求されていない場合はNULLです。

2番目の通常の結果セットの後、1つ以上のCopyResponse結果が送信されます。主データディレクトリ用に1つ、pg_default、pg_global以外の追加のテーブル空間ごとに1つ送信されます。CopyResponse結果内のデータは、テーブル空間の内容のtar形式(POSIX 1003.1-2008標準で規定された「ustar交換形式」に従う)ダンプです。ただし標準で規定された最後の2つのゼロブロックは省略されています。このtarデータが終わった後、バックアップマニフェストが要求された場合、現在のベースバックアップのマニフェストデータを含む別のCopyResponse結果が送信されます。いずれの場合も、バックアップのWAL終了位置を含む最終の通常の結果セットが、開始位置と同じ形式で送信されます。

データディレクトリと各テーブル空間のtarアーカイブには、そのディレクトリ内のファイルがPostgreSQLファイルかそのディレクトリに追加された他のファイルかに関係なく、すべて含まれます。以下に除かれるファイルを示します。

- postmaster.pid
- postmaster.opts
- pg_internal.init(複数のディレクトリに在ります)
- PostgreSQLサーバの操作中に作成される種々の一時ファイルおよびディレクトリで、pgsql_tmpで始まるすべてのファイルおよびディレクトリ、および一時リレーション。
- ログを取らないリレーション。ただし、リカバリでログを取らないリレーションの再作成に必要なinitフォークは除かれない。
- サブディレクトリを含むpg_wal。バックアップがwalファイルを含めて実行される場合、合成された版のpg_walが含まれます。これにはバックアップが動作するために必要なファイルのみが含まれ、残りの内容は含まれません。
- pg_dynshmem、pg_notify、pg_replslot、pg_serial、pg_snapshots、pg_stat_tmp、pg_subtransは(それがシンボリックリンクであったとしても)空のディレクトリとしてコピーされます。
- シンボリックリンク(上記で列挙したディレクトリは除きます)や特殊デバイスファイルなど、通常のファイルとディレクトリ以外のものは省略されます。(pg_tblspc中のシンボリックリンクは保持されます。)

サーバ上の基盤となるファイルシステムがサポートする場合、所有者、グループ、ファイルのモードが設定されます。

52.5. 論理ストリーミングレプリケーションのプロトコル

この節では論理レプリケーションのプロトコルについて説明します。このプロトコルはレプリケーションコマンドSTART_REPLICATION SLOT slot_name LOGICALで始まるメッセージフローです。

論理ストリーミングレプリケーションのプロトコルは、物理レプリケーションプロトコルの基本要素の上に構築されています。

52.5.1. 論理ストリーミングレプリケーションのパラメータ

論理レプリケーションのSTART_REPLICATIONコマンドは以下のパラメータを受け付けます。

proto_version

プロトコルのバージョンです。現在はバージョン1のみが受け付けられます。

publication_names

サブスクライブする(変更を受け取る)対象となるパブリケーション名をカンマで区切ったリストです。個々のパブリケーション名は標準的なオブジェクト名と扱われ、必要に応じて引用符で括ることができます。

52.5.2. 論理レプリケーションのプロトコルのメッセージ

個々のプロトコルのメッセージについては以降の副節で説明します。個々のメッセージについては52.9で説明されています。

トップレベルのプロトコルのメッセージはすべてメッセージタイプのバイトで始まります。コード内では文字として表現されますが、これは文字符号化のないバイト(符号付き)です。

ストリーミングレプリケーションのプロトコルはメッセージ長を含むため、トップレベルのプロトコルのメッセージはそのヘッダに長さを埋め込む必要がありません。

52.5.3. 論理レプリケーションのプロトコルのメッセージフロー

START_REPLICATIONコマンドと再生進捗のメッセージを除き、すべての情報はバックエンド側からフロントエンド側にのみ流れます。

論理レプリケーションのプロトコルは、個々のトランザクションを一つずつ送信します。これはつまり、BeginとCommitのメッセージの対の間にある全てのメッセージは同じトランザクションに属するということです。

送信されるすべてのトランザクションにはゼロ個以上のDMLメッセージ(Insert、Update、Delete)が含まれます。カスケードの設定がされている場合は、Originメッセージを含めることができます。Originメッセージはトランザクションの起点が別のレプリケーションノードであることを示します。論理レプリケーションのプロトコルという観点では、レプリケーションノードはほぼ何でも良いため、唯一の識別子はOriginの名前です。(必要なら)必要に応じてこれを処理するのは下流側の責任です。Originメッセージは必ずトランザクション内のどのDMLよりも前に送信されます。

すべてのDMLメッセージは任意のリレーションIDを含んでおり、これをRelationメッセージ内のIDと関連付けることができます。Relationメッセージはあるリレーションのスキーマを記述します。あるリレーションについてのRelationメッセージは、現在のセッションでそのリレーションについて初めてDMLメッセージを送信する場合、あるいはRelationメッセージが最後に送信された後でリレーションの定義が変更された場合に送信さ

れます。このプロトコルは、クライアントが必要なだけ多くのリレーションについて、メタデータをキャッシュできることを前提としています。

52.6. メッセージのデータ型

本節ではメッセージの中で使われる基本的なデータ型を説明します。

Intn(i)

ネットワークバイト順（最上位バイトが先頭）におけるnビットの整数。もしiが指定されていれば、それがそのまま使われます。さもなければ変数です。例えばInt16、Int32(42)などです。

Intn[k]

nビット整数の要素数kの配列で、それぞれはネットワークバイト順です。配列サイズkは常にメッセージの前のフィールドで決定されます。例えばInt16[M]です。

String(s)

NULL終端文字列（C様式文字列）。文字列には長さ制限の指定はありません。sが指定されていれば、それがそのまま使われます。さもなければ値は変数です。例えばString、String("user")などです。

> **注記**
>
> バックエンドから返すことができる文字列の長さには事前に定義された制限はありません。フロントエンドではメモリに収まるものはすべて受け入れられるように拡張可能なバッファを使用するコーディング戦略を勧めます。これが実行できないのであれば、文字列全体を読み取り、固定長バッファに合わない後の方の文字を破棄してください。

Byten(c)

厳密にnバイト。フィールド幅nが定数でない場合、メッセージの前のフィールドから決定されます。cが指定されていれば、それがそのまま使われます。例えば Byte2、Byte1('\n')などです。

52.7. メッセージの書式

本節ではそれぞれのメッセージの詳細書式について説明します。それぞれにはフロントエンド（F）、バックエンド（B）あるいは双方（F & B）から送出されることを示す印が付いています。各メッセージには先頭にバイト数を持っていますが、バイト数を参照しなくてもメッセージの終わりを検知できるようにメッセージ書式は定義されています。これは有効性検査を補助します。（CopyDataメッセージはデータストリームの一部を形成しますので例外です。個々のCopyDataメッセージの内容は自身でも解釈することができません。）

AuthenticationOk (B)

Byte1('R')

認証要求としてメッセージを識別します。

Int32(8)

自身を含む、メッセージ内容の長さ（バイト単位）。

Int32(0)

認証が成功したことを指定します。

AuthenticationKerberosV5 (B)

Byte1('R')

メッセージを認証要求として識別します。

Int32(8)

自身を含む、メッセージ内容の長さ（バイト単位）。

Int32(2)

Kerberos V5認証が必要であることを指定します。

AuthenticationCleartextPassword (B)

Byte1('R')

メッセージを認証要求として識別します。

Int32(8)

自身を含む、メッセージ内容の長さ（バイト単位）。

Int32(3)

平文パスワードが必要であることを指定します。

AuthenticationMD5Password (B)

Byte1('R')

メッセージが認証要求であることを識別します。

Int32(12)

自身を含む、メッセージ内容の長さ（バイト単位）。

Int32(5)

MD5暗号化パスワードが必要であることを指定します。

Byte4

パスワード暗号化用ソルトです。

AuthenticationSCMCredential (B)

Byte1('R')

メッセージが認証要求であることを識別します。

Int32(8)

自身を含む、メッセージ内容の長さ（バイト単位）。

Int32(6)

SCM資格証明メッセージが必要であることを指定します。

AuthenticationGSS (B)

Byte1('R')

メッセージが認証要求であることを識別します。

Int32(8)

自身を含む、メッセージ内容の長さ（バイト単位）。

Int32(7)

GSSAPI認証証明メッセージが必要であることを指定します。

AuthenticationSSPI (B)

Byte1('R')

メッセージが認証要求であることを識別します。

Int32(8)

自身を含む、メッセージ内容の長さ（バイト単位）。

Int32(9)

SSPI認証証明メッセージが必要であることを指定します。

AuthenticationGSSContinue (B)

Byte1('R')

メッセージが認証要求であることを識別します。

Int32

自身を含む、メッセージ内容の長さ（バイト単位）。

Int32(8)

このメッセージがGSSAPIまたはSSPIデータを含むことを指定します。

Byten

GSSAPIまたはSSPI認証データです。

AuthenticationSASL (B)

Byte1('R')

メッセージが認証要求であることを識別します。

Int32

自身を含む、メッセージ内容の長さ（バイト単位）。

Int32(10)

SASL認証が必要であることを指定します。

メッセージ本体はSASL認証機構をサーバにとって望ましい順に並べたリストです。最後の認証機構名の後に終端子としてゼロのバイトを置く必要があります。各機構は以下のようになります。

String

SASL認証機構の名前です。

AuthenticationSASLContinue (B)

Byte1('R')

このメッセージが認証要求であることを識別します。

Int32

自身を含む、メッセージ内容の長さ（バイト単位）。

Int32(11)

このメッセージがSASLのチャレンジを含むことを指定します。

Byten

使用するSASL機構に固有のSASLデータです。

AuthenticationSASLFinal (B)

Byte1('R')

このメッセージが認証要求であることを識別します。

Int32

自身を含む、メッセージ内容の長さ（バイト単位）。

Int32(12)

SASL認証が完了したことを指定します。

Byte*n*

SASLの結果の「追加データ」で、使用するSASL機構に固有のものです。

BackendKeyData (B)

Byte1('K')

メッセージが取り消しのキーデータであることを識別します。フロントエンドが後でCancelRequestメッセージを発行できるようにするためには、これらの値を保存しておかなければなりません。

Int32(12)

自身を含む、メッセージ内容の長さ（バイト単位）。

Int32

このバックエンドのプロセスIDです。

Int32

このバックエンドの秘密鍵です。

Bind (F)

Byte1('B')

このメッセージがBindコマンドであることを識別します。

Int32

自身を含む、メッセージ内容の長さ（バイト単位）。

String

宛先ポータルの名前（空文字列にすると無名ポータルを選択します）。

String

入力元のプリペアド文の名前（空文字列にすると無名のプリペアド文を選択します）。

Int16

その後に続くパラメータ書式コードの数（以下では*C*で表します）。これは、パラメータがない、またはパラメータはすべてデフォルトの書式（テキスト）を使うことを示す0、あるいは指定の書式コードがすべてのパラメータに適用されることを示す1にすることができます。そうでなければ、実際のパラメータの数と同じになります。

Int16[*C*]

パラメータ書式コードです。現在はそれぞれが0（テキスト）あるいは1（バイナリ）でなければなりません。

Int16

後続するパラメータ値の数 （ゼロの場合もあります）。これは問い合わせが必要とするパラメータ数と一致する必要があります。

次に、各パラメータに対して、以下のフィールドのペアが現れます。

Int32

パラメータ値のバイト単位の長さ（これには自身は含まれません）。ゼロにすることもできます。特別な場合として、-1はNULLというパラメータ値を示します。NULLの場合、後続の値用のバイトはありません。

Byte*n*

関連する書式コードで示される書式におけるパラメータの値。*n*は上述の長さです。

最後のパラメータの後に、以下のフィールドが現れます。

Int16

後続する結果列書式コードの数（以下では*R*で表します）。これは、結果列が存在しないことを示す0、あるいはすべての結果列が（もしあれば）デフォルトの書式コード（テキスト）を使用することを示す1にすることができます。さもなくば、問い合わせの結果列の実際の数と同じになります。

Int16[*R*]

結果列の書式コード。現在それぞれは0（テキスト）もしくは1（バイナリ）のいずれかでなければなりません。

BindComplete (B)

Byte1('2')

メッセージがBind完了指示子であることを識別します。

Int32(4)

自身を含む、メッセージ内容の長さ（バイト単位）。

CancelRequest (F)

Int32(16)

自身を含む、メッセージ内容の長さ（バイト単位）。

Int32(80877102)

取消要求コードです。この値は、最上位16ビットに1234が、下位16ビットに5678を持つように選択されます。（混乱を防ぐため、このコードはプロトコルバージョン番号と同一になってはいけません。）

Int32

対象バックエンドのプロセスIDです。

Int32

対象バックエンドの秘密鍵です。

Close (F)

Byte1('C')

メッセージがCloseコマンドであることを識別します。

Int32

自身を含む、メッセージ内容の長さ(バイト単位)。

Byte1

プリペアド文を閉ざす時は'S'。ポータルを閉ざす時は'P'です。

String

閉ざすプリペアド文またはポータルの名前です (空文字列で無名のプリペアド文または無名ポータルを選択します)。

CloseComplete (B)

Byte1('3')

メッセージがClose完了指示子であることを識別します。

Int32(4)

自身を含む、メッセージ内容の長さ(バイト単位)。

CommandComplete (B)

Byte1('C')

メッセージがコマンド完了応答であることを識別します。

Int32

自身を含む、メッセージ内容の長さ(バイト単位)。

String

コマンドタグです。これは通常どのSQLコマンドが完了したかを表す単一の単語です。

INSERTコマンドの場合、タグはINSERT oid rowsです。ここでrowsは挿入された行数です。かつてoidは、rowsが1、かつ、対象テーブルがOIDを持つ場合、挿入された行のオブジェクトIDでしたが、もはやOID列はサポートされていません。ですからoidは常に0です。

DELETEコマンドの場合、タグはDELETE rowsです。ここでrowsは削除された行数です。

UPDATEコマンドの場合、タグはUPDATE rowsです。ここでrowsは更新された行数です。

SELECTまたはCREATE TABLE ASの場合、タグはSELECT rowsとなります。ここでrowsは取り込んだ行数です。

MOVEコマンドの場合、タグはMOVE rowsです。ここでrowsは、カーソル位置が何行変更されたかを示す数です。

FETCHコマンドの場合、タグはFETCH rowsです。ここでrowsは、カーソルから何行取り出したかを示す行数です。

COPYコマンドの場合、タグはCOPY rowsです。ここでrowsは、コピーされた行数です（注意：この行数はPostgreSQL 8.2以降でのみ出力されます）。

CopyData (F & B)

Byte1('d')

メッセージがデータのCOPYであることを識別します。

Int32

自身を含む、メッセージ内容の長さ（バイト単位）。

Byten

COPYデータストリームの一部を形成するデータです。バックエンドから送信されるメッセージは、常に1つのデータ行に対応します。しかし、フロントエンドから送信されるメッセージは任意のデータストリームに分割される可能性があります。

CopyDone (F & B)

Byte1('c')

メッセージがCOPY完了指示子であることを識別します。

Int32(4)

自身を含む、メッセージ内容の長さ（バイト単位）。

CopyFail (F)

Byte1('f')

メッセージがCOPY失敗指示子であることを識別します。

Int32

自身を含む、メッセージ内容の長さ（バイト単位）。

String

失敗の原因を報告するエラーメッセージです。

CopyInResponse (B)

Byte1('G')

メッセージがStart Copy Inの応答であることを識別します。フロントエンドはここで必ずコピーインデータを送信しなければなりません（まだ準備ができていない場合はCopyFailメッセージを送信してください）。

Int32

自身を含む、メッセージ内容の長さ（バイト単位）。

Int8

0はCOPY全体の書式がテキスト（行は改行で区切られ、列は区切り文字などで区切られます）であることを示します。1は、コピー全体の書式がバイナリ（DataRowの書式と同様）であることを示します。より詳細についてはCOPYを参照してください。

Int16

コピーされるデータ内の列数です （以下ではNと表されます）。

Int16[N]

各列で使用される書式コードです。現在それぞれは0（テキスト）または1（バイナリ）でなければなりません。コピー全体の書式がテキストの場合、すべてが0でなければなりません。

CopyOutResponse (B)

Byte1('H')

メッセージがStart Copy Outの応答であることを識別します。このメッセージの後にコピーアウトデータが続きます。

Int32

自身を含む、メッセージ内容の長さ（バイト単位）。

Int8

0はCOPY全体の書式がテキスト（行は改行で区切られ、列は区切り文字などで区切られます）であることを示します。1はコピー全体の書式がバイナリ（DataRowの書式同様）であることを示します。詳細についてはCOPYを参照してください。

Int16

コピーされるデータ内の列数です （以下ではNと表されます）。

Int16[N]

各列で使用される書式コードです。現在それぞれは0（テキスト）または1（バイナリ）でなければなりません。コピー全体の書式がテキストの場合、すべてが0でなければなりません。

CopyBothResponse (B)

Byte1('W')

メッセージがStart Copy Bothの応答であることを識別します。このメッセージはストリーミングレプリケーションのみで使用されます。

Int32

自身を含む、メッセージ内容の長さ（バイト単位）。

Int8

0はCOPY全体の書式がテキスト（行は改行で区切られ、列は区切り文字などで区切られます）であることを示します。1はコピー全体の書式がバイナリ（DataRowの書式同様）であることを示します。詳細についてはCOPYを参照してください。

Int16

コピーされるデータ内の列数です（以下ではNと表されます）。

Int16[N]

各列で使用される書式コードです。現在それぞれは0（テキスト）または1（バイナリ）でなければなりません。コピー全体の書式がテキストの場合、すべてが0でなければなりません。

DataRow (B)

Byte1('D')

メッセージがデータ行であることを識別します。

Int32

自身を含む、メッセージ内容の長さ（バイト単位）。

Int16

後に続く列値の数です（ゼロの場合もあります）。

次に、各列について以下のフィールドのペアが現れます。

Int32

列値のバイト単位の長さです（これには自身は含まれません）。ゼロとすることもできます。特別な場合として、-1はNULLという列値を示します。NULLの場合、後続の値用のバイトはありません。

Byten

関連する書式コードで示される書式における列の値。nは上述の長さです。

Describe (F)

Byte1('D')

メッセージがDescribeコマンドであることを識別します。

Int32

自身を含む、メッセージ内容の長さ（バイト単位）。

Byte1

プリペアド文の記述の場合は'S'。ポータルの記述の場合は'P'です。

String

記述を求めるプリペアド文またはポータルの名前です（空文字列で無名のプリペアド文または無名ポータルを選択します）。

EmptyQueryResponse (B)

Byte1('I')

メッセージが空の問い合わせ文字列に対する応答であることを識別します（これはCommandCompleteを置き換えます）。

Int32(4)

自身を含む、メッセージ内容の長さ（バイト単位）。

ErrorResponse (B)

Byte1('E')

メッセージがエラーであることを識別します。

Int32

自身を含む、メッセージ内容の長さ（バイト単位）。

このメッセージの本体には、ゼロバイトを終端として後続する、1つ以上の識別されるフィールドが含まれます。フィールドは任意の順番で現れる可能性があります。各フィールドには以下があります。

Byte1

フィールド種類を識別するコードです。ゼロならば、メッセージの終端であり、後続する文字列がないことを表します。52.8に、現時点でフィールド種類として定義されているものを列挙します。将来もっと多くのフィールド種類が追加される可能性がありますので、フロントエンドは、認知できない種類のフィールドに対して何もせずに無視すべきです。

String

フィールド値です。

Execute (F)

Byte1('E')

メッセージがExecuteコマンドであることを識別します。

Int32

自身を含む、メッセージ内容の長さ（バイト単位）。

String

実行するポータルの名前です。（空文字列で無名ポータルを選択します）。

Int32

ポータルが行を返す問い合わせの場合、返される行数の最大値です（他の問い合わせでは無視されます）。ゼロは「無制限」を表します。

Flush (F)

Byte1('H')

メッセージがFlushコマンドであることを識別します。

Int32(4)

自身を含む、メッセージ内容の長さ（バイト単位）。

FunctionCall (F)

Byte1('F')

メッセージが関数呼び出しであることを識別します。

Int32

自身を含む、メッセージ内容の長さ（バイト単位）。

Int32

呼び出す関数のオブジェクトIDを指定します。

Int16

後述する引数書式コード数です（以下ではCと表します）。これは、引数が存在しない、あるいは、すべての引数がデフォルトの書式（テキスト）を使用することを示す0に、指定する書式コードをすべての引数に適用することを示す1にすることができます。さもなくば、これは実際の引数の数と同じになります。

Int16[C]

引数の書式コードです。それぞれは、0（テキスト）もしくは1（バイナリ）でなければなりません。

Int16

関数に提供する引数の数を指定します。

次に、各引数に対して以下のフィールドのペアが現れます。

Int32

引数の値のバイト単位の長さです（これには自身は含まれません）。0とすることもできます。特別な場合として、-1はNULLという引数の値を示します。NULLの場合、後続の値用のバイトはありません。

Byten

関連する書式コードで示される書式における引数の値。nは上述の長さです。

最後の引数の後に、以下のフィールドが現れます。

Int16

関数結果用の書式コードです。現在、0（テキスト）または1（バイナリ）でなければなりません。

FunctionCallResponse (B)

Byte1('V')

メッセージが関数呼び出しの結果であることを識別します。

Int32

自身を含む、メッセージ内容の長さ（バイト単位）。

Int32

関数の結果の値のバイト単位の長さです（これには自身は含まれません）。ゼロとすることもできます。特別な場合として、-1はNULLという関数の結果の値を示します。NULLの場合、後続の値用のバイトはありません。

Byte*n*

関連する書式コードで示される書式における関数の結果の値。*n*は上述の長さです。

GSSResponse (F)

Byte1('p')

このメッセージがGSSAPIまたはSSPI応答であることを識別します。これはSASLおよびパスワードの応答メッセージにも使用されることに注意してください。厳密なメッセージ種別は、その状況から推論できます。

Int32

自身を含む、メッセージ内容の長さ（バイト単位）。

Byte*n*

GSSAPI/SSPIに固有のメッセージデータ。

NegotiateProtocolVersion (B)

Byte1('v')

メッセージが、プロトコルバージョン交渉メッセージであることを識別します。

Int32

自身を含む、メッセージ内容の長さ（バイト単位）。

Int32

クライアントが要求したメジャープロトコルバージョンに対し、サーバがサポートする最新のマイナープロトコルバージョン。

Int32

サーバが認識しなかったプロトコルオプションの数。

続いて、サーバが認識しなかったプロトコルオプションに対して以下が続きます。

String

オプション名。

NoData (B)

Byte1('n')

メッセージがデータなしの指示子であることを識別します。

Int32(4)

自身を含む、メッセージ内容の長さ(バイト単位)。

NoticeResponse (B)

Byte1('N')

メッセージが警報であることを識別します。

Int32

自身を含む、メッセージ内容の長さ(バイト単位)。

このメッセージの本体には、ゼロバイトを終端として後続する、1つ以上の識別されるフィールドが含まれます。フィールドは任意の順番で現れる可能性があります。各フィールドには以下があります。

Byte1

フィールド種類を識別するコードです。ゼロならば、メッセージの終端であり、後続する文字列がないことを表します。52.8に、現時点でフィールド種類として定義されているものを列挙します。将来もっと多くのフィールド種類が追加される可能性がありますので、フロントエンドは、認知できない種類のフィールドに対して何もせずに無視すべきです。

String

フィールドの値です。

NotificationResponse (B)

Byte1('A')

メッセージが通知応答であることを識別します。

Int32

自身を含む、メッセージ内容の長さ(バイト単位)。

Int32

通知元バックエンドのプロセスIDです。

String

通知の発生元となったチャネル名です。

String

通知プロセスから渡される「ペイロード」文字列です。

ParameterDescription (B)

Byte1('t')

メッセージがパラメータ記述であることを識別します。

Int32

自身を含む、メッセージ内容の長さ（バイト単位）。

Int16

文で使用されるパラメータ数です（ゼロとすることができます）。

そして、各パラメータに対して、以下が続きます。

Int32

パラメータのデータ型のオブジェクトIDを指定します。

ParameterStatus (B)

Byte1('S')

メッセージが実行時パラメータ状態報告であることを識別します。

Int32

自身を含む、メッセージ内容の長さ（バイト単位）。

String

報告される実行時パラメータの名前です。

String

そのパラメータの現在値です。

Parse (F)

Byte1('P')

メッセージがParseコマンドであることを識別します。

Int32

自身を含む、メッセージ内容の長さ（バイト単位）。

String

宛先のプリペアド文の名前です（空文字列で無名のプリペアド文を選択します）。

String

解析される問い合わせ文字列です。

Int16

指定されるパラメータデータ型の数です（ゼロとすることができます）。これは、問い合わせ文字列内にあるパラメータの数を示すものではないことに注意してください。フロントエンドが型指定を希望するパラメータの数でしかありません。

その後、各パラメータに対し、以下が続きます。

Int32

パラメータのデータ型のオブジェクトIDを指定します。ここにゼロを書くことは型指定を行わないことと同じです。

ParseComplete (B)

Byte1('1')

メッセージがParse完了指示子であることを識別します。

Int32(4)

自身を含む、メッセージ内容の長さ（バイト単位）。

PasswordMessage (F)

Byte1('p')

メッセージがパスワード応答であることを識別します。これがGSSAPI、SSPIまたはSASL応答メッセージでも使用されることに注意してください。厳密なメッセージ種別は、その状況から推論できます。

Int32

自身を含む、メッセージ内容の長さ（バイト単位）。

String

パスワードです（必要ならば暗号化されています）。

PortalSuspended (B)

Byte1('s')

メッセージがポータル中断指示子であることを識別します。これは、Executeメッセージの行数制限に達した場合にのみ現れることに注意してください。

Int32(4)

自身を含む、メッセージ内容の長さ（バイト単位）。

Query (F)

Byte1('Q')

メッセージが簡易問い合わせであることを識別します。

Int32

自身を含む、メッセージ内容の長さ（バイト単位）。

String

問い合わせ文字列自体です。

ReadyForQuery (B)

Byte1('Z')

このメッセージ種類を識別します。バックエンドで新しい問い合わせサイクルの準備が整った時には常にReadyForQueryが送信されます。

Int32(5)

自身を含む、メッセージ内容の長さ（バイト単位）。

Byte1

現在のバックエンドのトランザクション状態指示子です。取り得る値は、待機状態（トランザクションブロックにない状態）に'I'、トランザクションブロック内の場合に'T'、失敗したトランザクションブロック（ブロックが終わるまで問い合わせは拒絶されます）内の場合に'E'です。

RowDescription (B)

Byte1('T')

メッセージが行の記述であることを識別します。

Int32

自身を含む、メッセージ内容の長さ（バイト単位）。

Int16

行内のフィールド数を指定します（ゼロとすることができます）。

その後、各フィールドに対して以下が続きます。

String

フィールド名です。

Int32

フィールドが特定のテーブルの列として識別できる場合、テーブルのオブジェクトIDです。さもなくばゼロです。

Int16

フィールドが特定のテーブルの列として識別できる場合、列の属性番号です。さもなくばゼロです。

Int32

フィールドのデータ型のオブジェクトIDです。

Int16

データ型の大きさ(pg_type.typlenを参照)です。負の値が可変長の型を表すことに注意してください。

Int32

型修飾子(pg_attribute.atttypmodを参照)です。修飾子の意味は型に固有です。

Int16

フィールドに使用される書式コードです。現在、0(テキスト)または1(バイナリ)のいずれかになります。RowDescriptionがステートメント用のDescribeから返された場合、書式コードはまだ不明ですので、常に0になります。

SASLInitialResponse (F)

Byte1('p')

メッセージが最初のSASL応答であることを識別します。これがGSSAPI、SSPIまたはパスワード応答メッセージでも使用されることに注意してください。厳密なメッセージ種別は、その状況から推論できます。

Int32

自身を含む、メッセージ内容の長さ(バイト単位)。

String

クライアントが選択したSASL認証機構の名前。

Int32

それに続くSASLの機構固有の「Initial Client Response(最初のクライアントの応答)」の長さ、またはInitial Responseがなければ-1。

Byten

SASLの機構固有の「Initial Response(最初の応答)」。

SASLResponse (F)

Byte1('p')

メッセージがSASL応答であることを識別します。これがGSSAPI、SSPIまたはパスワード応答メッセージでも使用されることに注意してください。厳密なメッセージ種別は、その状況から推論できます。

Int32

自身を含む、メッセージ内容の長さ（バイト単位）。

Byten

SASLの機構固有のメッセージデータ

SSLRequest (F)

Int32(8)

自身を含む、メッセージ内容の長さ（バイト単位）。

Int32(80877103)

SSL要求コードです。この値は最上位の16ビットに1234が、最下位の16ビットに5679が含まれるように選択されます。（混乱を防ぐため、このコードはどのプロトコルのバージョン番号とも同じになってはいけません。）

GSSENCRequest (F)

Int32(8)

自身を含む、メッセージ内容の長さ（バイト単位）。

Int32(80877104)

GSSAPI暗号化要求コードです。この値は最上位の16ビットに1234が、最下位の16ビットに5680が含まれるように選択されます。（混乱を防ぐため、このコードはどのプロトコルのバージョン番号とも同じになってはいけません。）

StartupMessage (F)

Int32

自身を含む、メッセージ内容の長さ（バイト単位）。

Int32(196608)

プロトコルのバージョン番号です。最上位の16ビットはメジャーバージョン番号（ここで説明しているプロトコルでは3）です。最下位の16ビットはマイナーバージョン番号（ここで説明しているプロトコルでは0）です。

プロトコルのバージョン番号の後には、パラメータ名と値文字列の対が1つ以上続きます。最後の名前／値の対の後に終端子としてゼロのバイトが必要です。パラメータは任意の順番に並べることができます。userが必須、他はオプションです。各パラメータは以下のように指定します。

String

パラメータ名です。現在認識される名前を以下に示します。

user

接続するデータベースユーザ名です。必須。デフォルトはありません。

database

接続するデータベースです。デフォルトはユーザ名です。

options

バックエンド用のコマンドライン引数です。(これは廃棄予定であり、個々の実行時パラメータを設定する方が好ましいです。) この文字列の中の空白は、バックスラッシュ(\)でエスケープされていなければ、引数を分けるためのものとみなされます。バックスラッシュそのものを表すためには\\と書いてください。

replication

ストリーミングレプリケーションモードで接続するのに使用され、SQL文の代わりにレプリケーションコマンドの小さな集合を発行することができます。値はtrue、falseまたはdatabaseをとることができ、デフォルトはfalseです。詳細は52.4を参照してください。

上記に加え、他のパラメータが列挙される可能性があります _pq_ .で始まるパラメータ名は、プロトコルの拡張用途のために予約されています。それ以外は、バックエンド開始時に設定される実行時パラメータとして扱われます。こうした設定は、バックエンド起動時に(もしあればコマンドライン引数の解析の後に)適用されます。この値はセッションのデフォルトとして動作します。

String

パラメータの値です。

Sync (F)

Byte1('S')

メッセージがSyncコマンドであることを識別します。

Int32(4)

自身を含む、メッセージ内容の長さ(バイト単位)。

Terminate (F)

Byte1('X')

メッセージが終了であることを識別します。

Int32(4)

自身を含む、メッセージ内容の長さ(バイト単位)。

52.8. エラーおよび警報メッセージフィールド

本節では、ErrorResponseおよびNoticeResponseメッセージ内で現れる可能性があるフィールドについて説明します。それぞれのフィールド種類は、単一バイトの識別子トークンを持ちます。メッセージ内に与えられる任意のフィールド種類は、多くてもメッセージ当たり1つでなければならないことに注意してください。

S

深刻度です。フィールドの内容はERROR、FATAL、PANIC（エラーメッセージ内）、WARNING、NOTICE、DEBUG、INFO、LOG（警報メッセージ内）、もしくはこれらの1つの地域化された翻訳です。常に存在します。

V

深刻度です。フィールドの内容はERROR、FATAL、PANIC（エラーメッセージ内）、WARNING、NOTICE、DEBUG、INFO、LOG（警報メッセージ内）です。これは、その内容が決して地域化されないという点以外はSフィールドと同一です。これはPostgreSQLバージョン9.6以降で生成されたメッセージにだけあります。

C

コード、そのエラー用のSQLSTATEコードです（付録Aを参照）。地域化されません。常に存在します。

M

メッセージ、主に人にわかりやすいエラーメッセージです。これは正確、簡潔でなければなりません（通常は1行です）。常に存在します。

D

詳細です。問題のより詳細を説明する省略可能な二次的なエラーメッセージです。複数行にまたがる可能性があります。

H

ヒントです。その問題にどう対応するかを表す省略可能な提言です。これは、詳細と異なり、事実ではなく提案（不適切な場合もありますが）を提供することを目的としたものです。複数行にまたがる可能性があります。

P

位置です。フィールド値は、エラーカーソルの位置を示すもので、元の問い合わせ文字列へのインデックスを10進ASCIIで表した整数です。先頭の文字がインデックス1になり、位置はバイトではなく文字で数えられます。

p

内部的位置です。これはPと同じ定義ですが、カーソルの位置がクライアントによって発せられたコマンドではなく内部的に生成されたコマンドを参照する場合に使用されます。このフィールドが現れる時には常にqも現れます。

q

内部的問い合わせ。失敗した、内部生成のコマンドテキストです。これは例えば、PL/pgSQL関数によって発行されたSQL問い合わせなどです。

W

場所です。エラーが発生したコンテキストを示します。現在ここには、実行中の手続き言語関数と内部生成問い合わせの呼び出しスタックトレースバックが含まれます。この追跡情報は、1行当たり1項目として、最も最近のものが初めに現れます。

s

スキーマ名。エラーが特定のデータベースオブジェクトに関連する場合、そのオブジェクトを含むスキーマ名。無名でなければ。

t

テーブル名。エラーが特定のテーブルに関連する場合、そのテーブル名。(スキーマ名フィールドにおいて、そのテーブルのスキーマ名を参照します。)

c

列名。エラーが特定のテーブルの列に関連する場合、その列名。(テーブルを識別するため、スキーマ名とテーブル名のフィールドを参照します。)

d

データ型名。エラーが特定のデータ型に関連する場合、そのデータ型名。(スキーマ名フィールドにおいて、そのデータ型のスキーマ名を参照します。)

n

制約名。エラーが特定の制約に関連する場合、その制約名。上に列挙したフィールドにおいて、関連するテーブルまたはドメインを参照します。(この目的のために、制約の構文のもとに作成されていない場合でも、インデックスは制約として扱われます。)

F

ファイルです。エラーを報告した、ソースコードのファイル名です。

L

行です。エラーを報告した、ソースコードの行番号です。

R

ルーチンです。エラーを報告した、ソースコードのルーチン名です。

> **注記**
>
> スキーマ名、テーブル名、列名、データ型名および制約名のフィールドは、限られたエラー型のためにしか提供されません。付録Aを参照してください。フロントエンドは、これらのフィールドの一部の存在が、他のフィールドの存在も保障すると仮定してはいけません。上記の相互関係により主なエラーの原因を探す方法がありますが、ユーザが定義した関数は他の方法でこれらのフィールドを利用できるかもしれません。同様の理由により、クライアントはこれらのフィールドが、現在のデータベースにおける適切なオブジェクトを示すと仮定してはいけません。

クライアントには、必要な情報を表示する際、整形する責任があります。具体的には、必要に応じて長い行を分割しなければなりません。エラーメッセージフィールド内にある改行文字は、改行ではなく、段落の区切りとして扱わなければなりません。

52.9. 論理レプリケーションのメッセージ書式

本節では論理レプリケーションの各メッセージの書式の詳細について説明します。これらのメッセージはレプリケーションスロットのSQLインタフェースから返されるか、あるいはwalsenderから送信されるかのいずれかです。walsenderの場合は、52.4で説明されているようにレプリケーションプロトコルのWALメッセージ内でカプセル化され、通常は物理レプリケーションと同じメッセージフローに従います。

Begin

Byte1('B')

メッセージが開始メッセージであることを識別します。

Int64

トランザクションの最後のLSNです。

Int64

トランザクションのコミット時刻です。その値はPostgreSQLのエポック（2000-01-01）からのマイクロ秒数です。

Int32

トランザクションのXIDです。

Commit

Byte1('C')

メッセージがCommitメッセージであることを識別します。

Int8

フラグですが現在は未使用です（0でなければなりません）。

Int64

コミットのLSNです。

Int64

トランザクションの終了LSNです。

Int64

トランザクションのコミット時刻です。その値はPostgreSQLのエポック（2000-01-01）からのマイクロ秒数です。

Origin

Byte1('O')

メッセージがOriginメッセージであることを識別します。

Int64

Originサーバ上のコミットのLSNです。

String

Originの名前です。

一つのトランザクション内で複数のOriginメッセージがあり得ることに注意してください。

Relation

Byte1('R')

メッセージがRelationメッセージであることを識別します。

Int32

リレーションのIDです。

String

名前空間（pg_catalogの場合は空文字列）。

String

リレーション名。

Int8

リレーションのレプリカ識別子の設定（pg_classのrelreplidentと同じ）。

Int16

列数。

次に、各列について以下のメッセージ部分があります。（生成列を除く）

Int8

列のフラグ。現在は、フラグがないことを示す0か、列がキーの一部であることを示す1のいずれかにできます。

String

列名。

Int32

列のデータ型のID。

Int32

列の型修飾子（atttypmod）。

Type

Byte1('Y')

メッセージがTypeメッセージであることを識別します。

Int32

データ型のID。

String

名前空間（pg_catalogの場合は空文字列）。

String

データ型の名前。

Insert

Byte1('I')

メッセージがInsertメッセージであることを識別します。

Int32

Relationメッセージ中のIDに対応するリレーションのID。

Byte1('N')

以下のTupleDataメッセージが新しいタプルであることを識別します。

TupleData

新しいタプルの内容を表すTupleDataメッセージ部分です。

Update

Byte1('U')

メッセージがUpdateメッセージであることを識別します。

Int32

Relationメッセージ中のIDに対応するリレーションのID。

Byte1('K')

これに続くTupleData副メッセージがキーであることを識別します。このフィールドはオプションで、UPDATEがREPLICA IDENTITYインデックスの一部となっている列のどれかを変更したときにのみ存在します。

Byte1('O')

これに続くTupleData副メッセージが古いタプルであることを識別します。このフィールドはオプションで、UPDATEが発生したテーブルでREPLICA IDENTITYがFULLに設定されている場合にのみ存在します。

TupleData

古いタプルまたは主キーの内容を表すTupleDataメッセージ部分です。この前に'O'または'K'の部分が存在するときにのみ存在します。

Byte1('N')

これに続くTupleDataメッセージが新しいタプルであることを識別します。

TupleData

新しいタプルの内容を表すTupleDataメッセージ部分です。

Updateメッセージは'K'メッセージ部分と'O'メッセージ部分のいずれかを含むか、どちらも含まないかであり、その両方を含むことはできません。

Delete

Byte1('D')

メッセージがDeleteメッセージであることを識別します。

Int32

Relationメッセージ中のIDに対応するリレーションのID。

Byte1('K')

これに続くTupleData副メッセージがキーであることを識別します。このフィールドはDELETEが発生したテーブルがインデックスをREPLICA IDENTITYとして使用している場合にのみ存在します。

Byte1('O')

これに続くTupleDataメッセージが古いタプルであることを識別します。このフィールドはDELETEが発生したテーブルでREPLICA IDENTITYがFULLに設定されている場合にのみ存在します。

TupleData

直前のフィールドに従って、古いタプルまたは主キーの内容を表すTupleDataメッセージ部分です。

Deleteメッセージは'K'メッセージ部分と'O'メッセージ部分のいずれかを含みますが、両方を含むことはできません。

Truncate

Byte1('T')

メッセージをTruncateメッセージと識別します。

Int32

リレーション数

Int8

TRUNCATEに対するオプションビット。1はCASCADE、2はRESTART IDENTITY

Int32

リレーションメッセージのIDに一致するリレーションのID。このフィールドは各リレーション毎に繰り返されます。

以下のメッセージ部分は上記のメッセージに共通です。

TupleData

Int16

列数。

次に各列を表す以下の副メッセージの一つがあります。(生成列を除く)

Byte1('n')

データがNULL値であることを識別します。

または、

Byte1('u')

TOAST値が変更されないことを識別します(実際の値は送信されません)。

または

Byte1('t')

データがテキスト形式の値であることを識別します。

Int32

列値の長さ。

Byte*n*

テキスト形式での列の値。(将来のリリースでは他の形式もサポートするかもしれません。) nは上記の長さです。

52.10. プロトコル2.0からの変更点の要約

本節では、既存のクライアントライブラリをプロトコル3.0に更新しようとする開発者向けに、変更点の簡易チェックリストを示します。

最初の開始パケットは、固定書式ではなく、柔軟な文字列のリスト書式を使用します。実行時パラメータのセッションのデフォルト値が直接開始パケット内に指定できるようになった点に注意してください。(実際、以前でもoptionsフィールドを使用してこれを行うことができましたが、optionsには長さに制限があること、および値内の空白文字を引用符でくくる方法がないことから、あまり安全な技法ではありませんでした。)

すべてのメッセージが、メッセージ種類バイトの直後にバイト数を持つようになりました (種類バイトがない開始パケットは例外です)。また、PasswordMessageが種類バイトを持つようになったことにも注意してください。

ErrorResponseおよびNoticeResponse('E'および'N')メッセージが複数のフィールドを持つようになりました。これを使用して、クライアントコードは、必要な冗長度に合わせて、エラーメッセージを組み立てることができます。個々のフィールドが通常改行で終わらないことに注意してください。単一の文字列を送信する古いプロトコルでは、常に改行で終わっていました。

ReadyForQuery('Z')メッセージに、トランザクション状態指示子が含まれます。

BinaryRowとDataRowメッセージ種類間の区別がなくなりました。1つのDataRowメッセージ種類で、すべての書式で記述されたデータを返すことができます。DataRowのレイアウトが解析しやすいように変更されたことに注意してください。またバイナリ値の表現も変更されました。もはやサーバの内部表現に直接束縛されません。

新しい「拡張問い合わせ」サブプロトコルがあります。これにより、フロントエンドメッセージ種類にParse、Execute、Describe、Close、Flush、およびSyncが、バックエンドメッセージ種類にParseComplete、BindComplete、PortalSuspended、ParameterDescription、NoData、およびCloseCompleteが追加されました。既存のクライアントは、このサブプロトコルを意識する必要はありませんが、これを使用することで、性能や機能を向上させることができます。

COPYデータがCopyDataとCopyDoneメッセージにカプセル化されるようになりました。COPY中のエラーから復旧するための十分に定義された方法があります。特別な「\.」という最後の行はもはや不要で、COPY OUTで送信されません。(COPY INではまだ終端として認識されます。しかし、この使用は廃止予定で、最終的には削除されます。) バイナリCOPYがサポートされます。CopyInResponseとCopyOutResponseメッセージは、列数と各列の書式を示すフィールドが含まれます。

FunctionCallとFunctionCallResponseメッセージのレイアウトが変更されました。FunctionCallは関数へのNULL引数を渡すことができるようになりました。また、テキストとバイナリ書式のどちらでもパラメータの引き渡しと結果の取り出しを扱うことができます。サーバの内部データ表現への直接アクセスを提供しなくなりましたので、FunctionCallを潜在的なセキュリティホールとみなす理由はもはやありません。

バックエンドは、接続開始時にクライアントライブラリが興味を持つとみなされるすべてのパラメータのためにParameterStatus('S')メッセージを送信します。その後、これらのパラメータのいずれかの実際の値が変更された時は常に、ParameterStatusメッセージが送信されます。

RowDescription('T')メッセージは、新規に、記述する各列に対してテーブルのOIDと列番号フィールドを伝えます。また各列の書式コードも示します。

CursorResponse('P')メッセージはもはやバックエンドで生成されません。

NotificationResponse('A')メッセージは、NOTIFYイベントの送信者から渡される「ペイロード」文字列を伝えることができる追加文字列フィールドを持ちます。

EmptyQueryResponse('I')メッセージは、空の文字列パラメータを含めるために使用されていました。これは削除されました。

第53章 PostgreSQLコーディング規約

53.1. 書式

ソースコードの書式では、タブを4カラムとするスペーシングを使用し、現在はタブを保存しています（つまりタブをスペースに展開しません）。各論理インデントのレベルは、さらに1つのタブストップです。

配置規則（括弧の位置など）はBSDの慣例に従います。特にif、while、switchなどの制御ブロックのための中括弧はそれ自身を一行で表します。

コードが80カラムのウィンドウで読み易くなるように1行の長さを制限してください。（これは80カラムを超えてはならないことを意味していません。例えば、任意の場所にある長いエラーメッセージ文字列をコードが80カラムに収まるように改行を含めても、おそらく可読性を向上させることはありません。）

一貫したコーディング形式を保つため、C++形式のコメント（//コメント）は使用しないでください。pgindentは、それらを/* ... */で置き換えます。

複数行に渡るコメントブロックの推奨書式は以下のようになります。

```
/*
 * コメントテキストはここから始まり
 * ここまで続きます
 */
```

列1で始まるコメントブロックはpgindentによりそのまま維持されますが、字下げされたコメントブロックを、あたかも平文テキストのように還流します。ある字下げブロックの中で改行を維持したい場合は以下のようにダッシュを追加します。

```
/*----------

    * コメントテキストはここから始まり
    * ここまで続きます
    *----------
    */
```

登録されたパッチは完全にはこの書式規則に従う必要はありませんが、そのようにすることを勧めます。登録されるコードは次のリリースの前にpgindentを通します。ですので、他の書式規則に従って作成して、見た目を良くすることに意味がありません。優れたパッチに関する原則は、「新しいコードがその前後にある既存のコードのように見える」ことです。

src/toolsディレクトリには、確実に上記の規約に従った書式になることを補助する、emacs、xemacs、vimエディタで使用できるサンプルの設定ファイルがあります。

テキスト閲覧ツールmoreとlessでは以下のようにすればタブを適切に表示させることができます。

```
more -x4
less -x4
```

53.2. サーバ内部からのエラーの報告

サーバコード内から生成されるエラー、警告、ログメッセージは、ereportもしくはこれに似た古いelogを使用して作成してください。この関数の使用はいくつか説明が必要なほど複雑です。

すべてのメッセージには、深刻度レベル（DEBUGからPANICまでの範囲）と主メッセージテキストという、2つの必須要素があります。さらに、省略可能な要素があります。その中で最もよく使用されるのは、SQL仕様のSQLSTATE規則に従うエラー識別コードです。ereport自身はシェルマクロで、主に、メッセージ生成をCソースコード内のひとつの関数呼び出しのように行わせる、構文の便宜上存在します。ereportで直接受け付けられる唯一のパラメータは深刻度レベルです。主メッセージテキストと任意の省略可能なメッセージ要素は、ereport呼び出し内でerrmsgなどの補助関数を呼び出すことで生成されます。

典型的なereportの呼び出しは以下のようなものです。

```
ereport(ERROR,
        errcode(ERRCODE_DIVISION_BY_ZERO),
        errmsg("division by zero"));
```

これはエラー深刻度レベルERROR（普通のエラー）を指定します。errcode呼び出しは、src/include/utils/errcodes.hで定義されたマクロを使用してSQLSTATEエラーコードを指定します。errmsg呼び出しは主メッセージテキストを提供します。

また、補助関数の呼び出しを追加の括弧のセットで囲んだ、この古いスタイルもよく見ることでしょう。

```
ereport(ERROR,
        (errcode(ERRCODE_DIVISION_BY_ZERO),
         errmsg("division by zero")));
```

この余分な括弧はPostgreSQLバージョン12より前では必要でしたが、現在ではオプションです。

以下に、より複雑な例を示します。

```
ereport(ERROR,
        errcode(ERRCODE_AMBIGUOUS_FUNCTION),
        errmsg("function %s is not unique",
               func_signature_string(funcname, nargs,
                                     NIL, actual_arg_types)),
        errhint("Unable to choose a best candidate function. "
                "You might need to add explicit typecasts."));
```

これは、実行時の値をメッセージテキスト内に埋め込むための整形用コードの使用を示します。また、省略可能な「ヒント」メッセージも提供されています。補助関数の呼び出しは任意の順序で記述できますが、慣習的にerrcodeとerrmsgが最初に記述されます。

深刻度レベルがERROR以上であれば、ereportは現在の問い合わせの実行を中断し、呼び出し元には戻りません。深刻度レベルがERROR未満であれば、ereportは正常に戻ります。

ereportで使用可能な補助ルーチンを以下に示します。

- errcode(sqlerrcode)は、その条件用のSQLSTATEエラー識別コードを指定します。このルーチンが呼び出されないと、エラー識別子のデフォルトは、エラー深刻度レベルがERROR以上の場合にはERRCODE_INTERNAL_ERRORに、エラー深刻度レベルがWARNINGの場合にはERRCODE_WARNINGに、さもなければ(NOTICE以下)ERRCODE_SUCCESSFUL_COMPLETIONになります。これらのデフォルトはしばしば便利ですが、errcode()呼び出しを省略する前に、常に適切かどうかを検討してください。

- errmsg(const char *msg, ...)は、主エラーメッセージテキストを指定し、また、実行時の値をそこに挿入することもできます。挿入は、sprintf様式の書式コードで指定されます。sprintfで受け付けられる標準の書式コードに加え、%m書式コードを使用して、現在のerrnoの値用のstrerrorで返されるエラーメッセージを挿入することができます。[1] %mはerrmsgのパラメータリスト内に対応する項目を必要としません。メッセージ文字列は、書式コードの処理を行う前に、地域化のためにgettextを通ることに注意してください。

- errmsg_internal(const char *msg, ...)は、そのメッセージ文字列は変換されず、国際化用メッセージ辞書に含まれない点を除き、errmsgと同一です。これは、おそらく翻訳作業を行う価値がない「発生し得ない」場合用に使用すべきです。

- errmsg_plural(const char *fmt_singular, const char *fmt_plural, unsigned long n, ...)はerrmsgのようですが、メッセージの各種の複数書式があります。fmt_singularは英語の単数書式、fmt_pluralは英語の複数書式、nはどの複数書式が必要なのかを決定する整数値で、残りの引数は選択された書式文字列に従って書式化されます。より詳細な情報は54.2.2を参照してください。

- errdetail(const char *msg, ...)は省略可能な「詳細」メッセージを提供します。これは、主メッセージ内に記述するには不適切と考えられる追加情報がある場合に使用されます。このメッセージ文字列はerrmsgとまったく同じ方法で処理されます。

- errdetail_internal(const char *msg, ...)は、メッセージが翻訳されない、または、国際化メッセージ辞書内に含まれない点を除き、errdetailと同じです。これは、例えばほとんどのユーザに役に立つにはあまりにも技術的すぎるなど、翻訳する手間をかける価値がない詳細メッセージで有用であるはずです。

- errdetail_plural(const char *fmt_singular, const char *fmt_plural, unsigned long n, ...)はerrdetailと似ていますが、メッセージの各種の複数書式をサポートします。より詳細な情報は54.2.2を参照してください。

- この文字列がサーバログのみに渡り、クライアントに渡らない点を除きerrdetail_log(const char *msg, ...)はerrdetailと同一です。errdetail(や上記の等価物の1つ)とerrdetail_logが共に使用された場合、1つの文字列はクライアントに渡り、もう1つはログに渡ります。クライアントに送られるレポートに含めるにはセキュリティに対して慎重を期さなければならないもの、あるいは膨大すぎるエラー詳細に対して効果があります。

- errdetail_log_plural(const char *fmt_singular, const char *fmt_plural, unsigned long n, ...)はerrdetail_logと似ていますが、メッセージの各種の複数書式をサポートします。より詳細な情報は54.2.2を参照してください。

- errhint(const char *msg, ...)は、省略可能な「ヒント」メッセージを提供します。これは、何が悪かったのかについての事実に基づく詳細とは反対で、問題を解決させる方法に関する提言を提供するために使用されます。このメッセージ文字列はerrmsgとまったく同じ方法で処理されます。

[1] つまり、ereport呼び出しに達した時点の値です。補助報告ルーチン内でerrnoを変更しても効果はありません。errmsg内でstrerror(errno)を明示的に記述したとしても正確なものにはなりません。したがって、このようにはしないでください。

- errcontext(const char *msg, ...)は通常ereportメッセージ側から直接呼び出されません。エラーが発生したコンテキスト、例えばPL関数の現在位置の情報を提供するためにerror_context_stackコールバック関数内で使用されます。メッセージ文字列はerrmsgとまったく同じ方法で処理されます。他の補助関数とは異なり、1つのereport呼び出しで何度も呼び出すことができます。こうして提供される文字列の並びは、改行で区切った形で連結されます。

- errposition(int cursorpos)は、問い合わせ文字列内でエラーが発生した位置をテキストで指定します。現在、問い合わせ処理の字句解析および構文解析段階でエラーが検出された場合にのみ役に立ちます。

- errtable(Relation rel)はエラーレポートにおいて名前とスキーマ名が外部フィールドとして含まれなければならないリレーションを指定します。

- errtablecol(Relation rel, int attnum)はエラーレポートにおいて名前、テーブル名、およびスキーマ名が外部フィールドとして含まれなければならない列を指定します。

- errtableconstraint(Relation rel, const char *conname)はエラーレポートにおいて名前、テーブル名、およびスキーマ名が外部フィールドとして含まれなければならないテーブル制約を指定します。この目的のため、関連したpg_constraintエントリの有る無しに関わらず、インデックスは制約と見なされなければなりません。進行中のヒープリレーションを受け渡すにはインデックスではなくrelとすることに注意してください。

- errdatatype(Oid datatypeOid)はエラーレポートにおいて名前とスキーマ名が外部フィールドとして含まれなければならないデータ型を指定します。

- errdomainconstraint(Oid datatypeOid, const char *conname)はエラーレポートにおいて名前、ドメイン名、およびスキーマ名が外部フィールドとして含まれなければならないドメイン制約を指定します。

- errcode_for_file_access()は、ファイルアクセス関連のシステムコールでの失敗用のSQLSTATEエラー識別子を適切に選択する、便利な関数です。保存されたerrnoを使用して、どのエラーコードを生成するかを決定します。通常、これは主エラーメッセージテキスト内で%mと組み合わせて使用されなければなりません。

- errcode_for_socket_access()は、ソケット関連のシステムコールでの失敗用のSQLSTATEエラー識別子を適切に選択する、便利な関数です。

- errhidestmt(bool hide_stmt)は、postmasterのログ内のメッセージにおけるSTATEMENT:部分を抑制するために呼び出すことができます。通常これは、メッセージテキスト内にすでに現在の文が含まれている場合に適しています。

- errhidecontext(bool hide_ctx)は、postmasterのログ内のメッセージにおけるCONTEXT:部分を抑制するために呼び出すことができます。これは、冗長なデバッグメッセージにおいてコンテキストを繰り返し含めることがログのサイズを巨大にしてしまうような場合にのみ用いられるべきです。

> **注記**
>
> ereport呼び出しにおいて、最大限でもerrtable、errtablecol、errtableconstraint、errdatatype、またはerrdomainconstraintのうちの一つの関数が使用されなければなりません。これらの関数は、アプリケーションが自動エラー対応であって欲しいと期待するエラーレポートにおいて

> 使用されるべきです。PostgreSQL 9.3の時点で、この機能を完璧に保証する範囲はSQLSTATEクラス23（整合性制約違反）のみですが、将来的には十中八九拡張されそうです。

まだ頻繁に使用されている、古めのelog関数があります。以下のelog呼び出しは、

```
elog(level, "format string", ...);
```

以下とまったく同じです。

```
ereport(level, errmsg_internal("format string", ...));
```

SQLSTATEエラーコードが常にデフォルトになること、メッセージ文字列が変換されないことに注意してください。したがって、elogは、内部エラーと低レベルなデバッグ用ログにのみ使用すべきです。一般ユーザを対象とする任意のメッセージはereportを使用すべきです。それでもなお、システム内の「発生できなかった」内部エラーの検査にelogがまだ多く使用されています。これは、こうした単純な表記のメッセージに適しています。

53.3に推奨するエラーメッセージの作成についての提言を示します。

53.3. エラーメッセージのスタイルガイド

このスタイルガイドでは、PostgreSQLで生成されるすべてのメッセージに対する、一貫性維持、ユーザに親切なスタイルについての希望を説明します。

何がどこで起こったか

主メッセージは、簡潔に事実を示すものにすべきです。特定の関数名など実装の詳細への参照は止めるべきです。「簡潔」は「ごく普通の条件下で1行に収まる」ことを意味します。主メッセージを簡潔にするために必要であれば、また、特定のシステムコールが失敗したなど実装の詳細について記載したいのであれば、詳細メッセージを使用してください。主メッセージ、詳細メッセージの両方は事実を示すものにすべきです。どうすれば問題を解決できるかに関する提言には、その提言が常に適切とは限らない場合は特に、ヒントメッセージを使用してください。

例えば、

```
IpcMemoryCreate: shmget(key=%d, size=%u, 0%o) failed: %m
(plus a long addendum that is basically a hint)
```

ではなく、以下のように記述してください。

```
Primary:    could not create shared memory segment: %m
Detail:     Failed syscall was shmget(key=%d, size=%u, 0%o).
Hint:       the addendum
```

理論的根拠：主メッセージを簡潔にすることは、要点を維持することを助けます。そして、クライアントは、エラーメッセージ用に1行分確保すれば十分であるという仮定の下で画面設計を行うことができます。詳細メッセージやヒントメッセージを冗長モードに格下げしたり、エラーの詳細を表示するウィンドウをポップアップさせることもできます。また、詳細メッセージやヒントメッセージは通常ディスク容量を節約するためにサーバログには出力されません。ユーザがその詳細を知っているとは期待できないので、実装の詳細への参照を避けることが最善です。

整形

メッセージテキストの整形に関して、特定の前提を行わないでください。クライアントやサーバログでは、自身の必要性に合わせて行を改行すると想定してください。長めのメッセージでは、改行文字（\n）を推奨する段落の区切りを示すものとして使用することができます。メッセージの終わりに改行を付けないでください。タブや他の整形用文字を使用しないでください。（エラーの内容の表示では、関数呼び出しなどのコンテキストのレベルを区切るために、改行が自動的に追加されます。）

理論的根拠：メッセージは必ずしも端末型のディスプレイに表示されるとは限りません。GUIのディスプレイやブラウザでは、こうした書式指示はうまくいったとしても無視されます。

引用符

英文では、引用が適切な場合には二重引用符を使用すべきです。他の言語でのテキストは、出版上の慣習や他のプログラムのコンピュータ出力と矛盾しない種類の引用符の1つを一貫して使用してください。

理論的根拠：二重引用符と単一引用符の選択はどちらかでもよいものですが、推奨する方がよく使われています。SQL規約（すなわち文字列には単一引用符、識別子には二重引用符）に従って、オブジェクトの種類に応じて引用符を選択することを推奨する人もいます。しかし、これは言語内部の技術的な事項であり、多くのユーザが理解できるものではありません。また、他の種類の引用手法には拡張できません。他の言語へ翻訳できません。ですので、あまり意味がありません。

引用符の使用

ファイル名、ユーザ提供の識別子、その他の変数に単語が含まれている可能性がある場合には、必ず引用符で区切ってください。単語を含まない変数（例えば演算子名）をマークアップする際には引用符を使用しないでください。

バックエンドには必要に応じて出力に二重引用符を付与する関数(例えば`format_type_be()`)があります。こうした関数の出力の前後にさらに引用符を追加しないでください。

理論的根拠：オブジェクトの名前をメッセージ内に埋め込む際に曖昧さが生じることがあります。埋め込む名前がどこから始まりどこで終わるかついての表記には一貫性を持たせてください。しかし、不必要にメッセージをまとめたり、引用符を二重にすることは止めてください。

文法と句読点

この規則は、主エラーメッセージと詳細/ヒントメッセージとで異なります。

主エラーメッセージ: 最初の文字を大文字にしないでください。メッセージの最後にピリオドを付けないでください。メッセージの終わりに感嘆符を付けようとは考えないでください。

詳細メッセージとヒントメッセージ: 完全な文章を使用し、終わりにピリオドを付けてください。文章の最初の単語は大文字にしてください。他の文が続く場合はピリオドの後に空白を2つ入れてください(英文の場合です。他の言語では不適切かもしれません)。

エラー文脈文字列: 先頭文字を大文字にせず、また、終わりにはピリオドを付けないでください。文脈文字列は通常完全な文章にすべきではありません。

理論的根拠: 句読点の禁止により、クライアントアプリケーションでは、そのメッセージを各種文法的なコンテキストに埋め込みやすくなります。主メッセージはしばしば文法的に完全な文章になっていません。(そして、1行以上の長さになりそうであれば、主メッセージと詳細メッセージに分割すべきです。) しかし、詳細メッセージとヒントメッセージは、より長く、かつ複数の文章を持つ必要があるかもしれません。一貫性のため、これらは、たとえ文章が1つだけであっても、完全な文章形式に従うべきです。

大文字と小文字

メッセージの言葉使いでは小文字を使用してください。主エラーメッセージの場合は先頭文字も含みます。SQLコマンドとキーワードがメッセージ内に出現する場合は大文字を使用してください。

理論的根拠: メッセージは完全な文章かもしれませんし、そうではないかもしれませんので、この方法は、より簡単にすべての見た目の一貫性を向上します。

受動態の禁止

能動態を使用してください。能動的な主語がある(「AはBを行うことができない」)場合は完全な文章を使用してください。主語がプログラム自体である場合は、主語を付けずに電報様式を使用してください。プログラムに「I(私)」を使用しないでください。

理論的根拠: プログラムは人間ではありません。他を真似ないでください。

現在形と過去形

試行が失敗したが、次は(何らかの問題を修正した後に)成功するかもしれない場合は過去形を使用してください。失敗が永続するようであれば、現在形を使用してください。

以下の2つの意味には無視できないほどの違いがあります。

```
could not open file "%s": %m
```

および

```
cannot open file "%s"
```

最初のものは、ファイルを開くことに失敗したことを意味します。メッセージには、「ディスクが一杯」や「ファイルが存在しない」といった、その理由を付けるべきです。次回はディスクに空きがあるかもしれませんし、問題のファイルが存在するかもしれませんので過去形が適切です。

2番目の形式は、そのプログラム内の指定されたファイルを開く機能が存在しない、あるいは、概念的に不可能であることを示します。この条件は永遠に続きますので現在形が適切です。

理論的根拠：仮定ですが、一般的なユーザは単なるメッセージの時制から多くの意味を引き出すことはできないでしょう。しかし、言語が文法を提供してくれますので、それを正確に使用すべきでしょう。

オブジェクトの種類

オブジェクトの名前を引用する時、そのオブジェクトの種類を記載してください。

理論的根拠：さもないと、「foo.bar.baz」が何なのか誰もわかりません。

角括弧

角括弧は、(1)コマンドの概要にて省略可能な引数を表す、(2)配列の添字を表す、ためだけに使用されます。

理論的根拠：広く知られる慣習に対応するものがなく、人々を混乱させることになります。

エラーメッセージの組み立て

メッセージに、他で生成されるテキストを含める場合、以下の様式で埋め込んでください。

```
could not open file %s: %m
```

理論的根拠：すべての起こり得るエラーコードを単一のなめらかな文章に埋め込むことを考えることは困難です。ですので、何らかの句読点が必要とされます。括弧の中にテキストを埋め込むこともまた推奨されていますが、よくあるように埋め込むテキストがそのメッセージの最も重要となる場合は不自然です。

エラーの理由

メッセージは常にエラーが発生した理由を記述すべきです。以下に例を示します。

```
BAD:    could not open file %s
BETTER: could not open file %s (I/O failure)
```

理由がわからない場合はコードを直すべきです。

関数名

エラーテキストには、それを報告したルーチンの名前を含めないでください。必要に応じて、そのルーチンを見つける他の機構がありますし、また、ほとんどのユーザにとって役に立つ情報ではありません。関数名がないと、エラーメッセージにあまり意味がないのであれば、言葉使いを変えてください。

```
BAD:    pg_strtoint32: error in "z": cannot parse "z"
```

```
BETTER: invalid input syntax for type integer: "z"
```

同時に呼び出した関数名の記述も止めてください。代わりにそのコードが何をしようとしたのかを記述してください。

```
BAD:    open() failed: %m
BETTER: could not open file %s: %m
```

もし本当に必要であれば、詳細メッセージにそのシステムコールを記載してください（詳細メッセージの情報としてシステムコールに実際に渡した値を与えることが適切な場合もあります）。

理論的根拠：ユーザはその関数が何を行うのかを知りません。

ややこしい単語の防止

Unable. 「Unable」はほとんど受動態です。「cannot」または「could not」の適切な方を使用してください。

Bad. 「bad result」といったエラーメッセージは、知的に解釈することが非常に困難です。結果が「bad」である理由、例えば「invalid format」を記述してください。

Illegal. 「Illegal」は規則違反を表します。他は「invalid」です。より良くするために、なぜ無効なのかについても記述してください。

Unknown. 「unknown」は極力使用しないでください。「error: unknown response」について考えてみましょう。どんな応答であるかわからなければ、どうやって何がエラーなのかわかるでしょうか。「Unrecognized」を選んだ方が良い場合がしばしばあります。また、その警告の中に値が含まれていることを確認してください。

```
BAD:    unknown node type
BETTER: unrecognized node type: 42
```

Find対Exists. プログラムがリソースの場所について無視できないアルゴリズム（例えばパスの検索）を使用し、そのアルゴリズムが失敗した場合、プログラムがリソースを「find」できなかったと記述すべきでしょう。一方、想定したリソースの場所はわかっているが、プログラムがアクセスできなかった場合は、リソースが「exist」しなかったと記述してください。この場合に「find」を使用すると、弱く取られ、問題が混乱します。

May、Can、Might. 「May」は許可を示し、文書やエラーメッセージではあまり使われません（たとえば、熊手を借りられます）。「Can」は能力を示し（たとえば、丸太を持ち上げることができます）、「might」は可能性を示します（たとえば、雨が降るかもしれません）。意味を明確にし、翻訳を補助するために適切に使用してください。

短縮. 「can't」などの短縮は避けてください。代わりに「cannot」を使用してください。

適切なスペル

単語の完全なスペルを使用してください。例えば、以下は止めてください。

- spec
- stats
- parens
- auth
- xact

理論的根拠：これは一貫性を向上します。

地域化

エラーメッセージは他の言語に翻訳される必要があることを忘れないでください。54.2.2のガイドラインに従い、翻訳者に苦労を強いることを防いでください。

53.4. その他のコーディング規約

標準C

PostgreSQLのコードはC99の標準で利用可能な言語機能にのみ依存することになっています。つまり、C99に準拠したコンパイラであれば、少数のプラットフォーム依存の部分を除けばpostgresをコンパイルできるはずです。

現時点では、C99の標準に含まれる機能のいくつかはPostgreSQLのコアコードでは使うことが許可されていません。今のところ、可変長配列、型宣言とコードの混在、//コメント、汎用文字名が含まれます。その理由には、移植性と歴史的な慣例が含まれます。

代替策が用意されているのであれば、それより後のバージョンの標準Cの機能、あるいはコンパイラに依存した機能を使用することもできます。

例えば、_Static_assert()と__builtin_constant_pは、それぞれ、標準Cのより新しいバージョン由来、GCC拡張ですが、現在、使用されています。それらが利用できない場合は、それぞれ、同じチェックをする（ただし、やや暗号的なメッセージを発する）C99互換のもので代用し、__builtin_constant_pは使いません。

関数のようなマクロとインライン関数

引数付きのマクロとstatic inlineの関数のどちらも使用することができます。マクロとして記述した場合に、複数回評価されるリスクがあるならば、後者を選択します。例えば次のような場合です。

```
#define Max(x, y)       ((x) > (y) ? (x) : (y))
```

あるいは、マクロにすると非常に長くなる場合も、インライン関数を選択します。その他に、マクロだけしか利用できない、あるいはマクロの方が使いやすい場合があります。例えば、マクロに様々な型の式を渡す必要がある場合などです。

インライン関数の定義がバックエンドの一部としてのみ利用可能なシンボル（つまり、変数、関数）を参照する場合、その関数はフロントエンドのコードからインクルードされたときに不可視かもしれません。

```
#ifndef FRONTEND
static inline MemoryContext
MemoryContextSwitchTo(MemoryContext context)
{
    MemoryContext old = CurrentMemoryContext;

    CurrentMemoryContext = context;
    return old;
}
#endif   /* FRONTEND */
```

この例では、バックエンドのみで利用可能なCurrentMemoryContextが参照されているため、関数は#ifndef FRONTENDで隠されています。一部のコンパイラはインライン関数に含まれるシンボルの参照を、その関数が使われていなくても吐き出すため、この規則があります。

シグナルハンドラの作成

シグナルハンドラの内部で実行されるのに適切であるためには、注意深くコードを書く必要があります。根本的問題は、シグナルハンドラは、ブロックされない限り、いつでもコードに対して割り込むことができるということです。シグナルハンドラの内側のコードが、外側のコードと同じ状態を使うと、混沌が発生するかもしれません。例えば、シグナルハンドラが、割り込まれたコードで既に保持されているロックを獲得しようとしたら何が起きるか考えてみてください。

特別に準備された状況を別にすると、シグナルハンドラのコードは、（POSIXで定義される通りの）非同期シグナルで安全な関数だけを呼ぶことができ、型volatile sig_atomic_tの変数だけにアクセスできます。postgresでも、いくつかの関数はシグナルで安全とされており、なかでも重要なのはSetLatch()です。

ほとんどの場合、シグナルハンドラはシグナルが到着したことを記録し、ハンドラの外部で動作しているコードをラッチを使って呼び起こす以上のことをすべきではありません。以下はそのようなハンドラの例です。

```
static void
handle_sighup(SIGNAL_ARGS)
{
    int         save_errno = errno;

    got_SIGHUP = true;
    SetLatch(MyLatch);

    errno = save_errno;
}
```

errnoはSetLatch()によって変更されるかもしれないので、保存して、リストアされます。そうしなければ、割り込まれたコードが、現在errnoを参照している場合、誤った値を見ることになるかもしれません。

関数ポインタの呼び出し

明快にするため、ポインタが単純な変数である場合に指し示す関数を呼び出すときには、関数ポインタを以下の例のように明示的に参照することが望ましいです。

```
(*emit_log_hook) (edata);
```

(emit_log_hook(edata)でも動作するとしても)。関数ポインタが構造体の一部であるときには、以下のように、追加的な区切りは省略してよいし、通常は省略すべきです。

```
paramInfo->paramFetch(paramInfo, paramId);
```

第54章 多言語サポート

54.1. 翻訳者へ

PostgreSQLプログラムは（サーバ、クライアントともに）メッセージが翻訳されていれば、使い慣れた言語でそのメッセージを出すことができます。翻訳メッセージセットの作成と保守は、自分の言語を上手に話しPostgreSQLの成果に貢献したい人々の協力が必要です。この作業にはプログラマである必要はまったくありません。本節ではお手伝いの仕方を説明します。

54.1.1. 必要事項

協力者の言語の熟練度については判断しません。本節ではソフトウェアツールについて説明します。理論的には、テキストエディタのみが必要です。しかし、これは自分で作成した翻訳メッセージを試そうとはしないという、あまりあり得ない場合のみです。ソースツリーを構築する際に、`--enable-nls`オプションを使用していることを確認してください。これにより、全てのエンドユーザがとにかく必要とする、libintlライブラリと`msgfmt`プログラムも同時に検査されます。作業を試す際には、インストール手順の適切な部分に従ってください。

新規に翻訳作業を始めるか、（後述の）メッセージカタログのマージを行いたい場合、GNU版と互換性を持った実装の`xgettext`と`msgmerge`という2つのプログラムがそれぞれ必要です。将来はパッケージ化されたソース配布物を使用する場合に`xgettext`を必要としないように変更する予定です（Git版で作業をしているのであれば、まだこれが必要です）。現在GNU Gettext 0.10.36以上を推奨します。

使用するマシンのgettextの実装については、文書が一緒に付いてくると思います。以下のいくつかはおそらく重複していますが、追補すべき詳細についてはその文書を参照してください。

54.1.2. 概念

英語による元のメッセージとそれを基に（場合によって）翻訳されたメッセージの組み合わせは*メッセージカタログ*に、（関連するプログラムはメッセージカタログを共有することができますが）各プログラム、対象とする言語に対して一つずつ保持されます。メッセージカタログには2つのファイル形式があります。1つは「PO」ファイル（移植可能オブジェクト-Portable Object-を意味します）で翻訳者が編集する特別な構文を持った平文ファイルです。2番目は「MO」ファイル（マシンオブジェクト-Machine Object-を意味します）で対応するPOファイルから生成され、国際化プログラムの実行の際に使用されるバイナリファイルです。翻訳者は、MOファイルを扱いません。実際に扱うことは困難です。

メッセージカタログファイルの拡張子は想像していたかもしれませんが`.po`もしくは`.mo`です。基本名（拡張子を除いた部分）は、プログラムが伴っている名前、もしくは翻訳目的とする言語の名前のいずれかで、状況によって変わります。少し混乱するかもしれません。例えば、`psql.po`（psql用のPOファイル）、もしくは`fr.mo`（フランス語用のMOファイル）です。

POファイルの書式を以下に示します。

```
# comment

msgid "original string"
```

```
msgstr "translated string"

msgid "more original"
msgstr "another translated"
"string can be broken up like this"

...
```

msgidはプログラムのソースから抽出されます。(これは必要はありませんが最も一般的な方法です。)
msgstr行は初期状態では空であり、翻訳者によって有益な文字列が埋め込まれます。この文字列には、C言語形式のエスケープ文字を含めることも、上に示したように複数行にまたがって続けることもできます。(継続行は必ず行の先頭から始まらなければなりません。)

#文字はコメントの開始を示します。#文字の直後に空白文字があった場合、それは翻訳者によって保守されるコメントです。#の直後に非空白文字が付く、自動的に付与されるコメントもあります。これらは、POファイルに対する操作を行う各種ツールによって保守され、翻訳者の補助を意図しています。

```
#. automatic comment
#: filename.c:1023
#, flags, flags
```

#.スタイルのコメントはそのメッセージが使用されているソースファイルから抽出されます。おそらくプログラマが、翻訳者のために追加した、そのようにしてほしいと考える体裁についてなどの情報です。#:コメントは、ソース内でそのメッセージが使用される正確な場所を示します。翻訳者はプログラムソースを参照する必要はありませんが、翻訳の正確さに疑問がある場合にソースを参照することができます。#,コメントは何らかの方法でメッセージを説明するフラグです。現在、2つのフラグがあります。そのメッセージがプログラムソースの変更によって古いものとなった可能性がある場合、fuzzyが設定されます。翻訳者はこれを検証し、fuzzyフラグを削除できます。fuzzyメッセージはエンドユーザからは利用できないことに注意してください。もう1つのフラグはc-formatで、そのメッセージがprintf形式の書式テンプレートであることを示します。これは、翻訳側もプレースホルダの型と同じ番号を持った書式文字列でなければならないことを意味します。これを検証するツールがあり、それらはc-formatフラグを入力として受け取ります。

54.1.3. メッセージカタログの作成と保守

さて、「空の」メッセージカタログをどうやって作成するのでしょうか。まず、翻訳したいメッセージを持つプログラムが存在するディレクトリに移動します。nls.mkファイルがあればこのプログラムは翻訳の準備が整っています。

もし、.poファイルが数個既にあれば、誰かがある翻訳作業を行っています。そのファイルはlanguage.poと名前が付けられています。ここで、languageはISO 639-1の2文字言語コード(小文字)[1]を表します。例えば、fr.poはフランス語用です。1つの言語に複数の翻訳成果が必要である場合そのファイルの名前はlanguage_region.poのようになります。ここで、regionはISO 3166-1の2文字国コード(大文字)[2]を表します。例えば、pt_BR.poはブラジルでのポルトガル語用を示します。翻訳対象とする言語用のファイルがあれば、それを基に作業を始めることができます。

[1] https://www.loc.gov/standards/iso639-2/php/English_list.php

[2] https://www.iso.org/iso-3166-country-codes.html

新規に翻訳を始める必要がある場合、以下のコマンドを最初に実行してください。

```
make init-po
```

これは、progname.potファイルを作成します。(.potは「実用の」POファイルと区別するためのものです。Tは「テンプレート」を意味します。) このファイルをlanguage.poにコピーして編集します。新規の言語が利用可能になったことを知らせるために、nls.mkを編集し、言語(もしくは言語と国)コードを以下のように追加してください。

```
AVAIL_LANGUAGES := de fr
```

(もちろん他の言語があるかもしれません。)

対象のプログラムやライブラリの変更に伴い、メッセージはプログラマによって変更、追加されます。この場合は始めからやり直す必要はありません。その代わりに以下のコマンドを実行してください。

```
make update-po
```

そうすると新しい空のメッセージカタログファイル(最初に使用したpotファイル)が作成され、既存のPOファイルにマージされます。このマージのアルゴリズムが特定のメッセージに関して確実でない場合、それは上で説明した「fuzzy」となります。新規POファイルは.po.newという拡張子付きで保存されます。

54.1.4. POファイルの編集

POファイルは普通のテキストエディタで編集することができます。翻訳者はmsgstrディレクティブの後の引用符の間の部分の変更、コメントの追加、fuzzyフラグの変更のみを行えばよいのです。Emacsには(予想通り)POモードがあり、非常に使いやすいものです。

POファイルを完全に埋めることは必要ありません。ソフトウェアは使用できる翻訳がない(もしくは翻訳が空の)場合自動的に元の文字列に戻します。他の人が作業を引き継ぐことができますので、ソースツリー内に不完全な翻訳を含めることにも問題はありません。しかし、マージの後のfuzzyフラグを削除することを優先に考えることが推奨されています。fuzzyエントリはインストールされないことを忘れないでください。これらは何が正しい翻訳になり得るかの参照のためにのみ提供されています。

以下に、翻訳の編集を行う際に注意すべき点を示します。

- 元の文字列の終端が改行の場合、翻訳も同様になっていることを確認してください。タブなども同様です。
- 元がprintf書式文字列の場合、翻訳も同じでなければなりません。また、翻訳は同じ書式識別子を同じ順番で持たなければなりません。言語固有の規則によっては不可能な場合や扱いづらい場合も起こります。このような場合は、以下の書式指定子を使用することができます。

  ```
  msgstr "Die Datei %2$s hat %1$u Zeichen."
  ```

  そして、リストの最初のプレースホルダが実際にはこのリストの2番目の引数に使用されます。digits$は%の直後に続く必要があり、他の書式の操作の前に使用する必要があります。(この機能はprintf系の関数に本当に存在するものです。メッセージ国際化以外ではほとんど使用されませんので、聞いたことがないかもしれません。)

- 元の文字列に言語上の間違いがある場合、それを報告し(もしくはプログラムソースを直して)、普通に翻訳してください。修正された文字列は、プログラムのソースが修正された時にマージ可能になります。元の文字列が事実と異なる場合、それを報告し(もしくは自ら直して)、翻訳を行わないでください。その代わりに、POファイルのその文字列にコメントを付けてください。

- 元の文字列のスタイルや調子を維持してください。特に、(cannot open file %sなど)文章になっていないメッセージは、(翻訳する言語が大文字小文字を区別するのであれば)おそらく最初の文字を大文字にしてはなりませんし、(翻訳する言語が句読点として使用しているのであれば)ピリオドを終わりに付けてはいけません。53.3を読むと参考になります。

- メッセージの意味がわからない時や、曖昧な場合は、開発者用のメーリングリストに問い合わせてください。英語を話すエンドユーザも理解できない、または曖昧であると判断することができる機会となり、メッセージの改良を行う最善のものです。

54.2. プログラマへ

54.2.1. 仕組み

本節では、PostgreSQL配布物の一部であるプログラムやライブラリにおける各国語サポートの実装方法を説明します。現在はCプログラムにのみ適用できます。

プログラムにNLSサポートを追加する

1. プログラムの起動処理に以下のコードを追加してください。

```
#ifdef ENABLE_NLS
#include <locale.h>
#endif

...

#ifdef ENABLE_NLS
setlocale(LC_ALL, "");
bindtextdomain("progname", LOCALEDIR);
textdomain("progname");
#endif
```

   (prognameは実際には自由に選択できます。)

2. どこであろうと翻訳の候補となるメッセージが見つかったら、gettext()の呼び出しが追加される必要があります。例えば、

```
fprintf(stderr, "panic level %d\n", lvl);
```

   は、次のように変更されます。

```
fprintf(stderr, gettext("panic level %d\n"), lvl);
```

（NLSサポートが組み込まれていない場合、gettextはノーオペレーション命令として定義されます。）

これは混乱しがちになります。一般的なショートカットは以下のものです。

```
#define _(x) gettext(x)
```

他の解決方法は、バックエンドにおけるereport()のように、そのプログラムが通信のほとんどを1つまたは数個の関数で行っている場合有効です。その場合、この関数の内部で全ての入力文字列に対しgettextを呼び出すようにすることになります。

3. プログラムのソースのあるディレクトリにnls.mkを追加してください。これはmakefileとして読まれます。以下の変数への代入をここで設定する必要があります。

   CATALOG_NAME

   textdomain()の呼び出しに使用されるプログラム名です。

   AVAIL_LANGUAGES

   用意された翻訳のリストです。初めは空です。

   GETTEXT_FILES

   翻訳可能文字列を含むファイルの一覧です。つまり、これらはgettextもしくは他の解決法として印が付けられます。結局、これはプログラムのほとんど全てのソースファイルを含むことになります。この一覧があまりに長くなる場合、最初の「file」を+とし、2番目の単語を1行に対して1つのファイル名を持ったファイルとすることができます。

   GETTEXT_TRIGGERS

   翻訳者が作業を行う上で、どの関数呼び出しが翻訳可能文字列を含むかを知る必要に迫られた場合に、メッセージカタログを生成するツールです。デフォルトでは、gettext()呼び出しのみを認識します。_や他の識別子を使用した場合、ここに記載しなければなりません。翻訳可能文字列がその最初の引数ではない場合、その項目は（例えば2番目の引数の場合）func:2という形式でなければなりません。複数形メッセージをサポートする関数がある場合、その項目は（単一形および複数形メッセージ引数を特定する）func:1,2のようになります。

ビルドシステムは、自動的にメッセージカタログの構築およびインストールを行います。

54.2.2. メッセージ記述の指針

メッセージの翻訳を簡単にするために以下に指針をいくつか示します。

- 以下のように実行時に文章を構築することはしないでください。

```
printf("Files were %s.\n", flag ? "copied" : "removed");
```

文章内の単語の順番は言語によって異なる可能性があります。さらに全ての断章に対してgettext()を呼び出すことを覚えていたとしても、断章が個別に的確に翻訳されるわけではありません。それぞれのメッセージが全て調和して翻訳されるかどうか、ちょっとしたコードの複製を用意するとよいかもしません。番号、ファイル名、および実行時変数のみメッセージテキストに実行時に挿入するべきです。

- 同様の理由で、以下も上手くいきません。

```
printf("copied %d file%s", n, n!=1 ? "s" : "");
```

これは複数形がどのように形成されるかを決めてかかっているからです。もし、以下のようにして回避できたと考えると、

```
if (n==1)
    printf("copied 1 file");
else
    printf("copied %d files", n):
```

失望することになります。言語の中には、独特の規則によって2つ以上の形式になるものもあります。問題全体を回避するためこのメッセージを設計することが最善です。たとえば以下のようにします。

```
printf("number of copied files: %d", n);
```

適切に複数形を持つメッセージを構築したいと本当に思うのなら、これに対するサポートがありますが、多少厄介です。ereport()内の主たる、または詳細なエラーメッセージを生成する場合、以下のように書くことができます。

```
errmsg_plural("copied %d file",
              "copied %d files",
              n,
              n)
```

最初の引数は英文の単数形に適切な書式文字列で、二番目は英文の複数形に適切な書式文字列、そして三番目はどの複数形を使用するのかを決定する整数制御値です。引き続く引数はいつものように書式文字列毎に書式化されます。(通常、複数化制御値は書式化されるべき値の内の1つです。)英語ではnが1であるか、そうでないかのみ重要ですが、他の言語では数多くの複数書式が存在します。翻訳者にはグループとして2つの英文書式を参照し、nの実行時の値に基づいて選択される適切な1つでもって、複数の代替文字列を供給する機会があります。

errmsgあるいはerrdetail報告に直接行かない複数形メッセージが必要であれば、基礎となっている関数、ngettextを使用する必要があります。gettextのドキュメントを参照してください。

- メッセージをどのように他の出力と合わせる予定なのかなど翻訳者と何か連絡を取り合いたい場合、translatorで始まるコメントを最初に付けてどうなるかを知らせてください。以下のようにします。

```
/* translator: This message is not what it seems to be. */
```

これらのコメントはメッセージカタログにコピーされますので翻訳者は参照できます。

第55章 手続き言語ハンドラの作成

現在のコンパイル言語用「Version-1」インタフェース以外のある言語で作成された関数の呼び出しはすべて、特定の言語用の呼び出しハンドラを経由します（これには、ユーザ定義手続き言語で作成された関数、SQLで作成された関数が含まれます）。提供されたソーステキストを解釈するなどによって、関数の実行を意味のある方法で行うことは、呼び出しハンドラの責任です。本章では、どのように新しい手続き言語の呼び出しハンドラを作成できるかについて概要を示します。

手続き言語用の呼び出しハンドラは「通常」の関数で、Cなどのコンパイル言語で作成し、Version-1インタフェースを使用し、引数を取らずにlanguage_handlerを返すものとしてPostgreSQLに登録しなければなりません。この特殊な仮想型は、その関数を呼び出しハンドラとして識別し、SQLコマンド内で直接その関数が呼び出されることを防止します。C言語の呼び出し規約と動的ロードについては37.10を参照してください。

呼び出しハンドラは、他の関数と同じ方法で呼び出されます。引数値と呼び出された関数についての情報を含むFunctionCallInfoBaseData structのポインタを受け取り、Datum型の結果を返すもの（そして、SQLのNULLという結果を返そうとする場合に、FunctionCallInfoBaseData構造体のisnullフィールドを設定するかもしれないもの）と想定されています。呼び出しハンドラと通常の呼び出される関数との違いは、FunctionCallInfoBaseData構造体のflinfo->fn_oidに、呼び出しハンドラ自身ではなく、実際に呼び出される関数のOIDが含まれるという点です。呼び出しハンドラはこのフィールドを使用して、どの関数を呼び出すのかを決定しなければなりません。また、渡された引数リストは、呼び出しハンドラの宣言ではなく、目的とする関数の宣言に従うよう設定されています。

pg_procシステムカタログから関数のエントリを取り出し、呼び出される関数の引数と戻り値の型を解析するまでを呼び出しハンドラが行います。関数のCREATE FUNCTIONコマンドのAS句は、pg_procの行のprosrc列にあります。これは通常、手続き言語自体で作成されたソーステキストですが、理論上はファイルへのパス名や、何らかの呼び出しハンドラに詳細に何をすべきかを通知するものとすることもできます。

1つのSQL文で同じ関数が何回も呼び出されることがよくあります。呼び出しハンドラは、flinfo->fn_extraフィールドを使用して、呼び出す関数に関する情報を繰り返し検索することを防ぐことができます。これは初期状態ではNULLですが、呼び出しハンドラによって呼び出す関数の情報を指すように設定することもできます。その後の呼び出しでは、flinfo->fn_extraが非NULLであれば、それを使用して、情報検索の段階を省略することができます。呼び出しハンドラはflinfo->fn_extraが少なくとも現在の問い合わせの終了まで有効なメモリを指しているかどうかを確認しなければなりません。FmgrInfoデータ構造体は長く保持される可能性があるからです。この方法の1つとして、flinfo->fn_mcxtで指定されたメモリコンテキスト内に余分なデータを割り当てることです。このデータは通常FmgrInfo自身と同期間有効です。しかし、ハンドラはまた、長時間メモリコンテキストにあるものを使用するかどうかを選ぶこともできます。これにより関数定義情報を、問い合わせをまたいでキャッシュすることができます。

手続き言語関数がトリガとして呼び出された場合、引数は通常の方法では渡されず、FunctionCallInfoBaseDataのcontextフィールドが、普通の関数呼び出しのようにNULLにはならずに、TriggerData構造体を指しています。呼び出しハンドラは、手続き言語に対しトリガ情報を取り出す機構を提供しなければなりません。

以下は、Cで作成した手続き言語ハンドラの雛型です。

```
#include "postgres.h"
#include "executor/spi.h"
#include "commands/trigger.h"
```

```
#include "fmgr.h"
#include "access/heapam.h"
#include "utils/syscache.h"
#include "catalog/pg_proc.h"
#include "catalog/pg_type.h"

PG_MODULE_MAGIC;

PG_FUNCTION_INFO_V1(plsample_call_handler);

Datum
plsample_call_handler(PG_FUNCTION_ARGS)
{
    Datum          retval;

    if (CALLED_AS_TRIGGER(fcinfo))
    {
        /*

         * トリガ関数として呼び出された
         */
        TriggerData    *trigdata = (TriggerData *) fcinfo->context;

        retval = ...
    }
    else
    {
        /*

         * 関数として呼び出された
         */

        retval = ...
    }

    return retval;
}
```

数千行のコードを上のドットの代わりに追加するだけで、呼び出しハンドラを完成することができます。

ハンドラ関数を動的ロード可能なモジュールにコンパイル(37.10.5を参照してください)した後、以下のコマンドでサンプルの手続き言語を登録することができます。

```
CREATE FUNCTION plsample_call_handler() RETURNS language_handler
    AS 'filename'
    LANGUAGE C;
CREATE LANGUAGE plsample
```

```
    HANDLER plsample_call_handler;
```

最低限の手続き言語を作成する場合には呼び出しハンドラを提供するだけで十分ですが、他にも省略可能ですが、その言語の利用をより簡便にするために提供できる2つの関数があります。これらは有効性検証関数とインラインハンドラです。有効性検証関数を提供して、CREATE FUNCTION時に言語固有の検査を行うことができます。インラインハンドラを提供して、言語にDOコマンド経由の匿名コードブロック実行をサポートさせることができます。

有効性検証関数が手続き言語により提供される場合、oid型の単一パラメータを取る関数として宣言しなければなりません。有効性検証関数の結果は無視されます。そのためよくvoidを返すものと宣言されます。有効性検証関数はその手続き言語で関数を作成または置換するCREATE FUNCTIONの最後に呼び出されます。渡されるOIDは関数のpg_proc行のOIDです。有効性検証関数は通常の方法でこの行を取り出さなければならず、そして適切な検査を実行します。まずユーザがCREATE FUNCTIONで到達できない有効性検証関数への明示的な呼び出しを診断するため、CheckFunctionValidatorAccess()を呼び出します。典型的な検査として、さらに関数引数および結果の型がその言語でサポートされているかや関数本体がその言語において文法的に正しいかどうかを検証することが挙げられます。有効性検証関数がその関数に問題がないことを判定したら、単に戻ります。エラーがあることを判定したら、通常のereport()エラー報告機構を使用して報告しなければなりません。エラーを返すことで、強制的にトランザクションはロールバックされ、不正な関数定義がコミットされることを防ぎます。

有効性検証関数は通常check_function_bodiesパラメータを遵守しなければなりません。これが無効な場合、高価な、または文脈次第の検査を飛ばさなければなりません。言語がコンパイル時のコード実行を提供するのであれば、有効性検証関数はそのような実行を引き起こす検査を抑制しなければなりません。特にこのパラメータは、副作用や関数本体の他のデータベースオブジェクトへの依存を心配することなく手続き言語関数をロードできるように、pg_dumpにより無効にされます。(この仕様のため呼び出しハンドラは有効性検証関数が完全にその関数を検査したことを前提としてはいけません。有効性検証関数を持つ目的は、呼び出しハンドラが検査を省略できることではなく、明確なエラーがCREATE FUNCTIONコマンド内に存在する場合、それを即座にユーザに通知することです。) 厳密に何を検査すべきかの選択は主として有効性検査関数の裁量に委ねられていますが、check_function_bodiesが有効な場合にはCREATE FUNCTIONの中心となるコードは関数に関連づけられたSET句を実行するだけですので注意して下さい。そのため、その結果がGUCパラメータの影響を受ける検査は、ダンプをリロードする時の偽の失敗を避けるために、check_function_bodiesが無効な場合には確実に飛ばさなければなりません。

インラインハンドラが手続き言語により提供される場合、その関数はinternal型の単一パラメータを取るものとして宣言されなければなりません。インラインハンドラの結果は無視されます。そのためよくvoidを返すものと宣言されます。インラインハンドラは特定の手続き言語でDO文が実行された時に呼び出されます。実際に渡されるパラメータはInlineCodeBlock構造体のポインタです。ここにはDO文のパラメータ、具体的には実行される匿名コードブロックのテキスト、に関する情報が含まれています。インラインハンドラはこのコードを実行し、戻らなければなりません。

簡単なCREATE EXTENSIONコマンドで言語をインストールすることが十分にできるように、これらの関数宣言とCREATE LANGUAGEコマンド自身を拡張としてまとめることを勧めます。拡張の作成方法については37.17を参照してください。

独自の言語ハンドラを作成する際、標準配布物に含まれる手続き言語は優れたリファレンスです。ソースツリーのsrc/plサブディレクトリを調べてください。CREATE LANGUAGEマニュアルページもまた有用な情報を含みます。

第56章 外部データラッパの作成

外部テーブルへの全ての操作は、コアサーバから呼び出される関数のセットで構成される、外部データラッパで処理されます。外部データラッパは、リモートデータソースからデータを取り出し、そのデータをPostgreSQLエクゼキュータに返却することを担当します。外部テーブルの更新をサポートする場合、ラッパはそれも扱わなければなりません。本章では、新しい外部データラッパを作成する方法の概要を示します。

独自の外部データラッパを作成する際、標準配布物に含まれているものは優れたリファレンスです。ソースツリーのcontribサブディレクトリを調べてください。CREATE FOREIGN DATA WRAPPERマニュアルページにも有用な情報があります。

> **注記**
>
> 標準SQLでは外部データラッパを作成するインタフェースを定義しています。しかしながら、PostgreSQLに適応させる労力が大きく、また標準のAPIが広く採用されているわけでもないので、PostgreSQLはそのAPIを実装していません。

56.1. 外部データラッパ関数

FDWの作者は、ハンドラ関数と、オプションで検証関数を実装する必要があります。両関数とも、version-1インタフェースを使用して、Cなどのコンパイル言語で作成しなければなりません。C言語の呼び出し規約と動的ロードについては37.10を参照してください。

ハンドラ関数は単に、プランナやエクゼキュータ、様々なメンテナンスコマンドから呼び出されるコールバック関数ポインタを含む構造体を返します。FDWを作成するための労力のほとんどは、これらのコールバック関数を実装することに費やされます。ハンドラ関数は、引数を取らず特殊な仮想型であるfdw_handlerを返す関数としてPostgreSQLに登録しなければなりません。コールバック関数は通常のC言語関数で、SQLレベルでは参照も呼び出しもできません。コールバック関数の説明は56.2にあります。

検証関数は、そのラッパを使用する外部サーバ、ユーザマッピング、外部テーブルだけでなく、外部データラッパ自身のCREATEやALTERといったコマンドで指定されたオプションの妥当性の検証を担当します。検証関数は、検証するオプションを含むtext配列と、オプションを関連付けるオブジェクトの種類を表すOID（そのオブジェクトが格納されるシステムカタログのOIDで次のいずれか、ForeignDataWrapperRelationId、ForeignServerRelationId、UserMappingRelationId、ForeignTableRelationId）という二つの引数を取るものとして登録しなければなりません。検証関数が指定されなかった場合、オブジェクト作成時やオブジェクト変更時にオプションはチェックされません。

56.2. 外部データラッパのコールバックルーチン

FDWハンドラ関数は、以下で説明するコールバックの関数ポインタを含む、pallocされたFdwRoutine構造体を返します。スキャンに関連した関数は必須で、それ以外は省略可能です。

FdwRoutine構造体はsrc/include/foreign/fdwapi.hで宣言されていますので、追加情報はそちらを参照してください。

56.2.1. 外部テーブルスキャンのためのFDWルーチン

```
void
GetForeignRelSize(PlannerInfo *root,
                  RelOptInfo *baserel,
                  Oid foreigntableid);
```

外部テーブルのリレーションサイズ見積もりを取得します。この関数は、ある外部テーブルをスキャンするクエリのプラン作成の開始時に呼び出されます。rootはそのクエリに関するプランナのグローバル情報です。baserelはそのテーブルに関するプランナの情報です。そして、foreigntableidはその外部テーブルのpg_class OIDです。(foreigntableidはプランナデータ構造体からも取得できますが、手間を省くために明示的に渡されます。)

この関数は、検索条件によるフィルタリングも考慮に入れた、そのテーブルスキャンが返すと見込まれる件数にbaserel->rowsを更新するべきです。baserel->rowsの初期値は固定のデフォルト見積もりなので、可能な限り置き換えられるべきです。この関数は、行の幅のよりよい見積もりを計算できるのであれば、baserel->widthを更新することも選択出来ます。(初期値は列の型と最後に実行されたANALYZEから計測された平均列幅に基づいています。) また、外部テーブルの総行数の見積もりをより正しく計算できる場合、この関数は、baserel->tuplesを更新しても構いません。(初期値はpg_class.reltuplesで、最後に実行されたANALYZEによって確認された総行数です。)

追加情報については56.4を参照してください。

```
void
GetForeignPaths(PlannerInfo *root,
                RelOptInfo *baserel,
                Oid foreigntableid);
```

外部テーブル対するスキャンとして可能なアクセスパスを作成します。この関数はクエリのプラン作成中に呼び出されます。引数は、先に呼ばれているGetForeignRelSizeと同じです。

この関数は、少なくとも一つのアクセスパス(ForeignPathノード)を作成して、それぞれのパスをbaserel->pathlistに追加するためにadd_pathを呼ばなければなりません。ForeignPathノードを構築するにはcreate_foreignscan_pathを使うことが推奨されています。この関数は、たとえばソート済みの結果を表現する有効なpathkeysを持つパスのような複数のアクセスパスを作成することが出来ます。それぞれのアクセスパスはコスト見積もりを含まねばならず、また意図した特定のスキャン方式を識別するのに必要なFDW固有の情報を持つことが出来ます。

追加情報については56.4を参照してください。

```
ForeignScan *
GetForeignPlan(PlannerInfo *root,
               RelOptInfo *baserel,
               Oid foreigntableid,
               ForeignPath *best_path,
               List *tlist,
```

```
                List *scan_clauses,
                Plan *outer_plan);
```

選択された外部アクセスパスからForeignScanプランノードを作成します。この関数はクエリプラン作成の最後に呼び出されます。引数は、GetForeignRelSizeと同じものに、選択されたForeignPath(事前にGetForeignPaths、GetForeignJoinPathsまたはGetForeignUpperPathsによって作成されたもの)、そのプランノードによって出力されるターゲットリスト、そのプランノードで強制される条件句、およびRecheckForeignScanが実行する再検査で使用されるForeignScanの外側のサブプランが追加されます。(パスがベースリレーションではなく結合のためのものの場合、foreigntableidはInvalidOidになります。)

この関数はForeignScanプランノードを作成して返さなければなりません。ForeignScanノードを構築するにはmake_foreignscanを使うことが推奨されています。

追加情報については56.4を参照してください。

```
void
BeginForeignScan(ForeignScanState *node,
                 int eflags);
```

外部テーブルスキャンの実行を開始します。この関数はエクゼキュータの起動中に呼び出されます。スキャンを開始できるようになる前に、あらゆる必要な初期化を実行するべきですが、実際のスキャンの実行を始めるべきではありません(それは最初のIterateForeignScan呼び出しにおいて行われるべきです)。ForeignScanStateノードは作成されていますが、そのfdw_stateフィールドはNULLのままです。スキャンするテーブルの情報は、ForeignScanStateノード(特に、その先にあるGetForeignPlanから提供されたFDWプライベート情報を含む、ForeignScanプランノード)を通じてアクセス可能です。eflagsは、このプランノードに関するエクゼキュータの操作モードを表すフラグビットを含みます。

(eflags & EXEC_FLAG_EXPLAIN_ONLY)が真の場合、この関数は外部に見える処理を実行すべきではないことに注意してください。ExplainForeignScanやEndForeignScan用にノード状態を有効にするのに必要とされる最小限のことだけをすべきです。

```
TupleTableSlot *
IterateForeignScan(ForeignScanState *node);
```

外部ソースから一行を取り出して、それをタプルテーブルスロットに入れて返します(この用途にはノードのScanTupleSlotを使うべきです)。利用可能な行がない場合は、NULLを返します。タプルテーブルスロット機構を使うと、物理タプルと仮想タプルのどちらでも返せます。ほとんどの場合、パフォーマンスの観点から後者を選ぶのが良いでしょう。この関数は、呼出しごとにリセットされる短命なメモリコンテキスト内で呼び出されることに注意してください。より長命なストレージが必要な場合は、BeginForeignScanでメモリコンテキストを作成するか、ノードのEStateに含まれるes_query_cxtを使用してください。

返される行は、ターゲットリストfdw_scan_tlistが提供されたなら、それとマッチしなければならず、提供されていない場合はスキャンされている外部テーブルの行型とマッチしなければなりません。不要な列を取り出さないように最適化することを選ぶなら、それらの列の位置にNULLを入れるか、あるいはそれらの列を除いたfdw_scan_tlistリストを生成するべきです。

PostgreSQLのエクゼキュータは返された行が外部テーブルに定義された制約に違反しているかどうかは気にしません。しかし、プランナはそれに着目するので、宣言された制約に反する行が外部テーブル上にあっ

た場合に、不正な最適化をするかもしれません。ユーザが制約が成り立つと宣言したのに制約に違反した場合は(データ型が一致しなかった場合にする必要があるのと同様に)エラーを発生させるのが適切でしょう。

```
void
ReScanForeignScan(ForeignScanState *node);
```

先頭からスキャンを再開します。スキャンが依存するいずれかのパラメータが値を変更しているかもしれないので、新しいスキャンが必ずしも厳密に同じ行を返すとは限らないことに注意してください。

```
void
EndForeignScan(ForeignScanState *node);
```

スキャンを終了しリソースを解放します。通常、pallocされたメモリを解放することは重要ではありませんが、たとえば開いたファイルやリモートサーバへの接続などはクリーンアップするべきです。

56.2.2. 外部テーブルの結合をスキャンするためのFDWルーチン

FDWが外部テーブルの結合を(両方のテーブルのデータをフェッチして、ローカルで結合するのでなく)リモートで実行することをサポートする場合、次のコールバック関数を提供します。

```
void
GetForeignJoinPaths(PlannerInfo *root,
                    RelOptInfo *joinrel,
                    RelOptInfo *outerrel,
                    RelOptInfo *innerrel,
                    JoinType jointype,
                    JoinPathExtraData *extra);
```

同じ外部サーバにある2つ(またはそれ以上)の外部テーブルの結合のための可能なアクセスパスを作成します。このオプション関数は、問い合わせの計画時に呼び出されます。GetForeignPathsと同じく、この関数は提供されたjoinrelのためのForeignPathパスを生成し(そのためにcreate_foreign_join_pathを使用します)、add_pathを呼んで、それらのパスを結合のために考慮されるパスの集合に追加します。しかし、GetForeignPathsとは異なり、この関数が少なくとも1つのパスの作成に成功することは必要ではありません。なぜなら、ローカルの結合を含んだパスはいつでも可能だからです。

この関数は、同じ結合のリレーションに対して、内側と外側のリレーションの異なる組み合わせで繰り返し呼び出されることに注意して下さい。同じ作業の繰り返しを最小化することはFDWの責任です。

ForeignPathパスが結合のために選択されると、それは結合プロセス全体を代表することになり、構成テーブルとその関連の結合のために生成されたパスは使われなくなります。結合パスの以降の処理は、単一の外部テーブルをスキャンするパスとほぼ同様に進みます。1つの相違点は、結果として作られるForeignScan計画ノードのscanrelidが0にセットされるべき、ということで、これはそれが表現する単一のリレーションがないためです。その代わりに、ForeignScanノードのfs_relidsフィールドが結合されるリレーションの集合を表します。(後者のフィールドはコアのプランナのコードによって自動的にセットされるので、

FDWによって設定される必要はありません。) 他の相違点は、リモートの結合についての列リストがシステムカタログにはないため、FDWはfdw_scan_tlistに適切なTargetEntryノードのリストを入れて、実行時に返されるタプル内の列の集合を表すようにしなければならないということです。

追加情報については56.4を参照してください。

56.2.3. スキャン/結合後の処理をプラン生成するためのFDWルーチン

FDWがリモート集約など、リモートでのスキャン/結合後の処理をサポートする場合、次のコールバック関数を提供します。

```
void
GetForeignUpperPaths(PlannerInfo *root,
                     UpperRelationKind stage,
                     RelOptInfo *input_rel,
                     RelOptInfo *output_rel,
                     void *extra);
```

上位リレーション処理のための、ありうるアクセスパスを作成します。上位リレーションはプランナ用語で、ウィンドウ関数、ソート、テーブル更新など、全てのスキャン/結合後の問い合わせのことです。この省略可能な関数は問い合わせのプラン作成時に呼ばれます。今のところ、これは問い合わせに含まれる全てのベースリレーションが同じFDWに属する場合だけ呼ばれます。この関数では、FDWがどのようにリモートで実行するか分かっている全てのスキャン/結合後の処理にForeignPathパスを生成し(そのためにcreate_foreign_upper_pathを使用します)、それらパスを指定された上位リレーションに加えるためにadd_pathを呼び出してください。GetForeignJoinPathsの時と同様に、この関数が何らかのパス作成に成功する必要はありません。なぜなら、ローカル処理を含んでいるパスはいつでも可能だからです。

stageパラメータはどのスキャン/結合後の処理が現在考慮されているかを定めます。output_relは本処理の計算方法をあらわすパスを受け取るであろう上位リレーションで、input_relは本処理への入力をあらわすリレーションです。extraパラメータは追加の詳細を指定し、今のところUPPERREL_PARTIAL_GROUP_AGGとUPPERREL_GROUP_AGGに対して指定できて、この場合はGroupPathExtraData構造体へのポインタです。さらにUPPERREL_FINALに対しても指定できて、この場合はFinalPathExtraData構造体へのポインタです。(注意:これらの処理は外部で実行されると考えられるため、output_relに加えられるForeignPathパスは、一般的にinput_relのパスへの直接の依存を全く持たないでしょう。しかしながら、手前の処理段階のために以前に生成されたパスを検査することは、冗長なプラン作成活動を回避するのに役立ちます。)

追加情報については56.4を参照してください。

56.2.4. 外部テーブル更新のためのFDWルーチン

もしFDWが更新可能な外部テーブルをサポートする場合、FDWのニーズと能力に応じて、以下のコールバック関数の一部または全てを提供する必要があります。

```
void
AddForeignUpdateTargets(Query *parsetree,
                        RangeTblEntry *target_rte,
                        Relation target_relation);
```

UPDATEとDELETEの操作は、テーブルスキャン関数によって事前にフェッチされた行に対して実行されます。FDWは、更新や削除の対象行を厳密に識別できるように行IDや主キー列の値といった追加情報を必要とするかもしれません。それをサポートするために、この関数はUPDATEやDELETEの間に外部テーブルから取得される列のリストに追加の隠された(または「ジャンクの」)ターゲット列を追加することができます。

これを実行するには、フェッチする追加の値の式を含むTargetEntryエントリをparsetree->targetListに追加します。それぞれのエントリはresjunk = trueとマークされなければならず、また実行時にエントリを識別できる異なるresnameを持つ必要があります。コアシステムがそのような名前のジャンク列を生成できるように、ctidNやwholerow、wholerowNと一致する名前は使用しないでください。

追加の式が単純な(Var型の)変数よりも複雑な場合は、それらをターゲットリストに追加する前にeval_const_expressionsを実行する必要があります。

この関数はプラン生成中に呼び出されますが、提供される情報は他のプラン生成ルーチンで使用できる情報とは少し異なります。parsetreeはUPDATEやDELETEコマンドの解析ツリーで、target_rteとtarget_relationは対象の外部テーブルを表します。

もしAddForeignUpdateTargetsポインタがNULLに設定されている場合は、追加のターゲット式は追加されません。(FDWが行を識別するのに不変の主キーに依存するのであればUPDATEは依然として実現可能かもしれませんが、DELETE操作を実装することは不可能になるでしょう。)

```
List *
PlanForeignModify(PlannerInfo *root,
                  ModifyTable *plan,
                  Index resultRelation,
                  int subplan_index);
```

外部テーブルに対する挿入、更新、削除に必要となる、追加のプラン生成アクションを実行します。この関数は、更新処理を実行するModifyTableプランノードに追加されるFDW固有の情報を生成します。この固有情報はList形式でなければならず、また実行段階の間にBeginForeignModifyに渡されます。

rootはそのクエリに関するプランナのグローバル情報です。planはfdwPrivListsフィールドを除いて完成しているModifyTableプランノードです。resultRelationは対象の外部テーブルをレンジテーブルの添字で識別します。subplan_indexはModifyTableプランノードの対象がどれであるかを0始まりで識別します。この情報はplan->plansなどのplanの下位構造を指定したい場合に使用してください。

追加情報は56.4を参照してください。

もしPlanForeignModifyポインタがNULLに設定されている場合は、追加のプラン作成時処理は実行されず、BeginForeignModifyに渡されるfdw_privateリストはNILになります。

```
void
BeginForeignModify(ModifyTableState *mtstate,
```

```
                    ResultRelInfo *rinfo,
                    List *fdw_private,
                    int subplan_index,
                    int eflags);
```

外部テーブルへの変更操作の実行を開始します。このルーチンはエクゼキュータの起動中に呼び出されます。実際のテーブル変更に先立って必要なあらゆる初期化処理を実行する必要があります。その後、各タプルが挿入、更新、削除されるようにExecForeignInsert、ExecForeignUpdate、ExecForeignDeleteのいずれかが呼ばれます。

mtstateは実行されているModifyTableプランノード全体の状態です。プランに関する全般的なデータと実行状態はこの構造体経由で利用可能です。rinfoは対象の外部テーブルを表すResultRelInfo構造体です。(ResultRelInfoのri_FdwStateフィールドはこの操作で必要となる固有の状態をFDWが格納するのに利用できます。) fdw_privateは、もしあればPlanForeignModifyで生成された固有データを含みます。subplan_indexは、これがModifyTableプランノードのどのターゲットであるかを識別します。eflagsは、このプランノードに関するエクゼキュータの操作モードを表すフラグビットを含みます。

(eflags & EXEC_FLAG_EXPLAIN_ONLY)が真の場合、この関数は外部に見える処理を実行すべきではないことに注意してください。ExplainForeignModifyやEndForeignModify用にノード状態を有効にするのに必要な最小限のことだけを実行するべきです。

もしBeginForeignModifyポインタがNULLに設定されている場合は、エクゼキュータ起動時には追加処理は何も実行されません。

```
TupleTableSlot *
ExecForeignInsert(EState *estate,
                  ResultRelInfo *rinfo,
                  TupleTableSlot *slot,
                  TupleTableSlot *planSlot);
```

外部テーブルにタプルを一つ挿入します。estateはそのクエリのグローバルな実行状態です。rinfoは対象の外部テーブルを表すResultRelInfo構造体です。slotには挿入されるタプルが含まれます。その行型定義は外部テーブルと一致します。planSlotにはModifyTableプランノードのサブプランが生成したタプルが含まれます。追加の「ジャンク」列を含みうる点において、slotとは異なります。(planSlotは一般的にINSERTのケースにおいてはそれほど意味を持ちませんが、完全性のために提供されます。)

戻り値は実際に挿入されたデータ(例えばトリガー処理の結果などにより、提供されたデータとは異なるかもしれません)を含むスロットか、または(こちらも一般的にトリガーの結果)実際には挿入されなかった場合はNULLです。渡されたslotはこの用途に再利用可能です。

返却されたスロット内のデータはINSERT文がRETURNING句を持っているか、WITH CHECK OPTIONを伴うビューに影響を及ぼす場合、もしくは、外部テーブルがAFTER ROWトリガを持っていた場合にのみ使われます。トリガは全列を必要としますが、FDWはRETURNING句やWITH CHECK OPTIONの制約の内容に応じて、返却する列を一部にするかすべてにするかを最適化する余地があります。それとは関係なく、処理成功を表すためになんらかのスロットは返却しなければなりません。さもないと、報告されるクエリの結果行数が誤った値になってしまいます。

もしExecForeignInsertポインタがNULLに設定されている場合は、外部テーブルへの挿入の試みはエラーメッセージとともに失敗します。

この関数は外部テーブルパーティションに転送対象のタプルを挿入する際、あるいはCOPY FROMを外部テーブルに対して実行する際にも呼び出されることに注意してください。COPY FROMの場合、INSERTとはこの関数の呼び出され方は異なります。FDWがそれをサポートすることを可能にする以下で説明するコールバック関数をご覧ください。

```
TupleTableSlot *
ExecForeignUpdate(EState *estate,
                  ResultRelInfo *rinfo,
                  TupleTableSlot *slot,
                  TupleTableSlot *planSlot);
```

外部テーブル内のタプルを一つ更新します。estateはそのクエリのグローバルな実行状態です。rinfoは対象の外部テーブルを表すResultRelInfo構造体です。slotにはタプルの新しいデータが含まれます。その行型定義は外部テーブルと一致します。planSlotにはModifyTableプランノードのサブプランが生成したタプルが含まれます。追加の「ジャンク」列を含みうる点において、slotとは異なります。実際、AddForeignUpdateTargetsが要求するジャンク列はこのスロットから利用可能です。

戻り値は実際に更新されたデータ(例えばトリガー処理の結果などにより、提供されたデータとは異なるかもしれません)を含むスロットか、または(こちらも一般的にトリガーの結果)実際には更新されなかった場合はNULLです。渡されたslotはこの用途に再利用可能です。

返却されたスロット内のデータはUPDATE文がRETURNING句を持っているか、WITH CHECK OPTIONを伴うビューに影響を及ぼす場合、もしくは外部テーブルがAFTER ROWトリガを持っていた場合にのみ使われます。トリガは全列を必要としますが、FDWはRETURNING句やWITH CHECK OPTIONの制約の内容に応じて返却する列を一部にするか全てにするかを最適化する余地があります。それとは関係なく、処理成功を表すためになんらかのスロットは返却しなければなりません。さもないと、報告されるクエリの結果行数が誤った値になってしまいます。

もしExecForeignUpdateポインタがNULLに設定されている場合は、外部テーブルへの更新の試みはエラーメッセージとともに失敗します。

```
TupleTableSlot *
ExecForeignDelete(EState *estate,
                  ResultRelInfo *rinfo,
                  TupleTableSlot *slot,
                  TupleTableSlot *planSlot);
```

外部テーブルからタプルを一つ削除します。estateはそのクエリのグローバルな実行状態です。rinfoは対象の外部テーブルを表すResultRelInfo構造体です。slotにはタプルの新しいデータが含まれます。その行型定義は外部テーブルと一致します。planSlotにはModifyTableプランノードのサブプランが生成したタプルが含まれます。実際、AddForeignUpdateTargetsが要求するジャンク列はこのスロットが運びます。ジャンク列は削除されるタプルを識別するために使用しなければなりません。

戻り値は実際に削除されたデータを含むスロットか、または(一般的にトリガーの結果)実際には削除されなかった場合はNULLです。渡されたslotは返却するタプルを保持する用途に利用可能です。

返却されたスロット内のデータはDELETEクエリがRETURNING句を持っていた場合もしくは外部テーブルがAFTER ROWトリガを持っていた場合にのみ使われます。トリガは全列を必要としますが、FDW

はRETURNING句の内容に応じて返却する列を一部にするか全てにするかを最適化する余地があります。それとは関係なく、処理成功を表すためになんらかのスロットは返却しなければなりません。さもないと、報告されるクエリの結果行数が誤った値になってしまいます。

もしExecForeignDeleteポインタがNULLに設定されている場合は、外部テーブルからの削除の試みはエラーメッセージとともに失敗します。

```
void
EndForeignModify(EState *estate,
                 ResultRelInfo *rinfo);
```

テーブル更新を終えてリソースを解放します。通常、pallocされたメモリを解放することは重要ではありませんが、たとえば開いたファイルやリモートサーバへの接続などはクリーンアップするべきです。

もしEndForeignModifyポインタがNULLに設定されている場合は、エクゼキュータ終了時には追加処理は何も実行されません。

INSERTあるいはCOPY FROMでパーティション化テーブルに挿入されたタプルはパーティションに転送されます。FDWが外部テーブルのパーティションへの転送をサポートしているなら、以下のコールバック関数も提供すべきです。これらの関数は、外部テーブルでCOPY FROMが実行された時に呼び出されます。

```
void
BeginForeignInsert(ModifyTableState *mtstate,
                   ResultRelInfo *rinfo);
```

外部テーブルへの挿入操作の実行を開始します。このルーチンは、タプル転送のためにパーティションが選択された場合か、COPY FROMコマンドでターゲットが指定された場合に、最初の行が外部テーブルに挿入される直前に呼び出されます。この関数は、実際の挿入に先立つすべての必要な初期化を実行すべきです。続いて、ExecForeignInsertが外部テーブルに挿入される個々のタプル毎に呼び出されます。

mtstateは、実行中のModifyTableプランノードの全体的な状態です。プランのグローバルデータと実行状態がこの構造体を通じて得られます。rinfoはResultRelInfo構造体で、ターゲットの外部テーブルを記述します。(この操作中に必要なFDWのプライベート状態を保存するためにResultRelInfoのri_FdwStateフィールドが利用可能です。)

この関数がCOPY FROMコマンドで呼ばれると、外部テーブルがタプル転送で選択された対象なのか、あるいはコマンドがターゲットを指定したのかに関わらず、mtstate中のプランに関係するグローバルデータは提供されず、次に個々の挿入されるタプルに対して呼び出されるExecForeignInsertのplanSlotパラメータはNULLとなります。

BeginForeignInsertポインターがNULLなら、初期化処理は実施されません。

FDWが外部テーブルパーティションのタプル転送をサポートしていないか、または外部テーブルに対してCOPY FROMの実行をサポートしていないか、あるいはその両方なら、この関数あるいは以後呼ばれたExecForeignInsertは、必ず必要なだけエラーを引き起こします。

```
void
EndForeignInsert(EState *estate,
```

```
                ResultRelInfo *rinfo);
```

挿入操作を終了してリソースを解放します。通常、pallocされたメモリを解放することは重要ではありませんが、たとえば開いたファイルやリモートサーバへの接続などはクリーンアップするべきです。

EndForeignInsertポインターがNULLなら、終了処理は実施されません。

```
int
IsForeignRelUpdatable(Relation rel);
```

指定された外部テーブルがどの更新処理をサポートしているかを報告します。戻り値は、その外部テーブルがサポートする操作を表すルールイベント番号のビットマスクである必要があります。UPDATE用の(1 << CMD_UPDATE) = 4、INSERT用の(1 << CMD_INSERT) = 8、DELETE用の(1 << CMD_DELETE) = 16といったCmdType列挙値を使います。

もしIsForeignRelUpdatableポインタがNULLに設定されていると、外部テーブルはExecForeignInsert、ExecForeignUpdate、ExecForeignDeleteを提供していると、それぞれ挿入、更新、削除をサポートしていると判断します。この関数は、FDWが一部のテーブルについてのみ更新をサポートする場合にのみ必要です。(そのような場合でも、この関数でチェックする代わりにクエリ実行関数でエラーにしても構いません。しかしながら、この関数はinformation_schemaのビューの表示で更新可否を判定するのに使用されます。)

外部テーブルへの挿入、更新、削除は、代替インタフェース一式を実装することで最適化できます。通常の挿入、更新、削除のインタフェースは行をリモートサーバから取得し、その後、それらの行を一つずつ変更します。一部の場合にはこの一行ごとのやり方は必要ですが、非効率とも言えます。外部サーバについて行が本当はそれらを引き出すことなしに変更されるべきと判断できて、操作に影響を与える仕組み(行レベルのローカルトリガ、格納生成列、あるいは、親ビューからのWITH CHECK OPTIONの制約)が無いならば、操作全体がリモートサーバで実行されるように計画することができます。以下に示すインタフェースはこれを可能にします。

```
bool
PlanDirectModify(PlannerInfo *root,
                 ModifyTable *plan,
                 Index resultRelation,
                 int subplan_index);
```

リモートサーバ上で直接変更を実行することが安全かを判断します。そうであれば、そのために必要なプラン作成の動作を実行した後にtrueを返します。さもなくば、falseを返します。この省略可能な関数は問い合わせのプラン作成時に呼ばれます。この関数が成功すると、BeginDirectModify、IterateDirectModify、EndDirectModifyが実行段階で代わりに呼び出されます。成功しなければ、テーブルの変更は前述のテーブル更新関数を使って実行されます。パラメータはPlanForeignModifyに対するものと同じです。

リモートサーバで直接変更を実行するには、本関数は対象サブプランをリモートサーバ上で直接変更するForeignScanプランノードで書き換えしなければなりません。ForeignScanのoperationフィールドにはCmdType列挙値を適切に、すなわち、UPDATEにはCMD_UPDATE、INSERTにはCMD_INSERT、DELETEにはCMD_DELETEを設定しなければいけません。

追加情報は56.4を参照してください。

PlanDirectModifyポインタにNULLが設定されている場合、リモートサーバでの直接変更の実行は試みられません。

```
void
BeginDirectModify(ForeignScanState *node,
                  int eflags);
```

リモートサーバでの直接変更を実行する準備をします。この関数はエクゼキュータが開始するときに呼び出されます。この関数は(最初のIterateDirectModify呼び出しで実行されるであろう)直接変更より前に必要とされる全ての初期化を実行するべきです。ForeignScanStateノードはすでに作られていますが、fdw_stateがフィールドはまだNULLです。変更するテーブルに関する情報はForeignScanStateノードを通して(具体的にはPlanDirectModifyで提供されるFDWプライベート情報を含む、元となるForeignScanプランノードから)入手可能です。eflagsは、このプランノードに関するエクゼキュータの操作モードを表すフラグビットを含みます。

(eflags & EXEC_FLAG_EXPLAIN_ONLY)が真の場合、この関数は外部に見える処理を実行すべきではないことに注意してください。ExplainDirectModifyやEndDirectModify用にノード状態を有効にするのに必要な最小限のことだけを実行するべきです。

BeginDirectModifyポインタがNULLに設定されている場合、リモートサーバでの直接変更の実行は試みられません。

```
TupleTableSlot *
IterateDirectModify(ForeignScanState *node);
```

INSERT、UPDATE、または、DELETEの問い合わせがRETURNING句を持たないときには、リモートサーバでの直接変更の後、単にNULLが返ります。問い合わせがRETURNING句を持つときには、RETURNING計算に必要なデータを含む結果を一つ取り出し、タプルテーブルスロットでそれを返します(この用途にはノードのScanTupleSlotを使うべきです)。実際に挿入、更新、削除されたデータはノードのEStateのes_result_relation_info->ri_projectReturning->pi_exprContext->ecxt_scantupleに格納されなければなりません。有効な行がそれ以上なければNULLを返します。これは呼び出しの間でリセットされる寿命の短いメモリコンテキストで呼び出されることに注意してください。より長命な格納場所を必要とするなら、BeginDirectModifyでメモリコンテキストを作るか、ノードのEStateのes_query_cxtを使ってください。

返される行は、ターゲットリストfdw_scan_tlistが提供されたなら、それとマッチしなければならず、提供されていない場合は変更されている外部テーブルの行型とマッチしなければなりません。RETURNING計算に不要な列を取り出さないように最適化することを選ぶなら、それらの列の位置にNULLを入れるか、あるいはそれらの列を除いたfdw_scan_tlistリストを生成するべきです。

問い合わせが句をもつかどうかによらず、問い合わせが報告する行数はFDW自身によって増加されなければなりません。問い合わせが句を持たないときも、FDWはEXPLAIN ANALYZEの場合のForeignScanState nodeむけに行数を増加させなければなりません。

IterateDirectModifyポインタがNULLに設定されている場合、リモートサーバでの直接変更の実行は試みられません。

```
void
```

```
EndDirectModify(ForeignScanState *node);
```

リモートサーバでの直接変更の後、クリーンアップします。通常、pallocされたメモリを解放することは重要ではありませんが、たとえば開いたファイルやリモートサーバへの接続などはクリーンアップするべきです。

EndDirectModifyポインタがNULLに設定されている場合、リモートサーバでの直接変更の実行は試みられません。

56.2.5. 行ロックのためのFDWルーチン

FDWが(56.5で説明される)遅延行ロックをサポートする場合は、以下のコールバック関数を提供する必要があります。

```
RowMarkType
GetForeignRowMarkType(RangeTblEntry *rte,
                      LockClauseStrength strength);
```

行の印付けでどのオプションを外部テーブルに使うかを報告します。rteはテーブルのRangeTblEntryノードで、strengthは関連するFOR UPDATE/SHARE句があれば、それが要求するロックの強さを表します。その結果は、RowMarkType列挙型のメンバーでなければなりません。

この関数はUPDATE、DELETE、SELECT FOR UPDATE/SHAREの問い合わせに現れ、かつUPDATEあるいはDELETEの対象ではない各外部テーブルについて、問い合わせの計画時に呼び出されます。

GetForeignRowMarkTypeのポインタがNULLに設定されていると、必ずROW_MARK_COPYオプションが使われます。(これはRefetchForeignRowが決して呼び出されないので、それを提供する必要もない、ということを意味します。)

さらなる情報については56.5を参照してください。

```
void
RefetchForeignRow(EState *estate,
                  ExecRowMark *erm,
                  Datum rowid,
                  TupleTableSlot *slot,
                  bool *updated);
```

必要ならロックした後で、外部テーブルから1つのタプルスロットを再フェッチします。estateは問い合わせのグローバルな実行状態です。ermは対象の外部テーブルおよび獲得する行ロックの種別(あれば)を記述するExecRowMark構造体です。rowidはフェッチするタプルを特定するものです。slotは呼び出しで役立つ内容は含みませんが、返されたタプルを保持するために使用できます。updatedは出力パラメータです。

この関数はタプルを与えられたスロットに格納するか、あるいは行ロックが取得できなければタプルをクリアします。獲得する行ロックの種別はerm->markTypeで指定されます。この値は事前にGetForeignRowMarkTypeから返されたものです。(ROW_MARK_REFERENCEは行のロックを獲得せずに、単にタプルを再フェッチすることを意味し、また、ROW_MARK_COPYはこのルーチンで使われることはありません。)

そして、*updatedはフェッチしたタプルが、以前に取得したものと同じではなく、更新されたバージョンであったときにtrueにセットされます。(どちらなのかFDWが判断できない場合は、trueを返すことが推奨されます)。

デフォルトでは、行ロックの獲得に失敗したときはエラーを発生させるべきであることに注意してください。空スロットを返すのが適切なのは、erm->waitPolicyでSKIP LOCKEDオプションが指定されている場合だけです。

rowidは、再フェッチする行を以前読んだ時のctid値です。rowid値はDatumとして渡されますが、現在はtidにしかなりません。将来は行ID以外のデータ型が可能になることを期待して、関数APIとすることが選択されました。

RefetchForeignRowポインタがNULLの場合、行を再フェッチする試みは失敗し、エラーメッセージを発行します。

さらなる情報については56.5を参照してください。

```
bool
RecheckForeignScan(ForeignScanState *node,
                   TupleTableSlot *slot);
```

以前に戻されたタプルが、関連するスキャンおよび結合の制約とまだ一致しているか再検査し、更新されたバージョンのタプルを提供する場合もあります。結合のプッシュダウンを行わない外部データラッパでは、通常は、これをNULLにセットし、代わりにfdw_recheck_qualsを適切にセットする方が便利でしょう。しかし、外部結合をプッシュダウンする場合、すべてのベーステーブルに関する検査を結果のタプルに適用するだけでは、たとえすべての必要な属性がそこにあったとしても十分ではありません。なぜなら一部の制約が一致しないことで、タプルが戻されない代わりに、一部の属性がNULLになってしまうかもしれないからです。RecheckForeignScan制約を再検査し、それが依然として満たされていれば真を、そうでなければ偽を返すことができます。それだけでなく、置換されたタプルを提供されたスロットに格納することもできます。

結合のプッシュダウンを実装する場合、外部データラッパは通常、再検査のためだけに使用される代替のローカル結合プランを構築します。これがForeignScanの外部サブプランとなります。再検査が必要な時は、このサブプランを実行して、結果のタプルをスロットに格納することができます。どのベーステーブルも最大で1行しか返さないので、このプランは効率的である必要はありません。例えば、すべての結合をネステッドループで実装することもできます。関数GetExistingLocalJoinPathは、存在するパスから代替ローカルの結合プランとして使用可能な適当なローカル結合パスを検索するのに使われるかもしれません。GetExistingLocalJoinPathは指定された結合リレーションのパスリストのパラメータ化されていないパスを検索します。(そのようなパスが見つからなかった場合はNULLを返します。この場合、外部データラッパはそれ自身によりローカルパスを構築するかもしれず、あるいは、その結合むけのアクセスパスを作らないことを選択するかもしれません。)

56.2.6. EXPLAINのためのFDWルーチン

```
void
ExplainForeignScan(ForeignScanState *node,
                   ExplainState *es);
```

外部テーブルスキャンの追加のEXPLAIN出力を表示します。EXPLAIN出力にフィールドを追加するためにExplainPropertyTextや関連する関数を呼び出すことができます。esの中のフラグフィールドは何を表示するかを決めるのに使用できます。また、EXPLAIN ANALYZEの場合には、実行時統計情報を提供するためにForeignScanStateノードの状態を調べることができます。

もしExplainForeignScanポインタがNULLに設定されている場合は、EXPLAIN中に追加情報は表示されません。

```
void
ExplainForeignModify(ModifyTableState *mtstate,
                     ResultRelInfo *rinfo,
                     List *fdw_private,
                     int subplan_index,
                     struct ExplainState *es);
```

外部テーブル更新の追加のEXPLAIN出力を表示します。EXPLAIN出力にフィールドを追加するためにExplainPropertyTextや関連する関数を呼び出すことができます。esの中のフラグフィールドは何を表示するかを決めるのに使用できます。また、EXPLAIN ANALYZEの場合には、実行時統計情報を提供するためにModifyTableStateノードの状態を調べることができます。最初の4つの引数はBeginForeignModifyと同じです。

もしExplainForeignModifyポインタがNULLに設定されている場合は、EXPLAIN中に追加情報は表示されません。

```
void
ExplainDirectModify(ForeignScanState *node,
                    ExplainState *es);
```

リモートサーバでの直接変更について追加EXPLAIN出力を表示します。この関数はEXPLAIN出力にフィールドを加えるためにExplainPropertyTextと関連の関数を呼ぶことができます。esの中のフラグフィールドは何を表示するかを決めるのに使用できます。また、EXPLAIN ANALYZEの場合には、実行時統計情報を提供するためにForeignScanStateノードの状態を調べることができます。

ExplainDirectModifyポインタがNULLに設定されている場合は、EXPLAIN中に追加情報は表示されません。

56.2.7. ANALYZEのためのFDWルーチン

```
bool
AnalyzeForeignTable(Relation relation,
                    AcquireSampleRowsFunc *func,
                    BlockNumber *totalpages);
```

この関数はANALYZEが外部テーブルに対して実行されたときに呼び出されます。もしFDWがこの外部テーブルの統計情報を収集できる場合は、そのテーブルからサンプル行を集める関数のポインタとページ単位でのテーブルサイズの見積もりをそれぞれfuncとtotalpagesに渡しtrueを返す必要があります。そうでない場合は、falseを返します。

もしFDWが統計情報の収集をどのテーブルについてもサポートしない場合は、AnalyzeForeignTableポインタをNULLにすることもできます。

もし提供される場合は、サンプル収集関数はこのようなシグネチャを持つ必要があります。

```
int
AcquireSampleRowsFunc(Relation relation,
                      int elevel,
                      HeapTuple *rows,
                      int targrows,
                      double *totalrows,
                      double *totaldeadrows);
```

最大targrows行のランダムサンプルをテーブルから収集し、呼び出し元が提供するrows配列に格納する必要があります。実際に収集された行の数を返す必要があります。さらに、テーブルに含まれる有効行と不要行の合計数の見積もりを出力パラメータのtotalrowsとtotaldeadrowsに返す必要があります。(もしFDWが不要行という概念を持たない場合はtotaldeadrowsを0に設定してください。)

56.2.8. IMPORT FOREIGN SCHEMAのためのFDWルーチン

```
List *
ImportForeignSchema(ImportForeignSchemaStmt *stmt, Oid serverOid);
```

外部テーブル作成コマンドのリストを取得します。この関数はIMPORT FOREIGN SCHEMAを実行する時に呼び出され、その文の解析木と外部サーバが使用するOIDとを渡されます。C文字列のリストを返し、その各文字列はCREATE FOREIGN TABLEコマンドを含んでいる必要があります。これらの文字列はコアサーバが解析して実行します。

ImportForeignSchemaStmt構造体において、remote_schemaはリモートスキーマの名前で、そこからテーブルがインポートされます。list_typeはテーブル名のフィルタ方法を指定します。ここで、FDW_IMPORT_SCHEMA_ALLはリモートスキーマのすべてのテーブルをインポートすること(この場合、table_listは空にします)、FDW_IMPORT_SCHEMA_LIMIT_TOはtable_listに列挙されたテーブルだけを含めること、そしてFDW_IMPORT_SCHEMA_EXCEPTはtable_listに列挙されたテーブルを除外することを意味します。optionsはインポートのプロセスで使用されるオプションのリストです。オプションの意味はFDWに依存します。例えば、FDWは列のNOT NULL属性をインポートするかどうかを定めるオプションを使うことができます。これらのオプションはFDWがデータベースオブジェクトのオプションとしてサポートするものと何ら関係ある必要はありません。

FDWはImportForeignSchemaStmtのlocal_schemaフィールドを無視しても良いです。なぜなら、コアサーバは解析されたCREATE FOREIGN TABLEコマンドにその名前を自動的に挿入するからです。

FDWはlist_typeおよびtable_listで指定されるフィルタの実装にも注意する必要はありません。なぜなら、コアサーバはそれらのオプションによって除外されるテーブルに対して戻されたコマンドをすべて自動的にスキップするからです。しかし、除外されるテーブルについてコマンドを作成する作業を回避するのは、そもそも役立つことが多いです。関数IsImportableForeignTable()は指定の外部テーブル名がフィルタを通るかどうかの検査に役立つかもしれません。

FDWがテーブル定義のインポートをサポートしない場合は、ImportForeignSchemaポインタをNULLにセットすることができます。

56.2.9. パラレル実行のためのFDWルーチン

ForeignScanノードは、オプションとして、パラレル実行をサポートします。並列ForeignScanは複数プロセスで実行され、全ての協調プロセスにわたって各行が一度だけ返るようにしなければなりません。これを行うため、プロセスは動的共有メモリの固定サイズチャンクを通して調整をはかることができます。この共有メモリは全プロセスで同じアドレスに割り当てされることが保証されませんので、ポインタを含まないようにしなければなりません。以下のコールバックは一般に全て省略可能ですが、パラレル実行をサポートするためには必要です。

```
bool
IsForeignScanParallelSafe(PlannerInfo *root, RelOptInfo *rel,
                          RangeTblEntry *rte);
```

スキャンがパラレルワーカーで実行できるかテストします。この関数はプランナが並列プランが可能であろうと考えるときだけ呼ばれます。また、そのスキャンにとってパラレルワーカーで実行するのが安全であるとき真を返すべきです。リモートデータソースがトランザクションのセマンティクスを持つ場合は、一般にあてはまりません。ただし、ワーカーのデータへの接続を何らかの形でリーダーとして同じトランザクション文脈を共有させることができる場合を除きます。

この関数が定義されていない場合、スキャンはパラレルリーダー内で実行しなければならないと想定されます。真を返すことは、スキャンがパラレルワーカーで実行可能であるだけで、パラレルに実行可能であることを意味するのでは無いことに注意してください。そのため、この関数を定義することはパラレル実行がサポートされていないときでも役立つ可能性があります。

```
Size
EstimateDSMForeignScan(ForeignScanState *node, ParallelContext *pcxt);
```

並列操作に必要とされるであろう動的共有メモリ量を推定します。これは実際に使われる量よりも大きくてよいですが、小さくてはいけません。戻り値はバイト単位です。この関数はオプションであり、必要でない場合は省略することができます。しかし省略された場合、FDWの使用のために共有メモリが割り当てられないため、次の3つの関数も省略しなければなりません。

```
void
InitializeDSMForeignScan(ForeignScanState *node, ParallelContext *pcxt,
                         void *coordinate);
```

並列処理で必要とされる動的共有メモリを初期化します。coordinateはEstimateDSMForeignScanの戻り値に等しいサイズの共有メモリ領域へのポインタです。この関数はオプションであり、必要でない場合は省略することができます。

```
void
ReInitializeDSMForeignScan(ForeignScanState *node, ParallelContext *pcxt,
                           void *coordinate);
```

外部スキャンプランノードが再スキャンされようとしているときに、並列操作に必要な動的共有メモリーを再初期化します。この関数はオプションであり、必要でない場合は省略することができます。ReScanForeignScan関数はローカル状態のみをリセットし、この関数は共有状態のみをリセットすることをお勧めします。現在、この関数はReScanForeignScanより前に呼び出されますが、その順序に依存しないようにする方が良いでしょう。

```
void
InitializeWorkerForeignScan(ForeignScanState *node, shm_toc *toc,
                            void *coordinate);
```

InitializeDSMForeignScanでリーダーがセットアップした共有状態に基づくパラレルワーカーのローカル状態を初期化します。この関数はオプションであり、必要でない場合は省略することができます。

```
void
ShutdownForeignScan(ForeignScanState *node);
```

ノードが完了するまで実行されないことが予想されるときにリソースを解放します。これはすべてのケースで呼ばれるわけではありません。EndForeignScanは、この関数が最初に呼び出されなくても呼び出されることがあります。このコールバックが呼び出された直後に、並列クエリで使用されるDSM(動的共有メモリ)セグメントが破棄されるため、DSMセグメントがなくなる前に何らかのアクションを実行する外部データラッパーがこのメソッドを実装する必要があります。

56.2.10. パスの再パラメータ化のためのFDWルーチン

```
List *
ReparameterizeForeignPathByChild(PlannerInfo *root, List *fdw_private,
                                 RelOptInfo *child_rel);
```

この関数は、child_relで指定された子リレーションの最上位の親によってパラメータ化されたパスを、子リレーションによってパラメータ化されたパスに変換する際に呼び出されます。この関数はパスをパラメータ化する、あるいはForeignPathのfdw_privateメンバーに保存されている式ノードを変換するために使用されます。このコールバックは必要に応じて、reparameterize_path_by_child、adjust_appendrel_attrsあるいはadjust_appendrel_attrs_multilevelを呼び出すことができます。

56.3. 外部データラッパヘルパ関数

FDWオプションのようなFDW関連オブジェクトの属性に外部データラッパの作者が簡単にアクセスできるように、いくつかのヘルパ関数がコアサーバからエクスポートされています。これらの関数を使用するには、 ヘッダファイルforeign/foreign.hをあなたのソースファイルにインクルードする必要があります。このヘッダはまたこれらの関数が返す構造体も定義しています。

```
ForeignDataWrapper *
GetForeignDataWrapperExtended(Oid fdwid, bits16 flags);
```

この関数は、与えられたOIDの外部データラッパーに対してForeignDataWrapperオブジェクトを返します。ForeignDataWrapperオブジェクトにはFDWの属性（詳しくはforeign/foreign.hを参照）が含まれます。flags は追加的なオプション一式を示すビット毎にorを取ったビットマスクです。ここにはFDW_MISSING_OK値を指定できて、この場合、未定義オブジェクトに対するエラーの代わりに呼び出し元にNULL結果が返されます。

```
ForeignDataWrapper *
GetForeignDataWrapper(Oid fdwid);
```

この関数は指定されたOIDを持つ外部データラッパのForeignDataWrapperオブジェクトを返します。ForeignDataWrapperオブジェクトはFDWのプロパティを含みます(詳細はforeign/foreign.hを参照して下さい)。

```
ForeignServer *
GetForeignServerExtended(Oid serverid, bits16 flags);
```

この関数は、与えられたOIDの外部サーバに対してForeignServerオブジェクトを返します。ForeignServerオブジェクトにはサーバの属性（詳しくはforeign/foreign.hを参照）が含まれます。flagsは追加的なオプション一式を示すビット毎にorを取ったビットマスクです。ここにはFSV_MISSING_OK値を指定できて、この場合、未定義オブジェクトに対するエラーの代わりに呼び出し元にNULL結果が返されます。

```
ForeignServer *
GetForeignServer(Oid serverid);
```

この関数は指定されたOIDを持つ外部サーバのForeignServerオブジェクトを返します。ForeignServerオブジェクトは外部サーバのプロパティを含みます(詳細はforeign/foreign.hを参照して下さい)。

```
UserMapping *
GetUserMapping(Oid userid, Oid serverid);
```

この関数は指定されたロールと指定された外部サーバのユーザマッピングのUserMappingオブジェクトを返します。(もし特定のユーザのマッピングがない場合は、PUBLICのためのマッピングを返すか、それもなければエラーを発生させます。) UserMappingオブジェクトはユーザマッピングのプロパティを含みます(詳細はforeign/foreign.hを参照して下さい)。

```
ForeignTable *
GetForeignTable(Oid relid);
```

この関数は指定されたOIDを持つ外部テーブルのForeignTableオブジェクトを返します。ForeignTableオブジェクトは外部テーブルのプロパティを含みます(詳細はforeign/foreign.hを参照して下さい)。

```
List *
GetForeignColumnOptions(Oid relid, AttrNumber attnum);
```

この関数は指定された外部テーブルOIDと属性番号に該当する列の列単位のFDWオプションをDefElemのリスト形式で返します。その列がオプションを持たない場合はNILが返ります。

いくつかのオブジェクト種別については、OIDベースのものに加えて名前ベースの検索関数もあります。

```
ForeignDataWrapper *
GetForeignDataWrapperByName(const char *name, bool missing_ok);
```

この関数は指定された名前の外部データラッパのForeignDataWrapperオブジェクトを返します。外部データラッパが見つからない場合は、missing_okがtrueの場合はNULLを返し、それ以外の場合はエラーを発生させます。

```
ForeignServer *
GetForeignServerByName(const char *name, bool missing_ok);
```

この関数は指定された名前の外部サーバのForeignServerオブジェクトを返します。外部サーバが見つからない場合は、missing_okがtrueの場合はNULLを返し、それ以外の場合はエラーを発生させます。

56.4. 外部データラッパのクエリプラン作成

FDWコールバック関数のGetForeignRelSize、GetForeignPaths、GetForeignPlan、PlanForeignModify、GetForeignJoinPaths、GetForeignUpperPaths、PlanDirectModifyはPostgreSQLプランナの動作と協調しなければなりません。ここでは、これらの関数がすべき事に関するいくつかの注意事項を述べます。

rootとbaserelに含まれる情報は、外部テーブルから取得する必要のある情報の量(とそれによるコスト)を削減するために使用できます。baserel->baserestrictinfoは、取得される行をフィルタリングする検索条件(WHERE句)を含んでいるため、特に興味深いものです。(コアのエクゼキュータが代わりにそれらをチェックできるので、FDW自身がこれらの制約を適用しなければならないわけではありません。) baserel->reltarget->exprsはどの列が取得される必要があるかを決定するのに使用できます。ただし、このリストはForeignScanプランノードから出力すべき列しか含んでおらず、条件検査には必要だがクエリからは出力されない列は含まないことに注意してください。

様々なプライベートフィールドがFDWのプラン作成関数で情報を格納する目的で利用できます。一般的に、プラン作成の最後に回収できるように、FDW固有フィールドに格納するものは全てpallocで確保すべきです。

baserel->fdw_privateは、voidポインタで、FDWのプラン作成関数で特定の外部テーブルに関する情報を格納する目的で利用できます。コアプランナは、RelOptInfoノードが作成されるときにNULLで初期化するときを除いて、このフィールドに一切に触れません。このフィールドは、GetForeignRelSizeからGetForeignPathsやGetForeignPathsからGetForeignPlanといったように情報を順次伝えるの便利で、結果として再計算を省くことができます。

GetForeignPathsでは、ForeignPathノードのfdw_privateフィールドに固有情報を格納することで、異なるアクセスパスを区別できます。fdw_privateはListポインタとして宣言されていますが、コアプランナがこのフィールドを操作することはないため、実際にはなんでも格納できます。しかし、バックエンドのデバッグサポート機能を利用できるようにnodeToStringでダンプ出来る形式を使うのが最良の手法です。

GetForeignPlanでは、選択されたForeignPathノードのfdw_privateフィールドを調べて、ForeignScanプランノード内に格納されプラン実行時に利用可能なfdw_exprsとfdw_privateの二つのリストを生成することが

できます。これらは両方ともcopyObjectがコピーできる形式でなければなりません。fdw_privateリストにはこれ以外に制約はなく、コアバックエンドによって解釈されることはありません。fdw_exprsリストがNILでない場合は、クエリ実行時に実行されることを意図した式ツリーが含まれていることが期待されます。これらのツリーは、完全に実行可能な状態にするためにプランナによる後処理を受けます。

GetForeignPlanでは、一般的に渡されたターゲットリストはそのままプランノードにコピーできます。渡されたscan_clausesリストはbaserel->baserestrictinfoと同じ句を含みますが、実行効率のよい別の順番に並べ替えることもできます。FDWにできるのがRestrictInfoノードをscan_clausesリストから(extract_actual_clausesを使って)抜き出して、全ての句をプランノードの条件リストに入れるだけ、といった単純なケースでは、全ての句は実行時にエクゼキュータによってチェックされます。より複雑なFDWは内部で一部の句をチェックできるかもしれませんが、そのような場合には、エクゼキュータが再チェックのために時間を無駄にしないように、それらの句はプランノードの条件リストから削除できます。

たとえば、ローカル側で評価されたsub_expressionの値があればリモートサーバ側で実行出来るとFDWが判断するような、foreign_variable = sub_expressionといった形式の条件句をFDWが識別するかもしれません。パスのコスト見積もりに影響するので、そのような句の実際の識別はGetForeignPathsでなされるべきです。おそらく、そのパスのfdw_privateフィールドは識別された句のRestrictInfoノードをさすポインタを含むでしょう。そして、GetForeignPlanはその句をscan_clausesから取り除き、実行可能な形式にほぐされることを保障するためにsub_expressionをfdw_exprsに追加するでしょう。また、おそらく、実行時に何をすべきかをプラン実行関数に伝えるためにプランノードのfdw_privateフィールドに制御情報を入れるでしょう。リモートサーバに送られたクエリは、実行時にfdw_exprs式ツリーを評価して得られた値をパラメータ値とするWHERE foreign_variable = $1のようなものを伴うでしょう。

READ COMMITTED分離レベルでの正しい動作を保証するため、プランノードの条件リストから除かれた句はすべて、代わりにfdw_recheck_qualsに追加されるか、RecheckForeignScanで再検査される必要があります。問い合わせに含まれる他のテーブルで同時更新があった場合、エクゼキュータはタプルが元の条件を、それも場合によっては異なるパラメータ値の組み合わせに対して満たすことを確認する必要があるかもしれません。fdw_recheck_qualsを使うのは、RecheckForeignScanの内部で検査を実装するより、通常は簡単でしょう。しかしこの方法は、外部結合がプッシュダウンされる場合は不十分です。なぜなら、この場合の結合タプルはタプル全体を拒絶せずに、一部のフィールドをNULLにしてしまうからです。

FDWがセットできる別のForeignScanフィールドにfdw_scan_tlistがあります。これはこのプランノードについてFDWが返すタプルを記述するものです。単純な外部テーブルスキャンに対しては、これをNILにセットすることができ、それは戻されるタプルが外部テーブルで宣言された行型を持つことを意味します。NILでない値はVar型の変数、あるいは返される列を表す式、あるいはその両方を含む対象のリスト(TargetEntryのリスト)でなければなりません。これは例えば、FDWが問い合わせのために必要ないと気づいた列を無視したことを示すのに使えるかもしれません。また、FDWが問い合わせで使われる式をローカルで計算するより安価に計算できるなら、それらの式をfdw_scan_tlistに追加することができます。結合プラン(GetForeignJoinPathsが作るパスから作成される)は、それが返す列の集合を記述するfdw_scan_tlistを必ず提供しなければならないことに注意して下さい。

FDWはそのテーブルの条件句のみに依存するパスを常に少なくとも一つは生成すべきです。結合クエリでは、例えばforeign_variable = local_variableといった結合句に依存するパス(群)を生成することもできます。そのような句はbaserel->baserestrictinfoには見つからず、リレーションの結合リストにあるはずです。そのような句を使用するパスは「パラメータ化されたパス」と呼ばれます。このようなパスでは、選択された結合句(群)で使用されているリレーション(群)をparam_infoの適合する値から特定しなければなりません;その値を計算するにはget_baserel_parampathinfoを使用します。GetForeignPlanでは、結合句のlocal_variable部分がfdw_exprsに追加され、実行時には通常の条件句と同じように動作します。

FDWがリモートでの結合をサポートする場合、GetForeignPathsがベーステーブルに対して処理するのとほぼ同じように、GetForeignJoinPathsは潜在的なリモートの結合に対してForeignPathを生成することになります。意図した結合に関する情報は、上記と同じ方法でGetForeignPlanに送ることができます。しかし、baserestrictinfoは結合のリレーションには関連がなく、代わりに、特定の結合に関連するJOIN句はGetForeignJoinPathsに別のパラメータ(extra->restrictlist)として渡されます。

FDWはグルーピングや集約のような、スキャンや結合のレベルより上位のプラン動作の直接実行を追加的にサポートできるかもしれません。このような方法を行うには、FDWはパスを生成して、それを適切な上位リレーションに挿入する必要があります。例えば、リモート集約をあらわすパスはadd_pathを使ってUPPERREL_GROUP_AGGリレーションに挿入されるべきです。このパスは外部リレーションに対する単純なスキャンパスを読むことによるローカル集約実行とコストに基づいて比較されます(このようなパスが提供されなければならないことに注意してください、さもないとプラン時にエラーになります)。リモート集約パスが、通常そうなりますが、勝った場合には、パスはGetForeignPlanを呼ぶ通常の手段でプランに変換されます。もし問い合わせの全てのベースリレーションが同じFDWから来るなら、このようなパスを生成するのに推奨される場所は、各上位リレーション(すなわち各スキャン/結合後の処理の段階)に対して呼び出されるGetForeignUpperPathsコールバック関数の中です。

PlanForeignModifyと56.2.4で記述された他のコールバックは、外部リレーションは通常の方法でスキャンされ、それから個別の行変更がローカルのModifyTableプランノードで駆動されるという想定をもとに設計されています。この方法は変更が外部テーブルと同様にローカルテーブルを読む必要がある一般的な場合に必要です。しかしながら、操作が全体的に外部サーバで実行できるなら、FDWはそのようにするパスを生成してUPPERREL_FINAL上位リレーションに挿入することができます。ここではModifyTable方式に対して競合します。この方式は、56.2.5で記述された行ロックコールバックを使うのでなしに、リモートSELECT FOR UPDATEを実装するのにも使われます。UPPERREL_FINALに挿入されたパスは問い合わせの全ての振る舞いの実装に責任があることに留意してください。

UPDATEやDELETEのプランを生成しているとき、PlanForeignModifyとPlanDirectModifyは、事前にスキャンプラン生成関数で作られたbaserel->fdw_privateデータを使うために、その外部テーブルのためのRelOptInfo構造体を検索することができます。しかしながら、INSERTでは対象テーブルはスキャンされないので対応するRelOptInfoは存在しません。PlanForeignModifyから返されるListには、ForeignScanプランノードのfdw_privateリストと同様に、copyObjectがコピーの仕方を知っている構造体しか保持してはいけないという制約があります。

ON CONFLICT句のあるINSERTは競合の対象の指定をサポートしません。なぜなら、リモートのテーブルの一意制約や排他制約についての情報がローカルにはないからです。これは結果的にON CONFLICT DO UPDATEがサポートされないことを意味します。なぜなら、競合の対象の指定が必須だからです。

56.5. 外部データラッパでの行ロック

FDWの元になる記憶機構が、行の同時更新を防ぐために個々の行をロックするという概念を持っているなら、PostgreSQLの通常のテーブルで使われている意味にできる限り現実的で近い行単位のロックをFDWが実施することは価値があるでしょう。これに関していくつかの考慮点があります。

なされるべき重要な決定の一つは、早いロックを実行するか遅いロックを実行するか、です。早いロックでは、行は、元となる記憶機構から最初に取り出されたときにロックされます。一方、遅いロックでは、行は、それがロックされる必要があることがわかってからロックされます。(この違いは、一部の行がローカルで検査される制約や結合条件によって除外されるために発生します。) 早いロックの方がずっと単純ですし、リモー

トの記憶機構との間の余分なやりとりもなくて済みますが、ロックしなくても良い行をロックするかもしれませんし、結果的に同時実行性が低下したり、予期しないデッドロックさえ発生します。一方で、遅いロックは、ロックすべき行が後で一意に再識別できる場合にのみ可能です。できれば、PostgreSQLのTIDがそうしているように、行識別子は行の特定のバージョンを識別できるのが望ましいです。

デフォルトではPostgreSQLはFDWとのやりとりにおいてロックの考慮をしませんが、FDWはコアのコードからの明示的なサポートなしに、早いロックを実行することができます。PostgreSQLバージョン9.5で追加された56.2.5に記載されたAPI関数を使うことで、望むならFDWで遅いロックを使うことも可能です。

さらなる考慮点は、READ COMMITTED分離モードにおいて、PostgreSQLは対象のタプルの更新されたバージョンに対して制約と結合条件の再検査を行う必要があるかもしれないということです。結合条件を再検査するには、前回取得対象のタプルと結合された、取得対象外の行の複製を再取得する必要があります。PostgreSQLの標準テーブルを使うときは、結合を通じて生成される列リストに対象でないテーブルのTIDを含めて、必要な時には対象でない行を再フェッチすることで解決しています。この方法は結合のデータセットを小さくできますが、安価な再フェッチ機能と再フェッチすべきバージョンの行を一意に特定できるTIDが必要になります。そのためデフォルトで外部テーブルに対して使われる方法は、外部テーブルからフェッチされた行全体を結合を通じて生成した列リストに含めるというものです。これによりFDWに対する特別な要請はなくなりますが、マージ結合およびハッシュ結合に置いてパフォーマンスが低下する結果となるかもしれません。再フェッチの要求を満たすことができるFDWでは最初の方法を選択するのも良いでしょう。

外部テーブルに対するUPDATEやDELETEでは、対象テーブルに対するForeignScan操作はフェッチする行を、恐らくはSELECT FOR UPDATEと同等なものを用いてロックすることが推奨されます。FDWはテーブルがUPDATEまたはDELETEの対象かどうかを、計画時にそのrelidをroot->parse->resultRelationと比較することで、あるいは実行時にExecRelationIsTargetRelation()を使うことで検知できます。これに代わる可能性として、ExecForeignUpdateまたはExecForeignDeleteのコールバック内で遅いロックを実行することがありますが、これについて特別なサポートは提供されません。

SELECT FOR UPDATE/SHAREコマンドによりロックすることが指定された外部テーブルについて、ForeignScanの操作ではSELECT FOR UPDATE/SHAREと同等なものを使ってタプルをフェッチすることで、ここでも早いロックを実行できます。逆に遅いロックを実行するには、56.2.5で定義されるコールバック関数を提供して下さい。GetForeignRowMarkTypeでは、要求されたロックの強度に応じて、rowmarkのオプションROW_MARK_EXCLUSIVE、ROW_MARK_NOKEYEXCLUSIVE、ROW_MARK_SHAREまたはROW_MARK_KEYSHAREを選択して下さい。(コアのコードは、この4つのオプションのどれが選ばれたかに関係なく、同じ動作をします。) その他には、この種のコマンドによって外部テーブルのロックが指定されたかどうかを、計画時にget_plan_rowmarkを使うことで、あるいは実行時にExecFindRowMarkを使うことで検知できます。このとき、NULLでないrowmark構造体が戻されるかどうかだけでなく、そのstrengthフィールドがLCS_NONEでないことも確認しなければなりません。

最後に、UPDATE、DELETEまたはSELECT FOR UPDATE/SHAREコマンドで使用されたが、行ロックの指定はされなかった外部テーブルについて、ロック強度がLCS_NONEになっているときにGetForeignRowMarkTypeでオプションROW_MARK_REFERENCEを選択すれば、すべての行を複製するというデフォルトの動作を変更することができます。これにより、markTypeにその値を入れてRefetchForeignRowが呼び出されるようになります。このとき、新しいロックを取得することなく行を再取得します。(GetForeignRowMarkType関数を使うが、ロックしていない行を再フェッチしたくない場合は、LCS_NONEについてオプションROW_MARK_COPYを選択して下さい。)

さらなる情報は、src/include/nodes/lockoptions.h、src/include/nodes/plannodes.hでのRowMarkTypeとPlanRowMarkについてのコメント、src/include/nodes/execnodes.hでのExecRowMarkについてのコメントを参照して下さい。

第57章 テーブルサンプリングメソッドの書き方

PostgreSQLによるTABLESAMPLE句の実装は、SQL標準が求めるBERNOULLIとSYSTEMのメソッドに加え、ユーザ定義のテーブルサンプリングメソッドをサポートしています。サンプリングメソッドは、TABLESAMPLE句が使用された際にどの行が選択されるかを決定します。

SQLレベルでは、テーブルサンプリングメソッドは、以下の呼び出し形式を持ち、典型的にはCで実装された単一のSQLの関数で表現されます。

```
method_name(internal) RETURNS tsm_handler
```

関数の名前はTABLESAMPLE句に現れるメソッド名と同じです。internal引数は、ダミー(常に0の値を持つ)で、単にこの関数がSQLコマンドから直接呼ばれるのを防ぐために用意されています。関数の戻り値は、pallocされたTsmRoutine型の構造体でなければなりません。その構造体の中には、サンプリングメソッド用のサポート関数へのポインタが含まれています。サポート関数は普通のC関数で、SQLレベルからは見ることも呼び出すこともできません。サポート関数は57.1で説明されています。

関数へのポインタに加え、TsmRoutine構造体は以下の追加のフィールドを提供しなければなりません。

`List *parameterTypes`

> このサンプリングメソッドが使用される際に、TABLESAMPLE句が受け付ける引数のデータ型のOIDが格納された、OIDのリストです。たとえば、組み込みのメソッドに対しては、このリストは、サンプリングのパーセンテージを表すFLOAT4OIDという値を持つ単一の要素が含まれています。カスタムサンプリングメソッドは、複数の異なるパラメータを持つことができます。

`bool repeatable_across_queries`

> trueの場合、もし毎回同じ引数とREPEATABLEシード値が提供され、テーブル内容に変更がないならば、サンプリングメソッドは実行されたどの問い合わせに対しても、同一のサンプルを出力することができます。falseの場合は、サンプリングメソッドを使用する際にREPEATABLE句を受け付けません。

`bool repeatable_across_scans`

> trueの場合、サンプリングメソッドは同じ問い合わせ内で実行されたどのスキャンに対しても、同一のサンプルを出力することができます(パラメータ、シード値、スナップショットに変更がない、という前提で)。falseの場合、プランナは、サンプル対象のテーブルを2度以上スキャンする必要のあるようなプランは選択しません。問い合わせの結果に不整合が発生する可能性があるからです。

TsmRoutine構造体はsrc/include/access/tsmapi.hで宣言されています。詳細はそちらをご覧ください。

標準配布物に含まれるテーブルサンプリングメソッドは、自分でサンプリングメソッドを書く際に良いお手本になります。組み込みのサンプリングメソッドに関しては、ソースツリー中のsrc/backend/access/tablesampleサブディレクトリを見てください。追加のメソッドに関してはcontribサブディレクトリを見てください。

57.1. サンプリングメソッドサポート関数

TSMハンドラ関数は、以下に示すサポート関数へのポインタを含むpallocされたTsmRoutine構造体を返します。大半の関数は必須ですが、いくつかオプションのものがあり、そうした関数へのポインタはNULLにできます。

```
void
SampleScanGetSampleSize (PlannerInfo *root,
                         RelOptInfo *baserel,
                         List *paramexprs,
                         BlockNumber *pages,
                         double *tuples);
```

この関数はプランニングの際に呼び出されます。サンプルスキャン中に読み出すリレーションのページ数と、スキャン中に選択されるタプル行数の見積もりを行わなければなりません。(たとえば、サンプル比率を推定し、それにbaserel->pagesとbaserel->tuplesを掛け、整数値になるように丸めることで見積もりが可能となるでしょう。) paramexprsリストは、TABLESAMPLE句への引数となる式を格納します。見積のためにその推測値が必要なら、estimate_expression_value()を使ってこれらの式を定数に簡約化してみることをおすすめします。ただし、関数は簡約化ができない場合でもサイズに関する見積は提供しなければなりませんし、値が正しくない場合でも関数がエラーになってはいけません(推測値は、実行時には値がそうなるであろうということに過ぎないことを思い出してください)。pagesとtuplesパラメータは出力です。

```
void
InitSampleScan (SampleScanState *node,
                int eflags);
```

SampleScan計画ノードを実行するための初期化を行います。この関数はエクゼキュータの起動時に呼び出されます。処理を開始する前に必要な初期化をすべて行う必要があります。SampleScanStateノードは作成済みですが、tsm_stateフィールドはNULLです。InitSampleScan関数はサンプリングメソッドが必要とする内部データをすべてpallocし、node->tsm_stateに格納します。スキャン対象のテーブルに関する情報はSampleScanStateノードの他のフィールドを通じてアクセスできます (ただし、node->ss.ss_currentScanDescスキャンディスクリプタはまだ設定されていません)。eflagsには、この計画ノードにおけるエクゼキュータの動作モードを記述するフラグビットが含まれます。

(eflags & EXEC_FLAG_EXPLAIN_ONLY)が真ならば、スキャンは実際には行われず、この関数はEXPLAINとEndSampleScanにとってノードの状態が意味のあるように最低限必要な処理を行うことになります。

この関数は(ポインタをNULLにすることにより)省略できますが、この場合、BeginSampleScanがサンプリングメソッドに必要なすべての初期化を行なわなければなりません。

```
void
BeginSampleScan (SampleScanState *node,
                 Datum *params,
                 int nparams,
                 uint32 seed);
```

サンプルスキャンの実行を開始します。これははじめてタプルを取得する直前に呼び出されます。また、再スキャンを行う必要が出た場合にも呼び出されます。スキャン対象のテーブルに関する情報はSampleScanStateノードのフィールドを通じてアクセスできます (ただし、node->ss.ss_currentScanDescスキャンディスクリプタはまだ設定されていません)。nparamsの長さを持つparams配列は、TABLESAMPLE句で指定された引数の値を保持しています。これらは、サンプリングメソッドのparameterTypesリストで指定された数と型を持ち、NULLでないことがチェック済みです。seedには、サンプリングメソッド内で使われる乱数のために使われるシードが格納されます。これは、REPEATABLEの値が指定されている場合はそこから派生したハッシュか、でなければrandom()の結果です。

この関数はnode->use_bulkreadとnode->use_pagemodeフィールドによって動作を変更します node->use_bulkreadがtrueなら(これはデフォルトです)、スキャンは使用後のバッファの再利用を推奨するバッファアクセス戦略を使います。テーブルのわずかな部分だけをスキャンがアクセスするようなら、falseにするのが妥当かもしれません。node->use_pagemodeがtrueなら(これはデフォルトです)、スキャンはアクセスするページ上のすべてのタプルに対して一括で可視性チェックを行います。スキャンがアクセスするページ上のわずかな部分のタプルだけを選択するのであれば、false にするのが妥当かもしれません。これにより、より少ないタプルに対して可視性チェックが行われます。ただし、個々の操作はより高くつきます。というのも、より多くのロックが必要になるからです。

サンプリングメソッドにrepeatable_across_scansという印があれば、最初にスキャンした時と同じタプルの集合を、再スキャンでも選択できることになります。つまり、新しいBeginSampleScanが、前回と同じタプルを選択することになるわけです。(もしTABLESAMPLEの引数とシードが変わらなければ、の話ですが)

```
BlockNumber
NextSampleBlock (SampleScanState *node, BlockNumber nblocks);
```

次にスキャンするページのブロック番号を返します。もはやスキャンするページがない場合にはInvalidBlockNumberを返します。

この関数は(ポインタをNULLにすることにより)省略できます。この場合コアのコードはリレーション全体を順スキャンします。そのようなスキャンは同期スキャンを行う可能性があるので、毎回のスキャンで同じ順番でリレーションのページをアクセスするとは、サンプリングメソッドは仮定できません。

```
OffsetNumber
NextSampleTuple (SampleScanState *node,
                 BlockNumber blockno,
                 OffsetNumber maxoffset);
```

サンプル対象の指定ページ内の次のタプルのオフセットを返します。サンプル対象のタプルが残っていない場合は、InvalidOffsetNumberを返します。maxoffsetは、使用中のページ中の最大オフセットです。

注記

NextSampleTupleは、範囲1 .. maxoffsetの中のどのオフセット番号が有効なタプルにあたるのかは明示的には教えてくれません。コアのコードは、存在しない、あるいは不可視のタプルを対象とするサンプルの要求は無視するため、通常これは問題にはなりません。サンプルの偏りも起きません。それでも必要ならば、関数はnode->donetuplesを使って、返されたタプルのうちのいくつが有効で可視であったのかを調べることができます。

注記

NextSampleTupleは、直近のNextSampleBlockの呼び出しが返したページ番号とblocknoが同じであると見なすべきではありません。ページ番号は、以前のNextSampleBlockの呼び出しが返したものではありますが、コアのコードは、先読みのために実際のスキャンに先立ってNextSampleBlockを呼び出すことが認められています。一旦あるページのサンプリングが開始すれば、InvalidOffsetNumberが返るまでは、続くNextSampleTupleに呼び出しがすべて同じページを参照すると見なすことは問題ありません。

```
void
EndSampleScan (SampleScanState *node);
```

スキャンを終了し、リソースを解放します。通常pallocされたメモリを解放するのは重要なことではありませんが、外部から見えるリソースはすべて解放しなければなりません。そのようなリソースが存在しない場合は、この関数は(ポインタをNULLにすることにより)省略できます。

第58章 カスタムスキャンプロバイダの作成

PostgreSQLでは、システムに新しいスキャン方式を追加する拡張モジュールを可能にするためのいくつかの実験的機構をサポートしています。外部データラッパが自分の外部テーブルのスキャン方法を知っていることだけを担当するのと異なり、カスタムスキャンプロバイダはシステム内のリレーションをスキャンする代替方式を提供することができます。典型的には、カスタムスキャンプロバイダを作成する理由は、キャッシュの利用や何らかの形式のハードウェアアクセラレーションといったコアシステムによってサポートされない最適化を利用可能にすることでしょう。本章では新しいカスタムスキャンプロバイダの作成方法について概要を説明します。

新しい種類のカスタムスキャンの実装には3段階のプロセスがあります。第一に、計画段階において、提案される戦略を用いたスキャンを表現するアクセスパスを生成する必要があります。第二に、プランナがあるリレーションをスキャンするための最適戦略として、そのアクセスパスの一つを選んだとき、アクセスパスが計画に変換されなければなりません。最後に、計画を実行して、同じリレーションを対象とする他のアクセスパスが生成するのと同じ結果を生成することが可能でなければなりません。

58.1. カスタムスキャンパスの作成

カスタムスキャンプロバイダは、典型的には、以下のフックを設定することでベースリレーションのためのパスを追加します。このフックはコアのコードがそのリレーションへのすべてのアクセスパスを生成した後で呼び出されます。(フックの呼び出しの後に作成されるギャザーパス(Gather path)を除きます。フックが追加した部分パスをギャザーパスが利用できるようにするためです。)

```
typedef void (*set_rel_pathlist_hook_type) (PlannerInfo *root,
                                            RelOptInfo *rel,
                                            Index rti,
                                            RangeTblEntry *rte);
extern PGDLLIMPORT set_rel_pathlist_hook_type set_rel_pathlist_hook;
```

このフックはコアシステムが生成したパスを検査し、修正し、あるいは削除するために使うことができますが、カスタムスキャンプロバイダは、典型的にはCustomPathオブジェクトを生成し、add_pathを使ってそれをrelに追加することのみを行います。カスタムスキャンプロバイダはCustomPathオブジェクトの初期化を担当します。このオブジェクトは以下のように宣言されています。

```
typedef struct CustomPath
{
    Path      path;
    uint32    flags;
    List     *custom_paths;
    List     *custom_private;
    const CustomPathMethods *methods;
} CustomPath;
```

pathは、他のすべてのパスと同じく、行数の推定値、開始とトータルのコスト、このパスで提供されるソート順を含めて初期化される必要があります。flagsはビットマスクで、カスタムパスが逆向きスキャンをサポートできるならCUSTOMPATH_SUPPORT_BACKWARD_SCANを、マークとリストアがサポートできるならCUSTOMPATH_SUPPORT_MARK_RESTOREを含めます。いずれの機能も必須ではありません。オプションのcustom_pathsはこのカスタムパスのノードで使用されるPathのノードのリストです。プランナがこれをPlanのノードに変換します。custom_privateはカスタムパスのプライベートデータを格納するために使うことができます。プライベートデータはnodeToStringが処理できるような形式で格納してください。そうすることで、カスタムパスを出力するデバッグルーチンが設計通りに動作します。methodsは要求されるカスタムパスのメソッドのオブジェクト(通常は静的に割り当てられる)を指している必要があり、現在は1つのみとなります。

カスタムスキャンプロバイダは結合(join)のパスを提供することもできます。ベースのリレーションの場合と同様、そのようなパスは置換される結合が普通に生成したであろうものと同じ結果を生成しなければなりません。そのために、結合のプロバイダは以下のフックをセットし、フック関数内で結合リレーション用にCustomPathのパスを作成します。

```
typedef void (*set_join_pathlist_hook_type) (PlannerInfo *root,
                                             RelOptInfo *joinrel,
                                             RelOptInfo *outerrel,
                                             RelOptInfo *innerrel,
                                             JoinType jointype,
                                             JoinPathExtraData *extra);
extern PGDLLIMPORT set_join_pathlist_hook_type set_join_pathlist_hook;
```

このフックは、同じ結合リレーションについて、内側あるいは外側のリレーションとの様々な組み合わせで繰り返し呼び出されます。繰り返しの作業を最小化するのはフック側の責任です。

58.1.1. カスタムスキャンパスのコールバック

```
Plan *(*PlanCustomPath) (PlannerInfo *root,
                         RelOptInfo *rel,
                         CustomPath *best_path,
                         List *tlist,
                         List *clauses,
                         List *custom_plans);
```

カスタムパスを完成した計画に変換します。戻り値は一般的にはCustomScanオブジェクトで、その領域はコールバックが割り当てて初期化しなければなりません。詳しくは58.2を参照してください。

58.2. カスタムスキャン計画の作成

カスタムスキャンは完成した計画ツリー内で、以下の構造体を使って表現されます。

```
typedef struct CustomScan
{
    Scan      scan;
```

```
    uint32    flags;
    List     *custom_plans;
    List     *custom_exprs;
    List     *custom_private;
    List     *custom_scan_tlist;
    Bitmapset *custom_relids;
    const CustomScanMethods *methods;
} CustomScan;
```

scanは他のすべてのスキャンと同じく、推定コスト、対象のリスト、制約などを含めて初期化される必要があります。flagsはCustomPathと同じ意味のビットマスクです。custom_plansは子のPlanノードを格納するために使うことができます。custom_exprsはsetrefs.cおよびsubselect.cによって作成される必要がある式のツリーを格納するために使われます。一方でcustom_privateはカスタムスキャンプロバイダ自体によってのみ使用されるその他のプライベートデータを格納するために使われます。custom_scan_tlistはベースリレーションをスキャンするときはNILとすることができます。これはカスタムスキャンがベースリレーションの行の型と一致するスキャンタプルを返すことを意味します。それ以外の場合は、実際のスキャンタプルを表現する対象のリストとなります。custom_scan_tlistは結合の場合には提供される必要があります。また、カスタムスキャンプロバイダがVarでない式を計算できる場合はスキャン用に提供することができます。custom_relidsは、コアコードにより、このスキャンノードが処理するリレーションの集合（範囲テーブルのインデックス）にセットされます。ただし、このスキャンが結合を置換する場合は例外で、ただ1つのメンバーだけになります。methodsは必要なカスタムスキャンメソッドを実装しているオブジェクト（通常は静的に割り当てられる）を指していなければなりません。これについては以下で詳しく説明します。

CustomScanがリレーションを1つだけスキャンするときは、scan.scanrelidはスキャンされるテーブルの範囲テーブルのインデックスである必要があります。結合を置換するときはscan.scanrelidはゼロになります。

計画ツリーはcopyObjectにより複製できる必要があるので、「custom」フィールド内に格納されるすべてのデータは、その関数が処理できるノードから構成されていなければなりません。また、カスタムスキャンプロバイダはCustomScanを組み込んでいる大きな構造体をCustomScanの構造体で代替することができません。CustomPathやCustomScanStateに対してはこれが可能です。

58.2.1. カスタムスキャン計画のコールバック

```
Node *(*CreateCustomScanState) (CustomScan *cscan);
```

このCustomScanにCustomScanStateの領域を割り当てます。多くのプロバイダは、より大きな構造体の最初のフィールドとしてこれを組み込もうとするので、実際の割り当ては通常のCustomScanStateが必要とするよりも多くなることが多いでしょう。戻り値では、ノードのタグとmethodsが適切に設定されている必要がありますが、その他のフィールドはこの段階ではゼロのままになっています。ExecInitCustomScanが基本的な初期化をした後、BeginCustomScanコールバックが呼び出されることで、カスタムスキャンプロバイダがその他の必要なことを実行する機会が与えられます。

58.3. カスタムスキャンの実行

CustomScanが実行されるとき、その実行状態はCustomScanStateで表現されます。これは次のように宣言されています。

```
typedef struct CustomScanState
{
    ScanState ss;
    uint32    flags;
    const CustomExecMethods *methods;
} CustomScanState;
```

ssは他のすべてのスキャン状態と同じく初期化されますが、スキャンがベースリレーションではなく結合を対象にしているときは例外で、ss.ss_currentRelationはNULLのままになります。flagsはCustomPathおよびCustomScanと同じ意味のビットマスクです。methodsは必要なカスタムスキャン状態のメソッドを実装するオブジェクト（通常は静的に割り当てられる）を指していなければなりません。これについては以下で詳しく説明します。CustomScanStateはcopyObjectをサポートしなくてもよく、典型的には上記を先頭のメンバーとして組み込んだより大きな構造体になっています。

58.3.1. カスタムスキャン実行のコールバック

```
void (*BeginCustomScan) (CustomScanState *node,
                         EState *estate,
                         int eflags);
```

提供されたCustomScanStateの初期化を完了します。標準的なフィールドはExecInitCustomScanで初期化が済んでいますが、プライベートフィールドはここで初期化されます。

```
TupleTableSlot *(*ExecCustomScan) (CustomScanState *node);
```

次のスキャンタプルをフェッチします。タプルが残っている場合は、現在のスキャン方向で次にあるタプルをps_ResultTupleSlotに入れます。タプルが残っていないときは、NULLまたは空のスロットが戻されます。

```
void (*EndCustomScan) (CustomScanState *node);
```

CustomScanStateに関連付けられたプライベートデータを整理します。このメソッドは必須ですが、関連付けられたデータがない場合、あるいはそれが自動的に整理される場合は、このメソッドは何もする必要はありません。

```
void (*ReScanCustomScan) (CustomScanState *node);
```

現在のスキャンを先頭まで巻き戻し、リレーションの再スキャンの準備をします。

```
void (*MarkPosCustomScan) (CustomScanState *node);
```

現在のスキャン位置を保存し、後でRestrPosCustomScanコールバックでリストアできるようにします。このコールバックは必須ではなく、CUSTOMPATH_SUPPORT_MARK_RESTOREフラグがセットされている場合にのみ、提供する必要があります。

```
void (*RestrPosCustomScan) (CustomScanState *node);
```

MarkPosCustomScanコールバックで保存された以前のスキャン位置をリストアします。このコールバックは必須ではなく、CUSTOMPATH_SUPPORT_MARK_RESTOREフラグがセットされている場合にのみ、提供する必要があります。

```
Size (*EstimateDSMCustomScan) (CustomScanState *node,
                               ParallelContext *pcxt);
```

並列操作に要求される動的共有メモリの使用量を予測します。使用を予測される量よりも多い量の結果が返しても良いですが、少なく返してはいけません。返り値の単位はバイトとなります。このコールバックは必須ではなく、カスタムスキャンプロバイダが並列実行をサポートする場合にのみ提供される必要があります。

```
void (*InitializeDSMCustomScan) (CustomScanState *node,
                                 ParallelContext *pcxt,
                                 void *coordinate);
```

並列操作に要求される動的共有メモリを初期化します。coordinateは、EstimateDSMCustomScanの返り値と大きさが一致する動的共有メモリ領域を指します。このコールバックは必須ではなく、カスタムスキャンプロバイダが並列実行をサポートする場合にのみ提供される必要があります。

```
void (*ReInitializeDSMCustomScan) (CustomScanState *node,
                                   ParallelContext *pcxt,
                                   void *coordinate);
```

カスタムスキャンプランノードが再スキャンしようとするときに、並列操作に必要な動的共有メモリを再初期化します。このコールバックは必須ではなく、カスタムスキャンプロバイダが並列実行をサポートする場合にのみ提供される必要があります。推奨する使い方としては、ReScanCustomScanコールバックはローカル状態だけをリセットし、このコールバックは共有状態だけをリセットするようにします。今のところ、このコールバックはReScanCustomScanの前に呼ばれますが、この順序関係には依存しない方が良いです。

```
void (*InitializeWorkerCustomScan) (CustomScanState *node,
                                    shm_toc *toc,
                                    void *coordinate);
```

InitializeDSMCustomScanによりリーダーにて設定された共有状態を元に、並列ワーカーのローカル状態を初期化します。このコールバックは必須ではなく、カスタムスキャンプロバイダが並列実行をサポートする場合にのみ提供される必要があります。

```
void (*ShutdownCustomScan) (CustomScanState *node);
```

ノードが実行を完了しないと思われるときに、リソースを解放します。これはすべての場合に呼ばれるわけではありません。ときには、この関数がまず呼ばれることなしに、EndCustomScanが呼ばれるかもしれません。パラレルクエリで使用されるDSMセグメントは、このコールバックが呼ばれた直後に削除されるので、DSMセグメントが削除される前に何らかのアクションを起こしたいカスタムスキャンプロバイダは、このメソッドを実装すべきです。

```
void (*ExplainCustomScan) (CustomScanState *node,
```

```
                            List *ancestors,
                            ExplainState *es);
```

カスタムスキャンの計画ノードのEXPLAINについて追加情報を出力します。このコールバックは必須ではありません。対象のリストやスキャンのリレーションなどScanStateに格納される共通データは、このコールバックがなくても表示されますが、このコールバックにより、追加のプライベートな状態が表示できるようになります。

第59章 遺伝的問い合わせ最適化

著者

このドキュメントはMartin Utesch（<utesch@aut.tu-freiberg.de>）によって、ドイツ、フライブルグにあるUniversity of Mining and TechnologyのInstitute of Automatic Controlのために書かれました。

59.1. 複雑な最適化問題としての問い合わせ処理

リレーショナル演算子の中で、処理と最適化が一番難しいのは結合です。実行可能な問い合わせ計画の数は問い合わせの中に含まれる結合の数によって指数関数的に増加します。個々の結合や、多様なインデックス（例えばPostgreSQLのB-tree、ハッシュ、GiST、GINなど）をリレーションのアクセスパスとして処理するため、様々な結合メソッド（例えばPostgreSQLのネステッドループ、ハッシュ結合、マージ結合など）を提供することが、さらなる最適化を行わなければならない腐心の原因となっています。

通常のPostgreSQL問い合わせオプティマイザは、候補ストラテジ空間にわたってしらみつぶしに近い検索を行います。IBMのSystem Rデータベースで初めて導入された、このアルゴリズムはほぼ最適な結合順を生成しますが、問い合わせ内の結合数が増えると膨大な処理時間とメモリ空間を必要とします。このため、通常のPostgreSQL問い合わせオプティマイザは結合するテーブル数の多い問い合わせには向いていません。

ドイツ、フライブルグにあるUniversity of Mining and TechnologyのInstitute of Automatic Controlでは、送電網の保守のための意志決定知識ベースシステムのためのバックエンドとしてPostgreSQL DBMSを使おうとしたため問題が起こりました。そのDBMSは知識ベースシステムの推論マシンのために、大規模な結合の問い合わせを処理する必要があったのです。こうした問い合わせに含まれる結合数を行うことは、通常の問い合わせオプティマイザでは実現不可能でした。

以下では、多数の結合を持つ問い合わせを効率的に行うことができるように、結合順問題を解決する遺伝的アルゴリズムの実装を説明します。

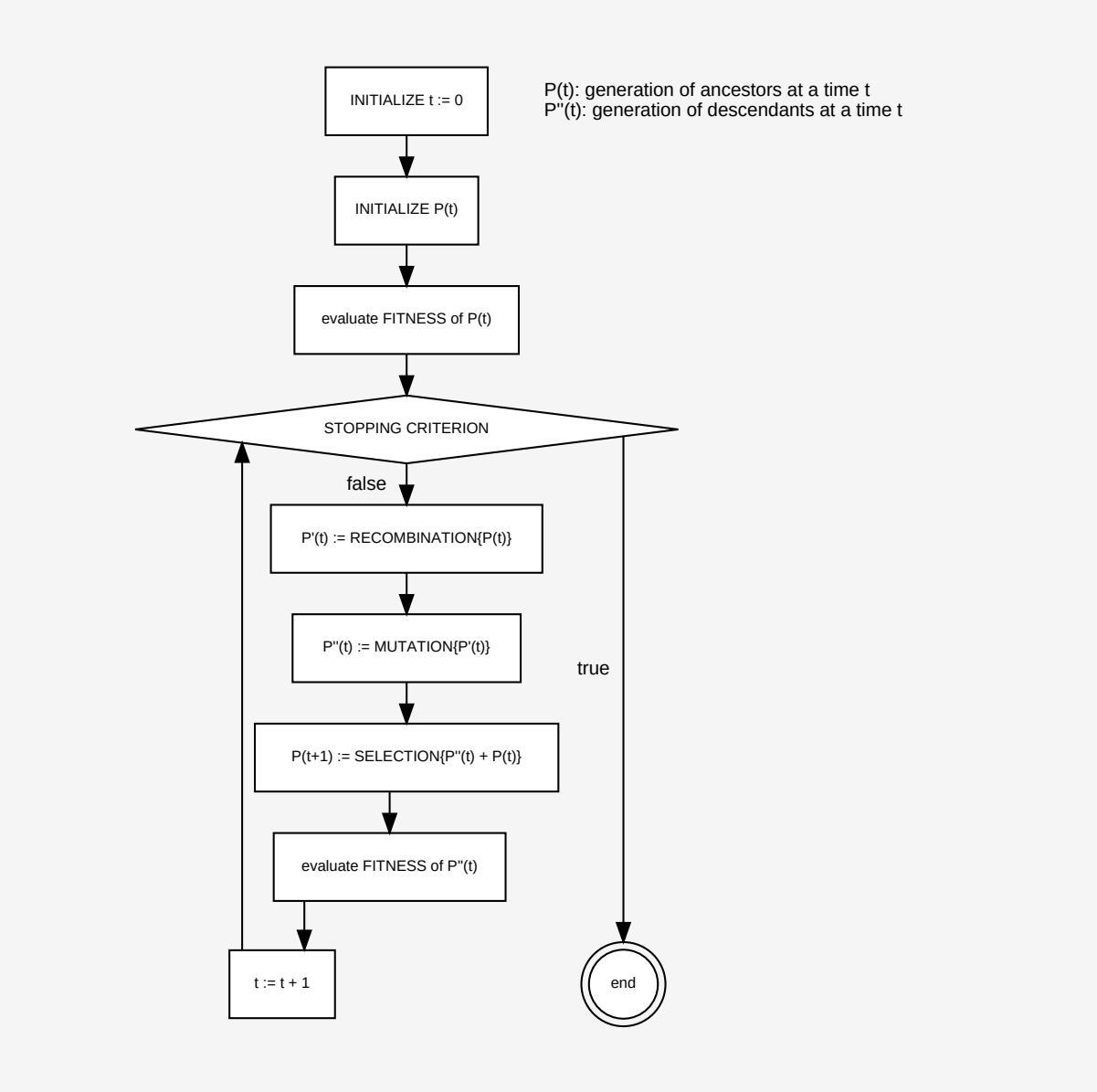
59.2. 遺伝的アルゴリズム

遺伝的アルゴリズム（GA）は発見的な最適化手法で、無作為の検索として働きます。最適化の問題に対する解の集合は個体群とみなされます。個体の環境への順応の度合は適応度によって指定されます。

検索空間の中で個体の同格性は、その実体が文字列の集合である染色体によって表現されます。遺伝子は最適化をしようとしている1つのパラメータの値を符号化する染色体の一部分です。遺伝子の符号化の典型的な例としてバイナリもしくは整数が挙げられます。

進化の過程のシミュレーションである、再組合せ、突然変異、淘汰を通して、祖先よりも適応度の平均が高い新世代の検索点が見つけられます。この段階を図 59.1で図解しています。

図59.1 遺伝的アルゴリズムの構造



comp.ai.geneticのFAQによると、GAが問題に対する純粋な無作為検索ではないことをどんなに強調してもし過ぎということはありません。GAは確率的なプロセスを使いますが、結果は明らかに（無作為よりもより良い）非無作為です。

59.3. PostgreSQLの遺伝的問い合わせ最適化（GEQO）

GEQOのモジュールは、問い合わせ最適化問題をあたかもよく知られている巡回セールスマン問題（TSP）のように扱います。可能な問い合わせプランは、整数の文字列として符号化されます。それぞれの文字列は、問い合わせの1つのリレーションから次へと結合の順番を表します。例えば、以下の結合ツリーは整数文字列「4-1-3-2」によって符号化されています。

```
/\
  /\ 2
 /\ 3
4  1
```

これが意味するのは、まずリレーション「4」と「1」を、次に「3」を、そして「2」を結合するということです。ここで1、2、3、4はPostgreSQLオプティマイザ内でリレーションIDを表します。

PostgreSQLにおけるGEQO実装の特有な特徴は下記の様なものです。

- 定常状態GAの使用（世代全体の置き換えではなく、個体の中で適応度の低いものだけの置き換え）は、改良された問い合わせ計画へ素早い収束を可能にします。これは、妥当な時間内での問い合わせ処理にはきわめて重要です。
- GAによるTSPの解決策の辺損失を低く抑えるため、非常に適した辺再組合せ交叉を使用します。
- TSPの合法な巡回を行うために必要な修復処理を要求しないように、遺伝的演算子の突然変異は無視しています。

GEQOモジュールの部品は D. WhitleyのGenitorアルゴリズムを適合させたものです。

GEQOモジュールにより、PostgreSQL問い合わせオプティマイザが、大きな結合問い合わせをしらみつぶし検索以外の方法で実行することが可能になります。

59.3.1. GEQOを使用した計画候補の生成

GEQOの計画作成では、個々のリレーションのスキャンに対する計画を生成するために標準のプランナが使用されます。そして、結合計画が遺伝的手法を用いて展開されます。上で示した通り、結合計画候補はそれぞれ、基本リレーションの結合順によって表現されています。初期段階では、GEQOコードは単純にランダムに取り得る結合順をいくつか生成します。考慮された結合順それぞれについて、標準プランナコードが呼び出され、その結合順を使用して問い合わせを行った場合のコストを推定します。（結合順の各段階において、全体で3つの取り得る結合戦略が考慮されます。そして、あらかじめ決められたリレーションスキャン計画もすべて利用可能です。推定コストとはこれらの可能性の中から最も安価なものです。）より低い推定コストの結合順を、より高い推定コストのものより「より高い適応度」と判断します。遺伝的アルゴリズムは適応度が低い候補を破棄します。そして、より多く合致する候補の遺伝子を組み合わせて、つまり、検討すべき新しい順序を作成するために既知の低コスト結合順をランダムに位置を選択して、新しい候補が生成されます。事前に設定された数まで結合順を検討するまで、この処理が繰り返されます。そして、この検索の間にもっとも優れたものが、最終的な計画を生成するために使用されます。

初期の群を選択する時、および、その後の最善の候補の「突然変異」の時に無作為な選択がなされますので、この処理は生来非決定論的なものです。選択された計画の予期せぬ変化を避けるために、GEQOアルゴリズムの各実行では乱数生成器を現在のgeqo_seedパラメータ設定で再スタートさせます。geqo_seedとその他のGEQOパラメータが変更されない限り、一定の問い合わせ（と統計のようなプランナへの他の入力）に対しては同じ計画が生成されます。異なる検索パスで実験するためには、geqo_seedを変更してみて下さい。

59.3.2. PostgreSQL GEQOの今後の実装作業

遺伝的アルゴリズムのパラメータ設定を改善するためにはまだ課題が残っています。src/backend/optimizer/geqo/geqo_main.cのgimme_pool_sizeとgimme_number_generationsというルーチンでは、次の2つの相反する要求を満たす妥協点を見つけなければいけません。

- 問い合わせ計画の最適性
- 計算時間

現在の実装では、各結合順候補の適応度は標準プランナの結合選択と、一から作成したコスト推定コードを実行して推定されます。異なる候補が同様の副結合順で使用されるにつれて、多くの作業が繰り返されることになります。これは、副結合のコスト推定を記憶することで、非常に高速になるはずです。この状態を記憶するために要するメモリ量が非合理的に拡大することを防止することが問題です。

最も基本的なレベルでは、TSP用に設計されたGAアルゴリズムを用いた問い合わせ最適化の解法が適切かどうかは明確ではありません。TSPの場合は、部分文字列（巡回経路の一部）に関連付けられたコストは残りの巡回経路と独立していますが、これは問い合わせ最適化の場合には確実に成り立ちません。したがって、辺再組合せ交叉が最も有効な突然変異手続きかどうかは疑わしいと言えます。

59.4. さらに深く知るには

次に示す資料は、さらに詳しい遺伝的アルゴリズムに関する情報が記載されています。

- The Hitch-Hiker's Guide to Evolutionary Computation[1], (FAQ for news://comp.ai.genetic)
- Evolutionary Computation and its application to art and design[2]、Craig Reynoldsによるもの
- [elma04]
- [fong]

[1] http://www.faqs.org/faqs/ai-faq/genetic/part1/

[2] https://www.red3d.com/cwr/evolve.html

第60章 テーブルアクセスメソッドのインタフェース定義

本章は、PostgreSQLのコアシステムと、テーブルの格納を制御するテーブルアクセスメソッドとのインタフェースを説明します。コアシステムはこのアクセスメソッドについて、ここで指定されたことのみを把握しています。これにより、追加コードを記述することで全く新しいアクセスメソッド種類を開発することができます。

各テーブルアクセスメソッドはpg_amシステムカタログの行で記述されます。pg_amのエントリではテーブルアクセスメソッドの名前とハンドラ関数を指定します。これらのエントリはSQLコマンドCREATE ACCESS METHODとDROP ACCESS METHODを使って、作成および削除することができます。

テーブルアクセスメソッドのハンドラ関数はinternal型の引数を一つ取り、table_am_handler疑似型を返すように宣言されなければなりません。この引数はハンドラ関数がSQLコマンドから直接呼び出されるのを防ぐためだけのダミーの値です。関数の結果はTableAmRoutine型の構造体のポインタでなければならず、そこにはテーブルアクセスメソッドを使用するためにコアコードが知る必要のあるすべてのことが含まれます。サーバ生存期間の間、戻り値は必要で、通常これはグローバルスコープでstatic const変数として定義することで達成されます。アクセスメソッドの*API*構造体とも呼ばれる、TableAmRoutine構造体はコールバックを使ってアクセスメソッドの振る舞いを定義します。これらのコールバックは通常のC関数へのポインタで、SQLレベルでは見ることも呼び出すこともできません。全てのコールバックとその振る舞いは、TableAmRoutine構造体（とコールバックの必要性を説明する構造体内のコメント）で定義されます。たいていのコールバックはラッパー関数を持ちます。これらはテーブルアクセスメソッドの（開発者ではなく）使用者の立場でドキュメント記載されています。詳細は、src/include/access/tableam.h[1]ファイルを参照してください。

アクセスメソッドを実装するには開発者は通常、タプルテーブルスロットのAM固有の型を実装する必要があります（src/include/executor/tuptable.h[2]を参照してください）。これはアクセスメソッド外のコードが、AMのタプルへの参照を保持できるようにして、そのタプルの列にアクセスできるようにするものです。

今のところAMが実際にデータを格納する方法は全く制限されていません。例えば、postgresの共有バッファキャッシュを使うことも、必須ではありませんが、可能です。使う場合、おそらく68.6に記述されたPostgreSQLの標準ページレイアウトを使うには有意義でしょう。

現在のテーブルアクセスメソッドAPIのそれなりに大きい制約は、AMが更新および/またはインデックスに対応したい場合、各タプルがブロック番号とアイテム番号から成るタプル識別子（TID）を持つ必要があることです（68.6も参照してください）。TIDsの下位要素が、例えばheapに対して持つのと同じ意味を持つことは、厳密には必要ありません。しかし、ビットマップスキャン対応（これは任意です）を望むなら、ブロック番号は局所性を備える必要があります。

クラッシュ安全性のために、AMはpostgresのWAL、あるいは、カスタム実装を使うことができます。WALを選んだ場合、汎用WALレコードが利用するか、新たなWALレコードタイプを実装することができます。汎用WALレコードは簡単ですが、大きなWAL容量を伴います。新たなWALレコード型を実装することは今のところコアコードの修正が必要です（具体的にはsrc/include/access/rmgrlist.h）。

[1] https://git.postgresql.org/gitweb/?p=postgresql.git;a=blob;f=src/include/access/tableam.h;hb=HEAD

[2] https://git.postgresql.org/gitweb/?p=postgresql.git;a=blob;f=src/include/executor/tuptable.h;hb=HEAD

異なるテーブルアクセスメソッドが単一トランザクション内でアクセスできるという類のトランザクション対応を実装するには、おそらくsrc/backend/access/transam/xlog.cの仕組みと注意深く統合することが必要でしょう。

新テーブルアクセスメソッドの開発者は、実装の詳細について、src/backend/access/heap/heapam_handler.cにある既存のheapの実装を参照できます。

第61章 インデックスアクセスメソッドのインタフェース定義

本章は、PostgreSQLのコアシステムと個々のインデックス種類を管理するインデックスアクセスメソッドとの間のインタフェースを定義します。コアシステムはインデックスの仕様のみを把握しています。したがって、追加コードを記述することで完全に新しいインデックス種類を開発することができます。

PostgreSQLのインデックスはすべて、技術的には補助的なインデックスとして知られるものです。つまり、インデックスは対象となるテーブルファイルとは物理的に分かれています。各インデックスは独自の物理的なリレーションとして格納され、また、pg_classカタログ内の項目として記述されます。インデックスの内容は完全にそのインデックスアクセスメソッドの制御下にあります。実際、すべてのインデックスアクセスメソッドは、通常の格納マネージャとバッファマネージャを使用してインデックスの内容にアクセスできるように、インデックスを標準サイズのページに分割します。(既存のすべてのインデックスアクセスメソッドではさらに、68.6で説明する標準ページレイアウトを使用し、そのほとんどは同じ書式をインデックスタプルヘッダに使用します。しかし、これはアクセスメソッドに強制されていることではありません。)

インデックスは効率的にあるデータキー値を、インデックスの親テーブル内の行バージョン(タプル)のタプル識別子言い換えるとTIDに関連付けます。TIDは、ブロック番号、ブロック内の項目番号(68.6を参照)から構成されます。これは、特定の行バージョンをテーブルから取り出すのに十分な情報です。MVCCでは1つの論理的な行に複数の現在のバージョンがあることを、インデックスが直接意識することはありません。インデックスでは、各タプルは、独自にインデックス項目を持たなければならない独立したオブジェクトです。したがって、行を更新すると、キーの値が変わっていなかってとしても、その行に対してまったく新しいインデックス項目が作成されます。(HOTタプルはこの説明の例外ですが、インデックスはこれらにも関与しません。)(バキューム実行によって)無効タプル自身が回収された時に、無効タプルに対するインデックス項目は回収されます。

61.1. インデックスの基本的API構造

各インデックスメソッドはpg_amシステムカタログの行で説明されます。pg_amエントリはインデックスアクセスメソッドの名前とハンドラ関数を指定します。これらのエントリはSQLコマンドCREATE ACCESS METHODとDROP ACCESS METHODを使って、作成および削除することができます。

インデックスメソッドのハンドラ関数は、internal型の引数を1つ取り、疑似型index_am_handlerを返すものとして宣言しなければなりません。引数は単にハンドラ関数がSQLコマンドから直接呼び出されるのを防ぐためのダミーの値です。関数の結果は型IndexAmRoutineのpallocされた構造体でなければならず、そこにはインデックスアクセスメソッドを使用するためにコアコードが知っている必要のあるすべてのことが含まれています。IndexAmRoutine構造体は、アクセスメソッドの*API*構造体とも呼ばれ、複数列のインデックスをサポートするかどうかなどといった、アクセスメソッドに関する様々な既定の属性を指定するフィールドが含まれます。さらに重要なことに、この構造体にはアクセスメソッドのサポート関数へのポインタが含まれ、これによってインデックスにアクセスするためのすべての実際の処理が行われます。これらのサポート関数は単なるCの関数で、SQLレベルでは見ることも呼び出すこともできません。サポート関数は61.2で説明されています。

構造体IndexAmRoutineは以下のように定義されています。

```
typedef struct IndexAmRoutine
{
    NodeTag     type;

    /*
     * Total number of strategies (operators) by which we can traverse/search
     * this AM.  Zero if AM does not have a fixed set of strategy assignments.
     */
    uint16      amstrategies;
    /* total number of support functions that this AM uses */
    uint16      amsupport;
    /* opclass options support function number or 0 */
    uint16      amoptsprocnum;
    /* does AM support ORDER BY indexed column's value? */
    bool        amcanorder;
    /* does AM support ORDER BY result of an operator on indexed column? */
    bool        amcanorderbyop;
    /* does AM support backward scanning? */
    bool        amcanbackward;
    /* does AM support UNIQUE indexes? */
    bool        amcanunique;
    /* does AM support multi-column indexes? */
    bool        amcanmulticol;
    /* does AM require scans to have a constraint on the first index column? */
    bool        amoptionalkey;
    /* does AM handle ScalarArrayOpExpr quals? */
    bool        amsearcharray;
    /* does AM handle IS NULL/IS NOT NULL quals? */
    bool        amsearchnulls;
    /* can index storage data type differ from column data type? */
    bool        amstorage;
    /* can an index of this type be clustered on? */
    bool        amclusterable;
    /* does AM handle predicate locks? */
    bool        ampredlocks;
    /* does AM support parallel scan? */
    bool        amcanparallel;
    /* does AM support columns included with clause INCLUDE? */
    bool        amcaninclude;
    /* does AM use maintenance_work_mem? */
    bool        amusemaintenanceworkmem;
    /* OR of parallel vacuum flags */
    uint8       amparallelvacuumoptions;
    /* type of data stored in index, or InvalidOid if variable */
    Oid         amkeytype;
```

```
    /* interface functions */
    ambuild_function ambuild;
    ambuildempty_function ambuildempty;
    aminsert_function aminsert;
    ambulkdelete_function ambulkdelete;
    amvacuumcleanup_function amvacuumcleanup;
    amcanreturn_function amcanreturn;   /* can be NULL */
    amcostestimate_function amcostestimate;
    amoptions_function amoptions;
    amproperty_function amproperty;     /* can be NULL */
    ambuildphasename_function ambuildphasename;   /* can be NULL */
    amvalidate_function amvalidate;
    ambeginscan_function ambeginscan;
    amrescan_function amrescan;
    amgettuple_function amgettuple;     /* can be NULL */
    amgetbitmap_function amgetbitmap;   /* can be NULL */
    amendscan_function amendscan;
    ammarkpos_function ammarkpos;       /* can be NULL */
    amrestrpos_function amrestrpos;     /* can be NULL */

    /* interface functions to support parallel index scans */
    amestimateparallelscan_function amestimateparallelscan;    /* can be NULL */
    aminitparallelscan_function aminitparallelscan;    /* can be NULL */
    amparallelrescan_function amparallelrescan;    /* can be NULL */
} IndexAmRoutine;
```

使い易くするために、インデックスアクセスメソッドはまた、pg_opfamily、pg_opclass、pg_amopおよびpg_amproc内で定義される、複数の演算子族と演算子クラスを持ちます。これらの項目により、プランナは、このアクセスメソッドのインデックスがどのような問い合わせ条件に対して使用できるかを決定することができます。演算子族と演算子クラスについては、37.16で説明します。これは本章を読む上で必要となる資料です。

個々のインデックスは、インデックスを物理的なリレーションとして記述するpg_class項目と、インデックスの論理的な内容、つまり、インデックスが持つインデックス列の集合とその列の意味を、関連する演算子クラスで再現されたものとして表すpg_index項目とで定義されます。インデックス列(キー値)は、背後のテーブルの単純な列、あるいは、テーブル行に対する式とすることができます。通常、インデックスアクセスメソッドはインデックスキー値が何を表すかについて考慮しません。(常に計算済みのキー値として扱われます。)しかし、pg_index内の演算子クラスの情報を深く考慮します。この両方のカタログ項目は、インデックスに対するすべての操作に渡されるRelationデータ構造の一部としてアクセスすることができます。

IndexAmRoutineのフラグフィールドの中には、意味がわかりにくいものがあります。amcanuniqueの必要条件は61.5で説明されています。amcanmulticolフラグはアクセスメソッドが複数列に対するインデックスをサポートすることを表し、amoptionalkeyは、インデックス可能な制限句が最初のインデックス列に指定されていないスキャンを許可することを表します。amcanmulticolが偽の場合、amoptionalkeyは基本的に、アクセスメソッドが制限句なしで完全なインデックススキャンをサポートするかどうかを表します。複数列に対するインデックスをサポートするアクセスメソッドは、最初の列以降のすべてまたは一部の列に関する制限がなくてもスキャンをサポートしなければなりません。しかし、最初のインデックス列にいくつかの制限を要求す

ることは認められています。これは、amoptionalkeyを偽に設定することで通知されます。インデックスアクセスメソッドがamoptionalkeyを偽にする1つの理由は、NULLをインデックス付けしない場合です。ほとんどのインデックス可能な演算子は厳密で、NULL値の入力に対して真を返すことができませんので、NULLに対してインデックス項目を格納しないことは一見魅力的です。これはインデックススキャンによって何も返しません。しかし、最初のインデックス列に対する制限がないインデックススキャンでは、この引数は失敗します。プランナがこうしたスキャンキーをまったく持たないインデックスを使用することを決定する可能性がありますので、実際これは、amoptionalkeyが真のインデックスはNULLインデックスを持たなければならないことを意味します。関連する制限として、プランナはこれらの列を制限しない問い合わせでインデックスを使用できることを前提とするため、複数のインデックス列をサポートするインデックスアクセスメソッドは1番目の後の列でNULL値のインデックスをサポートしなければならないということがあります。例えば、(a,b)に対するインデックスに、WHERE a = 4という条件で問い合わせを行うことを考えてみます。システムは、このインデックスをa = 4を持つ行をスキャンすることに使用できるものと仮定します。これはもし、bがNULLの場合の行をインデックスが省略する場合は間違っています。しかし、最初のインデックス列がNULLの場合に行を省略することは問題ありません また、NULLをインデックス付けするインデックスアクセスメソッドはamsearchnullsを設定する可能性があります。これは検索条件としてIS NULLおよびIS NOT NULL句をサポートすることを示します。

61.2. インデックスアクセスメソッド関数

インデックスアクセスメソッドがIndexAmRoutineで提供しなければならない、インデックス構築および保守関数を以下に示します。

```
IndexBuildResult *
ambuild (Relation heapRelation,
         Relation indexRelation,
         IndexInfo *indexInfo);
```

新しいインデックスを構築します。空のインデックスリレーションが物理的に作成されます。これは、アクセスメソッドが必要とする何らかの固定データと、テーブル内に既に存在するすべてのタプルに対応する項目が書き込まれなければなりません。通常、ambuild関数はtable_index_build_scan()を呼び出し、既存のタプルをテーブルからスキャンし、インデックスに挿入しなければならないキーを計算します。この関数は、 新しいインデックスに関する統計情報を含むpallocされた構造体を返さなければなりません。

```
void
ambuildempty (Relation indexRelation);
```

空のインデックスを構築し、それを指定されたリレーションの初期フォーク(INIT_FORKNUM)に書き出します。このメソッドはログを取らないインデックスに対してのみ呼び出されます。初期フォークに書き出された空のインデックスは、サーバの再起動の度に主リレーションフォークにコピーされます。

```
bool
aminsert (Relation indexRelation,
          Datum *values,
          bool *isnull,
          ItemPointer heap_tid,
```

```
          Relation heapRelation,
          IndexUniqueCheck checkUnique,
          IndexInfo *indexInfo);
```

既存のインデックスに新しいタプルを挿入します。values配列とisnull配列がインデックスされるキー値を提供するもので、heap_tidがインデックスされるTIDです。アクセスメソッドが一意なインデックスをサポートする場合（そのamcanuniqueが真の場合）、checkUniqueは実行する一意性検査の種類を示します。これは一意性制約が遅延可能か否かによって変わります。61.5を参照してください。通常アクセスメソッドは、一意性検査を行う時にheapRelationパラメータのみを必要とします (タプルの有効性を検証するためにヒープ内を検索しなければなりません)。

checkUniqueがUNIQUE_CHECK_PARTIALの場合、関数の論理型の結果値で十分です。この場合、真の結果は新しい項目は一意であることが確認されたことを、一方偽の結果は一意でない可能性があること(遅延一意性検査を予定しなければならないこと)を意味します。他の場合では、一定の偽という結果が推奨されます。

一部のインデックスではすべてのタプルをインデックス付けしない可能性があります。タプルがインデックス付けされない場合、aminsertは何も行わずに戻らなければなりません。

SQL文の中で、インデックスAMがインデックスへの連続的な挿入をまたがってデータをキャッシュすることが望ましい場合は、indexInfo->ii_Contextにメモリを確保し、そのデータへのポインタをindexInfo->ii_AmCache（初期値はNULLです)に格納することができます。

```
IndexBulkDeleteResult *
ambulkdelete (IndexVacuumInfo *info,
              IndexBulkDeleteResult *stats,
              IndexBulkDeleteCallback callback,
              void *callback_state);
```

インデックスからタプル（複数可）を削除します。これは「一括削除」操作を行いますが、インデックス全体をスキャンし、各項目に対して削除すべきかどうか検査を行うように実装されることが想定されています。渡されるcallback関数は、callback(TID, callback_state) returns boolという形で、参照用TIDで識別されるインデックス項目を削除すべきかどうか決定するために呼び出さなければなりません。NULLまたはpallocした削除操作の影響に関する統計情報を含む構造体を返さなければなりません。amvacuumcleanupに渡さなければならない情報がなければ、NULLを返しても問題ありません。

maintenance_work_memの制限により、多くのタプルが削除される時、ambulkdeleteを複数回呼び出す必要があるかもしれません。stats引数は、このインデックスに対する前回の呼び出し結果です。（VACUUM操作における最初の呼び出しではこれはNULLです。）これにより、アクセスメソッドは操作全体に跨った統計情報を計算することができます。典型的に、渡されたstatsがNULLでない場合、ambulkdeleteは同じ構造体を変更し、返します。

```
IndexBulkDeleteResult *
amvacuumcleanup (IndexVacuumInfo *info,
                 IndexBulkDeleteResult *stats);
```

VACUUM操作（0回以上のambulkdelete呼び出し）後の整理を行います。これは、インデックス統計情報を返す以上の処理を行う必要はありません。しかし、空のインデックスページの回収などの一括整理を行う可能性があります。statsは最後のambulkdelete呼び出しが返したものです。削除する必要があるタプルが存在

しなかったためにambulkdeleteが呼び出されなかった場合はNULLとなります。結果はNULLでなければ、pallocされた構造体でなければなりません。含まれる統計情報はpg_classを更新するために使用され、また、VERBOSEが指定されたVACUUMによって報告されます。VACUUM操作の間にインデックスがまったく変わらなかった場合はNULLを返しても問題ありません。しかし、そうでなければ正しい統計情報を返さなければなりません。

PostgreSQL 8.4の時点で、amvacuumcleanupもANALYZE操作の完了時点にも呼び出されます。この場合、statsは常にNULLで、戻り値はまったく無視されます。この事象はinfo->analyze_onlyを検査することで識別されます。アクセスメソッドがそのような呼び出しで挿入後の整理以外何もしないように、そしてそれは自動バキュームワーカプロセスのみであるようにすることを推奨します。

```
bool
amcanreturn (Relation indexRelation, int attno);
```

IndexTuple形式のインデックスエントリをインデックスが設定された列の値として返すことにより、そのインデックスが指定された列でインデックスオンリースキャンをサポートしているかどうかを判断します。属性番号は1始まり、すなわち最初の列の属性番号は1です。インデックスオンリースキャンがサポートされている場合は真が返され、サポートされていない場合は偽が返ります。アクセスメソッドがインデックスオンリースキャンをサポートしていない場合、IndexAmRoutine構造体のamcanreturnフィールドをNULLにセットすることができます。

```
void
amcostestimate (PlannerInfo *root,
                IndexPath *path,
                double loop_count,
                Cost *indexStartupCost,
                Cost *indexTotalCost,
                Selectivity *indexSelectivity,
                double *indexCorrelation,
                double *indexPages);
```

インデックススキャンのコストを推定します。この関数については後述の61.6で説明します。

```
bytea *
amoptions (ArrayType *reloptions,
           bool validate);
```

インデックス用のreloptionsの解析と検証を行います。インデックスに非NULLのreloptions配列が存在する場合にのみ呼び出されます。reloptionsは、name=value形式の項目からなる、text型の配列です。この関数はbytea型の値を生成しなければならず、この値はインデックスのrelcache項目のrd_optionsフィールドにコピーされます。bytea型の値の内容はアクセスメソッドが独自に定義できるように開放されています。標準のアクセスメソッドのほとんどはすべてStdRdOptions構造体を使用します。validateが真の場合、何らかのオプションが認識できなかった場合や無効な値が存在した場合、この関数は適切なエラーメッセージを報告しなければなりません。validateが偽の場合、無効な項目は単に無視されます。(読み込みオプションが既にpg_catalogに格納されている場合validateは偽です。アクセスメソッドがそのオプション用の規則を変更した場合にのみ、無効な項目が検出されます。そして、その場合、古い項目を無視することが適切です。)デフォルトの動作を行わせたい場合はNULLを返しても問題ありません。

```
bool
amproperty (Oid index_oid, int attno,
            IndexAMProperty prop, const char *propname,
            bool *res, bool *isnull);
```

amproperty メソッドにより、インデックスメソッドは pg_index_column_has_property および関連する関数のデフォルトの動作を上書きすることができます。インデックスアクセスメソッドがインデックスの属性の問い合わせについて特別な動作をしないのなら、IndexAmRoutine 構造体の amproperty フィールドは NULL にすることができます。そうでなければ、amproperty は pg_indexam_has_property の呼び出しに対し、index_oid と attno をいずれもゼロにして、pg_index_has_property の呼び出しに対して index_oid が有効、attno がゼロで、あるいは pg_index_column_has_property の呼び出しに対して index_oid が有効、attno が 1 以上で呼び出されます。prop は検査対象の属性を指定する列挙型の値、propname は元の属性の名称の文字列です。コアのコードが属性名を認識しない場合、prop は AMPROP_UNKNOWN になります。アクセスメソッドはカスタム属性名を定義して、マッチするものを propname で確認する（コアコードとの一貫性のため、pg_strcasecmp を使ってください）ことができます。コアコードに既知の名前については、prop を検査する方が良いです。amproperty メソッドが true を返すなら、それは属性検査の結果が決定したということで、*res を返すべき論理値にセットするか、NULL を返すために *isnull を true にセットするかしなければなりません。（どちらの参照変数も、呼び出しの前に false に初期化されます。）amproperty メソッドが false を返すなら、コアコードは属性検査の結果を決定するために、通常の手続きを進めます。

順序付け演算子をサポートするアクセスメソッドは、AMPROP_DISTANCE_ORDERABLE の属性検査を実装する必要があります。なぜなら、コアコードはそれをどうすれば良いか知らないため、NULL を返すからです。コアコードのデフォルトの動作であるインデックスのオープンと amcanreturn の呼び出しよりも安価にできるのであれば、AMPROP_RETURNABLE の検査を実装するのは利点となります。その他のすべての標準属性に対しては、デフォルトの動作が満足できるもののはずです。

```
char *
ambuildphasename (int64 phasenum);
```

指定されたビルドフェーズ番号のテキスト名を返します。フェーズ番号は、pgstat_progress_update_param インタフェースを介してインデックス構築中に報告されたものです。それから、フェーズ名は pg_stat_progress_create_index ビューで公開されます。

```
bool
amvalidate (Oid opclassoid);
```

指定の演算子クラスについて、アクセスメソッドが合理的にできる限りにおいて、カタログエントリを検証します。例えば、これには必要なすべてのサポート関数が提供されていることのテストが含まれるかもしれません。amvalidate 関数は演算子クラスが無効なときは偽を返さなければなりません。問題点は ereport メッセージにより報告されます。

当然ながらインデックスの目的は、よく修飾子やスキャンキーと呼ばれる、インデックス可能な WHERE 条件を満たすタプルのスキャンをサポートすることです。インデックススキャンのセマンティクスは後の 61.3 でより詳しく説明します。インデックスアクセスメソッドは「単純」インデックススキャン、「ビットマップ」インデックススキャン、またはこれら双方を提供します。インデックスアクセスメソッドが提供しなければならない、もしくは提供する可能性のあるスキャン関連の関数を以下に示します。

```
IndexScanDesc
ambeginscan (Relation indexRelation,
             int nkeys,
             int norderbys);
```

インデックススキャンを準備します。nkeysおよびnorderbysパラメータは、スキャンで使用される等価性演算子と順序付け演算子の個数を表します。これらは領域を割り当てる目的で便利かもしれません。スキャンキーの実値がまだ提供されていないことに注意してください。結果はpallocした構造体でなければなりません。実装上の理由により、インデックスアクセスメソッドはRelationGetIndexScan()呼び出しによってこの構造体を作成しなければなりません。ほとんどの場合、ambeginscanはこの呼び出しとおそらくロックの獲得の他にはほとんど何も行いません。インデックススキャンを始める際の興味深い部分は、amrescanにあります。

```
void
amrescan (IndexScanDesc scan,
          ScanKey keys,
          int nkeys,
          ScanKey orderbys,
          int norderbys);
```

インデックススキャンを起動または再起動します。スキャンキーを新しくすることもできます。(過去に渡されたキーを使用して再起動するには、key、orderbys、またはその両方にNULLを渡します。) ambeginscanに渡したキー演算子、順序付け演算子の個数より多くを使用することはできないことに注意してください。実際には、ネステッドループ結合によって新しい外部タプルが選択され、同じスキャンキー構造体で新しいキー比較値が必要とされた場合に、この再起動機能は使用されます。

```
boolean
amgettuple (IndexScanDesc scan,
            ScanDirection direction);
```

指定されたスキャン内から指定された方向(インデックス内の前方または後方)で次のタプルを取り出します。タプルを取り出した場合は真を返します。一致するタプルが残っていない場合は偽を返します。真の場合、そのタプルのTIDがscanに格納されます。「成功」とは、単にインデックスにスキャンキーに一致する項目があったことを意味しているだけです。タプルが必ずヒープ内に存在することや、呼び出し元のスナップショットの試験を通過したことを意味してはいません。成功の暁には、amgettupleはscan->xs_recheckを真か偽かに設定しなければなりません。偽の意味は、インデックス項目が確実にスキャンキーに一致することです。真の意味は、これが確かなことではなく、スキャンキーで表示された条件がヒープタプルを取り出された後で再検査されなければならないことです。この対策は「非可逆」インデックス演算子をサポートします。再検査はスキャン条件のみに拡大適用されることに注意してください。部分インデックス述語(もしあれば)はamgettuple呼び出し元で決して再検査されません。

そのインデックスがインデックスオンリースキャンをサポートしている場合(つまりamcanreturnが真を返す場合)、そのアクセスメソッドはスキャンが成功したならばscan->xs_want_itupも確認し、それが真の場合、 そのインデックスエントリに対応する元のインデックスされたデータを返さなければなりません。返却されるデータは、scan->xs_itupdescタプルディスクリプタとともにscan->xs_itupに格納されたIndexTupleポインタの形式か、あるいは、scan->xs_hitupdescタプルディスクリプタとともにscan->xs_hitupに格納されたHeapTupleポインタの形式です。(後者の形式は、再構成されたデータがIndexTupleに収まらない場合に

使用するべきです。) どちらの場合でも、そのポインタが参照するデータの管理はアクセスメソッドの責任です。データは少なくともamgettuple、amrescanまたはamendscanによってスキャンされるまでよい状態を保たなくてはなりません。

amgettuple関数は、アクセスメソッドが「単純」インデックススキャンをサポートするときのみ提供される必要があります。そうでなければ、IndexAmRoutine構造体のamgettupleフィールドはNULLに設定されなければなりません。

```
int64
amgetbitmap (IndexScanDesc scan,
             TIDBitmap *tbm);
```

指定されたスキャンから全てのタプルを取り出し、呼び出し側が提供するTIDBitmapにそれらを付加します(つまり、既にビットマップ内にある集合とタプルIDの集合とのORを取ります)。取り出されたタプル数が返されます(例えばいくつかのAMは重複を検出しませんので、これは単なる概算です)。タプルIDをビットマップに挿入する間、amgetbitmapは特定のタプルIDに必要なスキャン条件の再検査を示すことが可能です。これはamgettupleのxs_recheck出力パラメータに類似しています。注意:現在の実装においてこの機能の提供はビットマップそのものの非可逆格納を提供するのに結びついていて、したがって呼び出し側はスキャン条件と部分インデックスの述部(存在すれば)を再検査可能なタプルに対して再検査します。とは言っても常に正しいとは限りません。amgetbitmapおよびamgettupleを同じインデックススキャン内で使用することはできません。61.3で説明した通り、amgetbitmapを使用する場合には他にも制限があります。

amgetbitmap関数はアクセスメソッドが「ビットマップ」インデックススキャンをサポートしている場合のみ必要です。そうでなければ、IndexAmRoutine構造体の中のamgetbitmapフィールドはNULLに設定されなければなりません。

```
void
amendscan (IndexScanDesc scan);
```

スキャンを停止し、リソースを解放します。scan構造体自体は解放すべきではありません。アクセスメソッドで内部的に取られたロックやピンは、ambeginscanや他のスキャン関連の関数により確保されたメモリと同様に解放しなければなりません。

```
void
ammarkpos (IndexScanDesc scan);
```

現在のスキャン位置を記録します。アクセスメソッドは1スキャン当たり1つの記録済みスキャンのみをサポートしなければなりません。

ammarkpos関数はアクセスメソッドが順序付けされたスキャンをサポートする場合にのみ提供する必要があります。そうでなければ、そのIndexAmRoutine構造体のammarkposフィールドはNULLに設定しても構いません。

```
void
amrestrpos (IndexScanDesc scan);
```

もっとも最近に記録された位置にスキャンを戻します。

amrestrpos関数はアクセスメソッドが順序付けされたスキャンをサポートする場合にのみ提供する必要があります。そうでなければ、そのIndexAmRoutine構造体のamrestrposフィールドはNULLに設定しても構いません。

通常のインデックススキャンのサポートに加え、ある種のインデックスは、複数のバックエンドが協調してインデックススキャンを実行するパラレルインデックススキャンをサポートすることができます。インデックスアクセスメソッドは、協調するプロセスが、通常の非パラレルインデックススキャンが実行対象とする行のサブセットを返しつつ、しかもそれらのサブセットの合計が、通常の非パラレルインデックススキャンが返すタプルの集合と同じになるように調整しなければなりません。それだけでなく、パラレルスキャンが返すタプル全体の順序付けが想定されていない場合でも、協調するバックエンドが返すサブセットのタプルの順序付けは、要求された順序付けと一致しなければなりません。パラレルインデックススキャンをサポートするために、以下の関数を実装することができます。

```
Size
amestimateparallelscan (void);
```

パラレルスキャンを実行するために、アクセスメソッドによって必要とされる動的共有メモリのバイト数を推測し、返します。(この数値は、ParallelIndexScanDescDataのAM独立データに必要となる量に追加するための値であり、それを置き換えるものではありません。)

パラレルスキャンをサポートしない、あるいはメモリ領域への追加のバイト数が0のアクセスメソッドでは、この関数を実装する必要はありません。

```
void
aminitparallelscan (void *target);
```

この関数は、パラレルスキャンの最初に動的共有メモリを初期化するために呼ばれます。targetは、前もってamestimateparallelscanが返したバイト数を少なくとも持つ領域を指し、この関数はその分だけのスペースを使って必要なデータを保管することができます。

パラレルスキャンをサポートしない、あるいは共有メモリスペースの初期化が必要ないアクセスメソッドでは、この関数を実装する必要はありません。

```
void
amparallelrescan (IndexScanDesc scan);
```

実装された場合、この関数はパラレルインデックススキャンを再起動しなければならない時に呼ばれます。この関数は、aminitparallelscanが設定した共有状態を初期化し、スキャンが最初から再開できるようにします。

61.3. インデックススキャン

インデックススキャンでは、スキャンキーに一致するものと示したすべてのタプルのTIDを繰り返すことに関する責任をインデックスアクセスメソッドが持ちます。アクセスメソッドには、実際のインデックスの親テーブルからのタプルの取り出しやタプルがスキャンの可視性テストや他の条件を通過したかどうかの決定は含まれません。

スキャンキーは、index_key operator constantという形式のWHERE句の内部的表現です。ここで、index_keyは、インデックス列の1つで、operatorはインデックス列に関連した演算子族のメンバの1つです。インデックススキャンは、暗黙的にAND演算される0個以上のスキャンキーを持ちます。返されるタプルは指定された条件を満たすものと想定されます。

アクセスメソッドはインデックスがある特定の問い合わせに対し非可逆、または再検査を要求するかどうかを報告することができます。これは、インデックススキャンがスキャンキーを満たすすべての項目と、それに加えて、満たさない可能性のある項目を返すことを意味します。コアシステムのインデックススキャン機構はヒープタプルに対し、本当に選択されるべきかどうかを検証するためにその演算子をインデックス条件に再度適用します。再検査オプションが指定されない場合、インデックススキャンは一致する項目の集合を返さなければなりません。

確実に、指定されたスキャンキーすべてに一致するもののみをすべて正しく見つけ出すことは、完全にアクセスメソッドの責任であることに注意してください。また、コアシステムは、冗長かどうかや矛盾するかどうかを決定するための意味的な解析を行わず、単にインデックスキーと演算子族に一致するWHERE句をすべて渡します。例えば、WHERE x > 4 AND x > 14があり、xがB-treeインデックス列であったとすると、これは、B-tree amrescan関数に任されて、最初のスキャンキーが冗長であり、無視できることが認知されます。amrescanにおける前処理の必要性は、インデックスアクセスメソッドがスキャンキーを「正規化」形式にする必要があるかどうかに依存します。

一部のアクセスメソッドは、他では行いませんが、十分に定義された順序でインデックス項目を返します。アクセスメソッドが出力の順序付けをサポートできるようにする方法は、実質2種類存在します。

- 常にそのデータ(btreeなど)の自然な順序で項目を返すアクセスメソッドはamcanorderを真に設定しなければなりません。現在、こうしたアクセスメソッドは、その等価性と順序付け演算子でbtree互換の戦略番号を使用しなければなりません。

- 順序付け演算子をサポートするアクセスメソッドはamcanorderbyopを真に設定しなければなりません。これは、インデックスがORDER BY index_key operator constantを満たす順序で項目を返すことができることを示します。前述の通り、この形式のスキャン修飾子をamrescanに渡すことができます。

amgettuple関数はdirection引数を持ちます。これはForwardScanDirection(通常の場合)またはBackwardScanDirectionのいずれかを取ることができます。amrescan後の最初の呼び出しがBackwardScanDirectionを指定していた場合、一致したインデックス項目は通常の前から後ろという方向ではなく、後ろから前という方向でスキャンされます。そのため、amgettupleは通常ならばインデックス内の最初に一致したタプルを返すところですが、最後に一致したタプルを返さなければなりません。(これはamcanorderが真に設定されたアクセスメソッドでのみ発生します。) 最初の呼び出しの後、amgettupleは、最も最近に返された項目からどちらの方向にスキャンを進めるかを準備しなければなりません。(しかしamcanbackwardが偽であれば、引き続くすべての呼び出しは最初のものと同じ方向を持ちます。)

順序付けされたスキャンを提供するアクセスメソッドはスキャン内位置の「記録」をサポートしなければならず、また、後でその記録された位置に戻ることをサポートしなければなりません。同じ位置が複数回記録されるかもしれません。しかし、スキャン内の1つの位置のみを記録する必要があります。新しいammarkpos呼び出しにより前回記録された位置は上書きされます。順序付けされたスキャンをサポートしないアクセスメソッドはIndexAmRoutineでammarkpos関数およびamrestrpos関数を提供する必要はないので、これらのポインタをNULLにセットしてください。

スキャン位置と記録された位置(もしあれば)の両方は、インデックス内の同時挿入や削除という観点における一貫性を保持しなければなりません。スキャンが始まった時に存在していた場合、項目を見つけ出したス

キャンが新しく挿入された項目を返さなかったとしても問題ありません。このような場合のスキャンでは、再スキャンやバックアップによって、あたかも最初の時点で返されたものとして項目が返されます。同様に、同時実行削除によってスキャンの結果に影響が出るかもしれません。重要なことは、挿入や削除によって、その項目自体が挿入・削除されていない項目がスキャンで失われたり二重になったりすることが起こらないという点です。

インデックスが設定された列値がインデックスに格納されている(かつ、不可逆表現ではない)場合、ヒープタプルのTIDではなくインデックスに格納された実際のデータを返すインデックスオンリースキャンをサポートするのに有用です。これは、可視性マップによってTIDが全可視のページ上にあると判断できる場合にI/Oを避けるだけのことです。判断できない場合はMVCCを確認するためにヒープタプルにアクセスしなくてはなりません。しかしその動作はアクセスメソッドでは考慮されていません。

amgettupleを使用する代わりに、amgetbitmapを使用して、一回の呼出しですべてのタプルを取り出してインデックススキャンを行うことができます。これはアクセスメソッド内でのロック/ロック解除という過程を防ぐことができますので、amgettupleよりもかなり効率的です。実際には、amgetbitmapはamgettuple呼び出しを繰り返すことと同じ効果を持つはずですが、物事を単純化するために複数の制限を加えています。まず第一に、amgetbitmapは一回ですべてのタプルを返し、スキャン位置の記録と位置戻しをサポートしません。第二に、特定の順序付けをまったく持たないビットマップの中にタプルが返されます。これはamgetbitmapがdirection引数を取らない理由です。(順序付け演算子はこのようなスキャンでは決して与えられません。) また、amgetbitmapによるインデックスオンリースキャンは提供されていません。なぜなら、インデックスタプルの内容を返す手段がないからです。最後に、amgetbitmapは返されたタプルに関し、61.4に記載した意味でのロックを保証しません。

アクセスメソッドの内部実装がどちらか片方のAPIにそぐわない場合、amgettupleを実装せずamgetbitmapのみを実装、またはその逆も許されていることに注意してください。

61.4. インデックスのロック処理に関する検討

インデックスアクセスメソッドは、複数のプロセスによるインデックスの同時更新を取り扱えなければなりません。PostgreSQLコアシステムはインデックススキャン中にインデックスに対してAccessShareLockを獲得します。また、(通常のVACUUMを含む)インデックスの更新中にRowExclusiveLockを獲得します。これらの種類のロックは競合しませんので、アクセスメソッドは必要になるかもしれない粒度の細かなロック処理に関して責任を持ちます。インデックスの生成、破棄、REINDEX時にインデックス全体に対する排他ロックが獲得されます。

同時更新をサポートするインデックス種類を構築することは通常、必要な動作について広範かつ微細にわたる解析が必要です。B-treeおよびハッシュインデックス種類では、src/backend/access/nbtree/README と src/backend/access/hash/README にある設計に関する決定事項を読むことができます。

インデックス自身の内部的な一貫性要求の他に、同時実行更新には、親テーブル(ヒープ)とインデックス間の一貫性に関する問題が発生します。PostgreSQLはヒープへのアクセスおよび更新とインデックスへのアクセスおよび更新を分離していますので、インデックスとヒープとの間の一貫性が無くなる間隔が存在します。以下の規則でこうした問題を扱います。

- 新しいヒープ項目はインデックス項目を作成する前に作成されます。(このため、同時実行インデックススキャンはヒープエントリを確認する時によく失敗します。インデックスの読み取りは、未コミットの行を対象としませんので問題ありません。しかし、61.5を参照してください。)

- ヒープエントリが（VACUUMによって）削除される時、これに対するすべてのインデックス項目が先に削除されます。
- インデックススキャンは、最後にamgettupleが返した項目を保持するインデックスページ上のピンを管理しなければなりません。また、ambulkdeleteは、他のバックエンドがピンを持つページから項目を削除することはできません。この規則の必要性については後で説明します。

3番目の規則がないと、VACUUMによって削除される直前に、インデックス読み取りがインデックス項目を見つけ、そして、VACUUMによって削除された後に対応するヒープ項目に達する可能性があります。空の項目スロットはheap_fetch()で無視されますので、これは読み取りが達した時にその項目番号が未使用である場合でも大きな問題は起こりません。しかし、第三のバックエンドがすでにその項目スロットを他のものに再使用した場合はどうなるでしょうか？ そのスロット内の新しいものが、スナップショット試験を通過するには新しすぎることが確実ですので、MVCCに則ったスナップショットを使用する場合は問題ありません。しかし、MVCCに則らないスナップショット(SnapshotNowなど)では、実際にはスキャンキーに合わない行を受付け、返す可能性があります。すべての場合においてヒープ行に対しスキャンキーの再検査を行うことを必須とすることで、こうした状況から保護することができますが、これは高価すぎます。代わりに、読み取りがまだ一致するヒープ項目へのインデックス項目の「作業中」であることを示す代理として、インデックスページに対するピンを使用します。このピンに対してambulkdeleteがブロックするようにすることで、読み取りの作業が終わる前にVACUUMがそのヒープ項目を削除できないことを確実にします。実行時におけるこの対策のコストは小さく、実際に競合が発生するごく稀な場合にのみブロックするためのオーバーヘッドが加わります。

この対策は、インデックススキャンが「同期」していることを要求します。対応するインデックス項目のスキャンの後即座に各ヒープタプルを取り出さなければなりません。多くの理由のため、これは高価です。インデックスから多くのTIDを収集し、少し後でのみヒープタプルにアクセスする「非同期」スキャンでは、必要なロック処理オーバーヘッドがかなり少なくなり、また、より効率的なヒープへのアクセスパターンを取ることができます。上の解析に従うと、MVCCに則らないスナップショットでは同期方式を使用しなければなりませんが、問い合わせがMVCCスナップショットを使用する場合は非同期スキャンを使用することができます。

amgetbitmapインデックススキャンでは、アクセスメソッドは返されるタプル上にインデックスピンをまったく保持しません。したがって、MVCCに則ったスナップショットでこうしたスキャンを使用することのみが安全です。

ampredlocksフラグが設定されていない場合、シリアライザブルトランザクション内でそのインデックスアクセスメソッドを使用するスキャンはいずれもインデックス全体に対するブロックしない述語ロックを獲得します。これは、同時実行のシリアライザブルトランザクションによるそのインデックスへの何らかのタプル挿入で、読み書きの競合が発生することがあります。同時実行のシリアライザブルトランザクションの集合の中で特定の読み書きの競合パターンが検知された場合、データの整合性を保護するためにこれらのトランザクションの1つはキャンセルされます。このフラグが設定されている場合、こうしたトランザクションのキャンセルの頻度を低減することになる、より粒度の細かな述語ロックをインデックスアクセスメソッドが実装していることを示します。

61.5. インデックス一意性検査

PostgreSQLは、SQLの一意性制約を*一意性インデックス*を使用して強制します。このインデックスでは、同一キーに対し複数の項目を許しません。この機能をサポートするアクセスメソッドはamcanuniqueを真に設定します。（現時点ではb-treeのみがこれをサポートします。）INCLUDE句内の列のリストは、一意性制約の強制時には考慮されません。

MVCCのため、インデックス内に物理的に重複した項目が存在できることが常に必要です。これらの項目は1つの論理的な行の連続的なバージョンを示します。実際に強制させたい動作は、MVCCスナップショットが同じインデックスキーを持つ行を2つ含めないことです。一意性インデックスに新しい行を挿入する時に検査しなければならない状況を以下のように分割することができます。

- 競合する有効な行が現在のトランザクションで削除された場合は問題ありません。(具体的には、UPDATEは常に新しいバージョンを挿入する前に古い行バージョンを削除します。これによりキーを変更することなく行をUPDATEすることができます。)

- 競合する行が未コミットのトランザクションで挿入された場合、挿入しようとしている方はトランザクションのコミットが分かるまで待機しなければなりません。ロールバックした場合は競合しません。競合する行が削除されずにコミットした場合、一意性違反となります。(具体的には、他のトランザクションの終了をただ待機し、終了後に可視性の検査を完全に再実行します。)

- 同様に、競合する有効な行が未コミットのトランザクションで削除された場合、挿入しようとしている方はトランザクションのコミットまたはアボートを待機しなければならず、その後、試験を繰り返します。

さらに、上記規則に従った一意性違反を報告する直前に、アクセスメソッドは挿入される行の有効性を再度検査しなければなりません。もし、無効なコミットであれば、違反を報告してはいけません。(現在のトランザクションによって作成された通常の行の挿入という状況では、これは発生することはありません。しかし、これはCREATE UNIQUE INDEX CONCURRENTLY中に発生することがあります。)

インデックスアクセスメソッドにこうした試験を自身で行うことを要求します。これは、インデックスの内容に対して重複するキーを持つことを示している任意の行のコミット状態を検査するために、ヒープまでアクセスしなければならないことを意味します。これが醜くモジュール化されないことには疑う余地はありません。しかし、余計な作業を防ぐことができます。もし分離された探査を行ったとすると、新しいインデックス項目を挿入する場所を検索する時、競合する行に対するインデックス検索がどうしても繰り返されます。さらに、競合検査がインデックス行の挿入部分で統合されて行われない限り、競合状態を防ぐ明確な方法がありません。

一意性制約が遅延可能である場合はさらに複雑になります。新しい行向けのインデックス項目を挿入可能にする必要があります。しかし一意性違反エラーは文の終わりまたはそれ以降まで遅延されます。不要なインデックス検索の繰り返しを防ぐために、インデックスアクセスメソッドは初期の挿入の間に前座の一意性検査を行わなければなりません。これが現存するタプルとまったく競合がないことを示した場合、それで終了です。さもなければ、制約を強制する時に再検査を行うようスケジュールします。再検査の時点で対象のタプルと同じキーを持つ何らかの他のタプルが存在すると、エラーを報告しなければなりません。(この目的のために「存在する」は実際には「インデックス項目のHOTチェイン内に何らかのタプルが存在する」ことを意味します。) これを実装するために、aminsertは以下のいずれかの値を持つcheckUniqueパラメータを渡されます。

- UNIQUE_CHECK_NOは、一意性検査を行うことはない(これは一意性インデックスではない)ことを示します。

- UNIQUE_CHECK_YESは、上述の通り遅延がない一意性インデックスであり、一意性検査を即時に行わなければならないことを示します。

- UNIQUE_CHECK_PARTIALは一意性制約が遅延可能であることを示します。PostgreSQLはこのモードを使用して、各行のインデックス項目を挿入します。このアクセスメソッドはインデックス内の重複する項目を許さなければなりません。そしてaminsertから偽を返すことで重複の可能性があることを報告しなければなりません。偽が返された行それぞれに対して、遅延再検査が予定されます。

アクセスメソッドは一意性制約違反となるかもしれない行を識別しなければなりません。しかし間違った偽を報告することはエラーではありません。これにより他のトランザクションを待つことなく検査を行うことができます。ここで報告された重複はエラーとして扱われず、後で再検査されます。再検査時には重複しなくなっている可能性があります。

- UNIQUE_CHECK_EXISTINGは、一意性違反の可能性があると報告された行に対する遅延再検査であることを示します。これはaminsertを呼び出すことで実装されますが、アクセスメソッドはこの場合に新しいインデックス項目を挿入してはいけません。インデックス項目はすでに存在します。それよりも、アクセスメソッドは他に存在するインデックス項目があるか検査する必要があります。もし存在し、対象の行もまだ存在する場合エラーを報告します。

  UNIQUE_CHECK_EXISTING呼び出しでは、アクセスメソッドはさらに対象行が実際にインデックス内に既存の項目を持つか検証し、もしなければエラーを報告することを推奨します。aminsertに渡されるインデックスタプル値が再計算されているため勧めます。インデックス定義に実際には不変ではない関数が含まれる場合、インデックスの間違った領域を検査してしまうかもしれません。再検査にて対象行の存在を検査することで、元の挿入で使用されたものと同じタプル値をスキャンしていることを検証します。

61.6. インデックスコスト推定関数

amcostestimate関数には、インデックスと共に使用できることが決まっているWHERE句およびORDER BY句のリストを含む、インデックススキャンの可能性を記述する情報が与えられます。この関数はインデックスにアクセスするコストの概算とWHERE句の選択度（つまりインデックススキャンにて抽出される行の親テーブルにおける割合）を返さなくてはなりません。単純な場合だと、ほとんどすべてのコスト概算の作業は、オプティマイザの標準ルーチンを呼び出すことで行われます。amcostestimate関数を持つことの意味は、標準の概算を改善することができる場合に、インデックスアクセスメソッドがインデックス型固有の知識体系を提供することができるということです。

それぞれのamcostestimate関数は以下のシグネチャを持たなければいけません。

```
void
amcostestimate (PlannerInfo *root,
                IndexPath *path,
                double loop_count,
                Cost *indexStartupCost,
                Cost *indexTotalCost,
                Selectivity *indexSelectivity,
                double *indexCorrelation,
                double *indexPages);
```

最初の3つのパラメータは入力です。

root

処理されている問い合わせに関するプランナの情報。

path

考慮されるインデックスアクセスパス。コストと選択性値を除くすべてのフィールドが有効です。

loop_count

コスト概算の算出対象となるインデックススキャンが繰り返された回数です。これは通常、ネステッドループ結合の内部で利用されるパラメータ化されたスキャンの回数よりも大きい値になります。コスト概算は1回のスキャンのための値であることに注意してください。loop_countがより大きい場合、複数のスキャンにより得られる効果をみるには十分な値といえるでしょう。

最後の5つのパラメータは参照渡しの出力です。

*indexStartupCost

インデックスの起動処理にかかるコストに設定されます。

*indexTotalCost

インデックス処理の全体のコストに設定されます。

*indexSelectivity

インデックスの選択度に設定されます。

*indexCorrelation

インデックススキャンの順番と背後のテーブルの順番間の相関係数に設定されます。

*indexPages

インデックスのリーフページ数が設定されます。

コスト概算関数は、SQLやその他の手続き言語ではなく、C言語で書かれなければいけないことに注意してください。理由はプランナ/オプティマイザの内部データ構造にアクセスしなければいけないためです。

インデックスアクセスコストはsrc/backend/optimizer/path/costsize.cで使われる、逐次的なディスクブロックの取り出しにはseq_page_costのコストが、順不同の取り出しにはrandom_page_costのコストが、そして、1つのインデックス行の処理には通常cpu_index_tuple_costというコストがかかる、というパラメータで計算されなければなりません。さらに、インデックス処理（特にindexquals自体の評価）の間に呼び出される比較演算すべてに対して、cpu_operator_costに適当な係数をかけたコストがかかります。

アクセスコストは、インデックス自身のスキャンと関係するすべてのディスクとCPUコストも含むべきですが、インデックスで識別される親テーブルの行の処理や抽出にかかるコストは含めてはいけません。

「起動用コスト」は、最初の行を取り出し始めることができるようになる前に費やされなければならない総スキャンコストの一部です。ほとんどのインデックスでは、これはゼロとすることができます。しかし、高い起動用コストを持つインデックス種類ではこれを非ゼロにすることを勧めます。

indexSelectivityは、インデックススキャンの間に抽出される親テーブルの行の概算された割合として設定されるべきです。非可逆問い合わせの場合はこの値が、与えられた制約条件を実際に通過する行の割合よりも高くなることがよくあります。

indexCorrelationは、インデックスの順番とテーブルの順番の間の（-1.0から1.0までの間の値を取る）相関として設定されるべきです。この値は、メインテーブルから行を取り出すためのコスト概算を調整するために使用されます。

indexPagesは、リーフページ数が設定されるべきです。これは、パラレルインデックススキャンのワーカー数の見積もりに使用されます。

loop_countの値が1より大きい場合、戻り値はインデックスを利用した1回のスキャンを想定した平均値であるべきです。

コスト概算

典型的なコスト概算は次のように進められます。

1. 与えられた制約条件に基づいて訪れられるメインテーブルの行の割合を概算して返します。インデックス型固有の知識体系を持たない場合、標準のオプティマイザの関数であるclauselist_selectivity()を使用してください。

```
*indexSelectivity = clauselist_selectivity(root, path->indexquals,
                                           path->indexinfo->rel->relid,
                                           JOIN_INNER, NULL);
```

2. スキャン中に訪れられるインデックスの行数を概算します。多くのインデックス種類では、これはindexSelectivityとインデックスの中にある行数を掛けたものと等しいですが、それより多い場合もあります。(ページおよび行内のインデックスのサイズはpath->indexinfo構造体から得ることができることに注意してください。)

3. スキャン中に抽出されるインデックスページ数を概算します。これは単にindexSelectivityにページ内のインデックスのサイズを掛けたものになるでしょう。

4. インデックスアクセスコストを計算します。汎用的な概算においては以下のように行うでしょう。

```
/*

 * 一般的な仮定は、インデックスページは逐次的に読まれるので、
 * random_page_costではなく、それぞれseq_page_costが掛かるというものです。
 * 各インデックス行でのindexqualsの評価にもコストが掛かります。
 * コストはすべてスキャンの間に徐々に支払われると仮定します。
 */
cost_qual_eval(&index_qual_cost, path->indexquals, root);
*indexStartupCost = index_qual_cost.startup;
*indexTotalCost = seq_page_cost * numIndexPages +
    (cpu_index_tuple_cost + index_qual_cost.per_tuple) * numIndexTuples;
```

   しかし、上では繰り返されるインデックススキャンにかかるインデックス読み込みについて減価償却を考慮していません。

5. インデックスの相関を概算します。1つのフィールドに対する単純な順番のインデックスでは、これはpg_statisticから入手することができます。相関が未知の場合、概算を用心深く考えるとゼロ(無相関)となります。

コスト概算関数の例はsrc/backend/utils/adt/selfuncs.cにあります。

第62章 汎用WALレコード

組み込みのWALにログを書き込むすべてのモジュールは、それぞれに独自の型のWALレコードがありますが、ページへの変更を汎用的な方法で記述する汎用WALレコード型もあります。カスタムアクセスメソッドを提供する拡張では、独自のWALの再実行(redo)ルーチンを登録できないため、汎用WALレコードが役立ちます。

汎用WALレコードを構築するためのAPIはaccess/generic_xlog.hに定義されており、access/transam/generic_xlog.cで実装されています。

汎用WALレコードの機能を使ってWAL書き込みを伴うデータ更新を行うには、以下の手順に従ってください。

1. state = GenericXLogStart(relation) により、指定のリレーションについての汎用WALレコードの構築を開始します。

2. page = GenericXLogRegisterBuffer(state, buffer, flags) により、現在の汎用WALレコード内で更新されるバッファを登録します。この関数はバッファページの一時コピーへのポインタを返すので、更新はそれに対して行ってください。(バッファの内容は直接更新しないでください。) 3番目の引数は、操作についてのフラグのビットマスクです。現在のところ、使用できるフラグはGENERIC_XLOG_FULL_IMAGEのみで、これはWALレコードには変更の差分ではなく、ページ全体のイメージが含まれることを示します。典型的には、このフラグはページが新しいか、あるいは完全に書き換えられるときにセットされます。WAL書き込み対象の動作が複数のページを更新する必要がある場合は、GenericXLogRegisterBufferを繰り返すことができます。

3. 前の手順で取得したページのイメージに更新を適用する。

4. GenericXLogFinish(state)により、バッファの変更を適用し、汎用WALレコードを送出する。

WALレコードの構築は、上記の手順内の間のどこででも、GenericXLogAbort(state)を呼び出すことで中止できます。これによりページイメージのコピーに対する変更はすべて廃棄されます。

汎用WALレコードの機能を使うときは、以下の点に注意してください。

- バッファの直接更新は許されません! すべての更新はGenericXLogRegisterBuffer()で取得したコピーに対して行わなければなりません。言い換えれば、汎用WALレコードを使うコードではBufferGetPage()を呼び出してはいけません。しかし、適切なときにバッファにピンを立てる、外す、そしてロックする、解除するのが呼び出し側の責任であることに変わりはありません。各ターゲットバッファの排他的ロックをGenericXLogRegisterBuffer()の前からGenericXLogFinish()の後まで保持していなければなりません。

- 手順2のバッファの登録と、手順3のページイメージの更新は自由に混在させることができます。つまり、両方の手順を任意の順序で繰り返すことができます。バッファの登録は、再生時にロックを取得する順序と同じにすべきであることを覚えていてください。

- 汎用WALレコードに登録できるバッファの最大数はMAX_GENERIC_XLOG_PAGESです。この制限を超えるとエラーが発生します。

- 汎用WALでは、更新対象のページが標準的なレイアウトになっている、特にpd_lowerとpd_upperの間には意味のあるデータがないということを想定しています。

- ここではバッファページのコピーを更新するため、GenericXLogStart()はクリティカルセクションを開始しません。従って、GenericXLogStart()とGenericXLogFinish()の間では、メモリの割り当て、エラーの発生などを安全に実行できます。唯一の本当のクリティカルセクションはGenericXLogFinish()の内部にあります。エラー終了の中でGenericXLogAbort()を呼び出すことについても心配する必要はありません。

- GenericXLogFinish()はバッファをダーティにして、LSNの設定をすることの処理をします。これについて明示的な処理をする必要はありません。

- ログを取らないリレーションは、実際のWALレコードが送出されないことを除けば、すべてが同じように動作します。従って、通常は、ログを取らないリレーションについて明示的な検査をする必要はありません。

- 汎用WALを再生する機能は、バッファの排他的ロックを、バッファが登録されたのと同じ順序で取得します。すべての変更を再生した後で、ロックは同じ順序で解放されます。

- 登録バッファにGENERIC_XLOG_FULL_IMAGEが指定されない場合、汎用WALレコードは古いページイメージと新しいページイメージの間の差分を含むものとされます。この差分はバイト毎の比較に基づくものです。これはデータをページ内で移動する場合、あまり小さくなりませんが、将来は改善されるかもしれません。

第63章 B-Treeインデックス

63.1. はじめに

PostgreSQLは、標準的なbtree（multi-way balanced tree）インデックスデータ構造を実装しています。明確に定義された線形順にソート可能なデータ型は、すべてbtreeインデックスで索引付できます。唯一の制限は、一つのインデックスエントリが（適用可能であれば、TOAST圧縮後）ページの約1/3を超えられないことです。

btree演算子クラスはそのデータ型がソート順を持つことが必要なので、btree演算子クラス（実際には演算子族）は、PostgreSQLの一般的表現として、およびソートセマンティクスを理解するものとして利用されてきました。ですから、単にbtreeインデックスをサポートするだけに必要なもの以上の機能と、btree AMが使用するものからはかけ離れたシステムの部品を備えなければなりません。

63.2. B-Tree演算子クラスの振る舞い

表 37.3で示すように、btree演算子クラスは次の5つの比較演算子を提供しなければなりません。<、<=、=、>=、そして>です。<>も演算子クラスの一部であると期待する方もいるかもしれませんが、そうではありません。インデックス検索のWHERE句で<>を使うのは、ほとんど常に役に立たないからです。（ある種の目的のためにプランナは<>をbtree演算子クラスに関連しているものとして扱います。しかし、プランナはpg_amopから検索するのではなく=の否定子リンクから検索します。）

複数のデータ型がほとんど同じソートセマンティクスを共有している場合、それらの演算子クラスは演算子族にまとめることができます。そうすることによりプランナが型をまたがる比較を推論できるので、これはメリットがあります。ファミリー中の演算子クラスには、入力データ型のための単一型演算子（および関連するサポート関数）が含まれます。一方型をまたがる比較演算子とサポート関数は演算子族中で「ゆるやか」です。プランナが推移関係から推論するすべての比較条件を提示できるように、型をまたがる演算子の完全な集合を演算子族に入れておくことをお勧めします。

btree演算子族が満たさなければならない基本的な前提条件があります。

- =演算子は等号関係でなければなりません。つまり、そのデータ型のすべての非NULL値A、B、Cについて、
  - A = Aが真である（反射律）
  - A = Bなら、B = Aである（対称律）
  - A = BかつB= Cなら、A = Cである（推移律）
- <は強順序関係でなければなりません。つまり、すべての非NULL値A、B、Cに対して、
  - A < Aは偽である（非反射律）
  - A < BかつB < Cなら、A < Cである（推移律）
- 更に、順序は全である。すなわち、すべての非NULL値A、Bに対して、

- 厳密にA < B、A = B、B < Aのうちどれか一つが真（三分律）
（もちろん、三分律は比較サポート関数の定義を正当化します。）

他の3つの演算子は=と<に沿って自明に定義され、それらと一貫していなければなりません。

複数のデータ型をサポートする演算子族について、演算子族中のデータ型であるどんなA、B、Cも上記の法則を満たさなければなりません。型をまたがる際に2つあるいは3つの異なる演算子が一貫していることを表明することになるので、推移律を満たすことはもっとも困難です。例をあげると、少なくともfloat8と比較するためにnumeric値をfloat8に変換する現在の意味論のもとでは、float8とnumericを同じ演算子族に加えるのはうまくいかないでしょう。float8の精度に限りがあるからです。これは同じfloat8値に対して等号比較する複数の異なるnumeric値が存在することを意味し、したがって推移律は満たされません。

複数データ型ファミリーに関する別な要件は、演算子族に含まれるデータ型間に定義される暗黙的あるいは二値型強制(binary-coercion)キャストは、関係するソート順を変更してはならないことです。

単一のデータ型において、btreeインデックスがこれらの法則を守ることを要求するのはかなり明確です。これらの法則なしにはキー並べる順序がなくなってしまうからです。また、異なるデータ型の比較キーを使うインデックス検索では、2つのデータ型またがる比較が正常に動作することが必要です。演算子族中で3つ以上のデータ型に対する拡張はbtreeインデックスの機構自体では要求されませんが、プランナは最適化の目的でそれらに依存します。

63.3. B-Treeサポート関数

表 37.9で示すように、btreeでは一つの必須サポート関数と、4つの省略可能なサポート関数を定義します。5つのユーザ定義メソッドは以下の通りです。

order

btreeの演算子族が比較演算子を提供する各データ型の組み合わせに対して、比較サポート関数を提供しなければなりません。それらはサポート関数1番でpg_amprocに、また、比較での左右のデータ型と等しいamproclefttype/amprocrighttypeに、登録されます（すなわち、pg_amopに登録されている演算子が対応するものと同じデータ型です）。比較関数は2つの非NULL値AとBを取り、A < B、A = B、または、A > Bであるときにそれぞれ、< 0、0、または、> 0であるint32の値を返さなければなりません。NULLの結果は許されず、データ型の全ての値は比較可能でなければなりません。例としてsrc/backend/access/nbtree/nbtcompare.cを参照してください。

比較される値が照合順序が適用可能なデータ型のものである場合、比較サポート関数に適切な照合順序のOIDが渡され、標準のPG_GET_COLLATION()機構が使用されます。

sortsupport

任意で、btree演算子族はソートサポート関数を提供してもよいです。これはサポート関数2番で登録されます。この関数は、素朴に比較サポート関数を呼び出すよりも、ソート目的により効果的な方法での比較の実装を可能にします。これに関するAPIはsrc/include/utils/sortsupport.hで定義されています。

in_range

任意で、btree演算子族は*in_range*サポート関数を提供してもよいです。これはサポート関数3番に登録されます。これはbtreeインデックス操作中には使われません。そうではなく、演算子族のセマンティクスをRANGE offset PRECEDINGとRANGE offset FOLLOWINGフレーム境界タイプ（4.2.8を参照）を含むWINDOW句に対応できるように拡張します。基本的には、提供される拡張情報はどのように演算子族のデータ並び順と互換性のある方法でoffset値を足すか引くかです。

in_range関数は以下のシグネチャを持たなければなりません。

```
in_range(val type1, base type1, offset type2, sub bool, less bool)
returns bool
```

valとbaseは同じ型でなければならず、これは演算子族でサポートされる型の一つ（すなわち、並び順を提供する対象の型）です。しかしながら、offsetは異なる型のものでも可能です。それは演算子族でサポートされないものでもよいです。例としては、組み込みのtime_ops族がinterval型のoffsetを持つin_range関数を提供しています。演算子族は、任意のサポートされる型と一つまたは複数のoffset型に対するin_range関数を提供できます。各in_range関数は、pg_amprocにtype1と等しいamproclefttypeとtype2に等しいamproclefttypeで登録されるべきです。

in_range関数の本質的なセマンティクスは2つのBooleanフラグパラメータに依存します。これは以下のように、baseにoffsetを加算または減算して、それからvalを結果と比較すべきです。

- !subかつ !lessであるなら、val >= (base + offset)を返します
- !subかつ lessであるなら、val <= (base + offset)を返します
- subかつ !lessであるなら、val >= (base - offset)を返します
- subかつ lessであるなら、val <= (base - offset)を返します

このように実行する前に、本関数は、offsetの符号を検査すべきです。すなわち、負であったなら、エラーERRCODE_INVALID_PRECEDING_OR_FOLLOWING_SIZE(22013)、エラー文面としては「invalid preceding or following size in window function(ウィンドウ関数で先行または後続のサイズが不正です)」などを出すことです。（意味上の必要性が乏しいと見られることから非標準の演算子族はこの制限を無視することを選ぶかもしれませんが、これはSQL標準で必要とされています。）中核コードが特定のデータ型における「ゼロより小さい」ことの意味を理解しなくても良いように、この要件はin_range関数に委託されます。

さらに期待されることは、in_range関数は、実用的には、base + offsetやbase - offsetがオーバーフローする場合にエラーを投げるのを避けるべきです。たとえ値がデータ型の範囲を超えたとしても正しい比較結果は決定できます。データ型が「infinity」や「NaN」などの概念を含む場合には、in_rangeの結果が演算子族の通常のソート順序と一致するように特別な対応が必要となることに注意してください。

in_range関数の結果は、演算子族で規定されるソート順序と整合していなければなりません。正確には、与えらえれた任意のoffsetとsubの修正値は以下のようになります。

- less = trueのin_rangeがいくつかのval1とbaseに対して真であるなら、同じbaseの全てのval2 <= val1に対して真でなければなりません。
- less = trueのin_rangeが、いくつかのval1とbaseに対して偽であるなら、同じbaseの全てのval2 >= val1に対して偽でなければなりません。

- less = trueのin_rangeがいくつかのvalとbase1に対して真であるなら、同じvalの全てのbase2 >= base1に対して真でなければなりません。

- less = trueのin_rangeが一部のvalとbase1に対して偽であるなら、同じvalの全てのbase2 <= base1に対して偽でなければなりません。

less = falseのときには、逆条件の類似した命題が適用できます。

整列しようとしている型(type1)が照合可能であるなら、標準のPG_GET_COLLATION()機構を使って、in_range関数に適切な照合順序のOIDが渡されます。

in_range関数は、通例STRICTと印付けされ、NULL入力を処理する必要がありません。

equalimage

省略可能ですが、btree演算子族はequalimage (「イメージ等価を意味する等価」)サポート関数を提供してもよいです。これはサポート関数4番で登録されます。この関数は、中核コードがbtree重複排除の最適化を適用するのが安全かを決定できるようにします。今のところ、equalimage関数はインデックスの構築または再構築時にのみ呼び出されます。

equalimage関数は以下のシグネチャを持たなければなりません。

```
equalimage(opcintype oid) returns bool
```

戻り値は演算子クラスと照合順序についての静的な情報です。trueを返すことは、AおよびB引数が何らセマンティック情報を損失すること無しに交換可能でもあるとき、演算子クラスに対するorder関数が0(「引数が等しい」)だけを返すことが保証されていることを示します。equalimage関数が登録されていなかったり、falseを返すことは、この条件は守られないであろうことを示します。

opcintype引数は演算子クラスタがインデックスを作るデータ型のpg_type.oidです。これは同じ基となるequalimage関数を演算子クラスを横断して再利用できるようになる利便性があります。opcintypeが照合可能なデータ型である場合には、適切な照合順序のOIDが、標準のPG_GET_COLLATION()機構を使って、equalimage関数に渡されます。

演算子クラスに関する限り、trueを返すことは、重複排除が安全(あるいはequalimage関数に渡されたOIDの照合順序について安全)であることを示します。しかしながら、コアコードは、全てのインデックス列がequalimage関数を登録する演算子クラスを使っていて、各関数が呼ばれたとき実際にtrueを返すときに、そのインデックスに対して重複排除を安全と見做すだけです。

イメージ等価は単純にビット毎に等しいこととほとんど同じ条件です。一点微妙な違いがあります。varlenaデータ型にインデックス作成するとき、入力時の一貫性のないTOAST圧縮の適用のために、同じdatumの二つのイメージのディスク上の表現はビット毎には等しくないかもしれません。これまでは、演算子クラスのequalimage関数がtrueを返すときには、datum_image_eq() C関数が常に演算子クラスのorder関数と一致すると想定して安全でした(同じ照合順序のOIDがequalimageとorderの両関数に渡されるとして)。

コアコードは基本的に、複数データ型の族の中の演算子クラスの「等価性がイメージ等価性を含む」状態について、同族の他の演算子クラスの詳細に基づいた、いかなる推測もできません。また、ある演算子族が型にまたがってequalimage関数を登録していることを認識できず、そのような試みはエラーになります。これは「等価性がイメージ等価性を含む」状態は、演算子族の階層でおおむね定義されている、

ソートと等価性のセマンティクスに依存しているだけでは無いためです。一般に、ある特定データ型の実装によるセマンティクスは別個に考慮されなければなりません。

コアPostgreSQL配布物に含まれる演算子クラスが従う慣習は、標準品、すなわち、一般的なequalimage関数を登録することです。大部分の演算子クラスタはbtequalimage()を登録しています。これは重複排除が無条件に安全であることを示しています。textなどの照合可能なデータ型に対する演算子クラスはbtvarstrequalimage()を登録します。これは決定的な照合順序では重複排除が安全であることを示します。サードパーティ拡張におけるベストプラクティスは制御を保つためにそれら自身のカスタム関数を登録することです。

options

省略可能ですが、B-treeの演算子族はoptions(「演算子クラス固有オプション」)サポート関数を提供してもよいです。これはサポート関数5番に登録されます。この関数はユーザに見える演算子クラスの振る舞いを制御するパラメータの集合を定義します。

optionsサポート関数は以下のシグネチャを持たなければなりません。

```
options(relopts local_relopts *) returns void
```

関数にはlocal_relopts構造体へのポインタが渡されます。ここには演算子クラス固有のオプションの集合が書かれている必要があります。このオプションにはPG_HAS_OPCLASS_OPTIONS()およびPG_GET_OPCLASS_OPTIONS()マクロを使って他のサポート関数からアクセスが可能です。

今のところ、optionsサポート関数を持ったB-Treeの演算子クラスはありません。B-treeはGiST、SP-GiST、GINおよびBRINで行われてるような柔軟なキーの表現を許していません。そのため、おそらくはoptionsが現在のB-treeインデックスアクセスメソッドで多数適用されることはありません。それでも、統一性のためにサポート関数がB-treeに追加されました。おそらくPostgreSQLでのB-treeの更なる進化の過程で使用法を見つけ出すでしょう。

63.4. 実装

本節では、上級ユーザに役立つかもしれない、B-Treeインデックスの実装の詳細について説明します。更なる詳細、B-Tree実装の内部に焦点をあてた記述については、ソース配布物のsrc/backend/access/nbtree/README を参照してください。

63.4.1. B-Treeの構造

PostgreSQLのB-Treeインデックスは複数階層のツリー構造で、ツリーの各階層はページの双方向連結リストとして使用できます。一つのメタページがインデックスの最初のセグメントファイルの固定位置に格納されます。それ以外の全てのページはリーフページか内部ページのいずれかです。リーフページはツリーの最下階層にあるページです。各リーフページはテーブルの行を指すタプルを含みます。各内部ページはツリーの次の下位層を指すタプルを含みます。典型的には、全ページの99%以上がリーフページです。内部ページとリーフページは共に、68.6に記載されている標準のページ書式を使用します。

既存リーフページがやってくるタプルをはめ込むことができないとき、新たなリーフページがB-Treeインデックスに追加されます。ページ分割操作は一部のアイテムを新ページに動かすことで、当初は溢れている

ページに属していたアイテムのために空間を作ります。ページ分割は、また、新ページへの新たなダウンリンクを親ページに挿入しなければなりません。これは親ページの分割を同様に引き起こすかもしれません。ページは分割は再帰的に「上向きに連鎖」します。最終的にルートページが新たなダウンリンクをはめ込みできないときには、ルートページ分割が実施されます。これは元のルートページの一つ上の階層に新たなルートページを作ることで、ツリー構造に新しい階層を加えます。

63.4.2. 重複排除

重複とは、同じインデックスで全てのインデックスキー列が少なくとも一つの他のリーフページタプルの該当する列の値と一致する値をもっている、リーフページタプル（テーブルの行を指すタプル）です。重複タプルは実際によくあります。オプションの技法「重複排除」が有効にされているとき、B-Treeインデックスは、特別な重複に対する空間効率の良い表現方法を使用できます。

重複排除は重複タプルのグループを定期的に合併して、各グループに対する単一のポスティングリストタプルを形成することで機能します。この表現方法では列のキー値は一度だけ現れます。テーブルの行を指すTIDのソートされた配列がこれに続きます。概して各値（あるいは列値の異なる組み合わせ）が複数回出現する場合に、これは顕著にインデックスの格納サイズを減らします。問い合わせの遅延も顕著に削減できます。全体的な問い合わせのスループットも顕著に増加するかもしれません。インデックスのバキューム処理のオーバーヘッドも顕著に削減されるかもしれません。

> **注記**
>
> B-Tree重複排除は、B-Tree演算子クラスの=項に従ってNULL値が決して互いに等しくならないとしても、NULL値を含む「重複」に効果的です。ディスク上のB-Tree構造を解するいかなる実装部分に関しても、NULLはまさにインデックス値の定義域以外の一つの値です。

既存のリーフページにはめ込みできない新たな要素が挿入されたとき、重複排除の処理は怠惰に実行されます。これはリーフページの分割を防止（あるいは少なくとも遅延）します。GINのポスティングリストのタプルと違って、B-Treeのポスティングリストのタプルは新たな重複が挿入される度に拡張する必要がありません。それらはリーフページの元の論理内容に対する単なる代替の物理表現にすぎません。この設計は読み書き混合のワークロードでの性能の一貫性を重視しています。ほとんどのクライアントアプリケーションは重複排除を使うことで少なくとも控えめな性能の恩恵を確認することができるでしょう。

CREATE INDEXとREINDEXは、使用する手順が若干異なりますが、ポスティングリストタプルを作って重複排除を適用します。テーブルから取得されてソートされた入力で遭遇した重複した通常タプルの各グループは、現在のペンディングリーフページに追加される前に、ポスティングリストタプルにマージされます。個別のポスティングリストタプルには、可能な限り多数のTIDが詰め込まれます。リーフページは、重複排除用の別パスではなく、通常の方法で書き出されます。この戦略はCREATE INDEXとREINDEXに良く適合します。これらは1回で終わるバッチ操作であるからです。

インデックスの値に重複が無いか殆ど無いために重複排除から利益を得られない、書き込みの多いワークロードには、（重複排除が明示的に無効化されて居ない限り）固定のペナルティによる小さい負荷増があります。deduplicate_items格納パラメータは個別のインデックス内で重複排除を無効化するのに使うことができます。ポスティングリストタプルの読み込みは少なくとも通常タプル表現の読み込み程度に効率的であるため、読み込みのみのワークロードで性能ペナルティは一切ありません 通常は重複排除を無効化することは有益ではありません。

B-Treeインデックスは、MVCC下で同じ論理テーブル行が複数バージョン存在していることを、直接には認識していません。インデックスにとっては、各タプルは自身のインデックスエントリを必要とする独立したオブジェクトです。「バージョン重複」は時に積み重なって問い合わせのレイテンシとスループットに不利に作用するかもしれません。これは典型的には、(しばしば一つ以上のインデックス列が変更されて、新たなインデックスタプルバージョンの一式 ― それぞれのインデックスに対するもの、の設置が必要となるために)大部分の各更新がHOTを適用できないUPDATEが重たいワークロードで発生します。実際のところ、B-Treeの重複排除はバージョンの大量発生によるインデックス膨張を改善します。一意インデックスのタプルであっても、バージョン重複のため、ディスクに格納されるときに物理的に一意とは限らないことに注意してください。重複排除の最適化は一意性インデックス内に選択的に適用されます。バージョン重複を持つと見られるページが対象となります。高位の目標は、起こりうるバージョン大量発生によって生じる「不要な」ページ分割の前により多くVACUUM実行を施すことです。

> **ヒント**
>
> 一意性インデックスで重複排除パスを実行すべきかどうかの判断には、特別なヒューリスティックが適用されます。これは、しばしばリーフページ分割まで連続してスキップして、無益な重複排除パスでの無駄なサイクルによる性能ペナルティを回避できます。重複排除のオーバーヘッドを懸念するなら、選択的に設定deduplicate_items = offを検討してください。一意性インデックスでは重複排除を無効にすることに不都合はありません。

実装レベルの制限により、重複排除は全ての場合に使えるわけではありません。重複排除の安全性はCREATE INDEXあるいはREINDEXが実行されたときに決定されます。

等しいデータの間で意味的に明らかな違いを伴う以下の場合には、重複排除は安全でないと見做されて使用できないことに注意してください。

- 非決定的な照合順序が使われているときtext、varchar、および、charは重複排除を使えません。等しいデータの間で大文字小文字やアクセントの違いが維持されなければなりません。
- numericは重複排除を使えません。等しいデータの間で数の表示スケールが維持されなければなりません。
- jsonbのB-Tree演算子クラスは内部的にnumericを使っているため、jsonbは重複排除を使えません。
- float4およびfloat8は重複排除を使えません。これらの型は-0と0に異なる表現を持ち、にもかかわらずこれらは等しいと見做されます。この違いは維持されなければなりません。

さらに以下の実装レベルの制限があります。これはPostgreSQLの将来バージョンで解消されるかもしれません。

- コンテナ型(複合型、配列型、あるいは、範囲型など)は、重複排除を使えません。

さらに以下の実装レベルの制限があります。これは使われている演算子クラスや照合順序にかかわりなく該当します。

- INCLUDEインデックスには重複排除は使えません。

第64章 GiSTインデックス

64.1. はじめに

GiSTは汎用検索ツリー（Generalized Search Tree）を表します。これは、均衡な、ツリー構造のアクセスメソッドで、任意のインデックスの枠組みを実装する基本的なテンプレートとして動作します。B-tree、R-treeやその他多くのインデックスの枠組みをGiSTで実装することができます。

GiSTの利点の1つは、データベースの専門家ではなく、データ型分野の専門家によって、適切なアクセスメソッドで独自のデータ型を開発することができることです。

ここで示す情報の一部は、カリフォルニア大学バークレイ校のGiSTインデックスプロジェクト、ウェブサイト[1]およびMarcel Kornackerの論文、Access Methods for Next-Generation Database Systems[2]から派生したものです。PostgreSQLにおけるGiSTの実装は、主に、Teodor SigaevとOleg Bartunovによって保守されています。そして、彼らのウェブサイト[3]にも多くの情報があります。

64.2. 組み込み演算子クラス

PostgreSQLのコア配布物は表 64.1に示すGiST演算子クラスを含みます。(付録Fに記載された追加モジュールの中には追加のGiST演算子クラスを提供するものもあります。)

表64.1 組み込みGiST演算子クラス

名前	インデックスされるデータ型	インデックス可能な演算子	順序付け演算子
box_ops	box	&& &> &< &<| >> << <<| <@ @> @ |&> |>> ~ ~=	<->
circle_ops	circle	&& &> &< &<| >> << <<| <@ @> @ |&> |>> ~ ~=	<->
inet_ops	inet, cidr	&& >> >>= > >= <> << <<= < <= =	
point_ops	point	>> >^ << <@ <@ <@ <^ ~=	<->
poly_ops	polygon	&& &> &< &<| >> << <<| <@ @> @ |&> |>> ~ ~=	<->
range_ops	任意の範囲型	&& &> &< >> << <@ -| - = @> @>	
tsquery_ops	tsquery	<@ @>	
tsvector_ops	tsvector	@@	

歴史的な理由から、inet_ops演算子クラスは型inetとcidrのデフォルトクラスではありません。これを使用するには、CREATE INDEXでクラス名を指定します。例えば

[1] http://gist.cs.berkeley.edu/

[2] http://www.sai.msu.su/~megera/postgres/gist/papers/concurrency/access-methods-for-next-generation.pdf.gz

[3] http://www.sai.msu.su/~megera/postgres/gist/

```
CREATE INDEX ON my_table USING GIST (my_inet_column inet_ops);
```

のように、CREATE INDEXでクラス名を書いてください。

64.3. 拡張性

伝統的に、新しいインデックスメソッドの実装は、非常に難しい作業を意味していました。ロックマネージャやログ先行書き込みなどデータベースの内部動作を理解する必要がありました。GiSTインタフェースは高度に抽象化されており、アクセスメソッドの実装者には、アクセスするデータ型のセマンティクスのみの実装を要求します。GiST層自身が同時実行性、ログ処理、ツリー構造の検索処理に関する注意を行います。

この拡張性と、他の、扱うことができるデータを対象とした標準検索ツリーの拡張性とを混同すべきではありません。例えば、PostgreSQLは拡張可能なB-treeとハッシュインデックスをサポートしています。これは、PostgreSQLを使用して、任意のデータ型に対するB-treeやハッシュを構築することができることを意味します。しかし、B-treeは範囲述語(<、=、>)のみをサポートし、ハッシュインデックスは等価性問い合わせのみをサポートします。

ですから、PostgreSQLのB-treeで例えば画像群をインデックス付けする場合、「画像xは画像yと同じか」、「画像xは画像yより小さいか」、「画像xは画像yより大きいか」といった問い合わせのみ発行することができます。この文脈でどのように「同じか」や「より小さいか」、「より大きいか」を定義するかに依存して、これが有意なこともあるでしょう。しかし、GiSTを基にしたインデックスを使用すれば、問題分野に特化した、おそらくは、「馬の画像を全て見つけたい」、「露出オーバーの写真をすべて見つけたい」といった質問に答えられる手段を作成することができます。

GiSTアクセスメソッドを有効にし、実行するために行なわなければならないことは、ツリーのキーの動作を定義する、複数のユーザ定義のメソッドを実装することです。当然ながら、これらのメソッドは手の込んだ問い合わせをサポートするためかなり意匠を凝らす必要があります。しかし、すべての標準的な問い合わせ(B-treeやR-treeなど)ではこれらは、相対的に見てごく簡単です。まとめると、GiSTは汎用性、コード再利用、整理されたインタフェースと拡張性を兼ね備えたものです。

GiST用の演算子クラスが提供しなければならないメソッドが5つ、オプションで提供可能なメソッドが5つあります。インデックスの正確性は、same、consistent、unionメソッドを適切に実装することで保証されます。一方、インデックスの効率(容量と速度)はpenaltyとpicksplitメソッドに依存します。オプションのメソッドの2つは、compressとdecompressです。これによりインデックスはインデックス付けするデータと異なるデータ型のツリーデータを内部で持つことができるようになります。リーフはインデックス付けするデータ型となりますが、他のツリーノードは何らかのC構造体を取ることができます。(しかしここでもPostgreSQLのデータ型規約に従わなければなりません。容量が可変のデータに関してはvarlenaを参照してください。) ツリーの内部データ型がSQLレベルで存在する場合、CREATE OPERATOR CLASSコマンドのSTORAGEオプションを使用することができます。オプションの8番目のメソッドはdistanceです。これは演算子クラスに順序付けスキャン(最近傍検索)をサポートさせたい場合に必要です。オプションの9番目のメソッドfetchは、compressメソッドが省略されている場合を除き、演算子クラスがインデックスオンリースキャンをサポートしたい場合に必要になります。オプションの10番目のメソッドoptionsは、演算子クラスがユーザに固有のパラメータを提供する場合に必要です。

consistent

インデックス項目pと問い合わせ値qが与えられると、この関数はインデックス項目が問い合わせと「一貫性」があるかどうか、つまり、述語「indexed_columnindexable_operator q」が、インデックス項目で

表現される行に対して真かどうかを決定します。リーフインデックス項目では、これはインデックス付条件の試験と等価です。一方で内部ツリーノードでは、これはツリーノードで表現されるインデックスの副ツリーを走査する必要があるかどうかを決定します。結果がtrueならば、recheckフラグも返されなければなりません。これは、述語が確実に真なのか一部のみ真なのかを示します。recheck = falseならば、インデックスは述語条件を正確に試験されたことを示し、recheck= trueならば行が単に一致候補であることを示します。この場合、システムは自動的にindexable_operatorを実際の行値に対して評価し、本当に一致するかどうか確認します。この規則により、GiSTはインデックス構造が非可逆な場合でも可逆な場合でもサポートすることができます。

この関数のSQL宣言は以下のようになります。

```
CREATE OR REPLACE FUNCTION my_consistent(internal, data_type, smallint, oid, internal)
RETURNS bool
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

そして、Cモジュール内の対応するコードは以下のような骨格に従うことになります。

```
PG_FUNCTION_INFO_V1(my_consistent);

Datum
my_consistent(PG_FUNCTION_ARGS)
{
    GISTENTRY  *entry = (GISTENTRY *) PG_GETARG_POINTER(0);
    data_type  *query = PG_GETARG_DATA_TYPE_P(1);
    StrategyNumber strategy = (StrategyNumber) PG_GETARG_UINT16(2);
    /* Oid subtype = PG_GETARG_OID(3); */
    bool       *recheck = (bool *) PG_GETARG_POINTER(4);
    data_type  *key = DatumGetDataType(entry->key);
    bool        retval;

    /*

     * strategy、
      keyおよびqueryの関数として戻り値を決定してください。
     *
     * インデックスツリー内のどこで呼びだされているかを知るためGIST_LEAF(entry)を使用してください。
     * それは、
      例えば = 演算子をサポートする場合重宝です
     *（非リーフノードにおける空でないunion()とリーフノードにおける等価性を検査することができます）。
     */


    *recheck = true;        /* もしくは検査が正確であれば偽 */
```

```
    PG_RETURN_BOOL(retval);
}
```

ここで、keyはインデックス要素であり、queryはインデックスに対して検索される値です。StrategyNumberパラメータは、演算子クラスのどの演算子が適用されるかを示します。これはCREATE OPERATOR CLASSコマンドの演算子番号の1つに一致します。

演算子の右辺にはいかなる型も来ることがあり、それは左辺に現れるインデックス付けされたデータ型とは違うものかもしれませんので、このクラスにどの演算子を含めたかに依存して、queryのデータ型は演算子に応じて変動することがあります。(上のコードの骨格は型が1つだけ可能であることを仮定しています。そうでなければ、query引数の値を取得するのは演算子に依存しないといけないでしょう。)consistent関数のSQL宣言では、実際の型は演算子に依存して何か他のものであるとしても、query引数の演算子クラスのインデックス付けされたデータ型を使うことをお勧めします。

union

このメソッドはツリー内の情報を統合します。項目の集合が与えられると、この関数は与えられた項目すべてを表現するインデックス項目を新しく生成します。

この関数のSQL宣言は以下のようになります。

```
CREATE OR REPLACE FUNCTION my_union(internal, internal)
RETURNS storage_type
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

そして、Cモジュール内の対応するコードは以下のような骨格に従うことになります。

```
PG_FUNCTION_INFO_V1(my_union);

Datum
my_union(PG_FUNCTION_ARGS)
{
    GistEntryVector *entryvec = (GistEntryVector *) PG_GETARG_POINTER(0);
    GISTENTRY  *ent = entryvec->vector;
    data_type  *out,
               *tmp,
               *old;
    int         numranges,
                i = 0;

    numranges = entryvec->n;
    tmp = DatumGetDataType(ent[0].key);
    out = tmp;

    if (numranges == 1)
    {
```

```
        out = data_type_deep_copy(tmp);

        PG_RETURN_DATA_TYPE_P(out);
    }

    for (i = 1; i < numranges; i++)
    {
        old = out;
        tmp = DatumGetDataType(ent[i].key);
        out = my_union_implementation(out, tmp);
    }

    PG_RETURN_DATA_TYPE_P(out);
}
```

ご覧になったように、この骨格でunion(X, Y, Z) = union(union(X, Y), Z)であるようなデータ型を処理しています。このGiSTサポートメソッドに適切なunionアルゴリズムを実装することで、このような場合以外のデータ型をサポートすることは非常に容易です。

union関数の結果は、(インデックス付けされた列の型とは異なるかもしれないし、異ならないかもしれませんが)それが何であれインデックスの格納型の値でなければなりません。union関数は新たにpalloc()されたメモリへのポインタを返さなければなりません。型の変更がなかったとしても、入力値をそのまま返すことはできません。

上に示したように、union関数の1番目のinternal引数は実際はGistEntryVectorのポインタです。2番目の引数は整数の変数へのポインタであり、無視できます。(union関数がその結果値の大きさをその変数に保存するのに必要だったのですが、これはもはや必要ではありません。)

compress

データ項目をインデックスページ内の物理的な格納に適した形式に変換します。compressメソッドが省略されている場合、データ項目は変更されずにインデックスに格納されます。

この関数のSQL宣言は以下のようになります。

```
CREATE OR REPLACE FUNCTION my_compress(internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

そして、Cモジュール内の対応するコードは以下のような骨格に従うことになります。

```
PG_FUNCTION_INFO_V1(my_compress);

Datum
my_compress(PG_FUNCTION_ARGS)
{
    GISTENTRY  *entry = (GISTENTRY *) PG_GETARG_POINTER(0);
```

```
    GISTENTRY  *retval;

    if (entry->leafkey)
    {

        /* 圧縮バージョンで entry->key を差し替え */
        compressed_data_type *compressed_data = palloc(sizeof(compressed_data_type));


        /* entry->key ... から *compressed_data を補填 */

        retval = palloc(sizeof(GISTENTRY));
        gistentryinit(*retval, PointerGetDatum(compressed_data),
                      entry->rel, entry->page, entry->offset, FALSE);
    }
    else
    {

        /* 通常非リーフ項目に対して行うことはない */
        retval = entry;
    }

    PG_RETURN_POINTER(retval);
}
```

当然ながらcompressed_data_typeを、リーフノードを圧縮するために変換する特定の型に適合させなければなりません。

decompress

データ項目の格納された表現を、演算子クラスの他のGiSTメソッドで操作できる形式に変換します。decompressメソッドが省略された場合、他のGiSTメソッドが直接操作出来るデータ形式で格納されると想定されます。(decompressは、必ずしもcompressメソッドの逆になるわけではありません。特に、compressが不可逆な場合、decompressで元のデータを正確に再構築するのが不可能になります。他のGiSTメソッドはすべてのデータを再構築することは必要としないかもしれないので、decompressはfetchと等価であるとは限りません。)

この関数のSQL宣言は以下のようになります。

```
CREATE OR REPLACE FUNCTION my_decompress(internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

そして、Cモジュール内の対応するコードは以下のような骨格に従うことになります。

```
PG_FUNCTION_INFO_V1(my_decompress);
```

```
Datum
my_decompress(PG_FUNCTION_ARGS)
{
    PG_RETURN_POINTER(PG_GETARG_POINTER(0));
}
```

上記骨格は、伸長を必要としない場合に適したものです。

penalty

新しい項目をツリーの特定の分岐点に挿入するための「コスト」を示す値を返します。項目は、ツリー内でpenaltyが最小の経路に挿入されます。penaltyから返される値は非負でなければなりません。負の値が返された場合、ゼロとして扱われます。

この関数のSQL宣言は以下のようになります。

```
CREATE OR REPLACE FUNCTION my_penalty(internal, internal, internal)
RETURNS internal
AS 'MODULE_PATHNAME'

LANGUAGE C STRICT;  -- penalty関数は厳密である必要がない場合もあります
```

そして、Cモジュール内の対応するコードは以下のような骨格に従うことになります。

```
PG_FUNCTION_INFO_V1(my_penalty);

Datum
my_penalty(PG_FUNCTION_ARGS)
{
    GISTENTRY  *origentry = (GISTENTRY *) PG_GETARG_POINTER(0);
    GISTENTRY  *newentry = (GISTENTRY *) PG_GETARG_POINTER(1);
    float      *penalty = (float *) PG_GETARG_POINTER(2);
    data_type  *orig = DatumGetDataType(origentry->key);
    data_type  *new = DatumGetDataType(newentry->key);

    *penalty = my_penalty_implementation(orig, new);
    PG_RETURN_POINTER(penalty);
}
```

歴史的な理由により、penalty関数は単純にfloatの結果を返しません。その代わり、3番目の引数で指定された場所に値を格納しなければなりません。その引数のアドレスを戻すのが慣例ですが、戻り値それ自体は無視されます。

penalty関数は優れた性能のインデックスではきわめて重要です。これは、挿入の段階で新しい項目をツリーに追加する場所を決定する際にどの分岐に従うかを決定するために使用されます。問い合わせの際、インデックスのバランスが良ければ、検索が速くなります。

picksplit

インデックスページ分割が必要になった時、この関数は、ページ内のどの項目を古いページに残すか、および、どれを新しいページに移動するかを決定します。

この関数のSQL宣言は以下のようになります。

```
CREATE OR REPLACE FUNCTION my_picksplit(internal, internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

そして、Cモジュール内の対応するコードは以下のような骨格に従うことになります。

```
PG_FUNCTION_INFO_V1(my_picksplit);

Datum
my_picksplit(PG_FUNCTION_ARGS)
{
    GistEntryVector *entryvec = (GistEntryVector *) PG_GETARG_POINTER(0);
    GIST_SPLITVEC *v = (GIST_SPLITVEC *) PG_GETARG_POINTER(1);
    OffsetNumber maxoff = entryvec->n - 1;
    GISTENTRY  *ent = entryvec->vector;
    int         i,
                nbytes;
    OffsetNumber *left,
                *right;
    data_type  *tmp_union;
    data_type  *unionL;
    data_type  *unionR;
    GISTENTRY **raw_entryvec;

    maxoff = entryvec->n - 1;
    nbytes = (maxoff + 1) * sizeof(OffsetNumber);

    v->spl_left = (OffsetNumber *) palloc(nbytes);
    left = v->spl_left;
    v->spl_nleft = 0;

    v->spl_right = (OffsetNumber *) palloc(nbytes);
    right = v->spl_right;
    v->spl_nright = 0;

    unionL = NULL;
    unionR = NULL;
```

```
    /* 項目自体のベクタの初期化 */
    raw_entryvec = (GISTENTRY **) malloc(entryvec->n * sizeof(void *));
    for (i = FirstOffsetNumber; i <= maxoff; i = OffsetNumberNext(i))
        raw_entryvec[i] = &(entryvec->vector[i]);

    for (i = FirstOffsetNumber; i <= maxoff; i = OffsetNumberNext(i))
    {
        int         real_index = raw_entryvec[i] - entryvec->vector;

        tmp_union = DatumGetDataType(entryvec->vector[real_index].key);
        Assert(tmp_union != NULL);

        /*

         * インデックス項目の格納場所を決定し、
      それに合わせてunionLとunionRを更新
         * します。v->spl_left もしくは v->spl_right のどちらかに項目を追加します。
         * カウンタに留意してください。
         */

        if (my_choice_is_left(unionL, curl, unionR, curr))
        {
            if (unionL == NULL)
                unionL = tmp_union;
            else
                unionL = my_union_implementation(unionL, tmp_union);

            *left = real_index;
            ++left;
            ++(v->spl_nleft);
        }
        else
        {
            /*

             * 右と同じ
             */
        }
    }

    v->spl_ldatum = DataTypeGetDatum(unionL);
    v->spl_rdatum = DataTypeGetDatum(unionR);
    PG_RETURN_POINTER(v);
}
```

picksplit関数の結果は渡されたv構造体を修正することで返されることに注意してください。vのアドレスを戻すのが慣例ですが、戻り値それ自体は無視されます。

penalty同様、picksplit関数も優れた性能のインデックスのためにきわめて重要です。penaltyとpicksplitの実装を適切に設計することが、性能が良いGiSTインデックスの実装を行うことにつながります。

same

2つのインデックス項目が同一の場合に真、さもなくば偽を返します。(「インデックス項目」はインデックスの格納型の値であり、必ずしも元のインデックス付けされた列の型という訳ではありません。)

この関数のSQL宣言は以下のようになります。

```
CREATE OR REPLACE FUNCTION my_same(storage_type, storage_type, internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

そして、Cモジュール内の対応するコードは以下のような骨格に従うことになります。

```
PG_FUNCTION_INFO_V1(my_same);

Datum
my_same(PG_FUNCTION_ARGS)
{
    prefix_range *v1 = PG_GETARG_PREFIX_RANGE_P(0);
    prefix_range *v2 = PG_GETARG_PREFIX_RANGE_P(1);
    bool       *result = (bool *) PG_GETARG_POINTER(2);

    *result = my_eq(v1, v2);
    PG_RETURN_POINTER(result);
}
```

歴史的な理由により、same関数は単純に論理値の結果を返しません。その代わり、3番目の引数で指定された場所にフラグを格納しなければなりません。その引数のアドレスを戻すのが慣例ですが、戻り値それ自体は無視されます。

distance

インデックス項目pと問い合わせ値qを与えると、この関数は問い合わせ値からのインデックス項目の「距離」を決定します。この関数は、演算子クラスが何らかの順序付け演算子を含む場合には提供しなければなりません。順序付け演算子を使用する問い合わせは、まず最小の「距離」を持つインデックス項目を返すことで実装されます。このためこの結果は演算子の意味と一貫性を持たなければなりません。リーフインデックスノード項目では、結果は単にインデックス項目との距離を表します。内部ツリーノードでは、結果はすべての子項目が持つ中から最も最小の距離でなければなりません。

この関数のSQL宣言は以下のようにならなければなりません。

```
CREATE OR REPLACE FUNCTION my_distance(internal, data_type, smallint, oid, internal)
RETURNS float8
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

Cモジュールにおける対応するコードは次の骨格に従うことになります。

```
PG_FUNCTION_INFO_V1(my_distance);

Datum
my_distance(PG_FUNCTION_ARGS)
{
    GISTENTRY  *entry = (GISTENTRY *) PG_GETARG_POINTER(0);
    data_type  *query = PG_GETARG_DATA_TYPE_P(1);
    StrategyNumber strategy = (StrategyNumber) PG_GETARG_UINT16(2);
    /* Oid subtype = PG_GETARG_OID(3); */
    /* bool *recheck = (bool *) PG_GETARG_POINTER(4); */
    data_type  *key = DatumGetDataType(entry->key);
    double      retval;

    /*

     * strategy、keyおよびqueryの関数として戻り値を決定してください。
     */

    PG_RETURN_FLOAT8(retval);
}
```

distance関数の引数はconsistent関数の引数と同一です。

距離の決定において、その結果がエントリの実際の距離よりも大きくならない限り、多少の概算は許されます。したがって、例えば、幾何学に関するアプリケーションでは、通常は外接矩形への距離で十分です。内部ツリーノードについては、返される距離はどの子ノードへの距離よりも大きくなることは許されません。返される距離が正確でない場合、関数は*recheckを真にセットする必要があります。(内部ツリーノードについては、計算はいつでも不正確であると見なされるため、これは必要ありません。) この場合、エクゼキュータはヒープからタプルを取得した後で正確な距離を計算し、必要ならタプルを並べ替えます。

距離関数がリーフノードについて*recheck = trueを返す場合、元の順序づけ演算子の戻り型はfloat8またはfloat4でなければならず、また距離関数の結果の値は元の順序づけ演算子の戻り型と比較可能でなければなりません。なぜならエクゼキュータは距離関数の結果および再計算された順序づけ演算子の結果の両方を利用してソート処理を行うからです。その他の場合は、結果値の相対的な順序が順序づけ演算子が返す順序と一致する限り、距離関数の戻り値は任意の有限のfloat8の値とすることができます。(無限大とマイナス無限大は内部的にNULLなどの場合を処理するために利用するので、distance関数がこれらの値を戻すことは薦められません。)

fetch

インデックスオンリースキャンで使用するため、データ項目の圧縮されたインデックス表現を元のデータ型に変換します。返されたデータは元のインデックス値の正確で、何も失われていない複製でなければなりません。

この関数のSQL宣言は以下のようにならなければなりません。

```
CREATE OR REPLACE FUNCTION my_fetch(internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

引数はGISTENTRY構造体へのポインタです。関数が呼び出された時点では、そのkeyフィールドには、NULLでないリーフデータが圧縮形式で入っています。戻り値は別のGISTENTRY構造体で、そのkeyフィールドには、同じデータが元の非圧縮形式で入っています。opclassの圧縮関数がリーフのエントリに対して何もしないなら、fetchメソッドは引数をそのまま返すことができます。

Cモジュールにおける対応するコードは次の骨格に従うことになります。

```
PG_FUNCTION_INFO_V1(my_fetch);

Datum
my_fetch(PG_FUNCTION_ARGS)
{
    GISTENTRY  *entry = (GISTENTRY *) PG_GETARG_POINTER(0);
    input_data_type *in = DatumGetPointer(entry->key);
    fetched_data_type *fetched_data;
    GISTENTRY  *retval;

    retval = palloc(sizeof(GISTENTRY));
    fetched_data = palloc(sizeof(fetched_data_type));

    /*

     * fetched_dataを元のデータ型のデータに変換する。
     */



    /* fetched_dataを使って*retvalに値を入れる。 */
    gistentryinit(*retval, PointerGetDatum(converted_datum),
                  entry->rel, entry->page, entry->offset, FALSE);

    PG_RETURN_POINTER(retval);
}
```

compressメソッドがリーフエントリに対してデータ損失がある場合、演算子クラスはインデックスオンリースキャンをサポートすることができず、fetch関数を定義してはいけません。

options

演算子クラスの振舞いを制御するユーザに可視のパラメータの集合を定義します。

関数のSQL宣言は以下のようになります。

```
CREATE OR REPLACE FUNCTION my_options(internal)
RETURNS void
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

関数にはlocal_relopts構造体へのポインタが渡されますが、構造体を演算子クラスに固有のオプションの集合で満たすことが必要です。オプションはマクロPG_HAS_OPCLASS_OPTIONS()とPG_GET_OPCLASS_OPTIONS()を使って他のサポート関数からアクセスできます。

my_options()の実装と他のサポート関数からのパラメータの使用の例は以下の通りです。

```
typedef enum MyEnumType
{
    MY_ENUM_ON,
    MY_ENUM_OFF,
    MY_ENUM_AUTO
} MyEnumType;

typedef struct
{

    int32   vl_len_;    /* varlenaヘッダ(直接触らないこと!) */
    int     int_param;  /* 整数パラメータ */
    double  real_param; /* 実数パラメータ */
    MyEnumType enum_param; /* enumパラメータ */
    int     str_param;  /* 文字列パラメータ */
} MyOptionsStruct;


/* enum値の文字列表現 */
static relopt_enum_elt_def myEnumValues[] =
{
    {"on", MY_ENUM_ON},
    {"off", MY_ENUM_OFF},
    {"auto", MY_ENUM_AUTO},

    {(const char *) NULL}   /* リストの終端 */
};

static char *str_param_default = "default";
```

```
/*

 * 有効性検査の例: 文字列が8バイトより長くないことを検査します。
 */
static void
validate_my_string_relopt(const char *value)
{
    if (strlen(value) > 8)
        ereport(ERROR,
                (errcode(ERRCODE_INVALID_PARAMETER_VALUE),
                 errmsg("str_param must be at most 8 bytes")));
}

/*

 * 充填の例: 文字を小文字に交換します。
 */
static Size
fill_my_string_relopt(const char *value, void *ptr)
{
    char   *tmp = str_tolower(value, strlen(value), DEFAULT_COLLATION_OID);
    int     len = strlen(tmp);

    if (ptr)
        strcpy((char *) ptr, tmp);

    pfree(tmp);
    return len + 1;
}

PG_FUNCTION_INFO_V1(my_options);

Datum
my_options(PG_FUNCTION_ARGS)
{
    local_relopts *relopts = (local_relopts *) PG_GETARG_POINTER(0);

    init_local_reloptions(relopts, sizeof(MyOptionsStruct));
    add_local_int_reloption(relopts, "int_param", "integer parameter",
                            100, 0, 1000000,
                            offsetof(MyOptionsStruct, int_param));
    add_local_real_reloption(relopts, "real_param", "real parameter",
                             1.0, 0.0, 1000000.0,
                             offsetof(MyOptionsStruct, real_param));
    add_local_enum_reloption(relopts, "enum_param", "enum parameter",
                             myEnumValues, MY_ENUM_ON,
```

```
                              "Valid values are: \"on\", \"off\" and \"auto\".",
                              offsetof(MyOptionsStruct, enum_param));
    add_local_string_reloption(relopts, "str_param", "string parameter",
                               str_param_default,
                               &validate_my_string_relopt,
                               &fill_my_string_relopt,
                               offsetof(MyOptionsStruct, str_param));

    PG_RETURN_VOID();
}

PG_FUNCTION_INFO_V1(my_compress);

Datum
my_compress(PG_FUNCTION_ARGS)
{
    int     int_param = 100;
    double  real_param = 1.0;
    MyEnumType enum_param = MY_ENUM_ON;
    char   *str_param = str_param_default;

    /*

     * 通常は、
      演算子クラスが'options'メソッドを含む場合、
      optionsは常にサポート関数に
     * 渡されます。しかしながら、
      'options'メソッドを既存の演算子クラスに追加した場合、

     * 前に定義されたインデックスにoptionsがない場合、
      検査が必要です。
     */
    if (PG_HAS_OPCLASS_OPTIONS())
    {
        MyOptionsStruct *options = (MyOptionsStruct *) PG_GET_OPCLASS_OPTIONS();

        int_param = options->int_param;
        real_param = options->real_param;
        enum_param = options->enum_param;
        str_param = GET_STRING_RELOPTION(options, str_param);
    }


    /* サポート関数の残りの実装 */
}
```

GiSTでのキーの表現には柔軟性がありますので、ユーザに固有のパラメータに依存するかもしれません。例えば、キーの署名の長さが指定されるかもしれません。例についてはgtsvector_options()を参照してください。

すべてのGiSTサポートメソッドは通常短期間有効なメモリコンテキストで呼び出されます。つまりCurrentMemoryContextは各タプルが処理された後にリセットされます。そのためpallocしたすべてをpfreeすることに注意するのはあまり重要ではありません。しかし、サポートメソッドで、繰り返される呼び出しを跨がってデータをキャッシュすることが有用な場合があります。このためには、fcinfo->flinfo->fn_mcxtの中で長期間有効なデータを割り当て、そこへのポインタをfcinfo->flinfo->fn_extraの中に保持してください。こうしたデータはインデックス操作（例えば1つのGiSTインデックススキャン、インデックス構築、インデックスタプルの挿入）の間有効です。fn_extra値を置き換える時に以前の値をpfreeすることに注意してください。さもないと操作の間リークが蓄積されます。

64.4. 実装

64.4.1. バッファ付きGiST構築

すべてのタプルを単純に挿入することによって大規模なGiSTインデックスを構築することは、インデックスタプルがインデックス全体に分散し、インデックスがキャッシュに収まらない程大規模である場合、挿入の際に大量のランダムI/Oを行わなければなりませんので、低速になりがちです。バージョン9.2からPostgreSQLはバッファ処理に基づいてGiSTインデックスを構築するより効率的な方法をサポートします。これは、順序付けされていないデータ群に対して必要なランダムI/O数を劇的に減らします。十分に順序付けされたデータ群では、一度にわずかなページ数のみが新しいタプルを受け取り、そのためインデックス全体がキャッシュに収まらなくてもこれらのページがキャッシュ内に収まりますので、利点はより小さく、または利点がなくなります。

しかしバッファ付きインデックス構築は、CPUリソースを多少多く消費するpenalty関数をより多く呼び出さなければなりません。またバッファ付き構築で使用されるバッファは、最大作成されるインデックスと同じサイズまで、一時的にディスク容量を必要とします。バッファ処理は作成されるインデックスの品質にも、良くも悪くも、影響を与えます。この影響は、入力データの分布や演算子クラスの実装等、様々な要因に依存します。

デフォルトでは、インデックスのサイズがeffective_cache_sizeに達した時にGiSTインデックス構築はバッファ処理方式に切り替わります。CREATE INDEXコマンドのbufferingパラメータによって、手作業で有効または無効にすることができます。デフォルトの動作は大抵の場合良好です。しかし、入力データが順序付けされている場合、バッファ処理を無効にすることで構築が多少高速になります。

64.5. 例

PostgreSQLのソース配布物にはGiSTを使用したインデックスメソッドの実装のいくつかの事例が含まれています。コアシステムは現在全文検索サポート（tsvectorとtsqueryのインデックス付け）や組み込みの幾何データ型の一部に対するR-Treeと等価な機能を提供します（src/backend/access/gist/gistproc.cを参照してください）。以下のcontribモジュールも同時にGiST演算子クラスを含みます。

`btree_gist`

いくつかのデータ型に対するB-tree等価機能

`cube`

多次元の立方体用のインデックス

`hstore`

（キー、値）の組み合わせを格納するモジュール

`intarray`

int4値の1次元配列用のRD-Tree

`ltree`

疑似ツリー構造用のインデックス

`pg_trgm`

トライグラム一致を使用したテキストの類似性

`seg`

「浮動小数点範囲」のインデックス

第65章 SP-GiSTインデックス

65.1. はじめに

SP-GiSTは、空間分割された(Space-Partitioned)GiSTを短縮した語です。SP-GiSTは分割された探索木をサポートし、四分木、kd木、基数木(トライ木)など広範にわたる様々な非平衡データ構造の開発を可能にします。これらの構造に共通の特徴は、それらが探索空間を繰り返し小さな領域に分割し、その領域の大きさが必ずしも等しくない、ということです。分割規則によく適合した検索は非常に高速になります。

これらのよく使われるデータ構造は、元々はメモリ内での利用のために開発されたものでした。主記憶上では、それらは通常、ポインタにより接続され、動的に割り当てられるノードの集合として設計されます。このようなポインタのチェーンは長くなりがちで、非常に多くのディスクアクセスが必要となるため、ディスク上に直接格納するには適しません。これとは反対に、ディスクベースのデータ構造は、I/Oを最小化する、大きな論理出力数を持つべきです。SP-GiSTによって解決される困難とは、探索木のノードをディスクのページにマップするときに、多数のノードを通り抜ける場合であっても、探索ではごく少数のディスクページにしかアクセスしないですむようにすることです。

GiSTと同じく、SP-GiSTは適切なアクセス方法のある独自のデータ型の開発を可能にするためのもので、データベースのエキスパートよりもむしろ、そのデータ型の領域のエキスパートによる開発を可能にします。

ここで記述する情報の一部はPurdue大学のSP-GiSTインデックスプロジェクトWEBサイト[1]によるものです。PostgreSQLでのSP-GiSTの実装は、おもにTeodor SigaevとOleg Bartunovによって保守されており、詳しい情報は彼らのWEBサイト[2]にあります。

65.2. 組み込み演算子クラス

PostgreSQLのコアディストリビューションには表 65.1に示されるSP-GiSTの演算子クラスが含まれます。

表65.1 組み込みSP-GiST演算子クラス

名前	インデックスされるデータ型	インデックス可能な演算子	順序付け演算子
kd_point_ops	point	<< <@ <^ >> >^ ~=	<->
quad_point_ops	point	<< <@ <^ >> >^ ~=	<->
range_ops	任意の範囲型	&& &< &> -|- << <@ = >> @>	
box_ops	box	<< &< && &> >> ~= @> <@ &<| <<| |>> |&>	<->
poly_ops	polygon	<< &< && &> >> ~= @> <@ &<| <<| |>> |&>	<->
text_ops	text	< <= = > >= ~<=~ ~<~ ~>=~ ~>~ ^@	

[1] https://www.cs.purdue.edu/spgist/

[2] https://www.sai.msu.su/~megera/wiki/spgist_dev

名前	インデックスされるデータ型	インデックス可能な演算子	順序付け演算子
inet_ops	inet, cidr	&& >> >>= > >= <> << <<= < <= =	

point型の2つの演算子クラスのうち、quad_point_opsがデフォルトです。kd_point_opsは同じ演算子をサポートしますが、異なるインデックスデータ構造を使うため、アプリケーションによってはより良いパフォーマンスを提供することがあります。

quad_point_ops、kd_point_ops、poly_ops演算子クラスは<->順序付け演算子をサポートしますので、インデックス付けされた点や多角形のデータ集合に対してk近傍(k-NN)探索が可能です。

65.3. 拡張性

SP-GiSTは高度に抽象化されたインタフェースを提供します。アクセスメソッドの開発者は特定のデータ型専用のメソッドだけを開発する必要があります。SP-GiSTのコアは効率的なディスクマッピングと木構造の探索を担当します。また、同時実行制御とログ出力も担当します。

SP-GiSTのツリーのリーフタプルは、インデックスの付けられた列と同じデータ型の値を含んでいます。ルートレベルにあるリーフタプルは、必ずインデックスが付けられた元のデータの値を含んでいますが、より下のレベルのリーフタプルは、接尾辞など、圧縮された表現しか含んでいないかも知れません。この場合、演算子クラスのサポート関数が、内部タプルをリーフレベルまでたどりながら集める情報を使って元の値を再構築できる必要があります。

内部タプルは、探索木の分岐点となるため、もっと複雑です。それぞれの内部タプルは1つ以上の*ノード*の集合を含んでおり、ノードは類似のリーフ値のグループを表現します。ノードは下向きのリンクを含んでおり、これは下のレベルの別の内部タプルを指すか、あるいはすべて同じインデックスページ上に載っているリーフタプルの短いリストを指しています。それぞれのノードは、通常、それを記述する*ラベル*を持っています。例えば、基数木では、ノードのラベルは文字列の値の次の文字にすることができます。(あるいは、すべての内部タプルについて、決まったノードの集合しか扱わないのであれば、演算子クラスはノードのラベルを省略することができます。65.4.2を参照してください。) 省略可能ですが、内部タプルはそのすべてのメンバーを記述する*接頭辞*の値を持つことができます。基数木では、これは表現される文字列に共通の接頭辞とすることができます。接頭辞の値は、必ずしも本当の接頭辞である必要はなく、演算子クラスが必要とする任意の値で良いです。例えば四分木では、その中心点を保持し、4つの象限をそこから相対的に測るようにできます。そうすると、四分木の内部タプルはこの中心点の周りの象限に対応する4つのノードも含むことになるでしょう。

木構造のアルゴリズムには、現在のタプルのレベル(深さ)を知っていることが必要なものがあります。そこで、SP-GiSTのコアは、演算子クラスが木構造をたどって下がるときにレベル数の管理を可能にしています。また、必要であれば、表現される値を加算的に再構築すること、また木構造を下る間に追加データ(*探索値*と呼ばれます)を渡すこともサポートしています。

> **注記**
>
> SP-GiSTのコアのコードはnullエントリについても対応しています。SP-GiSTのインデックスはインデックス列がnullのエントリについても格納しますが、これはインデックスの演算子クラスのコードからは隠されているので、nullのインデックスエントリや検索条件が演算子クラスのメソッドに渡されること

はありません。(SP-GiSTの演算子は厳格なのでNULL値について成功を返すことはできないと想定されています。) 従って、ここではこれ以上、NULLについて議論しません。

SP-GiSTのインデックス演算子クラスが提供しなければならないユーザ定義メソッドは5つあり、加えて、オプションのメソッドが2つあります。5つの必須メソッド全ては2つのinternal引数を受け付けるというしきたりに従い、1つ目はサポートメソッドへの入力値を含むC構造体へのポインタで、一方2つ目は出力値が配置されるであろうC構造体へのポインタです。4つの必須メソッドでは、その結果がすべて出力構造体の中にあるので、単にvoidを返します。ですが、leaf_consistentはbooleanの結果を返します。メソッドは、その入力構造体のどのフィールドも変更してはいけません。どんな場合でも、出力構造体はユーザ定義メソッドを呼び出す前にゼロに初期化されます。オプションの6番目のメソッドcompressは、唯一の引数としてインデックス付けされるdatumを受け付け、リーフタプルの物理格納に適した値を返します。オプションの7番目のメソッドoptionsは、演算子クラスに固有のパラメータを入れるC構造体へのinternalポインタを受け付け、voidを返します。

5つの必須ユーザ定義メソッドは以下のとおりです。

config

接頭辞とノードラベルのデータ型のデータ型OIDを含め、インデックスの実装に関する静的情報を返します。

関数のSQL宣言は以下のようになります。

```
CREATE FUNCTION my_config(internal, internal) RETURNS void ...
```

1番目の引数はCのspgConfigIn構造体へのポインタで、関数の入力データを含みます。2番目の引数はCのspgConfigOut構造体へのポインタで、関数が結果のデータを入れます。

```
typedef struct spgConfigIn
{

    Oid         attType;        /* インデックス付けされるデータ型 */
} spgConfigIn;

typedef struct spgConfigOut
{

    Oid         prefixType;     /* 内部タプルの接頭辞のデータ型 */
    Oid         labelType;      /* 内部タプルのノードのラベルのデータ型 */
    Oid         leafType;       /* リーフタプル値のデータ型 */
    bool        canReturnData;  /* 演算子クラスは元のデータを再構築できる */
    bool        longValuesOK;   /* 演算子クラスは1ページより大きな値を扱える */
} spgConfigOut;
```

attTypeは多様のインデックス演算子クラスをサポートするために渡されます。通常の固定データ型の演算子クラスでは、それは常に同じ値を持っているので無視できます。

接頭辞を使わない演算子クラスでは、prefixTypeをVOIDOIDに設定することができます。同様に、ノードラベルを使わない演算子クラスでは、labelTypeをVOIDOIDに設定することができます。演算子クラスが、元々提供されていたインデックスの値を再構築できるときは、canReturnDataをtrueにします。attTypeが可変長で、演算子クラスが接尾辞付けの繰り返しによって長い値を分割できるときにのみ、longValuesOKをtrueにします(65.4.1参照)。

leafTypeは一般的にはattTypeと同じです。後方互換性のため、configメソッドはleafTypeを初期化しないままにすることができます。このことはleafTypeをattTypeと同一に設定するのと同じ効果をもたらすでしょう。attTypeとleafTypeが異なるときには、オプションメソッドcompressが提供されなければなりません。メソッドcompressはattTypeからleafTypeへのインデックス付けされたデータの変換に責任を持ちます。注意: 両consistent関数は、compressを使った変換を伴わない、変更されていないscankeysを取得します。

choose

内部タプルに新しい値を挿入するときのメソッドを選択します。

関数のSQL宣言は以下のようになります。

```
CREATE FUNCTION my_choose(internal, internal) RETURNS void ...
```

1番目の引数はCのspgChooseIn構造体へのポインタで、関数の入力データを含みます。2番目の引数はCのspgChooseOut構造体へのポインタで、関数が結果のデータを入れます。

```
typedef struct spgChooseIn
{

    Datum       datum;          /* インデックス付けされる元のデータ */
    Datum       leafDatum;      /* リーフに保存されている現在のデータ */
    int         level;          /* (0から数えた)現在のレベル */


    /* 現在の内部タプルからのデータ */
    bool        allTheSame;     /* タプルはall-the-sameの印を付けられているか */
    bool        hasPrefix;      /* タプルは接頭辞を持っているか */
    Datum       prefixDatum;    /* そうであれば、
      接頭辞の値 */
    int         nNodes;         /* 内部タプル内のノード数 */
    Datum      *nodeLabels;     /* ノードのラベルの値(なければNULL) */
} spgChooseIn;

typedef enum spgChooseResultType
{

    spgMatchNode = 1,           /* 既存のノードに下がる */
    spgAddNode,                 /* ノードに内部タプルを追加する */
    spgSplitTuple               /* 内部タプルを分割する(その接頭辞を変更する) */
} spgChooseResultType;
```

```
typedef struct spgChooseOut
{

    spgChooseResultType resultType;     /* アクションコード、
      上記参照 */
    union
    {

        struct                  /* spgMatchNodeの結果 */
        {

            int         nodeN;      /* このノードに下がる(0からのインデックス) */
            int         levelAdd;   /* この分だけレベルを増やす */
            Datum       restDatum;  /* 新しいリーフデータ */
        }           matchNode;

        struct                  /* spgAddNodeの結果 */
        {

            Datum       nodeLabel;  /* 新しいノードのラベル */
            int         nodeN;      /* 挿入する場所(0からのインデックス) */
        }           addNode;

        struct                  /* spgSplitTupleの結果 */
        {

            /* 子タプルを1つ持つ新しい上位のレベルの内部タプルを生成するための情報 */
            bool        prefixHasPrefix;    /* タプルは接頭辞を持つか */
            Datum       prefixPrefixDatum;  /* そうならば、
      その値 */
            int         prefixNNodes;       /* ノード数 */
            Datum      *prefixNodeLabels;   /* そのラベル(ラベルがなければNULL) */
            int         childNodeN;         /* どのタプルが子タプルを得るか */


            /* 古いノードをすべて持つ新しい低位の内部タプルを生成するための情報 */
            bool        postfixHasPrefix;   /* タプルは接頭辞を持つか */
            Datum       postfixPrefixDatum; /* そうならば、その値 */
        }           splitTuple;
    }           result;
} spgChooseOut;
```

datumはインデックスに挿入できたspgConfigIn.attType型の元データです。
leafDatumはspgConfigOut.leafType型の値です。これは最初は、メソッドcompressが提供されているならdatumに適用されたメソッドcompressの結果で、さもなくばdatumと同じ値です。leafDatumは、

chooseやpicksplitメソッドがこれを変更すると、ツリーのより低いレベルで変化することがあります。挿入の探索がリーフページに到達するとき、leafDatumの現在値は、新しく作成されるリーフタプルに格納される値です。levelは、ルートレベルを0として、現在の内部タプルのレベルを示します。現在の内部タプルが複数の同等なノードを含むとして印を付けられているとき、allTheSameをtrueにします(65.4.3参照)。現在の内部タプルが接頭辞を含むとき、hasPrefixをtrueにします。このとき、prefixDatumがその値になります。nNodesは内部タプルが含む子ノードの数で、nodeLabelsはそれらのラベル値の配列、あるいはラベルがなければNULLになります。

choose関数は、新しい値が既存の子ノードの1つとマッチするか、新しい子ノードを追加する必要があるか、あるいは新しい値がタプルの接頭辞と適合しないので内部タプルを分割してより制限のない接頭辞を作成する必要があるか、を決定することができます。

新しい値が既存の子ノードの1つにマッチしたときは、resultTypeをspgMatchNodeにセットします。nodeNはノードの配列中のそのノードの(0からの)番号にセットします。levelAddは、そのノードをたどって下がるときに生じたlevelの増分にセットします。あるいは演算子クラスがレベルを使っていなければ0のままにします。restDatumは、演算子クラスがデータをあるレベルから次のレベルに変更しないのであれば、datumに等しくセットします。そうでなければ、次のレベルでleafDatumとして使われる修正された値にセットします。

新しい子ノードを追加しなければならないときは、resultTypeをspgAddNodeにセットします。nodeLabelは、新しいノードで使われるラベルにセットし、nodeNはノードの配列中の挿入される場所のノードの(0からの)番号にセットします。ノードを追加した後で、choose関数を修正された内部タプルを使って再び呼び出しますが、このときは、spgMatchNodeという結果になるはずです。

新しい値がタプルの接頭辞と適合しないときは、resultTypeをspgSplitTupleにセットします。このアクションは、すべての既存のノードを新しい低位の内部タプルに移動し、新しい低位の内部タプルを指す単一の下向きのリンクを持つ新しいタプルで既存のタプルを置換します。prefixHasPrefixは新しい上位のタプルが接頭辞を持つかどうかを示し、持つ場合にはprefixPrefixDatumをその接頭辞の値にセットします。インデックスに追加される新しい値を受け入れるため、新しい接頭辞の値は元のものよりも十分に制限の緩いものになっていなければなりません。prefixNNodesは新しいタプルで必要なノード数にセットし、prefixNodeLabelsはラベルを保持するためにpallocされた配列に、ノードのラベルが必要でないときはNULLにセットします。新しい上位のタプルの全サイズは置き換えるタプルの全サイズよりも大きくはないことに注意してください。これは新しい接頭辞と新しいラベルの長さを制約します。childNodeNは、新しい低位の内部タプルへ下向きにリンクするノードの(0からの)番号にセットします。postfixHasPrefixは、新しい低位のタプルが接頭辞を持つかどうかを示し、持つときにはpostfixPrefixDatumを接頭辞の値にセットします。新しい低位に移動したタプルのノードのラベルを変更する機会も、子のインデックスのエントリを変更する機会もありませんから、これら2つの接頭辞と(もしあれば)下向きのリンクのノードのラベルの組み合わせは、元の接頭辞と同じ意味を持つ必要があります。ノードが分割された後で、置換した内部タプルを使ってchoose関数を再び呼び出します。この呼び出しは、spgSplitTupleアクションにより適切なノードが作られなければ、spgAddNodeという結果になります。そのうち、chooseがspgMatchNodeを返し、次のレベルに下がる挿入が可能となります。

picksplit

リーフタプルの集合に対し、新しい内部タプルをどうやって作るかを決定します。

関数のSQL宣言は以下のようになります。

```
CREATE FUNCTION my_picksplit(internal, internal) RETURNS void ...
```

1番目の引数はCのspgPickSplitIn構造体へのポインタで、関数の入力データを含みます。2番目の引数はCのspgPickSplitOut構造体へのポインタで、関数が結果のデータを入れます。

```
typedef struct spgPickSplitIn
{

    int         nTuples;        /* リーフタプルの数 */
    Datum      *datums;         /* そのデータ(長さnTuplesの配列) */
    int         level;          /* (0から数えた)現在のレベル */
} spgPickSplitIn;

typedef struct spgPickSplitOut
{

    bool        hasPrefix;      /* 新しい内部タプルは接頭辞を持つか */
    Datum       prefixDatum;    /* もしそうなら、
      その値 */


    int         nNodes;         /* 新しい内部タプルのノード数 */
    Datum      *nodeLabels;     /* そのラベル(ラベルがなければNULL) */


    int        *mapTuplesToNodes;   /* 各リーフタプルへのノードのインデックス */
    Datum      *leafTupleDatums;    /* 新しい各リーフタプルに保存されているデータ */
} spgPickSplitOut;
```

nTuplesは与えられるリーフタプルの個数です。datumsはspgConfigOut.leafType型のそれらのデータ値の配列です。levelはすべてのリーフタプルが共有する現在のレベルで、これが新しい内部タプルのレベルになります。

hasPrefixは新しい内部タプルが接頭辞を持つかどうかを示し、持つ場合はprefixDatumを接頭辞の値にセットします。nNodesは新しい内部タプルが含むノードの数を示し、nodeLabelsはそのラベル値の配列に、ノードのラベルが必要でないときはNULLにセットします。mapTuplesToNodesは、それぞれのリーフタプルが割り当てられるノードの(0からの)番号の配列にセットします。leafTupleDatumsは新しいリーフタプルに格納される値の配列にセットします(演算子クラスがデータをあるレベルから次のレベルに変更しなければこれらは入力のdatumsと同じになります)。picksplit関数は、nodeLabels、mapTuplesToNodes、leafTupleDatumsの配列についてpallocしなければならないことに注意してください。

2つ以上のリーフタプルを与えた場合、picksplit関数はそれらを2つ以上のノードに分類すると予想されます。そうでなければ、リーフタプルを複数のページにまたがって分割するという、この操作の究極の目的を実現できないからです。従って、picksplitがすべてのリーフタプルを同じノードに置くことになった場合には、SP-GiSTのコアのコードがその決定を覆して内部タプルを生成し、その中の複数の同一のラベルが付けられたノードに、リーフタプルが無作為に割り当てられます。そのようなタプルは、このことが発生したことを明示するため、allTheSameと印がつけられます。choose関数とinner_consistent関

数は、これらの内部タプルについて、適切な注意をして取り扱わなければなりません。詳細な情報は65.4.3を参照してください。

config関数がlongValuesOKをtrueにセットし、1ページよりも大きな入力値を与える場合にのみ、picksplitを1つだけのリーフタプルに適用できます。この場合の操作の重要な点は、接頭辞をはがして、新しい、より短いリーフデータの値を生成することです。この呼出は、1ページに収まる短さのリーフデータが生成されるまで繰り返されます。詳細な情報は65.4.1を参照してください。

inner_consistent

ツリーの探索でたどるべきノード(枝)の集合を返します。

関数のSQL宣言は以下のようになります。

```
CREATE FUNCTION my_inner_consistent(internal, internal) RETURNS void ...
```

1番目の引数はCのspgInnerConsistentIn構造体へのポインタで、関数の入力データを含みます。2番目の引数はCのspgInnerConsistentOut構造体へのポインタで、関数が結果のデータを入れます。

```
typedef struct spgInnerConsistentIn
{

    ScanKey     scankeys;       /* 演算子と比較する値の配列 */
    ScanKey     orderbys;       /* 順序付け演算子と比較する値の配列 */
    int         nkeys;          /* scankeys配列の長さ */
    int         norderbys;      /* orderbys配列の長さ */


    Datum       reconstructedValue;     /* 親で再構築された値 */
    void       *traversalValue; /* 演算子クラスに固有の探索値 */
    MemoryContext traversalMemoryContext;   /* 新しい探索値をここに入れる */
    int         level;          /* (0から数えた)現在のレベル */
    bool        returnData;     /* 元のデータを返さなければならないか */


    /* 現在の内部タプルからのデータ */
    bool        allTheSame;     /* タプルはall-the-sameと印が付けられているか */
    bool        hasPrefix;      /* タプルは接頭辞を持つか */
    Datum       prefixDatum;    /* もしそうなら、
      接頭辞の値 */
    int         nNodes;         /* 内部タプルの中のノード数 */
    Datum      *nodeLabels;     /* ノードのラベルの値(なければNULL) */
} spgInnerConsistentIn;

typedef struct spgInnerConsistentOut
{
```

```
    int         nNodes;         /* 訪れるべき子ノードの数 */
    int        *nodeNumbers;    /* ノードの配列でのそのインデックス */
    int        *levelAdds;      /* この分だけそれぞれレベルを挙げる */
    Datum      *reconstructedValues;    /* 関連する再構築された値 */
    void      **traversalValues;        /* 演算子クラスに固有の探索値 */
    double    **distances;              /* 関連する距離 */
} spgInnerConsistentOut;
```

配列scankeysは長さがnkeysで、インデックス検索の条件を記述します。複数の条件はANDで結合されます。つまり、条件のすべてを満たすインデックスエントリのみが対象となります。(nkeys = 0 は全インデックスエントリが問い合わせを満たす意味になる、ということに注意してください。) 通常、consistent関数では、配列のそれぞれのエントリのsk_strategyおよびsk_argumentフィールドのみが問題となります。これらのフィールドにはそれぞれインデックス付け可能な演算子と比較値が入ります。とりわけ、比較値がNULLかどうかを確認するためにsk_flagsを検査する必要はありません。なぜならSP-GiSTのコアのコードがそのような条件を除外するからです。配列orderbysは長さがnorderbysで、(もしあれば)順序付け演算子を同じように記述します。reconstructedValueは親タプルのために再構築された値で、ルートレベルの場合、あるいは親レベルのinner_consistent関数が値を返さなかった場合は(Datum) 0となります。reconstructedValueは常にspgConfigOut.leafType型です。traversalValueは親インデックスのタプルのinner_consistentの前の呼び出しから渡された探索データへのポインタで、ルートレベルならNULLです。traversalMemoryContextは出力探索値が格納されるメモリコンテキストです(以下を参照)。levelは現在の内部タプルのレベルを、ルートレベルを0として数えたものです。returnDataは、この問い合わせで再構築されたデータが必要な場合にtrueとなりますが、これはconfig関数がcanReturnDataであると主張した場合にのみ、そうなります。現在の内部タプルが「all-the-same」と印付けされているなら、allTheSameは真になります。この場合、(もしあるなら)全てのノードが同じラベルを持ち、問い合わせに全てが一致するか、全く一致しないかのいずれかになります(65.4.3を参照)。現在の内部タプルがプレフィックスを含んでいるならhasPrefixは真になります。その場合、prefixDatumがその値です。nNodesは内部タプルに含まれる子ノードの数で、nodeLabelsはそれらのラベル値の配列、あるいは、ノードがラベルを持たないならNULLです。

nNodesは探索で訪れる必要のある子ノードの数にセットされなければなりません。また、nodeNumbersはそれらの番号の配列にセットされなければなりません。演算子クラスがレベルを監視しているときは、それぞれのノードへと下って訪れるときに必要なレベルの増分の配列をlevelAddsにセットします。(この増分はすべてのノードについて同じになることも多いですが、必ずしもそうなるとは限らないので配列が使われます。) 値の再構築が必要なときには、訪れるそれぞれの子ノードについて再構築されたspgConfigOut.leafType型の値の配列をreconstructedValuesにセットします。再構築が必要でなければ、reconstructedValuesをNULLのままにします。順序付け検索を実行するなら、orderbys配列に従ってdistancesに距離の値の配列を設定します(距離の最も近いノードが最初に処理されます)。そうでなければNULLのままにします。追加の外部情報(「探索値」)をツリー探索の下位レベルに渡したい場合は、traversalValuesを適切な探索値、訪れるそれぞれの子ノードについて1つの配列にセットします。それ以外の場合はtraversalValuesをNULLのままにします。inner_consistent関数は、現在のメモリコンテキスト内のnodeNumbers、levelAdds、distances、reconstructedValues、traversalValuesの配列についてpallocしなければならないことに注意してください。ただし、traversalValues配列が指すすべての出力探索値はtraversalMemoryContext内に割り当てられます。それぞれの探索値は1つのpallocされた塊でなければなりません。

leaf_consistent

リーフタプルが問い合わせを満たす場合、trueを返します。

関数のSQL宣言は以下のようになります。

```
CREATE FUNCTION my_leaf_consistent(internal, internal) RETURNS bool ...
```

1番目の引数はCのspgLeafConsistentIn構造体へのポインタで、関数の入力データを含みます。2番目の引数はCのspgLeafConsistentOut構造体へのポインタで、関数が結果のデータを入れます。

```
typedef struct spgLeafConsistentIn
{

    ScanKey     scankeys;       /* 演算子と比較する値の配列 */
    ScanKey     orderbys;       /* 順序付け演算子と比較する値の配列 */
    int         nkeys;          /* scankeys配列の長さ */
    int         norderbys;      /* orderbys配列の長さ */


    Datum       reconstructedValue;     /* 親で再構築された値 */
    void       *traversalValue; /* 演算子クラスに固有の探索値 */
    int         level;          /* (0から数えた)現在のレベル */
    bool        returnData;     /* 元のデータを返さなければならないか */


    Datum       leafDatum;      /* リーフタプルのデータ */
} spgLeafConsistentIn;

typedef struct spgLeafConsistentOut
{

    Datum       leafValue;        /* もしあれば、再構築された元のデータ */
    bool        recheck;          /* 演算子を再チェックする必要があればtrue */
    bool        recheckDistances; /* 距離を再チェックする必要があればtrue */
    double     *distances;        /* 関連する距離 */
} spgLeafConsistentOut;
```

配列scankeysは長さがnkeysで、インデックス探索の条件を記述します。複数の条件はANDで結合されます。つまり、条件のすべてを満たすインデックスエントリのみが対象となります。(nkeysが0ならば、すべてのエントリが検索条件を満たすことになる、ということに注意してください。) 通常、consistent関数では、配列のそれぞれのエントリのsk_strategyおよびsk_argumentフィールドのみが問題となります。これらのフィールドにはそれぞれインデックス付け可能な演算子と比較値が入ります。なお、比較値がNULLかどうかを確認するためにsk_flagsを検査する必要はありません。なぜならSP-GiSTのコアのコードがそのような条件を除外するからです。配列orderbysは長さがnorderbysで、順序付け演算子を同じように記述します。reconstructedValueは親タプルのために再構築された値で、ルートレベルの場合、あるいは親レベルのinner_consistent関数が値を返さなかった場合は(Datum) 0となります。reconstructedValueは常にspgConfigOut.leafType型です。traversalValueは親インデックスのタプルのinner_consistentの前の呼び出しから渡された探索データへのポインタで、ルートレベルならNULLです。levelは現在のリーフタプルのレベルを、ルートレベルを0として数えたものです。returnDataは、この問い合わせで再構築されたデータが必要な場合にtrueとなりますが、これ

はconfig関数がcanReturnDataを確認した場合にのみ、そうなります。leafDatumは現在のリーフタプルに格納されているキーの値です。

この関数は、リーフタプルが問い合わせにマッチすればtrueを返し、マッチしなければfalseを返します。trueの場合、returnDataがtrueであれば、leafValueは、このリーフタプルにインデックス付けするために元々提供されたspgConfigIn.attType型の値にセットされなければなりません。また、マッチするかどうかが不確実で、マッチするかの確認のために実際のヒープタプルに演算子を再適用しなければならないときは、recheckがtrueにセットされることがあります。順序付け検索を実行するなら、orderbys配列に従ってdistancesに距離の値の配列を設定します。そうでなければNULLのままにします。返される距離の少なくとも1つが正確でないのなら、recheckDistancesにtrueを設定します。この場合、エクゼキュータはヒープからタプルを取得した後正確な距離を計算し、必要ならタプルを並べ替えます。

オプションのユーザ定義メソッドは以下です。

Datum compress(Datum in)

インデックページのリーフタプルでデータ項目を物理ストレージに適した形式に変換します。spgConfigIn.attTypeの値を受け付け、spgConfigOut.leafTypeの値を返します。出力値はTOASTされていないべきです。

options

演算子クラスの振舞いを制御するユーザに可視のパラメータの集合を定義します。

関数のSQL宣言は以下のようになります。

```
CREATE OR REPLACE FUNCTION my_options(internal)
RETURNS void
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

関数にはlocal_relopts構造体へのポインタが渡されますが、構造体を演算子クラスに固有のオプションの集合で満たすことが必要です。オプションはマクロPG_HAS_OPCLASS_OPTIONS()とPG_GET_OPCLASS_OPTIONS()を使って他のサポート関数からアクセスできます。

SP-GiSTでのキーの表現には柔軟性がありますので、ユーザに固有のパラメータに依存するかもしれません。

SP-GiSTのすべてのサポートメソッドは、通常は短期間有効なメモリコンテキスト内で呼び出されます。つまり、それぞれのタプルについて処理した後でCurrentMemoryContextはリセットされます。したがって、pallocしたものすべてについてpfreeすることを気にかけることはあまり重要ではありません。(configメソッドは例外で、メモリリークを避けるようにする必要があります。しかし、通常はconfigメソッドは、パラメータとして渡された構造体に定数を代入する以外、何もする必要がありません。)

インデックス付けされた列が照合可能なデータ型の場合、インデックスの照合は、標準的なPG_GET_COLLATION()の仕組みを使ってすべてのサポートメソッドに渡されます。

65.4. 実装

この節では、SP-GiSTの演算子クラスを実装する人にとって知っていると役に立つ、実装についての詳細とその他の秘訣について説明します。

65.4.1. SP-GiSTの制限

それぞれのリーフタプルおよび内部タプルは1つのインデックスページ内(デフォルトで8kB)に収まらなければなりません。従って、可変長のデータ型の値をインデックス付けするときは、長い値は基数木のようなメソッドによってのみサポートされます。つまり、ツリーのそれぞれのレベルではページに収まる短さの接頭辞を含み、最後のリーフレベルでは、やはりページに収まる短さの接尾辞を含む、というようなものです。このようなことが発生する場合の対応の準備ができている場合のみ、演算子クラスは`longValuesOK`を真にセットするべきです。そうでなければ、SP-GiSTのコアは、インデックスページに収めるには大きすぎる値についてのインデックス付け要求を拒絶します。

同様に、内部タプルが大きくなりすぎてインデックスページに収まらない、ということにならないようにするのは、演算子クラスの責任です。これにより、1つの内部タプルで使うことができる子ノードの数、および接頭辞の値の最大サイズが制限されます。

内部タプルのノードがリーフタプルの集合を指しているとき、それらのタプルはすべて同じインデックスページ内になければならない、という制限もあります。(これは、シークの回数を減らし、そのようなタプルを一つにつなげるリンクに必要なスペースを減らす、という設計上の決定によるものです。)リーフタプルの集合が大きくなって1ページに収まらなくなると、分割が実行され、中間の内部タプルが挿入されます。これで問題を解決するためには、新しい内部タプルは、リーフの値の集合を2つ以上のノードのグループに分割しなければなりません。演算子クラスの`picksplit`関数がそれをするのに失敗したときは、SP-GiSTのコアは、65.4.3に記述されている特別な手段に頼ることになります。

65.4.2. ノードラベルのないSP-GiST

木構造のアルゴリズムには、それぞれの内部タプルに対して固定された集合のノードを使うものがあります。例えば四分木では、内部タプルの中心点の周りの4つの象限に対応するちょうど4つのノードが必ずあります。このような場合、コードは典型的には数字を使ったノードで動作し、明示的なノードラベルは必要ありません。ノードラベルを使わない(そしてそれによりいくらかのスペースを節約する)ために、`picksplit`関数は`nodeLabels`配列として`NULL`を返すことができ、同様に`choose`関数は`spgSplitTuple`アクションの間`prefixNodeLabels`配列として`NULL`を返すことができます。この結果、その後の`choose`関数および`inner_consistent`関数の呼び出しにおいても`nodeLabels`は`NULL`になります。原則として、ノードラベルは同じインデックス中の一部の内部タプルに使い、他の内部タプルには省略する、ということができます。

ラベルのないノードを持つ内部タプルを処理するときに、`choose`が`spgAddNode`を返すのはエラーです。というのは、この場合、ノードの集合は固定されていると想定されるからです。

65.4.3. 「All-the-Same」内部タプル

`picksplit`が入力のリーフ値を少なくとも2つのノード分類に分割できなかった場合、SP-GiSTのコアは演算子クラスの`picksplit`関数の結果を無効にすることがあります。これが起きると、複数のノードを持

つ新しい内部タプルが作成されます。それぞれのノードは、picksplitが一つのノードに付与したもの(あれば)と同じラベルを持ち、リーフ値はこれらの等価なノード間でランダムに分割されます。内部タプルにはallTheSameのフラグがセットされ、choose関数およびinner_consistent関数に対し、そのタプルが通常期待されるようなノードの集合を持っていないことを警告します。

allTheSameの処理において、chooseのspgMatchNodeという結果は、新しい値は等価なノードのどれに割り当てられても良い、という意味に解釈されます。コアのコードは入力されたnodeNの値を無視し、(ツリーの平衡を保つために)ノードの1つにランダムに降りていきます。chooseがspgAddNodeを返すのはエラーです。というのは、そうするとすべてのノードが等価ではなくなるからです。挿入する値が既存のノードとマッチしない時は、spgSplitTupleのアクションを使わなければなりません。

allTheSameのタプルの処理において、すべてのノードは等価なので、inner_consistent関数は、インデックス検索を続けるためのターゲットとして、すべてのノードを返すか、ノードを1つも返さないかのいずれかであるべきです。このために、特殊ケースを扱うコードが必要になるかもしれませんし、必要ないかもしれません。それは、inner_consistent関数が、通常、ノードの意味についてどの程度のことを仮定しているかに依存します。

65.5. 例

PostgreSQLのソースコードの配布物には、表 65.1に示すように、SP-GiSTのインデックス演算子クラスの例がいくつか含まれています。コードを見るにはsrc/backend/access/spgist/とsrc/backend/utils/adt/を調べてみてください。

第66章 GINインデックス

66.1. はじめに

GINとは汎用転置インデックス(Generalized Inverted Index)を表します。GINは、以下のような状況を取り扱うために設計されました。(1)インデックス対象の項目が複合型である。(2)そのインデックスにより処理される問い合わせは、複合型の項目内に存在する要素の値を検索する必要がある。例えば、項目は文書であり、問い合わせは特定の単語を含む文書の検索です。

ここでは、インデックス対象の複合型の値を*項目*と呼びます。また、要素値を*キー*と呼びます。GINは項目の値自体ではなく、常にキーを格納し検索します。

GINインデックスは(キー、ポスティングリスト(posting list))の組み合わせの集合を格納します。ここで*ポスティングリスト*はキーが発生した行IDの集合です。項目は1つ以上のキーを含むことができますので、同じ行IDが複数のポスティングリストに現れることがあり得ます。キー値はそれぞれ一度のみ格納されます。このためGINインデックスの容量は、同じキーが何度も現れる場合に非常に小さくなります。

GINインデックスは、GINアクセスメソッドが高速化対象の操作を把握する必要がないという意味で汎用化されています。その代わり、特定のデータ型に対して定義された独自の戦略を使用します。戦略は、インデックス付けされた項目と問い合わせ条件からキーを抽出する方法および問い合わせ内のいくつかのキー値を含む行が実際に問い合わせを満たすかどうかを決定する方法を定義します。

GINの利点の1つは、データベース専門家ではなくデータ型の分野における専門家により、適切なアクセスメソッドを持つ独自のデータ型を開発できるという点です。これはGiSTの使用とほぼ同じ利点です。

PostgreSQLにおけるGINの実装は、主にTeodor SigaevとOleg Bartunovにより保守されています。GINに関する情報は彼らのwebサイト[1]により多く記載されています。

66.2. 組み込み演算子クラス

PostgreSQLのコア配布物は表 66.1に示すGIN演算子クラスを含みます。(付録Fに記載された追加モジュールの中には追加のGIN演算子クラスを提供するものもあります。)

表66.1 組み込みGIN演算子クラス

名前	インデックスされるデータ型	インデックス可能な演算子
array_ops	anyarray	&& <@ = @>
jsonb_ops	jsonb	? ?& ?| @> @? @@
jsonb_path_ops	jsonb	@> @? @@
tsvector_ops	tsvector	@@ @@@

[1] http://www.sai.msu.su/~megera/wiki/Gin

jsonb型の2つの演算子クラスのうち、jsonb_opsがデフォルトです。jsonb_path_opsはより少数の演算子しかサポートしませんが、その演算子に対してはより良いパフォーマンスを提供します。詳細は8.14.4を参照してください。

66.3. 拡張性

GINインタフェースは高度に抽象化されています。アクセスメソッド実装者に要求されることは、アクセスするデータ型の意味を実装することだけです。GIN層自体が同時実行性、ログ処理、ツリー構造の検索処理に関する面倒を見ます。

GINアクセスメソッドを動作させるために必要なことは、少数のユーザ定義関数を実装することだけです。これは、ツリー内のキーの動作とキーとインデックス付けされる項目、インデックス可能な問い合わせ間の関係を定義します。すなわち、GINは、一般化、コード再利用、整理されたインタフェースによる拡張性を組み合わせます。

GIN用の演算子クラスが提供しなければならないメソッドは2つあります。

```
Datum *extractValue(Datum itemValue, int32 *nkeys, bool **nullFlags)
```

インデックス対象値に与えられる、pallocで割り当てられたキーの配列を返します。返されるキーの数は*nkeysに格納しなければなりません。キーのいずれかがNULLになるかもしれない場合、*nkeys個のboolの配列をpallocで割り当てそのアドレスを*nullFlagsに格納し、必要に応じてNULLフラグを設定してください。すべてのキーが非NULLであれば、*nullFlagsをNULL（初期値）のままにすることができます。項目がキーを含まない場合、戻り値はNULLになるかもしれません。

```
Datum *extractQuery(Datum query, int32 *nkeys, StrategyNumber n, bool **pmatch, Pointer
**extra_data, bool **nullFlags, int32 *searchMode)
```

問い合わせ対象の値に与えられる、pallocで割り当てられたキーの配列を返します。つまり、queryはインデックス可能な演算子の右辺の値です。この左辺はインデックス対象の列です。nは演算子クラス内の演算子の戦略番号です（37.16.2を参照）。extractQueryはしばしば、queryのデータ型とキー値を抽出するために使用しなければならないメソッドを決定するために、nを調べなければなりません。返されるキーの数を*nkeysに格納しなければなりません。キーのいずれかがNULLとなる可能性がある場合はまた、*nkeys個のboolの配列をpallocで割り当て、*nullFlagsにそのアドレスを格納し、必要に応じてNULLフラグを設定してください。すべてのキーが非NULLならば*nullFlagsはNULL（初期値）のままにしておくことができます。queryがキーを含まない場合、戻り値をNULLにすることができます。

searchModeは出力引数です。これによりextractQueryは検索がどのように行われるかの詳細を指定することができます。*searchModeがGIN_SEARCH_MODE_DEFAULT（呼び出し前にこの値に初期化されます。）に設定された場合、返されるキーの少なくとも1つに一致する項目が合致候補とみなされます。*searchModeがGIN_SEARCH_MODE_INCLUDE_EMPTYに設定された場合、少なくとも1つの一致するキーを含む項目に加え、キーをまったく含まない項目が合致候補とみなされます。（このモードは例えば何のサブセットかを求める演算子を実装する際に有用です。）*searchModeがGIN_SEARCH_MODE_ALLに設定された場合、返されるキーのいずれかに一致するかどうかは関係なく、インデックス内の非NULLの項目すべてが合致候補とみなされます。（このモードは、基本的にインデックス全体のスキャン処理が必要ですので、他の2つの選択肢と比べてかなり低速になります。しかし境界条件を正確に実装するためにこれが必要になるかもしれません。おそらく、このモードを必要とする演算子はほとんどの場合、GIN演

算子クラス向けに優れた候補ではありません。) このモードを設定するために使用する記号はaccess/gin.hで定義されています。

pmatchは部分一致が提供されている場合に使用する出力引数です。使用するには、extractQueryが*nkeys個のboolの配列を割り当て、そのアドレスを*pmatchに格納しなければなりません。関連するキーが部分一致を必要とするとき、それぞれの配列要素は真に、そうでなければ偽に設定されなければなりません。*pmatchがNULLに設定されている場合、GINは部分一致が必要ないと想定します。呼び出し前に変数はNULLに初期化されますので、この引数は部分一致が提供されていない演算子クラスでは、単に無視できます。

extra_dataは、extractQueryがconsistentとcomparePartialメソッドに追加データを渡すことができるようにする出力引数です。使用するには、extractQueryが*nkeysポインタの配列を割り当て、そのアドレスを*extra_dataに格納し、そして望まれるものは何でも個別のポインタに格納しなければなりません。変数は呼び出し前にNULLに初期化されますので、追加データを必要としない演算子クラスでこの引数は単に無視できます。もし*extra_dataが設定されれば、配列全部がconsistentメソッドに、適切な要素がcomparePartialメソッドに渡されます。

演算子クラスは、インデックス付けされた項目が問い合わせに一致するか確認する関数も提供しなければなりません。それは2つの方法で行なわれます。2値のconsistent関数と3値のtriConsistent関数です。triConsistentが両方の機能を含みますので、triConsistentだけを提供しても十分です。しかし、2値の亜種を計算するのが著しく安価であれば、両方を提供することは役に立つかもしれません。2値の亜種のみが提供されていれば、すべてのキーを取得する前にインデックス項目が一致しないことを確認することに基づく最適化の中には無効となるものもあります。

```
bool consistent(bool check[], StrategyNumber n, Datum query, int32 nkeys, Pointer
extra_data[], bool *recheck, Datum queryKeys[], bool nullFlags[])
```

インデックス付けられた項目が戦略番号nを持つ問い合わせ演算子を満たす(または、recheck印が返されたときはたぶん満たすかもしれない)場合に真を返します。GINは項目を明示的に格納しませんので、この関数はインデックス付けされた項目の値に直接アクセスすることができません。どちらかというと、この問い合わせから取り出される指定された問い合わせで現れるキー値に関する知識が利用できるものです。check配列は長さnkeysであり、このqueryデータに対して事前に行われたextractQueryが返したキーの数と同じです。インデックス対象の項目が対応する問い合わせキーを持つ場合、check配列の各要素は真です。つまり、(check[i] == true)の場合、extractQueryの結果配列のi番目のキーがインデックス対象項目内に存在します。元のqueryデータは、consistentメソッドがそれを調査する必要がある場合に、渡されます。このためqueryKeys[]およびnullFlags[]は事前にextractQueryによって返されます。extra_dataはextractQueryにより返された追加データ配列で、ない場合はNULLです。

extractQueryがqueryKeys[]内でNULLキーを返す時、インデックス対象項目がNULLキーを含む場合は対応するcheck[]要素は真です、つまり、check[]の意味はIS NOT DISTINCT FROMのようなものです。consistent関数は、通常の値の合致とNULL合致との違いを通知する必要がある場合、対応するnullFlags[]要素を検査することができます。

成功の場合、*recheckはヒープタプルが問い合わせ演算子に対し再検査を必要とすれば真で、インデックス検査が的確であれば偽です。つまり、falseという戻り値はヒープタプルが問い合わせに合わないことを保証し、*recheckが付いたtrueという戻り値はヒープタプルが問い合わせに一致する可能性があるため、それを取り出し、元のインデックス付けされた項目を直接問い合わせ演算子で評価することで再検査する必要があることを意味します。

```
GinTernaryValue triConsistent(GinTernaryValue check[], StrategyNumber n, Datum query,
int32 nkeys, Pointer extra_data[], Datum queryKeys[], bool nullFlags[])
```

triConsistentはconsistentと似ていますが、checkベクターの論理値の代わりに、各キーに対して3つの可能な値があります。GIN_TRUE、GIN_FALSE、GIN_MAYBEです。GIN_FALSEとGIN_TRUEは通常の論理値と同じ意味であり、GIN_MAYBEはそのキーの存在が分からないこと意味します。GIN_MAYBE値があれば、インデックス項目が対応する問い合わせキーを含むかどうかに関わらず、項目が確実に一致する場合にのみ関数はGIN_TRUEを返すべきです。同様に、GIN_MAYBEを含むかどうかに関わらず項目が確実に一致しない場合にのみ関数はGIN_FALSEを返さなければなりません。結果がGIN_MAYBE項目に依存する、すなわち、分かっている問い合わせキーに基づいて、一致することもしないことも確認できない場合には、関数はGIN_MAYBEを返さなければなりません。

checkベクターにGIN_MAYBE値がなければ、GIN_MAYBE戻り値は論理値のconsistent関数でrecheckフラグを設定することと同じです。

さらに、GINにはインデックス内に格納されているキー値をソートする方法がなければなりません。演算子クラスは比較メソッドを指定することでソート順を定義できます。

```
int compare(Datum a, Datum b)
```

キー(インデックス付けされる項目ではありません)を比較し、0より小さい、0、または0より大きい整数を返します。それぞれ、最初のキーが2番目のキーより、小さい、等しい、または大きいことを示します。NULLキーがこの関数に渡されることはありません。

あるいは、演算子クラスがcompareメソッドを提供しない場合には、GINはそのインデックスキーデータ型に対するデフォルトのbtree演算子クラスを探し、その比較関数を使います。btree演算子クラスを探すのは処理に多少掛かりますので、GIN演算子クラスの中で比較関数を指定することを勧めます。それはただ一つのデータ型に対するものであることを意味します。しかし、(array_opsのような)多様GIN演算子クラスでは、通常は単一の比較関数を指定できません。

省略可能ですが、GINに対する演算子クラスは以下のメソッドを提供します。

```
int comparePartial(Datum partial_key, Datum key, StrategyNumber n, Pointer extra_data)
```

問い合わせキーとインデックスキーの部分一致を比較します。符号が結果を示す整数が返ります。ゼロ未満はインデックスキーは問い合わせに一致しないが、インデックススキャンを続けるべきであることを示します。ゼロはインデックスキーが問い合わせに一致することを示します。ゼロより大きな値はこれ以上の一致はありえないためインデックススキャンを停止すべきであることを示します。スキャンをいつ停止するかを決めるためにセマンティクスが必要とされる場合、部分一致問い合わせを生成した演算子の戦略番号nが提供されます。またextra_dataはextractQueryで作成される追加データ配列の対応する要素、もしなければNULLです。NULLキーがこの関数に渡されることはありません。

```
void options(local_relopts *relopts)
```

演算子クラスの振舞いを制御するユーザに可視のパラメータの集合を定義します。

options関数にはlocal_relopts構造体へのポインタが渡されますが、構造体を演算子クラスに固有のオプションの集合で満たすことが必要です。オプションはマクロPG_HAS_OPCLASS_OPTIONS()とPG_GET_OPCLASS_OPTIONS()を使って他のサポート関数からアクセスできます。

インデックス付けされた値からのキーの抽出にもGINでのキーの表現にも柔軟性がありますので、ユーザに固有のパラメータに依存するかもしれません。

「部分一致」問い合わせをサポートするためには、演算子クラスはcomparePartialメソッドを提供しなければなりません。またそのextractQueryは、部分一致問い合わせであった時にpmatchパラメータを設定しなければなりません。詳細については66.4.2を参照してください。

上記の各種Datum値の実データ型は、演算子クラスに依存して変動します。extractValueに渡される項目値は常に演算子クラスの入力型であり、キー値はすべてそのクラスのSTORAGE型でなければなりません。extractQuery、consistentおよびtriConsistentに渡されるquery引数の型は、戦略番号によって識別されるクラスのメンバ演算子の右辺入力型です。正しい型のキー値がそこから抽出できる限り、これはインデックス付けされた型と同じである必要はありません。しかしながら、この3つのサポート関数のSQL宣言では、実際の型は演算子に依存して何か他のものであるとしても、query引数には演算子クラスのインデックス付けされたデータ型を使うことを勧めます。

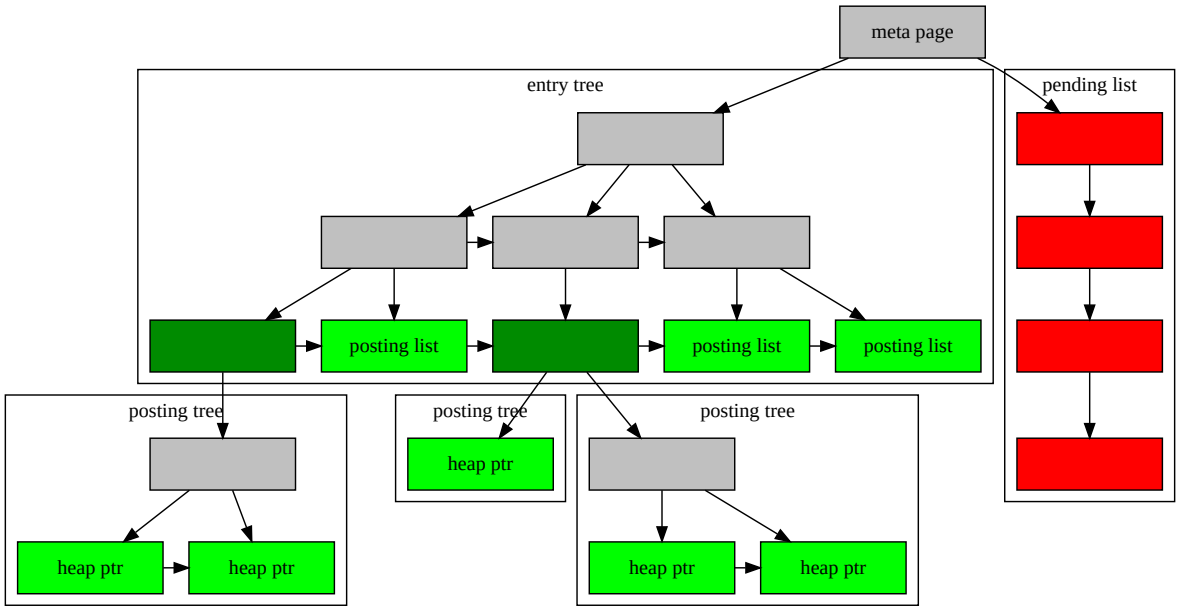
66.4. 実装

GINインデックスはキー全体に対するB-treeインデックスを持ちます。そのキーはそれぞれインデックス対象項目の要素（例えば配列のメンバ）であり、リーフページ内のタプルはそれぞれ、ヒープポインタのB-treeへのポインタ（「ポスティングツリー(posting tree)」）か、もしリストがキー値と共に単一インデックスタプルに合う程度十分に小さければヒープポインタの単純なリスト（「ポスティングリスト(posting list)」）です。図 66.1にGINインデックスのこれらのコンポーネントを示します。

PostgreSQL 9.1からNULLキー値をインデックスに含められるようになりました。またプレースホルダとしてのNULLが、NULLまたはextractValueによるとキーを含まないインデックス対象項目についてインデックスに含められます。これにより空の項目を見つけ出すための検索を行うことができます。

複数列に対するGINインデックスは複合型の値（列番号、キー値）全体について単一のB-treeを構築することで実装されます。異なる列に対するキー値は別の型となるかもしれません。

図66.1 GINの内部



66.4.1. GIN高速更新手法

1つのヒープ行の挿入または更新によりインデックスへの挿入が多く発生するという、転置インデックスの本質的な性質のためGINインデックスの更新は低速になりがちです。(各キー用のヒープ行はインデックス付けされた項目から取り出されます。) PostgreSQL 8.4からGINは、新しいタプルを一時的なソートされていない、待機中の項目リストに挿入することにより、この作業の大部分を遅延させることができるようになりました。テーブルがバキュームまたは自動解析された時、gin_clean_pending_list関数が呼ばれたとき、または、待機中のリスト(pending list)がgin_pending_list_limitよりも大きくなった時、初期のインデックス作成の際に使用されるものと同様の一括挿入技法を使用して、項目は主GINデータ構造に移動されます。これは、バキュームのオーバーヘッドが追加されることを考慮したとしても、GINインデックスの更新速度を著しく向上します。さらに、フォアグラウンドの問い合わせ処理ではなくバックグラウンド処理でこのオーバーヘッド作業を実行することができます。

この手法の大きな欠点は、検索時に通常のインデックス検索に加え待機中の項目リストのスキャンを行わなければならない点です。このため、待機中の項目リストが大きくなると検索が顕著に遅くなります。他の欠点は、ほとんどの更新は高速ですが、待機中の項目リストが「大きくなりすぎる」きっかけとなった更新は即時の整理処理を招くことになり、他の更新に比べ大きく低速になります。自動バキュームを適切に使用することで、これらの両方の問題を最小化することができます。

一貫した応答時間が更新速度より重要な場合、GINインデックスに対するfastupdate格納パラメータを無効にすることにより、待機中の項目の使用を無効にすることができます。詳細はCREATE INDEXを参照してください。

66.4.2. 部分一致アルゴリズム

GINは「部分一致」問い合わせをサポートすることができます。この問い合わせは1つ以上のキーに正確に一致することは決定しませんが、キー値の合理的に狭い(compareサポートメソッドで決まるキーのソート順に従った)範囲内に一致する可能性があります。extractQueryは、正確に一致したキー値を返す代わりに、検索される範囲の下限となるキー値を返し、pmatchフラグを真に設定します。そしてキー範囲をcomparePartialメソッドを使用して検索します。comparePartialは一致するインデックスキーではゼロを、一致しないが検索すべき範囲内にあればゼロ未満の値を、インデックスキーが一致可能な範囲を超えた場合はゼロより大きな値を返さなければなりません。

66.5. GINの小技

作成と挿入

各項目に対して多くのキーが挿入される可能性がありますので、GINインデックスへの挿入は低速になることがあります。ですので、テーブルに対する大量の挿入では、GINインデックスを削除し、大量の挿入が終わった段階で再作成することを勧めます。

PostgreSQL 8.4から遅延インデックス作成が使用されるため、この勧告は必要性が薄れました。(66.4.1を参照してください。) しかし非常に大規模な更新では、インデックスの削除と再作成がまだ最善かもしれません。

maintenance_work_mem

GINインデックスの構築時間はmaintenance_work_memの設定に非常に敏感です。インデックス作成時に作業メモリをより少なく使用しようとはしません。

gin_pending_list_limit

fastupdateが有効な既存のGINインデックスに対して挿入を繰り返す間、待機中の項目リストがgin_pending_list_limitより大きくなると、システムはこのリストを整理します。観測される応答時間の変動を防ぐためには、待機中リストの整理をバックグラウンド(すなわち自動バキューム)で起きるようにすることが望まれます。フォアグラウンドでの整理処理は、gin_pending_list_limitを大きくすること、もしくは自動バキュームをより積極的に行うことで防ぐことができます。しかし、整理処理の閾値を大きくすることは、フォアグラウンドで整理処理が発生した時により長い時間がかかることを意味します。

gin_pending_list_limitは格納パラメータを変更することで個々のGINインデックスに対して上書きでき、それにより各GINインデックスが自身の整理閾値を持てます。例えば、頻繁に更新される可能性のあるGINインデックスの閾値のみを増やして、それ以外は減らすことができます。

gin_fuzzy_search_limit

GINインデックス開発の主な目的は、スケーラビリティが高い全文検索のサポートをPostgreSQLで作成することでした。全文検索の結果は非常に大規模な結果セットを返します。さらに、問い合わせが非常に高頻度な単語を持つ場合、こうした状況はよく発生しますが、大規模な結果セットは有用ですらありません。ディスクから大量のタプルを読み、ソートすることは長い時間がかかりますので、実運用レベルでは受け入れられません。(インデックス検索自体は非常に高速であることに注意してください。)

こうした問い合わせの実行を簡単に制御できるように、GINは返される行数に対して設定可能なソフト上限、gin_fuzzy_search_limit設定パラメータを持ちます。これはデフォルトでは0です(無制限を意味します)。非0の制限が設定された場合、返されるセットは結果セット全体からランダムに選んだサブセットになります。

「ソフト」は、問い合わせとシステムの乱数ジェネレータの品質に依存して、返される結果の実際の数が指定した上限より多少異なることを意味します。

経験上、数千(例えば5000から20000)の値がうまく動作します。

66.6. 制限

GINは、インデックス付け可能な演算子は厳密であると仮定します。これはextractValueはNULL項目値についてはまったく呼び出されない(代わりにインデックス項目のプレースホルダが自動的に生成される)こと、 および、extractQueryは問い合わせの値がNULLの場合に呼び出されない(代わりに問い合わせは不一致であるとみなされる)ことを意味します。しかし非NULLの複合型項目内または問い合わせ値内のNULLキー値はサポートされます。

66.7. 例

PostgreSQLのコア配布物は以前表 66.1に示したGIN演算子クラスを含みます。以下のcontribモジュールにもGIN演算子クラスが含まれています。

btree_gin

さまざまなデータ型に対するB-tree等価の機能

hstore

(キー、値)の組み合わせを格納するモジュール

intarray

int[]に対する高度サポート

pg_trgm

トライグラム一致を使用したテキスト類似度

第67章 BRINインデックス

67.1. はじめに

BRINは「ブロックレンジインデックス」(Block Range Index)の略です。BRINは、ある列がテーブル内の物理的な位置と自然な関係があるような、非常に大規模なテーブルのために設計されています。ブロックレンジ*(block range)*は、テーブル内で物理的に隣接するページのグループです。それぞれのブロックレンジに対して、ある種の要約情報がインデックス内に格納されます。たとえば、店舗の受注情報を格納するテーブルは、各々の受注時期を格納する日付列を持つでしょう。大抵の場合、より前の受注エントリは、テーブルのより前の方にあるでしょう。郵便番号を管理するテーブルでは、ある市に属する郵便番号が自然にグループ化されることになるでしょう。

BRINインデックスは、通常のビットマップインデックススキャンを通じて要求されるクエリに使用することができます。すなわち、インデックス内のレンジ要約情報が検索条件と一致すれば、BRINインデックスは、レンジ内の全タプルを返します。クエリエクゼキュータの役割は、検索条件を再チェックし、条件に合致しないタプルを捨てることです。つまり言い換えると、BRINインデックスには損失性があります。BRINインデックスは非常に小さいため、それに対するスキャンは順スキャンに比べると小さなオーバヘッドしか与えません。しかし、あらかじめ条件に合致しないと分かっているテーブルの多くの部分をスキャンすることを避けることができます。

BRINインデックスに格納される特定のデータと、そのインデックスが対応できる特定のクエリは、インデックスに対応する各々の列に与えられた演算子クラスに依存します。線形のソート順を持つデータ型は、ブロックレンジ内の最小値と最大値と格納する演算子クラスを持つことができます。たとえば、幾何データ型は、ブロックレンジ内のすべてのオブジェクトを含む外接矩形を持つことでしょう。

ブロックレンジの大きさは、ストレージパラメータpages_per_rangeでインデックス作成時に決定されます。インデックスエントリの数は、リレーションのページ数をpages_per_rangeに設定した数で割ったものと等しくなります。ですから、pages_per_rangeの設定値が小さいほど、インデックスは大きくなります(より多くのインデックスエントリを格納する必要があるので)が、反面、格納されたサマリデータはより精密になり、インデックススキャンの際により多くのデータブロックをスキップすることができるようになります。

67.1.1. インデックスの保守

インデックスを作成した当初は、すべてのヒープページがスキャンされ、終端が不完全なものも含め、各々のレンジに対してサマリーインデックスタプルが作成されます。新しいページにデータが登録されると、新しいタプルのデータを元に、すでにサマリ済みのページレンジのサマリー情報が更新されます。最終サマリーレンジに適合しない新しいページが作成されると、そのレンジに対して自動的にはサマリタプルが作成されません。これらのタプルは、後でサマリー処理が走って初期サマリー情報が作成されるまではサマリーされません。この処理はbrin_summarize_range(regclass, bigint)またはbrin_summarize_new_values(regclass)関数を起動することで、手動で実行できます。あるいは、VACUUMがそのテーブルを処理する際に自動的に起動でき、また、行挿入があった際にautovacuumによって自動サマリ処理が実行されるときにも起動できます。(最後の自動処理起動はデフォルトでは無効になっていますが、autosummarizeパラメータで有効にできます。) 反対に、レンジはbrin_desummarize_range(regclass, bigint)関数で非サマリー化できます。これは、既存の値が変更されたためにインデックスタプルがもはや値の表現としては適当でなくなった場合に有効です。

自動サマリー機能が有効な場合、ページレンジが満たされる度に、そのレンジをターゲットとするサマリー機能を実行する要求が自動バキュームに送信されます。そしてそのリクエストは、同じデータベースに対してワーカーが次に走る際に、最後のところで実行されます。もしリクエストキューが満杯ならばそのリクエストは記録されず、次のメッセージがサーバのログに送信されます。

```
LOG:  request for BRIN range summarization for index "brin_wi_idx" page 128 was not recorded
```

この状態が発生すると、そのレンジはテーブルの次の通常バキュームで正常にサマライズされます。

67.2. 組み込み演算子クラス

PostgreSQLのコア配布物には、表 67.1で示されるBRIN演算子クラスが含まれます。

*minmax*演算子クラスは、インデックスが貼られた列の範囲内に現れる最小値と最大値を格納します。*inclusion*演算子クラスは、インデックスが貼られた列の範囲内に含まれる値を格納します。

表67.1 組み込みBRIN演算子クラス

名前	インデックスされるデータ型	インデックス可能な演算子
int8_minmax_ops	bigint	< <= = >= >
bit_minmax_ops	bit	< <= = >= >
varbit_minmax_ops	bit varying	< <= = >= >
box_inclusion_ops	box	<< &< && &> >> ~= @> <@ &<| <<| |>> |&>
bytea_minmax_ops	bytea	< <= = >= >
bpchar_minmax_ops	character	< <= = >= >
char_minmax_ops	"char"	< <= = >= >
date_minmax_ops	date	< <= = >= >
float8_minmax_ops	double precision	< <= = >= >
inet_minmax_ops	inet	< <= = >= >
network_inclusion_ops	inet	&& >>= <<= = >> <<
int4_minmax_ops	integer	< <= = >= >
interval_minmax_ops	interval	< <= = >= >
macaddr_minmax_ops	macaddr	< <= = >= >
macaddr8_minmax_ops	macaddr8	< <= = >= >
name_minmax_ops	name	< <= = >= >
numeric_minmax_ops	numeric	< <= = >= >
pg_lsn_minmax_ops	pg_lsn	< <= = >= >
oid_minmax_ops	oid	< <= = >= >

名前	インデックスされるデータ型	インデックス可能な演算子
range_inclusion_ops	any range type	<< &< && &> >> @> <@ -|- = < <= = > >=
float4_minmax_ops	real	< <= = >= >
int2_minmax_ops	smallint	< <= = >= >
text_minmax_ops	text	< <= = >= >
tid_minmax_ops	tid	< <= = >= >
timestamp_minmax_ops	timestamp without time zone	< <= = >= >
timestamptz_minmax_ops	timestamp with time zone	< <= = >= >
time_minmax_ops	time without time zone	< <= = >= >
timetz_minmax_ops	time with time zone	< <= = >= >
uuid_minmax_ops	uuid	< <= = >= >

67.3. 拡張性

BRINのインタフェースは高度に抽象化されており、アクセスメソッドを実装する人は、アクセスされるデータ型のセマンティクスを実装するだけで良いようになっています。BRIN層は、同時実行性、ログ、インデックス構造の検索を担当します。

BRINアクセスメソッドを動作させるために必要なのは、インデックスに格納された要約値の振る舞いと、それらがインデックススキャンする際にどう関係するのかを定義する少数のメソッドを実装することだけです。つまり、BRINは一般性、コードの再利用性、整理されたインタフェースと拡張性を同時に実現しています。

BRIN用の演算子クラスは、4つのメソッドを提供する必要があります。

`BrinOpcInfo *opcInfo(Oid type_oid)`

インデックスが貼られた列の要約データに関する内部情報を返します。返却値はpallocされたBrinOpcInfoへのポインタでなければなりません。BrinOpcInfoは以下の定義を持ちます。

```
typedef struct BrinOpcInfo
{
    /* Number of columns stored in an index column of this opclass */
    uint16      oi_nstored;

    /* Opaque pointer for the opclass' private use */
    void       *oi_opaque;

    /* Type cache entries of the stored columns */
    TypeCacheEntry *oi_typcache[FLEXIBLE_ARRAY_MEMBER];
} BrinOpcInfo;
```

BrinOpcInfo.oi_opaqueは、演算子クラスのルーチンが、インデックススキャン中にサポート関数同士で情報のやり取りをするために使うことができます。

```
bool consistent(BrinDesc *bdesc, BrinValues *column, ScanKey key)
```

ScanKeyがある範囲のインデックス値と一致しているかどうかを返します。属性の数はスキャンキーの一部として渡されます。

```
bool addValue(BrinDesc *bdesc, BrinValues *column, Datum newval, bool isnull)
```

追加された新しい値をインデックスが表現できるように、与えられたインデックスタプルとインデックス値にしたがい、タプルの指定アトリビュートを変更します。タプルの更新が行われれば、trueが返却されます。

```
bool unionTuples(BrinDesc *bdesc, BrinValues *a, BrinValues *b)
```

2つのインデックスタプルを統合します。与えられた2つのインデックスタプルのうち、最初のインデックスタプルを変更して、両方のタプルを表現できるようにします。2番目のタプルは変更されません。

省略可能ですが、BRINに対する演算子クラスは以下のメソッドを指定できます。

```
void options(local_relopts *relopts)
```

演算子クラスの振舞いを制御するユーザに可視のパラメータの集合を定義します。

options関数には`local_relopts`構造体へのポインタが渡されますが、構造体を演算子クラスに固有のオプションの集合で満たすことが必要です。オプションはマクロPG_HAS_OPCLASS_OPTIONS()とPG_GET_OPCLASS_OPTIONS()を使って他のサポート関数からアクセスできます。

インデックス付けされた値からのキーの抽出にもBRINでのキーの表現にも柔軟性がありますので、ユーザに固有のパラメータに依存するかもしれません。

コア配布物には、2種類の演算子クラスが含まれます。すなわち、minmaxとinclusionです。それらを使った演算子クラスの定義がコア配布物に必要に応じて含まれます。同じ定義を使って、ユーザは他のデータ型のために演算子クラスを定義することができます。そのためにソースコードを書く必要はありません。適切なシステムカタログの定義があれば十分です。演算子ストラテジのセマンティクスは、サポート関数のソースコード中に埋め込まれていることに注意してください。

前述の4つの主要なサポート関数を実装することにより、まったく異なるセマンティクスを実装する演算子クラスも可能です。なお、メジャーリリース間では下位互換性は保証されていません。たとえば、新しいリリースでは、サポート関数が追加で必要になるかもしれません。

表 67.2で示すように、全順序集合を実装するデータ型のための演算子クラスを書くために、関連する演算子とともにminmaxサポート関数を使うことができます。演算子クラスのメンバー(関数と演算子)はすべて必須です。

表67.2 Minmax演算子クラスの関数とサポート番号

演算子クラスメンバー	オブジェクト
Support Function 1	internal function `brin_minmax_opcinfo()`

演算子クラスメンバー	オブジェクト
Support Function 2	internal function `brin_minmax_add_value()`
Support Function 3	internal function `brin_minmax_consistent()`
Support Function 4	internal function `brin_minmax_union()`
Operator Strategy 1	operator less-than
Operator Strategy 2	operator less-than-or-equal-to
Operator Strategy 3	operator equal-to
Operator Strategy 4	operator greater-than-or-equal-to
Operator Strategy 5	operator greater-than

表 67.3で示すように、他のデータ型の値を含む複合データ型の演算子クラスを書くには、関連する演算子とともに、inclusionサポート関数を使うことができます。任意の言語で書かれたたったひとつの関数を追加するだけです。機能を追加するために関数を追加できます。すべての演算子はオプションです。表の中依存性の項目で示されているように、ある種の演算子は他の演算子を必要とすることもあります。

表67.3 Inclusion演算子クラスの関数とサポート番号

演算子クラスメンバー	オブジェクト	依存性
Support Function 1	internal function `brin_inclusion_opcinfo()`	
Support Function 2	internal function `brin_inclusion_add_value()`	
Support Function 3	internal function `brin_inclusion_consistent()`	
Support Function 4	internal function `brin_inclusion_union()`	
Support Function 11	function to merge two elements	
Support Function 12	optional function to check whether two elements are mergeable	
Support Function 13	optional function to check if an element is contained within another	
Support Function 14	optional function to check whether an element is empty	
Operator Strategy 1	operator left-of	Operator Strategy 4
Operator Strategy 2	operator does-not-extend-to-the-right-of	Operator Strategy 5
Operator Strategy 3	operator overlaps	
Operator Strategy 4	operator does-not-extend-to-the-left-of	Operator Strategy 1
Operator Strategy 5	operator right-of	Operator Strategy 2
Operator Strategy 6, 18	operator same-as-or-equal-to	Operator Strategy 7
Operator Strategy 7, 13, 16, 24, 25	operator contains-or-equal-to	
Operator Strategy 8, 14, 26, 27	operator is-contained-by-or-equal-to	Operator Strategy 3
Operator Strategy 9	operator does-not-extend-above	Operator Strategy 11

演算子クラスメンバー	オブジェクト	依存性
Operator Strategy 10	operator is-below	Operator Strategy 12
Operator Strategy 11	operator is-above	Operator Strategy 9
Operator Strategy 12	operator does-not-extend-below	Operator Strategy 10
Operator Strategy 20	operator less-than	Operator Strategy 5
Operator Strategy 21	operator less-than-or-equal-to	Operator Strategy 5
Operator Strategy 22	operator greater-than	Operator Strategy 1
Operator Strategy 23	operator greater-than-or-equal-to	Operator Strategy 1

サポート関数番号1から10は、BRINの内部関数用に予約されており、SQLレベルの関数は番号11から始まります。サポート関数11は、インデックスを構築するのに必要なメイン関数です。その関数は演算子クラスと同じデータ型を持つ2つの引数を受け取り、それらの和を返します。もしSTORAGEパラメータで定義されていれば、inclusion 演算子クラスは異なるデータ型の和を格納できます。和関数の返り値は、STORAGEデータ型と一致していなければなりません。

サポート関数番号12と14は、組み込みデータ型の例外事象をサポートするために提供されます。サポート関数番号12は、マージできない異なるファミリーのネットワークアドレスをサポートするために使用されます。サポート関数番号14は、空のレンジをサポートするために使用されます。サポート関数番号13はオプションですが、和関数に渡される前に新しい値のチェックを行うためのものとして推奨されます。BRINフレームワークは和が変化しない時に操作を省略することができるため、この関数を使うことによってインデックスの性能が向上する可能性があります。

minmaxとinclusion演算子クラスは、データ型をまたがる演算子をサポートします。しかし、これらを使用すると依存関係はより複雑になります。minmax演算子クラスは、両方の引数がデータ型が同じ型である完全な演算子のセットが必要になります。追加の演算子の組を定義することにより、追加のデータ型をサポートすることができます。表 67.3で示すように、inclusion演算子クラスのストラテジは、他の演算子クラスのストラテジに依存するか、自分自身の演算子クラスのストラテジに依存します。演算子クラスは、依存演算子がSTORAGEデータ型とともにサポートするデータ型の左辺引数、他のサポートするデータ型をサポートする演算子の右辺引数として定義される必要があります。minmaxの例として`float4_minmax_ops`、inclusionの例として`box_inclusion_ops`を参照してください。

第68章 データベースの物理的な格納

本章ではPostgreSQLデータベースで使用される物理的格納書式についての概要を説明します。

68.1. データベースファイルのレイアウト

本節ではファイルとディレクトリというレベルで格納書式について説明します。

伝統的に、データベースクラスタで利用される制御ファイルとデータファイルは、クラスタのデータディレクトリ内に一緒に格納され、通常（このディレクトリを定義するために使用できる環境変数名にちなんで）PGDATAとして参照されます。通常のPGDATAの位置は/var/lib/pgsql/dataです。異なるサーバインスタンスによって管理することで、複数のクラスタを同一のマシン上に存在させることができます。

表 68.1に示すように、PGDATAディレクトリには数個のサブディレクトリと制御ファイルがあります。これら必要な項目に加え、クラスタの設定ファイルであるpostgresql.conf、pg_hba.confおよびpg_ident.confが、他の場所にも置くことができますが、伝統的にPGDATA内に格納されます

表68.1 PGDATAの内容

項目	説明
PG_VERSION	PostgreSQLの主バージョン番号を保有するファイル
base	データベースごとのサブディレクトリを保有するサブディレクトリ
current_logfiles	ログ収集機構が現在書き込んでいるログファイルを記録するファイル
global	pg_databaseのようなクラスタで共有するテーブルを保有するサブディレクトリ
pg_commit_ts	トランザクションのコミット時刻のデータを保有するサブディレクトリ
pg_dynshmem	動的共有メモリサブシステムで使われるファイルを保有するサブディレクトリ
pg_logical	論理デコードのための状態データを保有するサブディレクトリ
pg_multixact	マルチトランザクションの状態のデータを保有するサブディレクトリ（共有行ロックで使用されます）
pg_notify	LISTEN/NOTIFY状態データを保有するサブディレクトリ
pg_replslot	レプリケーションスロットデータを保有するサブディレクトリ
pg_serial	コミットされたシリアライザブルトランザクションに関する情報を保有するサブディレクトリ
pg_snapshots	エクスポートされたスナップショットを保有するサブディレクトリ
pg_stat	統計サブシステム用の永続ファイルを保有するサブディレクトリ
pg_stat_tmp	統計サブシステム用の一時ファイルを保有するサブディレクトリ
pg_subtrans	サブトランザクションの状態のデータを保有するサブディレクトリ

項目	説明
pg_tblspc	テーブル空間へのシンボリックリンクを保有するサブディレクトリ
pg_twophase	プリペアドトランザクション用の状態ファイルを保有するサブディレクトリ
pg_wal	WAL(ログ先行書き込み)ファイルを保有するサブディレクトリ
pg_xact	トランザクションのコミット状態のデータを保有するサブディレクトリ
postgresql.auto.conf	ALTER SYSTEMにより設定された設定パラメータを格納するのに使われるファイル
postmaster.opts	最後にサーバを起動した時のコマンドラインオプションを記録するファイル
postmaster.pid	現在のpostmasterプロセスID(PID)、クラスタのデータディレクトリパス、postmaster起動時のタイムスタンプ、ポート番号、Unixドメインソケットのディレクトリパス(空も可)、有効な監視アドレスの一番目(IPアドレスまたは*、TCPを監視していない場合は空)および共有メモリのセグメントIDを記録するロックファイル(サーバが停止した後は存在しません)

クラスタ内の各データベースに対して、PGDATA/base内にサブディレクトリが存在し、サブディレクトリ名はpg_database内のデータベースOIDとなります。このサブディレクトリはデータベースファイルのデフォルトの位置であり、特にシステムカタログがそこに格納されます。

以下の節では、組み込みのheapテーブルアクセスメソッドと組み込みのインデックスアクセスメソッドの振舞いを説明していることに注意してください。PostgreSQLの拡張性のため、他のアクセスメソッドは異なる動作をするかもしれません。

各テーブルおよびインデックスは別個のファイルに格納されます。通常のリレーションでは、これらのファイル名はテーブルまたはインデックスのファイルノード番号となります。ファイルノード番号はpg_class.relfilenode内で見つけられます。しかし一時的なリレーションでは、ファイル名はtBBB_FFFという形になります。ここでBBBはファイルを生成したバックエンドのバックエンドID、FFFはファイルノード番号です。どちらの場合でも、主ファイル(いわゆる主フォーク)に加え、それぞれのテーブルとインデックスはリレーションに利用できる空き領域についての情報を格納する空き領域マップ(68.3参照)を持ちます。空き領域マップはファイルノード番号に接尾辞_fsmがついた名前のファイルに格納されます。テーブルは同時に、どのページが無効なタプルを持っていないと判断できるように追跡する可視性マップを持ち、フォークに接尾辞_vmを付けたファイルに格納します。可視性マップは68.4でより詳しく解説します。ログを取らないテーブルとインデックスは、初期化フォークという第3のフォークを持ち、フォークに接尾辞_initを付けたファイルに格納します(68.5参照)。

> **注意**
>
> テーブルにおけるファイルノード番号とOIDは多くの場合一致しますが、常に一致するとは限らないことに注意してください。TRUNCATE、REINDEX、CLUSTER等のいくつかの操作、およびALTER TABLEにおけるいくつかの構文は、OIDを保持したままファイルノード番号を変更できます。ファイルノード番号とテーブルOIDが同一であると仮定しないでください。またpg_class自身を含む特定のシステムカタログにおいて、pg_class.relfilenodeはゼロを持ちます。これらのカタログの実際のファイルノード番

> 号は低レベルなデータ構造内に保管されており、pg_relation_filenode()関数を使用して入手できます。

テーブルまたはインデックスが1ギガバイトを超えると、ギガバイト単位の*セグメント*に分割されます。最初のセグメントのファイル名はファイルノード番号と同一であり、それ以降は、ファイルノード番号.1、ファイルノード番号.2等の名称になります。この配置法によってファイル容量に制限のあるプラットフォームにおける問題を回避します。(実際、1ギガバイトは単なるデフォルトのセグメント容量です。セグメント容量はPostgreSQLを構築する際、--with-segsize設定オプションを使用して調整することができます。) 原理上、空き領域マップと可視性マップのフォークにおいても複数のセグメントも必要とする可能性がありますが、実際のところは起こりそうにありません。

項目が大きくなりそうな列を持ったテーブルは、連携した*TOAST*テーブルを保有する可能性があります。TOASTテーブルは、テーブル行の中には大き過ぎて適切に保持できないフィールド値を行の外部に格納するために使用されます。TOASTテーブルが存在する時、pg_class.reltoastrelidは元のテーブルとTOASTテーブルを結びつけます。68.2を参照してください。

テーブルおよびインデックスの内容は、68.6においてさらに考察されています。

テーブル空間は状況をさらに複雑にします。ユーザが定義したテーブル空間はそれぞれ、PGDATA/pg_tblspcディレクトリ内に物理的なテーブル空間ディレクトリ(つまりそのテーブル空間のCREATE TABLESPACEコマンドで指定された場所)を指し示す、シンボリックリンクを持ちます。シンボリックリンクの名称はテーブル空間のOIDとなります。物理的テーブル空間ディレクトリの内部では、PG_9.0_201008051などのPostgreSQLサーバのバージョンに依存した名前のサブディレクトリが存在します。(このサブディレクトリを使用する理由は、競合することなくCREATE TABLESPACEで指定する場所と同じものを将来のバージョンのデータベースでも使用できるようにするためです。) このバージョン固有のサブディレクトリの内部では、テーブル空間に要素を持つデータベースごとに、データベースOIDをディレクトリ名としたサブディレクトリが存在します。テーブルとインデックスは、ファイルノードの命名の規定に従って、そのディレクトリ内に格納されます。pg_defaultテーブル空間は pg_tblspcを通してアクセスされるのではなく、PGDATA/baseと連携します。同様に、pg_globalテーブル空間はpg_tblspcを通してアクセスされるのではなく、PGDATA/globalと連携します。

pg_relation_filepath()関数は任意のリレーションの(PGDATAから相対的な)パス全体を示します。これは上の規則の多くを記憶する必要がありませんので、しばしば有用です。しかし、この関数がリレーションの主フォークの最初のセグメントの名前だけを返すことに注意して下さい。リレーションに関したすべてのファイルを見つけるためにセグメント番号や_fsmや_vm、_initを追加する必要があるかもしれません。

一時ファイル(メモリ内に収まりきらないデータのソートなどの操作用)はPGDATA/base/pgsql_tmp内、または、pg_default以外のテーブル空間が指定されていた場合はテーブル空間ディレクトリ下のpgsql_tmpサブディレクトリ内に作成されます。一時ファイルの名前はpgsql_tmpPPP.NNNという形式です。ここで、PPPは所有するバックエンドのPIDであり、NNNで同一バックエンドで作成された別の一時ファイルと区別します。

68.2. TOAST

本節ではTOAST(過大属性格納技法:The Oversized-Attribute Storage Technique)の概要について説明します。

PostgreSQLは固定長のページサイズ（通常8キロバイト）を使用し、複数ページにまたがるタプルを許しません。そのため、大規模なフィールド値を直接格納できません。この限界を克服するため、大規模なフィールド値を圧縮したり、複数の物理的な行に分割したりしています。これはユーザからは透過的に発生し、また、バックエンドのコード全体には小さな影響しか与えません。この技法はTOAST（または「パンをスライスして以来最善のもの」）という愛称で呼ばれます。[訳注：TOASTはパンのトーストと綴りが同じなので、スライスしたパンを美味しく食べる方法に掛けて洒落ています。] TOASTの基盤は大きなデータ値のインメモリで処理の改善にも使用されています。

一部のデータ型のみがTOASTをサポートします。大規模なフィールド値を生成することがないデータ型にオーバーヘッドを負わせる必要はありません。TOASTをサポートするためには、データ型は可変長（*varlena*）表現を持たなければなりません。通常は、格納する値の最初の4バイトワードには値の長さ（このワード自体を含む）がバイト単位で含まれます。TOASTは残りのデータ型の表現について制限しません。*TOAST化された値*として集合的に呼ばれる特別な表現は、この先頭の長さのワードを更新または再解釈することで動作します。したがって、TOAST可能なデータ型をサポートするC言語関数は、潜在的にTOAST化されている入力値の扱い方に注意しなければなりません。つまり、入力が*TOAST解除*されなければ、それは実際には4バイトの長さのワードと内容から構成されていないかもしれないのです。（通常これは、入力に対して何か作業をする前にPG_DETOAST_DATUMを呼び出すことで行われますが、もっと効率的な方法が可能な場合もあります詳しくは37.13.1を参照してください）。

TOASTはvarlenaの長さワードの2ビット(ビッグエンディアンのマシンでは上位ビット、リトルエンディアンのマシンでは下位ビット)を勝手に使用します。そのため、すべてのTOAST可能なデータ型の値の論理サイズは1ギガバイト（2^{30} - 1バイト）までになります。両ビットが0の場合、値はそのデータ型の普通のTOAST化されていない値となり、長さワードの残りのビットはデータの（長さワードを含む）総サイズ（バイト単位）となります。上位側または下位側のどちらか片方のビットが設定された場合、値は通常の4バイトのヘッダを持たず1バイトのヘッダを持ちます。また、そのバイトの残りビットはデータの（長さワードを含む）総サイズ（バイト単位）となります。この方式により、127バイトより短い値の効率的な格納をサポートする一方で、データ型が必要なら1GBにまで大きくなることを可能にしています。1バイトのヘッダを持つ値は特定の境界に整列されませんが、4バイトのヘッダを持つ値は少なくとも4バイト境界の上に整列されます。このように整列のためのパディングを省略することで、短い値と比べて重要な追加のスペース節約ができます。特殊な状況として、1バイトのヘッダの残りビットがすべて0（自身の長さを含む場合はありえません）の場合、その値は行外データへのポインタで、以下に述べるようにいくつかの可能性があります。そのような*TOASTポインタ*の型とサイズはデータの2番目のバイトに格納されるコードによって決定されます。最後に上位側または下位側のビットが0で隣のビットが設定されている場合、データの内容は圧縮され、使用前に伸長しなければなりません。この場合、4バイトの長さワードの残りビットは元データのサイズではなく圧縮したデータの総サイズになります。圧縮が行外データでも起こりえますが、varlenaヘッダには圧縮されているかどうかについての情報がないことに注意してください。その代わりTOASTポインタの内容にこの情報が含まれています。

前に触れたように、TOASTポインタデータにはいくつかの型があります。最も古くて一般的な型は*TOASTテーブル*に格納されている行外データへのポインタです。TOASTテーブルは、TOASTポインタデータ自体を含むテーブルとは別の、しかし関連付けられるテーブルです。これらの*ディスク上*のポインタデータは、ディスク上に格納されるタプルが、そのまま格納するには大きすぎる時に、TOAST管理コード（access/common/toast_internals.cにあります）によって作られます。更なる詳細は68.2.1に記述されています。あるいはTOASTポインタデータは、メモリ内のどこかにある行外データへのポインタのこともあります。そのようなデータは短命で、ディスク上に現れることは決してありませんが、大きなデータ値を複製し、余分な処理をするのを避けるために有用です。更なる詳細は68.2.2に記述されています。

行内あるいは行外の圧縮データで使用される圧縮技術は、LZ系の圧縮技術の1つで単純かつ非常に高速なものです。詳細はsrc/common/pg_lzcompress.cを参照してください。

68.2.1. 行外ディスク上のTOAST格納

テーブルの列に1つでもTOAST可能なものがあれば、そのテーブルには連携したTOASTテーブルがあり、そのOIDがテーブルのpg_class.reltoastrelidエントリに格納されます。ディスク上のTOAST化された値は以下で詳しく説明する通り、TOASTテーブルに保持されます。

行外の値は(圧縮される場合は圧縮後に)最大TOAST_MAX_CHUNK_SIZEバイトの塊に分割されます (デフォルトではこの値は4チャンク行が1ページに収まり、およそ2000バイトになるように選ばれます)。各塊は、 データを持つテーブルと連携するTOASTテーブル内に個別の行として格納されます。すべてのTOAST テーブルはchunk_id列(特定のTOAST化された値を識別するOID)、chunk_seq列(値の塊に対する連番)、 chunk_data(塊の実際のデータ)列を持ちます。chunk_idとchunk_seqに対する一意性インデックスは値の抽出を高速化します。したがって、行外のディスク上のTOAST化された値を示すポインタデータには、検索先となるTOASTテーブルのOIDと指定した値のOID(chunk_id)を格納しなければなりません。簡便性のために、ポインタデータには論理データサイズ(元々の非圧縮のデータ長)と物理的な格納サイズ(圧縮時には異なります)も格納されます。varlenaヘッダバイトに収納するためにディスク上のTOASTポインタデータの総サイズは、表現される値の実サイズに関係なく、18バイトになります。

TOAST管理のコードは、テーブル内に格納される値がTOAST_TUPLE_THRESHOLDバイト(通常2キロバイト)を超える時にのみ実行されます。TOASTコードは、行の値がTOAST_TUPLE_TARGETバイト(こちらも通常2キロバイト、調整可能)より小さくなるかそれ以上の縮小ができなくなるまで、フィールド値の圧縮や行外への移動を行います。更新操作中、変更されない値は通常そのまま残ります。行外の値を持つ行の更新では、行外の値の変更がなければTOASTするコストはかかりません。

TOAST管理のコードでは、ディスク上にTOAST可能な列を格納するために、以下の4つの異なる戦略を認識します。

- PLAINは圧縮や行外の格納を防止します。さらにvarlena型での単一バイトヘッダの使用を無効にします。これはTOAST化不可能のデータ型の列に対してのみ取り得る戦略です。
- EXTENDEDでは、圧縮と行外の格納を許します。これはほとんどのTOAST可能のデータ型のデフォルトです。圧縮がまず行われ、それでも行が大き過ぎるのであれば行外に格納します。
- EXTERNALは非圧縮の行外格納を許します。EXTERNALを使用すると、textとbytea列全体に対する部分文字列操作が高速化されます。こうした操作は非圧縮の行外の値から必要な部分を取り出す時に最適化されるためです (格納領域が増加するという欠点があります)。
- MAINは圧縮を許しますが、行外の格納はできません (実際にはこうした列についても行外の格納は行われます。しかし、他に行を縮小させページに合わせる方法がない場合の最後の手段としてのみです)。

TOAST可能なデータ型はそれぞれ、そのデータ型の列用のデフォルトの戦略を指定します。しかしALTER TABLE ... SET STORAGEを使用して、あるテーブル列の戦略を変更することができます。

TOAST_TUPLE_TARGETはALTER TABLE ... SET (toast_tuple_target = N)を使って各テーブルで調整できます。

この機構には、ページをまたがる行の値を許可するといった素直な手法に比べて多くの利点があります。通常問い合わせは比較的小さなキー値に対する比較で条件付けされるものと仮定すると、エクゼキュータの仕事のほとんどは主だった行の項目を使用して行われることになります。TOAST化属性の大規模な値は、(それが選択されている時)結果集合をクライアントに戻す時に引き出されるだけです。このため、主テーブ

ルは行外の格納を使用しない場合に比べて、かなり小さくなり、その行は共有バッファキャッシュにより合うようになります。ソート集合もまた小さくなり、ソートが完全にメモリ内で行われる頻度が高くなります。小規模な試験結果ですが、典型的なHTMLページとそのURLを持つテーブルでは、TOASTテーブルを含め、元々のデータサイズのおよそ半分で格納でき、さらに、主テーブルには全体のデータのおよそ10%のみ（URLと一部の小さなHTMLページ）が格納されました。すべてのHTMLページを7キロバイト程度に切り詰めたTOAST化されない比較用テーブルと比べ、実行時間に違いはありませんでした。

68.2.2. 行外インメモリのTOAST格納

TOASTポインタは、ディスク上にあるデータだけでなく、現在のサーバプロセスのメモリ内の場所を指すこともできます。そのようなポインタは明らかに短命ですが、それでも有用です。現在のところ、間接データへのポインタと、展開データへのポインタの2つのケースがあります。

間接TOASTポインタは、単にメモリ上のどこかに格納されている間接的でないvarlena値を指すだけです。このケースは元々は単なる概念実証として作られたのですが、現在はロジカルデコーディング時に1GBを越える物理的タプルを作成する可能性を防ぐために使用されています。（すべての行外フィールド値をタプルに持ってこようとすると、そうなるかもしれません。）このケースでは、ポインタデータの作成者はポインタが存在可能な限り参照データが存在し続けることに全責任を負うため、利用が限られ、またこれを支援するための基盤もありません。

展開TOASTポインタは、ディスク上の表現が計算目的にあまり適さない複雑なデータ型で有用です。例えばPostgreSQLの配列の標準varlena表現には、次元の情報、NULLの要素があればNULLのビットマップ、そしてすべての要素の値が順番どおりに含まれます。要素型自体が可変長だと、N番目の要素を探す唯一の方法は前にある要素のすべてをスキャンすることです。この表現は、そのサイズの小ささからディスク上の記録には適していますが、配列を使った計算では、すべての要素の開始位置が特定されている「展開」または「解体」された表現があるとずっと良いです。TOASTポインタの機構では、参照渡しのデータが、標準のvarlena値（ディスク上の表現）あるいはメモリ上のどこかにある展開表現を指すTOASTポインタを指すことを許すことで、このニーズに応えています。この展開表現の詳細はデータ型に依存しますが、標準ヘッダを持ち、`src/include/utils/expandeddatum.h`にある他のAPIの要求を満たす必要があります。データ型を処理するc言語の関数は、どちらかの表現を扱うことを選ぶことができます。展開表現を認識せず、入力データに単に`PG_DETOAST_DATUM`を適用するだけの関数は、自動的に伝統的なvarlena表現を受け取ります。従って、展開表現のサポートは徐々に、1回に1つの関数だけ導入することができます。

展開された値へのTOASTポインタは、さらに読み書きのポインタと読み取りのみのポインタに分類されます。指された先の表現はどちらでも同じですが、読み書きのポインタを受け取った関数は、そこにある参照値を変更できるのに対し、読み取りのみのポインタを受け取った関数では変更が許されないため、値を変更したバージョンを作りたければ、まずその複製を作る必要があります。この区別と、関連したいくつかの慣習により、問い合わせの実行時に展開された値を不必要に複製するのを避けることが可能になります。

すべてのタイプのインメモリのTOASTポインタについて、TOAST管理のコードはそのようなポインタデータが偶然、ディスクに保存されてしまうことが決して起こらないようにします。インメモリのTOASTポインタは保存される前に自動的に展開されて通常の行内のvarlena値になります。その後、含んでいるタプルが大きすぎるような時には、ディスク上のTOASTポインタに変換されることもあります。

68.3. 空き領域マップ

ヒープとハッシュインデックス以外のインデックスリレーションはそれぞれ、そのリレーション内で利用可能な領域を継続して追跡するために、空き領域マップ(FSM)を持ちます。これは、個々のリレーションのフォーク内の主リレーションデータに沿って、リレーションのファイルノード番号に_fsmという接尾辞を付けた名前のファイルに格納されます。例えばリレーションのファイルノードが12345であれば、FSMは主リレーションファイルと同じディレクトリ内の12345_fsmという名前のファイルに格納されます。

空き領域マップはFSMページのツリーとして編成されます。最下位レベルのFSMページはすべてのヒープ(またはインデックス)ページで利用可能な空き領域を、各ページ毎に1バイト使用して格納します。上位レベルは下位レベルからの情報を集約します。

各FSMページの内部はノード当たり1バイトを持つ配列内に格納されたバイナリツリーです。各リーフノードはヒープページ、または下位レベルのFSMページを表現します。各非リーフノード内には、子の値より大きな値が格納されます。したがってリーフノード内の最大値がルートに格納されます。

FSMがどのように構築されるか、そしてどのように更新、検索されるかに関する詳細はsrc/backend/storage/freespace/READMEを参照してください。pg_freespacemapモジュールを使用して、空き領域マップに格納された情報を調べることができます。

68.4. 可視性マップ

各ヒープリレーションは、どのページがすべての実行中のトランザクションから可視であることが分かっているタプルだけを含むかを追跡する、可視性マップ(VM)を持ちます。どのページが凍結状態のタプルだけを含むのかも追跡します。これは、リレーションのファイルノード番号に_vmという接尾辞を付与した名前の別のリレーションフォーク内に主リレーションデータと並行して格納されます。例えばリレーションのファイルノードが12345の場合、VMは主リレーションファイルと同じディレクトリ内の12345_vmというファイル内に格納されます。インデックスはVMを持たないことに注意してください。

可視性マップはヒープページ当たり2ビットを保持します。最初のビットがセットされていれば、ページはすべて可視であること、すなわち、そのページにはバキュームが必要なタプルをまったく含んでいないことを示しています。またこの情報は、インデックスタプルのみを使用して問い合わせに答えるためにインデックスオンリースキャンによっても使用されます。2番目のビットがセットされていれば、そのページのタプルはすべて凍結状態であることを意味します。これは、周回対策のバキュームすらそのページを再び訪れる必要はないことを意味します。

このマップは、ビットがセットされている時は常にこの条件が真であることを確実に把握できるという点で保守的ですが、ビットがセットされていない場合は、真かもしれませんし偽かもしれません。可視性マップのビットはバキュームによってのみで設定されます。しかしページに対する任意のデータ編集操作によってクリアされます。

pg_visibilityモジュールは可視性マップに入っている情報を確かめるのに使えます。

68.5. 初期化フォーク

ログを取らないテーブルと、ログを取らないテーブルに対するインデックスは、それぞれ初期化フォークを持ちます。初期化フォークとは適切な種類の空テーブルと空インデックスです。ログを取らないテーブルをク

ラッシュのために再度空にしなければならない場合、初期化フォークで主フォーク全体をコピーし、その他のフォークは消去されます(これらは必要に応じて自動的に再作成されます)。

68.6. データベースページのレイアウト

本節ではPostgreSQLのテーブルおよびインデックスで使われるページ書式の概略について説明します。[1] TOASTのテーブルとシーケンスは、通常のテーブルと同様に整形されています。

以下の説明では1*バイト*は8ビットからなることを前提としています。さらに、*アイテム*という単語は、ページに格納される個別のデータ値のことを指しています。テーブル内ではアイテムは行であり、インデックス内ではアイテムはインデックスのエントリです。

テーブルとインデックスはすべて、固定サイズ(通常8キロバイト。サーバのコンパイル時に異なるサイズを設定可能)の*ページ*の集まりとして格納されます。テーブルでは、すべてのページは論理上等価です。したがって、あるアイテム(行)はどのページにでも格納することができます。インデックスでは、初めのページは通常、制御用の情報を保持する*メタページ*として予約されます。また、インデックスではインデックスアクセスメソッドに依存した様々なページ種類があります。

表 68.2はページの全体的なレイアウトを示しています。各ページには5つの部分があります。

表68.2 ページレイアウト全体

アイテム	説明
PageHeaderData	長さは24バイト。空き領域ポインタを含む、ページについての一般情報です。
ItemIdData	実際のアイテムを指すアイテム識別子の配列です。各項目は(オフセットと長さの)ペアです。1アイテムにつき4バイトです。
空き領域	割り当てられていない空間です。新規のアイテム識別子はこの領域の先頭から、新規のアイテムは最後から割り当てられます。
アイテム	実際のアイテムそのものです。
特別な空間	インデックスアクセスメソッド特有のデータです。異なるメソッドは異なるデータを格納します。通常のテーブルでは空です。

それぞれのページの最初の24バイトはページヘッダ(PageHeaderData)から構成されています。その書式を表 68.3にて説明します。最初のフィールドは、このページに関連する最も最近のWAL項目を表しています。2番目のフィールドにはdata checksumsが有効な場合にページチェックサムが格納されています。次にフラグビットを含む2バイトのフィールドがあります。その後に2バイトの整数フィールドが3つ続きます(pd_lower、pd_upper、pd_special)。これらには、割り当てられていない空間の始まり、割り当てられていない空間の終わり、そして特別な空間の始まりのバイトオフセットが格納されています。ページヘッダの次の2バイトであるpd_pagesize_versionは、ページサイズとバージョン指示子の両方を格納します。PostgreSQL 8.3以降のバージョン番号は4、PostgreSQL 8.1と8.2のバージョン番号は3、PostgreSQL 8.0のバージョン

[1] 実際には、テーブルアクセスメソッドもインデックスアクセスメソッドも、このページ書式を使用する必要はありません。heapテーブルアクセスメソッドは常にこの書式を使用します。既存のすべてのインデックスメソッドも、この基本書式を使用しています。しかし、インデックスメタページに保持されるデータは通常、アイテムレイアウト規則に従っていません。

番号は2、PostgreSQL 7.3と7.4のバージョン番号は1です。それより前のリリースのバージョン番号は0です（ほとんどのバージョン間で基本的なページレイアウトやヘッダの書式は変更されていませんが、ヒープ行ヘッダのレイアウトが変更されました）。ページサイズは基本的に照合用としてのみ存在しています。同一インストレーションでの複数のページサイズはサポートされていません。最後のフィールドはそのページの切り詰めが有益かどうかを示すヒントです。これはページ上で切り詰められていないもっとも古いXMAXが追跡するものです。

表68.3 PageHeaderDataのレイアウト

フィールド	型	長さ	説明
pd_lsn	PageXLogRecPtr	8バイト	LSN: このページへの最終変更に対応するWALレコードの最後のバイトの次のバイト
pd_checksum	uint16	2バイト	ページチェックサム
pd_flags	uint16	2バイト	フラグビット
pd_lower	LocationIndex	2 バイト	空き領域の始まりに対するオフセット
pd_upper	LocationIndex	2バイト	空き領域の終わりに対するオフセット
pd_special	LocationIndex	2バイト	特別な空間の始まりに対するオフセット
pd_pagesize_version	uint16	2バイト	ページサイズおよびレイアウトのバージョン番号の情報
pd_prune_xid	TransactionId	4バイト	ページ上でもっとも古い切り詰められていないXMAX。存在しなければゼロ。

詳細情報についてはsrc/include/storage/bufpage.hを参照してください。

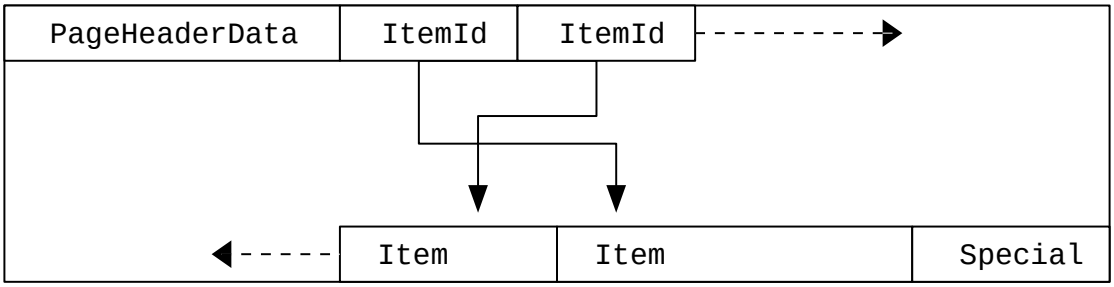
ページヘッダに続くのはアイテム識別子（ItemIdData）です。識別子ごとに4バイトを必要とします。アイテム識別子は、アイテムが開始されるバイトオフセット、バイト単位の長さ、そしてその解釈に影響する属性ビット群を持っています。新しいアイテム識別子は必要に応じて、未割当て空間の最初から割り当てられます。アイテム識別子の数は、新しい識別子を割り当てるために増加されるpd_lowerを見ることで決定できます。アイテム識別子は解放されるまで動かされることがないので、アイテム自体が空き領域をまとめるためにページ上で移動される場合でも、そのインデックスはアイテムを参照するために長期にわたって使うことができます。実際、PostgreSQLが作る、アイテムへのポインタ（ItemPointer、CTIDとも言います）はページ番号とアイテム識別子のインデックスによって構成されています。

アイテム自体は、未割り当て空間の最後から順番に割り当てられた空間に格納されます。正確な構造は、テーブルに何を含めるかによって異なります。テーブルとシーケンスの両方が、以下で説明するHeapTupleHeaderDataという構造を使用します。

最後のセクションは、アクセスメソッドが格納しようとするものを何でも含めることのできる「特別なセクション」です。例えば、B-treeインデックスは、そのページの両隣のページへのリンク、ならびに、インデックス構造体に関連したその他の何らかのデータを持ちます。通常のテーブルではこれはまったく使用されません（ページサイズを同じにするためにpd_specialを設定することで示されます）。

図 68.1は、これらの部分がページ内でどのようにレイアウトされているかを図解しています。

図68.1 ページレイアウト



68.6.1. テーブル行のレイアウト

テーブル行はすべて同じ方法で構成されています。固定サイズのヘッダ(ほとんどのマシンで23バイトを占有します)があり、その後にオプションのNULLビットマップ、オプションのオブジェクトIDフィールド、およびユーザデータが続きます。ヘッダについては表 68.4で説明します。実際のユーザデータ(行内の列)は、常にプラットフォームのMAXALIGN距離の倍数であるt_hoffで示されるオフセットから始まります。NULLビットマップは*HEAP_HASNULL*ビットがt_infomaskで設定されている場合にのみ存在します。存在する場合は、固定ヘッダのすぐ後ろから始まり、データ列ごとに1ビットとするのに十分なバイト数を占有します(すなわち、t_infomask2内の属性の個数と等しいビット数です)。このビットのリスト内では、1ビットは非NULLを、0ビットはNULLを示します。このビットマップが存在しない場合、すべての列が非NULLとみなされます。オブジェクトIDは*HEAP_HASOID_OLD*ビットがt_infomaskで設定されている場合にのみ存在します。存在する場合、これはt_hoff境界の直前に現れます。t_hoffをMAXALIGNの倍数とするために必要なパッドは全て、NULLビットマップとオブジェクトIDの間に現れます(このことにより、オブジェクトIDの位置揃えが確実に適切になります)。

表68.4 HeapTupleHeaderDataのレイアウト

フィールド	型	長さ	説明
t_xmin	TransactionId	4バイト	挿入XIDスタンプ
t_xmax	TransactionId	4バイト	削除XIDスタンプ
t_cid	CommandId	4バイト	挿入、削除の両方または片方のCIDスタンプ(t_xvacと共有)
t_xvac	TransactionId	4バイト	行バージョンを移すVACUUM操作用XID
t_ctid	ItemPointerData	6バイト	この行または最新バージョンの行の現在のTID
t_infomask2	uint16	2バイト	属性の数と各種フラグビット
t_infomask	uint16	2バイト	様々なフラグビット

フィールド	型	長さ	説明
t_hoff	uint8	1バイト	ユーザデータに対するオフセット

詳細情報についてはsrc/include/access/htup_details.hを参照してください。

実際のデータの解釈は、他のテーブル、ほとんどの場合、pg_attributeから取得された情報でのみ行うことができます。フィールド位置を識別するために必要なキー値は、attlenおよびattalignです。フィールドの幅が固定されていてNULL値が存在しない場合を除き、特定の属性を直接取得する方法はありません。この仕組みはすべて、*heap_getattr*、*fastgetattr*および*heap_getsysattr*関数にラップされています。

データを読むためには、それぞれの属性を順番に検査する必要があります。まず、NULLビットマップに従ってフィールドがNULLかどうかを検査します。もしNULLであれば、次に進みます。次に、位置揃えが正しいことを確認してください。フィールドの幅が固定されていれば、すべてのバイトが単純に配置されます。可変長のフィールド（attlen == -1）の場合はもう少し複雑です。可変長のデータ型はすべて、格納する値の長さといくつかのフラグビットを持つstruct varlenaという共通ヘッダ構造体を共有します。フラグによって、データは行内、または別のテーブル（TOAST）のいずれかとなったり、圧縮済みとなったりします（68.2を参照してください）。

第69章 システムカタログの宣言と初期内容

PostgreSQLは、テーブルや関数のようなデータベースオブジェクトの存在の有無と特性を追跡するために、多くの異なるシステムカタログを使用します。物理的な観点ではシステムカタログとユーザーテーブルの間に違いはありませんが、バックエンドのCコードはそれぞれのカタログの構造と特性を把握しており、直接カタログを低レベルで操作することができます。ですから、たとえばカタログの構造を思いつきで変更しようとするのはおすすめできません。そのことによって、Cのコードに組み込まれているカタログの行のレイアウトに関する前提を壊してしまうことになるからです。とはいえ、カタログの構造はメジャーバージョン間で変更されることがあります。

カタログの構造は、ソースツリーのsrc/include/catalog/ディレクトリの中の特殊な形式のCヘッダーファイルに宣言されています。とりわけ、個々のカタログに対応して、カタログと同じ名前のヘッダーファイルが存在し（たとえば、pg_classに対してpg_class.hというように）、カタログが持つ一連のカラムやOIDのような基本的な特性を定義しています。そのほか、カタログ構造を定義している重要なファイルとしては、すべてのシステムカタログに存在するインデックスの構造を定義するindexing.hや、カタログによっては必要としているTOASTテーブルの構造を定義するtoasting.hなどがあります。

SQLコマンドを実行可能な状態にまでシステムを持っていくために、多くのカタログはinitdbの「bootstrap」フェーズで読み込まなければならない初期データを持っています。（たとえば、pg_class.hは、他のシステムカタログとインデックス同様、自分自身のエントリを含んでいなければなりません。）この初期データも、src/include/catalog/ディレクトリに格納されているデータファイル中に編集可能な形式で保存されています。たとえば、pg_proc.datは、pg_procカタログに挿入されるべき初期の行を記述しています。

カタログファイルを作り初期データをそこにロードするために、ブートストラップモードで実行中のバックエンドは、コマンドと初期データを含むBKI（Backend Interface: バックエンドインタフェース）ファイルを読み込みます。このモードで使用されるpostgres.bkiは、genbki.plというPerlスクリプトを使って、PostgreSQLディストリビューションを構築する過程で前述のヘッダーとデータファイルから作成されます。postgres.bkiはPostgreSQLの特定のリリースに固有のものですが、プラットフォームからは独立しており、インストレーションツリーのshareサブディレクトリにインストールされます。

genbki.plは、他にも各々のカタログで使用する、たとえばpg_classのためのpg_class_d.hのような派生ファイルを生成します。このファイルには自動生成されたマクロ定義が含まれているほか、他のマクロとenum宣言も含まれており、特定のカタログを読み込むクライアントCコードに役立ちます。

ほとんどのPostgreSQL開発者は直接BKIファイルのことを気にかける必要はありませんが、バックエンドに些細ではない機能を追加する際にはカタログヘッダーファイル、あるいはまた初期データファイルの変更が必要になるでしょう。この章の残りの部分でそれについての情報をお届けします。また網羅性のために、BKIファイルのフォーマットも説明します。

69.1. システムカタログの宣言ルール

カタログヘッダーファイルの肝心な部分は、カタログにおける行の配置を記述するC構造体定義です。これはCATALOGマクロで始まりますが、Cコンパイラの観点からすると、単にtypedef struct

FormData_catalognameの短縮形です。構造体の各々のフィールドは、カタログのカラムを生成します。フィールドにはgenbki.hに記述されたBKIプロパティマクロを使って注釈を付けることができます。たとえば、フィールドのデフォルト値を定義したり、NULLが許されるかどうかのフラグを付けることができます。CATALOG行にも注釈が付けられます。genbki.hに記述されたBKIプロパティマクロを使って、共有リレーションであるかどうかといった、そのカタログ全体のプロパティを定義することができます。

システムカタログキャッシュのコード(そして一般的にたいていのカタログを触るコード)は、すべてのシステムカタログタプルに固定長部分が実際に存在するとみなします。システムカタログキャッシュのコードは、C構造体定義をその固定部分にマップするからです。したがって、すべての可変長フィールドと、NULLを許容するフィールドは、最後尾に置かれなければならず、また、構造体のフィールドとしてはアクセスできません。たとえば、pg_type.typrelidをNULLにしようとすると、他のコード部分がtypetup->typrelidを参照しようとして失敗します。(あるいはもっと悪いことにtypetup->typelemを参照中に失敗します。なぜなら、そのフィールドはtyprelidの後に来るからです。) これはランダムなエラーとなるか、あるいはセグメンテーション違反にすらなってしまいます。

この種のエラーから部分的に身を守るためには、可変長あるいはNULLを許容するフィールドはCコンパイラから直接見えないようにすべきです。これは#ifdef CATALOG_VARLEN ... #endifの中に入れることで達成できます。(ここで、CATALOG_VARLENは、決して定義されないシンボルです。) これにより、Cコードが不注意で存在しないフィールドにアクセスしようとしたり、オフセットが違うフィールドにアクセスしようとするのを防ぐことができます。不正な行を作るのを防ぐ独立したガードとして、NULLを許容しないすべてのカラムをpg_attributeでそのように宣言することを要求します。ブートストラップコードは、固定長でかつNULLを許容するカラムの次ではないすべてのカラムに対して自動的にNOT NULLのマークを付けます。このルールが不適切なら、BKI_FORCE_NOT_NULLとBKI_FORCE_NULLを必要に応じて使ってマーキングを修正できます。

フロントエンドのコードはすべてのpg_xxx.hカタログヘッダーファイルをincludeすべきではありません。バックエンド以外ではコンパイルできないCコードを含んでいるかもしれないからです。(典型的には、src/backend/catalog/ファイル中に関数宣言を含んでいることによって起こります。) その代わりに、フロントエンドは生成されたpg_xxx_d.hヘッダーをincludeできます。このファイルは、OIDの#defineと、クライアント側で必要になるデータを含んでいます。カタログヘッダー中のマクロやその他のコードをフロントエンドから見えるようにしたい場合は、#ifdef EXPOSE_TO_CLIENT_CODE ... #endifで該当セクションを囲むことにより、genbki.plがそのセクションをpg_xxx_d.hにコピーするように指示してください。

少数のカタログは非常に基本的なものなので、ほとんどのカタログで使用されるBKI createコマンドですら作成できません。そのコマンドが、新しいカタログの記述をこれらのカタログに書き込む必要があるからです。これらはブートストラップカタログと呼ばれ、定義するためには大量の追加の作業が必要です。pg_classとpg_typeのあらかじめロード済みの内容上に手動で適切なエントリを用意し、後のカタログ構造への変更に合わせてそれらのエントリを更新する必要があります。(また、ブートストラップカタログはpg_attribute中のロード済みのエントリを必要としますが、幸いにも最近はgenbki.plが適切に処理してくれます。) 可能ならば、新しいカタログをブートストラップカタログとして作るのは避けてください。

69.2. システムカタログ初期データ

手動で生成した初期データを持つ(いくつかのものは持っていません)各々のカタログには、編集可能なデータ形式の初期データを含み、対応する.datファイルがあります。

69.2.1. データファイル形式

個々の.datファイルにはPerlのデータ構造文字列が含まれます。それらは単に評価されることによって1個がカタログの1行に対応するハッシュ参照の配列を含むメモリ上のデータ構造を生成します。pg_database.datから抜きだしたものに些細な変更を加えたものを使って、鍵となる機能を示します。

```
[

# A comment could appear here.
{ oid => '1', oid_symbol => 'TemplateDbOid',
  descr => 'database\'s default template',
  datname => 'template1', encoding => 'ENCODING', datcollate => 'LC_COLLATE',
  datctype => 'LC_CTYPE', datistemplate => 't', datallowconn => 't',
  datconnlimit => '-1', datlastsysoid => '0', datfrozenxid => '0',
  datminmxid => '1', dattablespace => 'pg_default', datacl => '_null_' },

]
```

特筆すべきポイント:

- 全体的なファイルレイアウトは次のようになります。開き大括弧、カタログの行を表現する一つ以上の中括弧、閉じ大括弧。各々の閉じ中括弧の後にはカンマを書きます。

- 各々のカタログ行にカンマ区切りでkey=>valueペアを書きます。記述可能なkeyは、カタログのカラムに加えてメタデータキーであるoid、oid_symbol、array_type_oid、descrです。(oidとoid_symbolの使い方は後述の69.2.2で説明されていて、一方、array_type_oidは69.2.4で説明されています。descrはオブジェクトの説明文字列に使用し、pg_descriptionかpg_shdescriptionの適切な方に挿入されます。) メタデータキーは省略可能であるのに対し、カタログの.hファイルがカラムのデフォルト値を指定する場合を除いてカタログの定義済みカラムはすべて提供されなければなりません。(上記の例ではpg_database.hが適切なデフォルト値を供するのでdatdbaフィールドは省略されました。)

- すべての値は単一引用符で囲まなければなりません。値中の単一引用符はバックスラッシュでエスケープします。データを意味するバックスラッシュは二重にできますが、必須ではありません。これはPerlの単純な単一引用符で引用されたリテラルに関するルールに基づいています。データとして使われるバックスラッシュは、エスケープ文字列定数(4.1.2.2参照)と同じルールに基づき、ブートストラップスキャナーはエスケープと解釈することに注意してください。たとえば\tはタブへと変換されます。最終的な値としてバックスラッシュを使用したい場合は、4つ書く必要があります。Perlが2つ削除し、ブートストラップスキャナーが認識するために\\が残ります。

- NULL値は_null_で表します。(それと同じ文字列を作る方法はないことに注意してください。)

- コメントは#を前に置いてください。また同じ行上に置かなければなりません。

- 他のカタログエントリのOIDであるフィールド値は、実際の数値のOIDではなくシンボル名で記述されるべきです。(上記の例ではdattablespaceがこのような参照を含みます。) これは後述の69.2.3で説明します。

- ハッシュは順序付けられないデータ構造なので、フィールドの順や行の配置には重要な意味はありません。しかし、見た目を統一するために、フォーマットスクリプトreformat_dat_file.plが適用される少数のルールを設定しました。

- 中括弧のペアの中で、メタデータフィールドのoid、oid_symbol、array_type_oid、および、(もし存在するなら)descrがこの順で最初に来ます。そして、定義された順にカタログ自身のフィールドが現れます。
- 可能ならば、行の長さを80文字に制限するために、必要に応じてフィールドの間に改行を挿入します。改行はメタデータフィールドと通常のフィールドの間にも挿入します。
- カタログの.hファイルがカラムのデフォルト値を指定していて、データエントリが同じ値なら、reformat_dat_file.plはデータファイルからデータエントリを省略します。これでデータ表現が小さくなります。
- reformat_dat_file.plは空白行とコメント行をそのまま維持します。
カタログデータパッチを投稿する前に、reformat_dat_file.plを実行することをお勧めします。便利さのために、単にsrc/include/catalog/に変更を加えてmake reformat-dat-filesを実行することができます。

- データ表現をより小さくする新しい方法を付け加えたいのであれば、reformat_dat_file.plで実装し、また データを完全な表現に戻す方法をCatalog::ParseData()に指示しなければなりません。

69.2.2. OIDの割当

初期データに現れるカタログ行にはoid=> nnnnメタデータフィールドを書くことで手動で割り当てたOIDを与えることができます。それだけでなく、OIDを割り当てられたならば、oid_symbol => nameメタデータフィールドを書くことでそのOID用のCマクロを作ることができます。

他のプリロードカタログ行の中にそのOIDへの参照がある場合には、プリロードカタログ行は割当済みのOIDを持たなければなりません。Cコードから行OIDが参照されるときにも割当済みのOIDは必要です。どちらも当てはまらない場合は、oidメタデータフィールドは省略可能です。その場合、ブートストラップコードが自動的にOIDを割り当てます。実用的には、カタログの一部のみが実際に相互参照されている場合でも、与えられたプリロードカタログ行のOIDをすべて割当済みにするか、一つも割当済みにしないかのどちらかに通常します。

Cコード中でOIDの実際の数値を書くのは非常に良くないと考えられます。pg_procを直接参照するのは普通のことなので、自動的に必要なマクロを生成する特別な仕掛けがあります。src/backend/utils/Gen_fmgrtab.plを見てください。歴史的理由により、似ていはいますが同じではない方法によるpg_type OID用のマクロを自動生成する仕組みがあります。ですから、oid_symbolエントリはこれらの2つのカタログに必ずしも存在しなければならないというわけではありません。同様に、pg_classシステムカタログのOIDとインデックスマクロは自動的に設定されます。他のすべてのシステムカタログでは、oid_symbolを使って必要なマクロを手動で指定しなければなりません。

新しいプリロード行のために利用可能なOIDを見つけるには、src/include/catalog/unused_oidsスクリプトを実行してください。未使用のOIDの範囲が表示されます。(たとえば、出力行45-900はOIDs 45から900が利用されていないことを示します。) 今の所OID 1–9999 は手動での割当のために予約されています。unused_oidsスクリプトは、単にカタログヘッダーと.datを見てそこに出現していないOIDを探しているだけです。間違い見つけるためにduplicate_oidsを利用することもできます。(genbki.plは手動アサインされていない全ての行にOIDを割り当て、また、コンパイル時に重複OIDを検出します。)

即座にコミットされるとは期待できないパッチ用にOIDを選ぶときの最良の手法は、8000—9999の範囲でランダムに選択したところから始まるおおむね連続したOIDのグループを使うことです。これは同時に開発さ

れている他のパッチとのOID衝突の危険を最小化します。8000—9999の範囲を開発目的に空けておくため、パッチがマスタgitリポジトリにコミットされた後、そのOIDはこの範囲の下位の使用可能な場所に番号の振り直しをすべきです。通例これは各開発サイクルの終わり近くに行われ、同時にそのサイクルでコミットされたパッチで消費された全てのOIDを移動するでしょう。スクリプトrenumber_oids.plはこの目的に使用できます。コミットされていないパッチが最近コミットされたパッチとOID衝突していることに気づいた場合に、このような状況から回復するのにもrenumber_oids.plがおそらく役立つでしょう。

パッチに割り当てられたOIDを番号付け替えすることがあるこの慣習のため、パッチに割り当てられたOIDはそのパッチが正式リリースに含まれるまでは永続的と考えるべきではありません。さまざまな互換性の問題を生み出すかもしれないため、一度リリースされた手動でアサインされたオブジェクトのOIDは変更しません。

genbki.plは、手動アサインされたOIDを持たないカタログエントリにOIDを割り当てる必要がある場合、10000—11999範囲の値を使います。ブートストラップ実行開始の際にはサーバのOIDカウンタは12000に設定されます。したがって、information_schema.sqlスクリプトを実行して作られるオブジェクトなど、ブートストラップ後の段階において通常のSQLコマンドで作られたオブジェクトは12000以上のOIDを受け取ります。

通常のデータベース操作で割り当てられたOIDは16384以上に限定されます。これはgenbki.plやブートストラップの中で自動的に割り当てられたOIDに対して10000—16383の範囲が空いていること保証します。これらの自動割り当てされたOIDは永続的とはいえず、あるインストレーションから他のインストレーションで変更されるかもしれません。

69.2.3. OID参照検索

原則としては、ある初期カタログ行から他への相互参照は、参照しているフィールドで参照されている行の事前割り当てされたOIDを書くことだけで記述できます。しかしながら、間違いやすく、読みにくく、また、新たに割り当てられたOIDが番号付け直しされたときに破損しやすいため、これはプロジェクト方針に反します。そのため、genbki.plが代わりにシンボル参照を記述する機構を提供しています。そのルールは以下のとおりです。

- BKI_LOOKUP(lookuprule)をカラム定義に加えることで、特定のカタログカラムでのシンボル参照が利用可能になります。ここでlookuprule は参照されているカタログ名で、例えばpg_procです。BKI_LOOKUPを、Oid、regproc、oidvector、Oid[]のカラムに加えることができます。最後の2つにおいては、配列の個々の要素を検索することを暗に意味します。

- 文字セット符号化方式を参照する整数の列にBKI_LOOKUP(encoding)を加えることも許容されます。これは今のところカタログのOIDとして現れませんが、値の集合をgenbki.plに知らせます。

- そうしたカラムでは、InvalidOidの意味で0を用いる以外には、すべてのエントリでシンボル参照を使用しなければなりません。(カラムがregprocと宣言されている場合は、0の代わりに-と書くことができます。)genbki.plは認識できない名前には警告を出します。

- たいていのカタログオブジェクト類は単純にその名前で参照されます。型名は参照されているpg_typeのエントリのtypnameと正確に一致しなければならないことに注意してください。int4に対するintegerなどの別名は使えません。

- それがpg_proc.dat内でユニークなら、関数はpronameで表現できます。(regprocの入力はこのように働きます。) そうでなければ、regprocedureのように、proname(argtypename,argtypename,...)と書いてください。引数型名は正確にpg_proc.datエントリのproargtypesで指定しなければなりません。空白は挿入しないでください。

- 演算子はoprname(lefttype,righttype)で表現します。型名は正確にpg_operator.datエントリのoprleftフィールドとoprrightフィールドで記述します。(省略された単項演算子のオペランドは0と書きます。)
- opclassesとopfamiliesの名前はアクセスメソッド内でのみユニークなので、access_method_name/object_nameで表します。
- 以上のいずれの場合にもスキーマ修飾の規定はありません。ブートストラップ中に作成されるすべてのオブジェクトは、pg_catalogスキーマにあると期待されます。
- 一般的な検索の仕組みに加えて、PGNSPがpg_catalogスキーマのOIDで置換され、PGUIDがブートストラップのスーパーユーザロールのOIDで置換されるという特別な習慣があります。これらの使用方法はやや歴史的なものですが今のところ、これらを一般化する必要はありませんでした。

genbki.plは実行中にすべてのシンボル参照を解決し、生成したBKIファイルの中に単純な数字のOIDを設定します。ですから、ブートストラップバックエンドはシンボル参照にかかわる必要はありません。

69.2.4. 配列型の自動作成

たいていのスカラデータ型は対応する配列型を持つはずです(すなわち、要素がスカラ型の標準varlena配列型で、スカラ型のpg_typeエントリのtyparrayフィールドから参照されているもの)。genbki.plはたいていの場合に配列型に対するpg_typeエントリも自動的に生成できます。

この機能を使うには、スカラ型のpg_typeエントリにarray_type_oid => nnnnというメタデータフィールドを記述して、配列型に使用するOIDを指定します。自動的にそのOIDが書かれるため、このときtyparrayフィールドを省いてもよいです。

生成された配列型の名前は、スカラ型の名前の手前にアンダースコアを付けたものです。配列エントリの他のフィールドは、pg_type.hでBKI_ARRAY_DEFAULT(value)注釈から充当され、もし無ければスカラ型からコピーされます。(typalignに対する特別な場合もあります。) さらに両エントリのtypelemおよびtyparrayフィールドは相互参照するように設定されます。

69.2.5. データファイルの編集方法

カタログデータファイルを更新する共通の作業を実施するためのもっとも簡単な方法の提案を示します。

カタログにデフォルト付きの新しいカラムを追加する. BKI_DEFAULT(value)注釈付きでヘッダーファイルにカラムを追加します。非デフォルト値が必要な既存の行に対してのみフィールドを追加によるデータファイルの調整が必要です。

デフォルト値を持たない既存のカラムにデフォルト値を追加する. BKI_DEFAULT注釈をヘッダーファイルに追加し、冗長になったフィールドエントリを削除するためにmake reformat-dat-filesを実行します。

デフォルト値の有無にかかわらず、カラムを削除する. ヘッダーからカラムを削除し、make reformat-dat-filesを実行して不要になったフィールドエントリを削除します。

既存のデフォルト値を変更もしくは削除する. 現在のデータが正しく解釈されなくなるため、単にヘッダーファイルを変更することはできません。まずmake expand-dat-filesを実行し、すべてのデフォルト値が明

示的に挿入されるようにデータファイルを書き換えます。次にBKI_DEFAULT注釈を変更もしくは削除し、make reformat-dat-filesを実行して余分のフィールドを再び削除します。

特定の目的のための大量の編集: reformat_dat_file.plを使って色々な大量の変更を実施できます。一度限りのコードを挿入できることを示すブロックコメントを見つけます。次の例では、pg_proc中の2つの論理値型フィールドを一つの文字フィールドに統合します。

1. デフォルトがある新しいカラムをpg_proc.hに追加します。

```
+    /* see PROKIND_ categories below */
+    char        prokind BKI_DEFAULT(f);
```

2. 臨機応変に適当な値を挿入するために、reformat_dat_file.plを元に新しいスクリプトを作ります。

```
-           # At this point we have the full row in memory as a hash
-           # and can do any operations we want. As written, it only
-           # removes default values, but this script can be adapted to
-           # do one-off bulk-editing.
+           # One-off change to migrate to prokind
+           # Default has already been filled in by now, so change to other
+           # values as appropriate
+           if ($values{proisagg} eq 't')
+           {
+               $values{prokind} = 'a';
+           }
+           elsif ($values{proiswindow} eq 't')
+           {
+               $values{prokind} = 'w';
+           }
```

3. スクリプトを実行します。

```
$ cd src/include/catalog
$ perl  rewrite_dat_with_prokind.pl  pg_proc.dat
```

この時点でpg_proc.datにはprokind、proisagg、proiswindowのすべての3つのカラムがありますが、非デフォルト値を持つ行だけに表れます。

4. pg_proc.hから古いカラムを削除します。

```
-    /* is it an aggregate? */
-    bool        proisagg BKI_DEFAULT(f);
-
-    /* is it a window function? */
-    bool        proiswindow BKI_DEFAULT(f);
```

5. 最後に、make reformat-dat-filesを実行してpg_proc.datから不要になった古いエントリを削除します。

さらなる大量編集スクリプトの例については、https://www.postgresql.org/message-id/CAJVSVGVX8gXnPm+Xa=DxR7kFYprcQ1tNcCT5D0O3ShfnM6jehA@mail.gmail.comに付随するconvert_oid2name.plとremove_pg_type_oid_symbols.plを見てください。

69.3. BKIファイル形式

本節ではPostgreSQLのバックエンドがどのようにしてBKIファイルを解釈するのかを説明します。例としてpostgres.bkiファイルが手元にあると、説明が一層理解しやすくなるでしょう。

BKIの入力は一連のコマンドで構成されます。コマンドはいくつものトークンから構成されていて、コマンドの構文に依存しています。トークンは通常空白で分離されていますが、どちらとも解釈されるような曖昧性がなければ必要性ありません。特別なコマンド区切り文字はありません。したがって、構文上その前のコマンドに属すことができない次のトークンは新たなコマンドとなります（通常、わかりやすくするために、新しいコマンドは新しい行に記述します）。トークンはある一定のキーワードや特別な文字（括弧やカンマなど）、数字、二重引用符で囲まれた文字列などが使用できます。大文字/小文字は全て区別されます。

#で始まる行は無視されます。

69.4. BKIコマンド

create tablename tableoid [bootstrap] [shared_relation] [rowtype_oid oid] (name1 = type1 [FORCE NOT NULL | FORCE NULL] [, name2 = type2 [FORCE NOT NULL | FORCE NULL], ...])

括弧で与えられた列と、OID tableoidを持つtablenameというテーブルを作成します。

次の列型はbootstrap.cで直接サポートされます。bool、bytea、char（1バイト）、name、int2、int4、regproc、regclass、regtype、text、oid、tid、xid、cid、int2vector、oidvector、_int4（配列）、_text（配列）、_oid（配列）、_char（配列）、_aclitem（配列）。この他の型を持つテーブルを作成することはできますが、pg_typeが完了し適切な項目で埋められるまで完了させることができません。（これらの列型のみブートストラップカタログで使用されますが、非ブートストラップカタログは如何なる組み込み型も含む事があるという事を実際に意味しています。）

bootstrapが指定された場合、テーブルはディスク上に作成されるだけで、pg_classやpg_attributeなどにその項目は登録されません。したがって、これらの項目が（insertコマンドで）固定化されるまで、普通のSQL操作でこのテーブルにアクセスできません。このオプションはpg_classなど自身を作成するために使用されます。

shared_relationが指定された場合、テーブルは共有として作成されます。テーブルの行型OID（pg_type OID）はrowtype_oid句で指定できます。指定されなければ、OIDは自動的に生成されます。（bootstrap が指定されていれば、rowtype_oid句は役に立ちません。しかし、文書化のためにともかく指定はできます。）

open tablename

データを挿入するためにtablenameと名前が付けられたテーブルを開きます。現在開いているテーブルは閉じられます。

close tablename

開いているテーブルを閉じます。照合用にテーブル名を指定しなければなりません。

insert ([oid_value] value1 value2 ...)

value1やvalue2などを列の値として、開いているテーブルに行を挿入します。

NULL値は特別なキーワード、_null_によって指定できます。識別子に見えない値、あるいは数値文字列は二重引用符で囲まなければなりません。

declare [unique] index indexname indexoid on tablename using amname (opclass1 name1 [, ...])

amnameアクセスメソッドを使用して、tablenameと名付けられたテーブル上に、OID indexoidを所有する、indexnameという名前のインデックスを作成します。インデックスが付けられるフィールドは、name1、name2など、そして使用される演算子クラスはopclass1、opclass2などとそれぞれ呼ばれます。このインデックスファイルは作成され、適切なカタログ項目が作成されますが、このコマンドではインデックスの内容の初期化を行いません。

declare toast toasttableoid toastindexoid on tablename

tablenameという名前のテーブル用のTOASTテーブルを作成します。このTOASTテーブルはOIDとしてtoasttableoidが割り当てられ、そのインデックスはOIDとしてtoastindexoidが割り当てられます。declare indexと同様、インデックスの作成は遅延されます。

build indices

前に宣言されたインデックスを作成します。

69.5. BKIファイルのブートストラップの構成

openコマンドは、テーブルが、使用するテーブルが存在し、開かれるテーブルに対しエントリを所有するまで使用できません。(これら最小限度のテーブルは、pg_class、pg_attribute、pg_proc、およびpg_typeです。)これらのテーブル自体が充填されるようにするには、bootstrapオプションを伴ったcreateが明示的にデータの挿入のために作成されたテーブルを開きます。

また、必要とするシステムカタログが作成され、値が設定されるまで、declare indexおよびdeclare toastコマンドは使用できません。

従い、postgres.bkiの構造は以下でなければなりません。

1. 1つの重要なテーブルをcreate bootstrap
2. 少なくとも重要なテーブルを記述するデータをinsert
3. close
4. その他の重要テーブルに対して反復。
5. 重要でないテーブルを(bootstrap無しで)create
6. open

7. 求められるデータのinsert

8. close

9. その他の重要でないテーブルに対して反復。

10. インデックスおよびTOASTテーブルの定義。

11. build indices

他にも確かに、ドキュメント化されていない順序に関する依存性があります。

69.6. BKIの例

次の一連のコマンドは、それぞれoid型、int4型、text型の3つの列、oid、cola、colbを持ち、OID 420 が付いたtest_tableテーブルを作成し、そして2つの行をテーブルに挿入します。

```
create test_table 420 (oid = oid, cola = int4, colb = text)
open test_table
insert ( 421 1 "value1" )
insert ( 422 2 _null_ )
close test_table
```

第70章 プランナは統計情報をどのように使用するか

本章は、14.1と14.2で扱われている題材を基にしていて、問い合わせの各段階において返される行数を推定するために、プランナがシステムの統計情報をどのように使用するかについて更なる詳細をいくつか説明します。これは計画作成処理において重要な部分で、コスト計算用の多くの情報を提供します。

本章の目的はコードを詳しく文書化することではありません。どのように動作するのかに関する概要を表すことが目的です。これによりおそらく、後にコードを参照するユーザの習得速度が向上するでしょう。

70.1. 行数推定の例

以下の例はPostgreSQLリグレッションテストデータベース内のテーブルを使用します。表示される出力はバージョン8.3で取得しました。以前の(または以降の)バージョンとは動作が変わっているかもしれません。また、ANALYZEは統計情報を生成する時にランダムなサンプリングを行いますので、結果はANALYZEを新しく行った後に多少変わることに注意してください。

非常に簡単な問い合わせから始めましょう。

```
EXPLAIN SELECT * FROM tenk1;

                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..458.00 rows=10000 width=244)
```

プランナがどのようにtenk1の濃度を決定するかについては14.2で説明しました。しかし、ここでは完全を期するために説明を繰り返します。ページ数および行数はpg_classから検索されます。

```
SELECT relpages, reltuples FROM pg_class WHERE relname = 'tenk1';

 relpages | reltuples
----------+-----------
      358 |     10000
```

これらの値は最後にそのテーブルをVACUUMまたはANALYZEを行った時点のものです。プランナはその後、テーブル内の実際のページ数を取り出します(これはテーブルスキャンを行わない安価な操作です)。それがrelpagesと異なる場合、reltuplesを得られたページ数の割合に応じて変更して現在の推定行数を求めます。上の例では、relpagesの値は最新のものなので、推定行数はreltuplesと同じです。

次にWHERE句に範囲条件を持つ例に進みましょう。

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 1000;

                                  QUERY PLAN
------------------------------------------------------------------------------
```

```
 Bitmap Heap Scan on tenk1  (cost=24.06..394.64 rows=1007 width=244)
   Recheck Cond: (unique1 < 1000)
   ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..23.80 rows=1007 width=0)
         Index Cond: (unique1 < 1000)
```

プランナはWHERE句の条件を検査し、pg_operator内の<演算子用の選択度関数を検索します。これはoprrest列に保持されます。今回の例ではこの項はscalarltselです。scalarltsel関数は、pg_statisticからunique1の度数分布を取り出します。手作業で問い合わせる場合は、より単純なpg_statsビューを検索した方が簡単です。

```
SELECT histogram_bounds FROM pg_stats
WHERE tablename='tenk1' AND attname='unique1';

                    histogram_bounds
------------------------------------------------------
 {0,993,1997,3050,4040,5036,5957,7057,8029,9016,9995}
```

次に、「< 1000」で占められる度数分布率を取り出します。これが選択度(selectivity)です。この度数分布は、範囲を等頻度のバケット(bucket)に分割します。ですので、行わなければならないことは、値が入るバケットを見つけ、その部分と、その前にあるバケット全体を数えることだけです。1000という値は明らかに2番目のバケット(993–1997)にあります。従って、値が各バケットの中で線形に分布していると仮定すると、選択度を以下のように計算することができます。

```
selectivity = (1 + (1000 - bucket[2].min)/(bucket[2].max - bucket[2].min))/num_buckets
            = (1 + (1000 - 993)/(1997 - 993))/10
            = 0.100697
```

つまり、1つのバケット全体に、2番目のバケットとの線形比率を加えたものを、バケット数で割ったものとなります。ここで、行の推定値は、選択度とtenk1の濃度を掛け合わせたものとして計算されます。

```
rows = rel_cardinality * selectivity
     = 10000 * 0.100697

     = 1007  (四捨五入)
```

次に、WHERE句に等価条件を持つ例を検討してみましょう。

```
EXPLAIN SELECT * FROM tenk1 WHERE stringu1 = 'CRAAAA';

                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=30 width=244)
   Filter: (stringu1 = 'CRAAAA'::name)
```

ここでも、プランナはWHERE句の条件を検査し、=用の選択度関数、この場合はeqselを検索します。等価性の推定では、度数分布は役に立ちません。代わりに、選択度の決定には頻出値（MCV）のリストが使用されます。MCVを見てみましょう。後で有用になる列がいくつかあります。

```
SELECT null_frac, n_distinct, most_common_vals, most_common_freqs FROM pg_stats
WHERE tablename='tenk1' AND attname='stringu1';

null_frac         | 0
n_distinct        | 676
most_common_vals  | {EJAAAA,BBAAAA,CRAAAA,FCAAAA,FEAAAA,GSAAAA,JOAAAA,MCAAAA,NAAAAA,WGAAAA}
most_common_freqs | {0.00333333,0.003,0.003,0.003,0.003,0.003,0.003,0.003,0.003,0.003}
```

CRAAAAがMCVのリスト内にありますので、選択度は単に頻出値の頻度（MCF）のリスト内の対応する項目になります。

```
selectivity = mcf[3]
            = 0.003
```

前と同様、推定される行数は単に前回同様、この値とtenk1の濃度との積です。

```
rows = 10000 * 0.003
     = 30
```

ここで、同じ問い合わせを見てみます。ただし、今回は定数がMCV内にありません。

```
EXPLAIN SELECT * FROM tenk1 WHERE stringu1 = 'xxx';

                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=15 width=244)
   Filter: (stringu1 = 'xxx'::name)
```

値がMCVの一覧にない場合、選択度をどのように推定するかは大きく異なります。値が一覧にない場合に使用される方法は、MCVすべての頻度に関する知識を組み合わせたものです。

```
selectivity = (1 - sum(mvf))/(num_distinct - num_mcv)
            = (1 - (0.00333333 + 0.003 + 0.003 + 0.003 + 0.003 + 0.003 +
                    0.003 + 0.003 + 0.003 + 0.003))/(676 - 10)
            = 0.0014559
```

つまり、MCVの頻度をすべて足し合わせたものを1から差し引き、そして、この他の個別値の数で除算します。これは、MCV以外の列の割合は、この他の個別値すべてに渡って一様に分布していることを前提としていることになります。NULL値が存在しないため、これを考慮する必要がないことに注意してください。（さもなくば、分子から同様にNULLの割合を差し引くことになります。）推定行数は以下のように普通に計算されます。

```
rows = 10000 * 0.0014559

     = 15  (四捨五入)
```

前述のunique1 < 1000を使用した例はscalarltselが本当は何を行うかについて、単純化しすぎたものでした。ここまでで、MCVを使用した例を見てきましたので、多少詳細に補てんすることができます。unique1は

一意な列であるため、MCVが存在しません（ある値が他の値と同じとなることがないことは明確です）ので、 例は計算自体は正確なものでした。一意ではない列では、通常度数分布とMCVリストの両方が存在します。そして、度数分布は、MCVで表される列母集団の位置を含みません。より正確な推定を行うことができるため、この方法を行います。この状況では、scalarltselは直接条件（例えば「< 1000」）をMCVリストの各値に適用し、条件を満たすMCVの頻度を足し合わせます。これがMCVのテーブル部分における正確な推定選択度です。その後度数分布が上と同様に使われ、MCV以外のテーブル部分における選択度を推定します。そしてこの2つの値を組み合わせて、全体の選択度を推定します。例えば、以下を検討します。

```
EXPLAIN SELECT * FROM tenk1 WHERE stringu1 < 'IAAAAA';

                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=3077 width=244)
   Filter: (stringu1 < 'IAAAAA'::name)
```

すでにstringu1のMCV情報は確認していますので、ここでは度数分布を見てみます。

```
SELECT histogram_bounds FROM pg_stats
WHERE tablename='tenk1' AND attname='stringu1';

                                histogram_bounds
--------------------------------------------------------------------------------
 {AAAAAA,CQAAAA,FRAAAA,IBAAAA,KRAAAA,NFAAAA,PSAAAA,SGAAAA,VAAAAA,XLAAAA,ZZAAAA}
```

MCVリストを検査すると、stringu1 < 'IAAAAA'条件は先頭の6項目で満たされ、最後の4項目で満たされないことがわかります。ですので、母集団のMCV部分における選択度は以下のようになります。

```
selectivity = sum(relevant mvfs)
            = 0.00333333 + 0.003 + 0.003 + 0.003 + 0.003 + 0.003
            = 0.01833333
```

MCFの総和はまた、MCVで表される母集団の合計割合が0.03033333であり、したがって度数分布で表される割合が0.96966667であることがわかります。（この場合もNULLは存在しません。もし存在する場合はここで除外しなければなりません。）IAAAAAという値は3番目のバケットの終端近辺になることを確認することができます。異なる文字の頻度について多少安っぽい仮定を使用すると、プランナはIAAAAAより小さい母集団の度数分布の部分の推定値は0.298387になります。そしてMCVと非MCV母集団についての推定値を組み合わせます。

```
selectivity = mcv_selectivity + histogram_selectivity * histogram_fraction
            = 0.01833333 + 0.298387 * 0.96966667
            = 0.307669

rows        = 10000 * 0.307669

            = 3077  (四捨五入)
```

列の分布がかなり平坦ですので、この特定の例におけるMCVリストによる訂正はかなり小さなものです。(これらの特定の値が他より頻出するものと示す統計情報はほとんどサンプリングエラーによります。) より一般的な、一部の値が他より非常に多く頻出する場合では、頻出値に対する選択度が正確に検出されますので、この複雑な処理により精度が改良されます。

次にWHERE句に複数の条件を持つ場合を検討しましょう。

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 1000 AND stringu1 = 'xxx';

                                  QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=23.80..396.91 rows=1 width=244)
   Recheck Cond: (unique1 < 1000)
   Filter: (stringu1 = 'xxx'::name)
   ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..23.80 rows=1007 width=0)
         Index Cond: (unique1 < 1000)
```

プランナは2つの条件が独立していると仮定します。このため、個々の句の選択度が掛け合わされます。

```
selectivity = selectivity(unique1 < 1000) * selectivity(stringu1 = 'xxx')
            = 0.100697 * 0.0014559
            = 0.0001466

rows        = 10000 * 0.0001466

            = 1  (四捨五入)
```

ビットマップインデックススキャンにより返されるものと推定される行数は、インデックスで使用される条件のみを反映することに注意してください。後続のヒープ取り出しのコスト推定に影響しますので、これは重要です。

最後に、結合を含む問い合わせを見てみましょう。

```
EXPLAIN SELECT * FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 50 AND t1.unique2 = t2.unique2;

                                     QUERY PLAN
-------------------------------------------------------------------------------------
 Nested Loop  (cost=4.64..456.23 rows=50 width=488)
   ->  Bitmap Heap Scan on tenk1 t1  (cost=4.64..142.17 rows=50 width=244)
         Recheck Cond: (unique1 < 50)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..4.63 rows=50 width=0)
               Index Cond: (unique1 < 50)
   ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.00..6.27 rows=1 width=244)
         Index Cond: (unique2 = t1.unique2)
```

tenk1「unique1 < 50」に関する制限が、ネステッドループ結合の前に評価されます。これは、前の範囲に関する例と同様に扱われます。しかし、今回の値50はunique1度数分布の最初のバケットにありますので、以下のようになります。

```
selectivity = (0 + (50 - bucket[1].min)/(bucket[1].max - bucket[1].min))/num_buckets
            = (0 + (50 - 0)/(993 - 0))/10
            = 0.005035

rows        = 10000 * 0.005035

            = 50  (四捨五入)
```

結合の制限はt2.unique2 = t1.unique2です。演算子はよく使用する単なる=ですが、選択度関数はpg_operatorのoprjoin列から入手され、eqjoinselとなります。eqjoinselはtenk2およびtenk1の両方の統計情報を検索します。

```
SELECT tablename, null_frac,n_distinct, most_common_vals FROM pg_stats
WHERE tablename IN ('tenk1', 'tenk2') AND attname='unique2';

tablename  | null_frac | n_distinct | most_common_vals
-----------+-----------+------------+------------------
 tenk1     |         0 |         -1 |
 tenk2     |         0 |         -1 |
```

今回の場合、すべての値が一意であるため、unique2に関するMCV情報がありません。ですので、両リレーションの個別値数とNULL値の部分のみに依存したアルゴリズムを使用することができます。

```
selectivity = (1 - null_frac1) * (1 - null_frac2) * min(1/num_distinct1, 1/num_distinct2)
            = (1 - 0) * (1 - 0) / max(10000, 10000)
            = 0.0001
```

これは、各リレーションにおいて、1からNULL部分を差し引き、個別値数の最大値で割った値です。この結合が生成するはずの行数は、2つの入力のデカルト積の濃度に、この選択度を掛けたものとして計算されます。

```
rows = (outer_cardinality * inner_cardinality) * selectivity
     = (50 * 10000) * 0.0001
     = 50
```

2つの列に対するMCVリストがありますので、eqjoinselはMCVで表される列母集団部分の結合選択度を決めるために、MCVリストを直接比較します。残りの母集団に対する推定はここで示した同じ手法に従います。

inner_cardinalityを10000、つまりtenk2の変更がないサイズと示していることに注意してください。EXPLAINの出力を検査すると、結合行の推定が50 * 1、つまり、外側の行数とtenk2上の内側のインデックススキャン毎に得られる推定行数を掛けた数から来ていると思うかもしれません。しかし、実際はそうではありません。結合リレーションサイズは、具体的な結合計画が検討される前に推定されます。もしすべてがうまくいけば、結合サイズを推定する2つの方法は同じ答えを導きます。しかし、四捨五入誤差などの要因により多少異なる場合があります。

詳細に興味を持った方向けに、テーブル（すべてのWHERE句の前にあるもの）のサイズ推定はsrc/backend/optimizer/util/plancat.cで行われます。句の選択度に関する一般的なロジックについてはsrc/backend/optimizer/path/clausesel.cにあります。演算子固有の選択度関数についてはたいていsrc/backend/utils/adt/selfuncs.c内にあります。

70.2. 多変量統計の例

70.2.1. 関数従属性

非常に単純なデータ集合で、多変量相関係数の例を示すことができます。2つの列を持ち、両方の列が同じ値を持つテーブルです。

```
CREATE TABLE t (a INT, b INT);
INSERT INTO t SELECT i % 100, i % 100 FROM generate_series(1, 10000) s(i);
ANALYZE t;
```

14.2で説明されているように、pg_classから得られるページ数と行数を使って、tの濃度を決定できます。

```
SELECT relpages, reltuples FROM pg_class WHERE relname = 't';

 relpages | reltuples
----------+-----------
       45 |     10000
```

データの分布はとても単純です。各々の列にはわずか100の異なる値があるだけであり、かつ均一に分布しています。

次の例では、a列に関するWHERE条件の見積もり結果を示しています。

```
EXPLAIN (ANALYZE, TIMING OFF) SELECT * FROM t WHERE a = 1;
                                 QUERY PLAN
-----------------------------------------------------------------------------
 Seq Scan on t  (cost=0.00..170.00 rows=100 width=8) (actual rows=100 loops=1)
   Filter: (a = 1)
   Rows Removed by Filter: 9900
```

プランナは、この条件を調べ、この句の選択度を1%と決定しました。この見積もりと、実際の行数を比較すると、見積もりは非常に正確であることがわかります。（テーブルがとても小さいので、実際には見積もり通りです。）WHERE条件を変更してb列を使うようにすると、同じプランが生成されます。では、AND条件でつないで、この二つの列に同じ条件を適用するとどうなるか見てみましょう。

```
EXPLAIN (ANALYZE, TIMING OFF) SELECT * FROM t WHERE a = 1 AND b = 1;
                                 QUERY PLAN
-----------------------------------------------------------------------------
 Seq Scan on t  (cost=0.00..195.00 rows=1 width=8) (actual rows=100 loops=1)
```

```
   Filter: ((a = 1) AND (b = 1))
   Rows Removed by Filter: 9900
```

個別に選択度を見積もった結果、上記と同じ1%の見積もりとなります 次に、その条件が独立であると見なし、それらの選択度を掛けあわせ、最終的な選択度の見積もりをわずか0.01%であるとします その条件に一致する実際の行数は2桁多いので(100)、これはかなり過小見積もりです。

この問題は、ANALYZEに二つの列について関数従属性多変量統計を計算させる、統計オブジェクトを作成することによって解決できます。

```
CREATE STATISTICS stts (dependencies) ON a, b FROM t;
ANALYZE t;
EXPLAIN (ANALYZE, TIMING OFF) SELECT * FROM t WHERE a = 1 AND b = 1;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Seq Scan on t  (cost=0.00..195.00 rows=100 width=8) (actual rows=100 loops=1)
   Filter: ((a = 1) AND (b = 1))
   Rows Removed by Filter: 9900
```

70.2.2. 多変量N個別値計数

GROUP BY句が生成するグループ数のような、複数列の集合の濃度の見積もりについても、同様の問題が起きます。GROUP BYの対象が単一の列なら、N個別値の推定（HashAggregateノードが返す推定行数で示されます）はとても正確です。

```
EXPLAIN (ANALYZE, TIMING OFF) SELECT COUNT(*) FROM t GROUP BY a;
                                     QUERY PLAN
-------------------------------------------------------------------------------------
 HashAggregate  (cost=195.00..196.00 rows=100 width=12) (actual rows=100 loops=1)
   Group Key: a
   ->  Seq Scan on t  (cost=0.00..145.00 rows=10000 width=4) (actual rows=10000 loops=1)
```

しかし、多変量統計がないと、二つの列についてのGROUP BY問い合わせにおけるグループ数の見積もりは、 次の例のようにひと桁ずれてしまいます

```
EXPLAIN (ANALYZE, TIMING OFF) SELECT COUNT(*) FROM t GROUP BY a, b;
                                     QUERY PLAN
-------------------------------------------------------------------------------------
 HashAggregate  (cost=220.00..230.00 rows=1000 width=16) (actual rows=100 loops=1)
   Group Key: a, b
   ->  Seq Scan on t  (cost=0.00..145.00 rows=10000 width=8) (actual rows=10000 loops=1)
```

二つの列についてのN個別値計数を含むように統計オブジェクトを再定義することにより、見積もりは大きく改善されます。

```
DROP STATISTICS stts;
```

```
CREATE STATISTICS stts (dependencies, ndistinct) ON a, b FROM t;
ANALYZE t;
EXPLAIN (ANALYZE, TIMING OFF) SELECT COUNT(*) FROM t GROUP BY a, b;
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 HashAggregate  (cost=220.00..221.00 rows=100 width=16) (actual rows=100 loops=1)
   Group Key: a, b
   ->  Seq Scan on t  (cost=0.00..145.00 rows=10000 width=8) (actual rows=10000 loops=1)
```

70.2.3. MCVリスト

70.2.1で説明したように、関数従属性は非常に安価で効率的な統計情報ですが、主要な制限はその大域的な性質です(列レベルだけでの従属性の追跡であり、個々の列の値の間のものではありません)。

この節ではMCV(最頻値)リストの多変量のもの、70.1で述べた行毎の統計情報の素直な拡張を導入します。この統計情報は格納された個々の値による制限を解決しますが、ANALYZEでの統計情報の構築や容量や計画作成時間に関して当然より高価です。

再び70.2.1の問い合わせを見てみましょう。ですが、今回は列の同じ集合に対してMCVリストを作ります(プランナが新しく作られた統計情報を確実に利用するよう、関数従属性を確実に削除してください)。

```
DROP STATISTICS stts;
CREATE STATISTICS stts2 (mcv) ON a, b FROM t;
ANALYZE t;
EXPLAIN (ANALYZE, TIMING OFF) SELECT * FROM t WHERE a = 1 AND b = 1;
                                    QUERY PLAN
--------------------------------------------------------------------------------
 Seq Scan on t  (cost=0.00..195.00 rows=100 width=8) (actual rows=100 loops=1)
   Filter: ((a = 1) AND (b = 1))
   Rows Removed by Filter: 9900
```

主に、テーブルがかなり小さく、異なる値の少ない単純な分布のおかげで、推定は関数従属性と同じくらい正確です。関数従属性では特に上手く扱えない2番目の問い合わせを見る前に、MCVリストを少し調べてみましょう。

MCVを調べるのは、集合を返すpg_mcv_list_items関数でできます。

```
SELECT m.* FROM pg_statistic_ext join pg_statistic_ext_data on (oid = stxoid),
                pg_mcv_list_items(stxdmcv) m WHERE stxname = 'stts2';
 index |  values  | nulls | frequency | base_frequency
-------+----------+-------+-----------+----------------
     0 | {0, 0}   | {f,f} |      0.01 |         0.0001
     1 | {1, 1}   | {f,f} |      0.01 |         0.0001
   ...
    49 | {49, 49} | {f,f} |      0.01 |         0.0001
    50 | {50, 50} | {f,f} |      0.01 |         0.0001
   ...
```

```
    97 | {97, 97} | {f,f} |      0.01 |          0.0001
    98 | {98, 98} | {f,f} |      0.01 |          0.0001
    99 | {99, 99} | {f,f} |      0.01 |          0.0001
(100 rows)
```

これで、2つの列の100個の個別の組み合わせがあり、すべてほぼ同様に確からしい(それぞれ1%の頻度)ことが確かめられます。基準となる頻度(base frequency)は、複数列の統計情報がないとして、列毎の統計情報から計算された頻度です。列のどちらか一方にでもNULL値があれば、nulls列で見分けられます。

選択性を推定する場合、プランナはMCVリストの項目にすべての条件を適用してから、一致するものの頻度を合計します。詳細はsrc/backend/statistics/mcv.cのmcv_clauselist_selectivityを参照してください。

関数従属性に比べて、MCVは主要な利点が2つあります。1つ目は、リストが実際の値を格納していることで、これによりどの組み合わせが適合するのか決定できます。

```
EXPLAIN (ANALYZE, TIMING OFF) SELECT * FROM t WHERE a = 1 AND b = 10;
                                 QUERY PLAN
-----------------------------------------------------------------------------
 Seq Scan on t  (cost=0.00..195.00 rows=1 width=8) (actual rows=0 loops=1)
   Filter: ((a = 1) AND (b = 10))
   Rows Removed by Filter: 10000
```

2つ目は、MCVリストが、関数従属性のような等式句だけでなく、より広い範囲の形の句を扱うことです。例えば、以下のような同じテーブルに対する範囲の問い合わせを考えてみましょう。

```
EXPLAIN (ANALYZE, TIMING OFF) SELECT * FROM t WHERE a <= 49 AND b > 49;
                                QUERY PLAN
-----------------------------------------------------------------------------
 Seq Scan on t  (cost=0.00..195.00 rows=1 width=8) (actual rows=0 loops=1)
   Filter: ((a <= 49) AND (b > 49))
   Rows Removed by Filter: 10000
```

70.3. プランナの統計情報とセキュリティ

テーブルpg_statisticへのアクセスはスーパーユーザのみに制限されているため、一般ユーザはこのテーブルを使って他のユーザのテーブル内容について調べることはできません。選択性推定関数には保存されている統計情報を解析するためにユーザ定義の演算子(問い合わせに現れる演算子あるいは関連する演算子)を使うものがあります。例えば、保存されている最頻値を適用できるかどうかを調べるためには、選択性推定関数は適切な=演算子を実行して問い合わせ内の定数を保存されている値と比較する必要があるでしょう。従って、pg_statistic内のデータは、潜在的に、ユーザ定義演算子に渡される可能性があります。巧妙に作られた演算子を使うと、渡された引数を意図的に漏らす(例えば、それをログに出力する、他のテーブルに書き出すなど)、あるいはその値をエラーメッセージに出力することで偶然に漏らすことが可能で、いずれにせよpg_statisticのデータを、それを見ることができないはずのユーザに対して露出する可能性があります。

このことを防ぐため、すべての組み込みの選択性推定関数には以下のことが適用されます。問い合わせの計画を作成するとき、保存されている統計情報を使用できるためには、現在のユーザはテーブルあるいは対象の列にSELECT権限を持っている必要がある、あるいは使用する演算子（正確には、演算子の元となる関数）がLEAKPROOFである必要があります。そうでないときは、選択性推定はあたかも利用可能な統計情報がないかのような動作をし、プランナはデフォルトあるいは代替の推定に従って処理をします。

ユーザがテーブルや列について必要な権限を持っていない場合、最終的には権限不足のエラーを受け取ることが多いでしょう。この場合、上記の仕組みは実際にはユーザからはわかりません。しかし、ユーザがセキュリティバリアビューから読み取ろうとしている場合、プランナはそのビューの元となっているテーブルの統計情報を検査したいと思うかもしれず、またユーザはそのテーブルにはアクセス権がないかもしれません。その場合は、演算子がリークプルーフ(leak-proof)でなければ、統計情報は使用されません。そのことについての直接的なフィードバックは何もなく、プランが理想的ではないかもしれないというだけです。このことが起きているかもしれないと思った場合は、より権限のあるユーザで問い合わせを実行して、異なる計画が得られるかどうか調べることができます。

この制限は、プランナがpg_statisticの1つ以上の値についてユーザ定義演算子を実行する必要がある場合にのみ適用されます。従って、列内でのNULL値の割合や異なる値の個数といった一般的な統計情報については、プランナはアクセス権限に関わらず使用することが許されています。

サードパーティの拡張に含まれる選択性推定関数で、ユーザ定義演算子で統計情報の演算をする可能性のあるものは、同じセキュリティ規則に従うべきです。そのための手引については、PostgreSQLのソースコードを参照してください。

第71章 バックアップマニフェスト書式

pg_basebackupで生成されるバックアップマニフェストはpg_verifybackupを用いてバックアップを検証できるようにすることを主目的としています。しかしながら、他のツールでバックアップマニフェストファイルを読んで中に含まれる情報を独自の目的に利用することも可能です。そのために、本章ではバックアップマニフェストファイルの書式を説明します。

バックアップマニフェストはUTF-8でエンコードされたJSONドキュメントです。(一般にJSONドキュメントはユニコードであることを必要としますが、PostgreSQLはjsonおよびjsonbデータ型にサポートされる全てのサーバエンコーディングを用いることを許しています。バックアップマニフェストに同様の例外はありません。)このJSONドキュメントは常に単一オブジェクトです。オブジェクトにあるキーについて、次章で説明します。

71.1. バックアップマニフェストの最上位レベルオブジェクト

バックアップマニフェストのJSONドキュメントには以下のキーがあります。

`PostgreSQL-Backup-Manifest-Version`

関連付けられた値は常に整数の1です。

`Files`

関連付けられた値は常にオブジェクトのリストで、それぞれがバックアップ中にある一つのファイルを記述しています。このリストにはバックアップを使うために必要なWALファイルやバックアップマニフェスト自体に対するエントリは含まれません。リスト内の各オブジェクトの構造は71.2で説明します。

`WAL-Ranges`

関連付けられた値は常にオブジェクトのリストで、それぞれがバックアップを使用するために特定タイムラインから読み込み可能でなければならないWALレコードの範囲を記述しています。これらオブジェクトの構造は後段の71.3で説明します。

`Manifest-Checksum`

このキーは常にバックアップマニフェストファイルの最後の行にあります。関連付けられた値はこれ以前の全行のSHA256チェックサムです。クライアントでマニフェストの逐次的な解析を可能とするため、ここでは固定のチェックサム方式を使います。SHA256チェックサムはCRC32Cチェックサムよりもかなり高コストですが、マニフェストは通常、追加の計算が大きな問題とならない程度に小さいはずです。

71.2. バックアップマニフェストのファイルオブジェクト

単一ファイルを記述するオブジェクトは、PathキーかEncoded-Pathキーを持ちます。通常はPathキーになります。関連付けられた文字列値はファイルのバックアップディレクトリからの相対パスです。ユーザ定義テー

ブル空間にあるファイルは、最初の2つの要素がpg_tblspcとテーブル空間のOIDであるパスを持ちます。パスがUTF-8として正当な文字列でなかったり、ユーザが全ファイルにエンコードされたパスが使われることを要求した場合には、代替にEncoded-Pathキーがあらわれます。これは同じデータを格納しますが、16進数の文字列としてエンコードされます。文字列における16進数の各2つ組で1オクテットを表現します。

以下の2つのキーは常に含まれます。

Size

ファイルの期待されるサイズです。整数として書かれます。

Last-Modified

バックアップ時にサーバによって報告されたファイルの最終変更時刻です。バックアップに格納された他フィールドと違い、本フィールドはpg_verifybackupでは使われません。情報提供のみを目的として含まれています。

ファイルチェックサムを有効にしてバックアップが取得された場合、以下のキーが含まれます。

Checksum-Algorithm

このファイルのチェックサム計算で使用されるチェックサムアルゴリズムです。今のところ、これはバックアップマニフェスト内の全ファイルで同じになりますが、将来のリリースで変更されるかもしれません。現在サポートされるアルゴリズムは、CRC32C、SHA224、SHA256、SHA384、および、SHA512です。

Checksum

このファイルに対して計算されたチェックサムで、チェックサムの各バイト毎に2文字の、16進数の文字列として格納されます。

71.3. バックアップマニフェストのWAL範囲オブジェクト

WAL範囲を記述するこのオブジェクトは常に以下の3つのキーを持ちます。

Timeline

WALレコードのこの範囲に対するタイムラインです。整数として書かれます。

Start-LSN

バックアップを使用するためにリプレイを開始しなければならない指定されたタイムライン上のLSNです。このLSNはPostgreSQLで標準的に使われる書式で格納されます。すなわち、2つの16進数文字列で構成され、それぞれの長さが1から8で、スラッシュで区切られた文字列です。

End-LSN

このバックアップを使うときにリプレイを終了してもよい、指定されたタイムライン上の最も早いLSNです。これはStart-LSNと同じ書式で格納されます。

通常、単一のWAL範囲だけが存在します。しかしながら、バックアップが、上流の昇格のためにバックアップ中にタイムラインを変更したスタンバイから取得された場合、それぞれ異なるタイムラインを持つ複数の範囲が存在する可能性があります。同じタイムラインに対して複数のWAL範囲があらわれることは決してありません。

パート VIII. 付録

目次

付録A PostgreSQLエラーコード

PostgreSQLサーバによって発行されるメッセージは全て、標準SQLにおける「SQLSTATE」コードの記述方法に従った、5文字のエラーコードが割り当てられています。どのようなエラー条件が発生したかを把握しなければならないアプリケーションは、通常テキスト形式のエラーメッセージを確認するのではなく、このエラーコードを検査すべきです。このエラーコードは、PostgreSQLリリースの違いによって変更することはあまりありません。また、エラーメッセージの各国言語化による変更にも影響されません。PostgreSQLで発行されるエラーコードのいくつか（全部ではありません）は、標準SQLで定義されていることに注意してください。標準SQLで定義されていない追加のエラーコードは、独自のものであったり他のデータベースから取り入れたものです。

標準に従い、エラーコードの最初の2文字はエラーのクラスを表し、残り3文字がそのクラス内の特定条件を表します。したがって、特定のエラーコードを認識できないアプリケーションであっても、エラークラスから何をすべきか推定できることがあります。

表 A.1は、PostgreSQL 13.1で定義されたエラーコードを全て一覧で示しています（標準SQLでは定義されていますが、現在実際に使用されていないものもあります）。エラークラスも示しています。各エラークラスには、残りの3文字が000となる「標準」エラーコードが存在します。あるクラスの範囲内で発生したが、より特定のコードが割り当てられていないエラー条件のためだけに、このコードが使用されます。

「条件名」列に示されているシンボルはPL/pgSQLで使用している条件名です。条件名は大文字もしくは小文字でも記述されます。（PL/pgSQLは、エラーの場合と異なり、警告の場合にはその状態名を認識しません。これらはクラス00と01と02です）。

いくつかのエラー型に対し、サーバはそのエラーに起因するデータベースオブジェクト（テーブル、テーブル列、データ型、もしくは制約）の名前を報告します。たとえば、unique_violationエラーの原因となった一意性制約の名前です。これら名前はエラーレポート文書の分離されたフィールドに与えられます。これにより、アプリケーションは読んで理解できる翻訳されているかも知れない文書のテキストからそれらを抽出しようと試みる必要がなくなります。PostgreSQL 9.3の時点で、この機能を完璧に保証する範囲はSQLSTATEクラス23（整合性制約違反）のみですが、将来的には十中八九拡張されそうです。

表A.1 PostgreSQLエラーコード

エラーコード	条件名
クラス 00 — 正常終了	
00000	successful_completion
クラス 01 — 警告	
01000	warning
0100C	dynamic_result_sets_returned
01008	implicit_zero_bit_padding
01003	null_value_eliminated_in_set_function
01007	privilege_not_granted
01006	privilege_not_revoked
01004	string_data_right_truncation
01P01	deprecated_feature

エラーコード	条件名
クラス 02 — データがない(標準SQLではこれは警告クラスでもある)	
02000	no_data
02001	no_additional_dynamic_result_sets_returned
クラス 03 — SQL文の未完了	
03000	sql_statement_not_yet_complete
クラス 08 — 接続の例外	
08000	connection_exception
08003	connection_does_not_exist
08006	connection_failure
08001	sqlclient_unable_to_establish_sqlconnection
08004	sqlserver_rejected_establishment_of_sqlconnection
08007	transaction_resolution_unknown
08P01	protocol_violation
クラス 09 — トリガによるアクションの例外	
09000	triggered_action_exception
クラス 0A — サポートされない機能	
0A000	feature_not_supported
クラス 0B — 無効なトランザクションの開始	
0B000	invalid_transaction_initiation
クラス 0F — 位置付け子の例外	
0F000	locator_exception
0F001	invalid_locator_specification
クラス 0L — 無効な権限付与	
0L000	invalid_grantor
0LP01	invalid_grant_operation
クラス 0P — 無効なロールの指定	
0P000	invalid_role_specification
クラス 0Z — 診断の例外	
0Z000	diagnostics_exception
0Z002	stacked_diagnostics_accessed_without_active_handler
クラス 20 — caseが存在しない	
20000	case_not_found
クラス 21 — 次数違反	
21000	cardinality_violation

エラーコード	条件名
クラス 22 — データ例外	
22000	data_exception
2202E	array_subscript_error
22021	character_not_in_repertoire
22008	datetime_field_overflow
22012	division_by_zero
22005	error_in_assignment
2200B	escape_character_conflict
22022	indicator_overflow
22015	interval_field_overflow
2201E	invalid_argument_for_logarithm
22014	invalid_argument_for_ntile_function
22016	invalid_argument_for_nth_value_function
2201F	invalid_argument_for_power_function
2201G	invalid_argument_for_width_bucket_function
22018	invalid_character_value_for_cast
22007	invalid_datetime_format
22019	invalid_escape_character
2200D	invalid_escape_octet
22025	invalid_escape_sequence
22P06	nonstandard_use_of_escape_character
22010	invalid_indicator_parameter_value
22023	invalid_parameter_value
22013	invalid_preceding_or_following_size
2201B	invalid_regular_expression
2201W	invalid_row_count_in_limit_clause
2201X	invalid_row_count_in_result_offset_clause
2202H	invalid_tablesample_argument
2202G	invalid_tablesample_repeat
22009	invalid_time_zone_displacement_value
2200C	invalid_use_of_escape_character
2200G	most_specific_type_mismatch
22004	null_value_not_allowed
22002	null_value_no_indicator_parameter

エラーコード	条件名
22003	numeric_value_out_of_range
2200H	sequence_generator_limit_exceeded
22026	string_data_length_mismatch
22001	string_data_right_truncation
22011	substring_error
22027	trim_error
22024	unterminated_c_string
2200F	zero_length_character_string
22P01	floating_point_exception
22P02	invalid_text_representation
22P03	invalid_binary_representation
22P04	bad_copy_file_format
22P05	untranslatable_character
2200L	not_an_xml_document
2200M	invalid_xml_document
2200N	invalid_xml_content
2200S	invalid_xml_comment
2200T	invalid_xml_processing_instruction
クラス 23 — 整合性制約違反	
23000	integrity_constraint_violation
23001	restrict_violation
23502	not_null_violation
23503	foreign_key_violation
23505	unique_violation
23514	check_violation
23P01	exclusion_violation
クラス 24 — 無効なカーソル状態	
24000	invalid_cursor_state
クラス 25 — 無効なトランザクション状態	
25000	invalid_transaction_state
25001	active_sql_transaction
25002	branch_transaction_already_active
25008	held_cursor_requires_same_isolation_level
25003	inappropriate_access_mode_for_branch_transaction

エラーコード	条件名
25004	inappropriate_isolation_level_for_branch_transaction
25005	no_active_sql_transaction_for_branch_transaction
25006	read_only_sql_transaction
25007	schema_and_data_statement_mixing_not_supported
25P01	no_active_sql_transaction
25P02	in_failed_sql_transaction
25P03	idle_in_transaction_session_timeout
クラス 26 — 無効なSQL文の名前	
26000	invalid_sql_statement_name
クラス 27 — トリガによるデータ変更違反	
27000	triggered_data_change_violation
クラス 28 — 無効な認証指定	
28000	invalid_authorization_specification
28P01	invalid_password
クラス 2B — 依存する権限記述子がまだ存在する	
2B000	dependent_privilege_descriptors_still_exist
2BP01	dependent_objects_still_exist
クラス 2D — 無効なトランザクションの終了	
2D000	invalid_transaction_termination
クラス 2F — SQLルーチン例外	
2F000	sql_routine_exception
2F005	function_executed_no_return_statement
2F002	modifying_sql_data_not_permitted
2F003	prohibited_sql_statement_attempted
2F004	reading_sql_data_not_permitted
クラス 34 — 無効なカーソル名称	
34000	invalid_cursor_name
クラス 38 — 外部ルーチン例外	
38000	external_routine_exception
38001	containing_sql_not_permitted
38002	modifying_sql_data_not_permitted
38003	prohibited_sql_statement_attempted
38004	reading_sql_data_not_permitted
クラス 39 — 外部ルーチン呼び出し例外	

エラーコード	条件名
39000	external_routine_invocation_exception
39001	invalid_sqlstate_returned
39004	null_value_not_allowed
39P01	trigger_protocol_violated
39P02	srf_protocol_violated
39P03	event_trigger_protocol_violated
クラス 3B — セーブポイント例外	
3B000	savepoint_exception
3B001	invalid_savepoint_specification
クラス 3D — 無効なカタログ名称	
3D000	invalid_catalog_name
クラス 3F — 無効なスキーマ名称	
3F000	invalid_schema_name
クラス 40 — トランザクションロールバック	
40000	transaction_rollback
40002	transaction_integrity_constraint_violation
40001	serialization_failure
40003	statement_completion_unknown
40P01	deadlock_detected
クラス 42 — 構文エラーもしくはアクセス規則違反	
42000	syntax_error_or_access_rule_violation
42601	syntax_error
42501	insufficient_privilege
42846	cannot_coerce
42803	grouping_error
42P20	windowing_error
42P19	invalid_recursion
42830	invalid_foreign_key
42602	invalid_name
42622	name_too_long
42939	reserved_name
42804	datatype_mismatch
42P18	indeterminate_datatype
42P21	collation_mismatch

エラーコード	条件名
42P22	indeterminate_collation
42809	wrong_object_type
428C9	generated_always
42703	undefined_column
42883	undefined_function
42P01	undefined_table
42P02	undefined_parameter
42704	undefined_object
42701	duplicate_column
42P03	duplicate_cursor
42P04	duplicate_database
42723	duplicate_function
42P05	duplicate_prepared_statement
42P06	duplicate_schema
42P07	duplicate_table
42712	duplicate_alias
42710	duplicate_object
42702	ambiguous_column
42725	ambiguous_function
42P08	ambiguous_parameter
42P09	ambiguous_alias
42P10	invalid_column_reference
42611	invalid_column_definition
42P11	invalid_cursor_definition
42P12	invalid_database_definition
42P13	invalid_function_definition
42P14	invalid_prepared_statement_definition
42P15	invalid_schema_definition
42P16	invalid_table_definition
42P17	invalid_object_definition
クラス 44 — WITH CHECK OPTION違反	
44000	with_check_option_violation
クラス 53 — リソース不足	
53000	insufficient_resources

エラーコード	条件名
53100	disk_full
53200	out_of_memory
53300	too_many_connections
53400	configuration_limit_exceeded
クラス 54 — プログラム制限超過	
54000	program_limit_exceeded
54001	statement_too_complex
54011	too_many_columns
54023	too_many_arguments
クラス 55 — 必要条件を満たさないオブジェクト	
55000	object_not_in_prerequisite_state
55006	object_in_use
55P02	cant_change_runtime_param
55P03	lock_not_available
クラス 57 — 操作の介入	
57000	operator_intervention
57014	query_canceled
57P01	admin_shutdown
57P02	crash_shutdown
57P03	cannot_connect_now
57P04	database_dropped
クラス 58 — システムエラー(PostgreSQL自体の外部のエラー)	
58000	system_error
58030	io_error
58P01	undefined_file
58P02	duplicate_file
クラス 72 — スナップショット失敗	
72000	snapshot_too_old
クラス F0 — 設定ファイルエラー	
F0000	config_file_error
F0001	lock_file_exists
クラス HV — 外部データラッパーエラー (SQL/MED)	
HV000	fdw_error
HV005	fdw_column_name_not_found

エラーコード	条件名
HV002	fdw_dynamic_parameter_value_needed
HV010	fdw_function_sequence_error
HV021	fdw_inconsistent_descriptor_information
HV024	fdw_invalid_attribute_value
HV007	fdw_invalid_column_name
HV008	fdw_invalid_column_number
HV004	fdw_invalid_data_type
HV006	fdw_invalid_data_type_descriptors
HV091	fdw_invalid_descriptor_field_identifier
HV00B	fdw_invalid_handle
HV00C	fdw_invalid_option_index
HV00D	fdw_invalid_option_name
HV090	fdw_invalid_string_length_or_buffer_length
HV00A	fdw_invalid_string_format
HV009	fdw_invalid_use_of_null_pointer
HV014	fdw_too_many_handles
HV001	fdw_out_of_memory
HV00P	fdw_no_schemas
HV00J	fdw_option_name_not_found
HV00K	fdw_reply_handle
HV00Q	fdw_schema_not_found
HV00R	fdw_table_not_found
HV00L	fdw_unable_to_create_execution
HV00M	fdw_unable_to_create_reply
HV00N	fdw_unable_to_establish_connection
クラス P0 — PL/pgSQLエラー	
P0000	plpgsql_error
P0001	raise_exception
P0002	no_data_found
P0003	too_many_rows
P0004	assert_failure
クラス XX — 内部エラー	
XX000	internal_error
XX001	data_corrupted

エラーコード	条件名
XX002	index_corrupted

付録B 日付/時刻のサポート

PostgreSQLは、全ての日付/時刻入力のサポートにおいて、内蔵しているヒューリスティックなパーサを使用します。日付と時刻は文字列で入力され、そのフィールドにはどのような種類の情報が入るのかが事前に決められている別個のフィールドに分割されます。それぞれのフィールドは解釈された後、数値を割り当てられたり、無視されたり、あるいははねられたりします。構文解析に際し、月、曜日、および時間帯を含む、テキストフィールドすべてに対する内部参照テーブルがあります。

この付録ではこれらの参照テーブルの内容についての情報と構文解析で日付と時刻を解読する手順を説明します。

B.1. 日付/時刻入力の解釈

日付/時刻入力文字列は以下の手続きを使って解読します。

1. 入力文字列をトークンに分割し、そしてそれぞれのトークンを文字列、時刻、時間帯、または数値というように分類します。

    a. 数値トークンにコロン(:)が含まれている場合は、時刻文字列です。そこに続く全ての数字とコロンを含みます。

    b. 数値トークンにハイフン(-)、スラッシュ(/)、または2つ以上のドット(.)が含まれている場合は、テキストの月名がある日付文字列です。日付トークンがすでに現れている場合は代わりに、時間帯名として解釈されます(例えばAmerica/New_York)。

    c. トークンが数値だけの場合、それは単項、もしくはISO 8601の連結形式の日付(例:1999年1月13日を示す19990113)、あるいは時刻(例:14:15:16を示す141516)のいずれかです。

    d. トークンがプラス記号(+)あるいはマイナス記号(-)で始まっている場合は、数値形式の時間帯フィールドか特殊なフィールドです。

2. もしトークンがアルファベット文字列の場合、以下のように可能性のある文字列と照合されます。

    a. トークンが既知の時間帯省略形と一致するかどうかを調べます。これらの短縮形はB.4で記述する設定ファイルで提供されます。

    b. 見つからなかった場合は、そのトークンに対し特殊文字列(たとえばtoday)、曜日(たとえばThursday)、月(たとえばJanuary)、ノイズ(たとえばat、on)に一致するかどうか、内部テーブルを検索します。

    c. それでも探し出せなかった場合、エラーを返します。

3. トークンが数値あるいは数値フィールドの場合を以下に示します。

    a. トークンが8桁または6桁、かつ、以前に他のどのような日付フィールドも読まれていない場合は、「連結された日付」(例えば、19990118または990118)として解釈されます。その解釈方法はYYYYMMDDまたはYYMMDDです。

    b. もしトークンが3桁で年が既に読み込まれている場合は年内の経過日数と解釈されます。

c. 4桁または6桁の場合で年が既に読み込まれている時は時刻(HHMMまたはHHMMSS)と解釈されます。

d. 3桁以上の場合、かつ、どの日付フィールドもまだ見つかっていない場合は年と解釈されます (この場合、残る日付フィールドの順序は強制的にyy-mm-ddと解釈されます)。

e. さもなければ、日付フィールドの順序は、DateStyleの設定mm-dd-yy、dd-mm-yy、yy-mm-ddに従うものと仮定されます。月や月内の日のフィールドが範囲外であれば、エラーになります。

4. もしBCが指定された場合は内部格納用に年を負の数にして1を加えます (グレゴリオ暦にはゼロ年がないので、数値的には1BC(紀元前1年)がゼロ年になります)。

5. BCが指定されず年フィールドの長さが2桁の場合、年は4桁になるよう調整されます。そのフィールドが70未満の場合は2000が加えられますが、その他の場合には1900が加えられます。

> **ヒント**
>
> (例えば、西暦99年を0099のように)グレゴリオ暦の西暦元年から99年までは、ゼロを前に付加して4桁で入力することができます。

B.2. 不正あるいは曖昧なタイムスタンプの扱い

日付/時刻文字列が構文的に正しいが、フィールドの範囲外の値を含んでいる場合、通常、エラーとなります。たとえば、2月31日を指定した入力は受け付けられません。

夏時間の移行期間では、一見正しく見えるタイムスタンプ文字列が、存在しない、あるいは曖昧なタイムスタンプを表現してしまうことがあります。そのような場合はエラーで弾くことはせず、どのUTCオフセットを適用するかを決定する過程で曖昧さが解消されます。たとえばTimeZoneパラメータがAmerica/New_Yorkに設定されているとして、以下の例を考えてみましょう。

```
=> SELECT '2018-03-11 02:30'::timestamptz;
      timestamptz
------------------------
 2018-03-11 03:30:00-04
(1 row)
```

その時間帯では、その日は春に時刻を進める(spring-forward transition)日なので、標準時で2:30AMは存在しません。2AM ESTから3AM EDTに時計がジャンプするからです。PostgreSQLはあたかも標準時(UTC-5)で時刻を与えられたかのように解釈し、続いて3:30AM EDT (UTC-4)として表示しました。

逆に、秋に時刻を戻す移行期間(fall-back transition)の振る舞いを考えます。

```
=> SELECT '2018-11-04 02:30'::timestamptz;
      timestamptz
------------------------
 2018-11-04 02:30:00-05
(1 row)
```

その日は、2:30AMに対してふた通りの解釈が可能でした。まず2:30AM EDTがあり、1時間後の標準時への遡行ののち、2:30AM ESTとなりました。ここでもPostgreSQLはあたかも標準時(UTC-5)で時刻を与えられたかのように解釈しました。夏時間を指定することにより、動作を強制できます。

```
=> SELECT '2018-11-04 02:30 EDT'::timestamptz;
      timestamptz
------------------------
 2018-11-04 01:30:00-05
(1 row)
```

このタイムスタンプは2:30 UTC-4あるいは1:30 UTC-5のどちらでも正しく表示できます。タイムスタンプの出力コードは後者を選びました。

このような場合に適用される正確なルールは次のようなものです。夏時間で時刻を進める移行期間に入る不正なタイムスタンプは、移行期間の直前の時間帯に適用されるUTCオフセットが割り当てられます。一方、時刻を戻す移行期間の前あるいは後のどちらでにでも入る可能性のある不正なタイムスタンプは、移行期間の後に相当するUTCオフセットが割り当てられます。ほとんどの時間帯にとってこれは「疑わしければ標準時間として解釈する」と言うのと同じです。

どんな場合でも、数字のUTCオフセットを使うか、あるいは時間帯省略形に関連する固定のUTCオフセットを使って、タイムスタンプに付随するUTCオフセットを明示的に指定できます。ここで説明したルールは、ある時間帯のUTCオフセットが変動し、UTCオフセットを推測する必要がある場合にのみ適用されます。

B.3. 日付/時刻キーワード

表 B.1に月名として認識されるトークンを示します。

表B.1 月名

月	簡略形
January	Jan
February	Feb
March	Mar
April	Apr
May	
June	Jun
July	Jul
August	Aug
September	Sep、Sept
October	Oct
November	Nov
December	Dec

表 B.2に、曜日名として認識されるトークンを示します。

表B.2 曜日名

曜日	簡略形
Sunday	Sun
Monday	Mon
Tuesday	Tue、Tues
Wednesday	Wed、Weds
Thursday	Thu、Thur、Thurs
Friday	Fri
Saturday	Sat

表 B.3に、様々な修飾子の目的を持つトークンを示します。

表B.3 日付/時刻フィールドの修飾子

識別子	説明
AM	12:00以前の時刻
AT	このキーワードは無視されます
JULIAN, JD, J	次のフィールドはユリウス日
ON	このキーワードは無視されます
PM	12:00と12:00以降の時刻
T	次のフィールドは時刻

B.4. 日付/時刻設定ファイル

時間帯省略形は十分に標準化されていませんので、PostgreSQLでは、サーバで受付け可能な省略形群をカスタマイズできる仕組みを提供します。timezone_abbreviations実行時パラメータは有効な省略形群を決定します。このパラメータはすべてのデータベースユーザで変更可能ですが、取り得る値はデータベース管理者により制御されます。これらは実際にはインストレーションディレクトリの.../share/timezonesets/内の設定ファイル名です。代替ファイルをこのディレクトリに追加することにより、管理者は時間帯省略形に対するローカルポリシーを設定することができます。

timezone_abbreviationsは、.../share/timezonesets/に存在する、名前のすべてがアルファベットである任意のファイルの名前を指定することができます。(timezone_abbreviations内の非アルファベット文字の禁止により、意図したディレクトリ以外のファイル読み取りを防ぐことができます。また、バックアップファイルやその他のファイルの読み取りを防ぐこともできます。)

時間帯省略形ファイルには空白行や#から始まるコメントを含めることができます。コメント以外は以下の書式を持たなければなりません。

```
zone_abbreviation offset
```

```
zone_abbreviation offset D
zone_abbreviation time_zone_name
@INCLUDE file_name
@OVERRIDE
```

zone_abbreviationは単なる定義された省略形です。offsetはUTCからの相当するオフセットを秒数で表した整数です。グリニッジより東にあれば正、西にあれば負の値となります。たとえば、-18000はグリニッジより西に5時間、すなわち、北アメリカ東海岸の標準時間を示します。Dは、時間帯名が標準時間ではなくローカル時間での夏時間を表すことを示します。

あるいは、IANA時間帯データベースで定義されている地域名を参照するtime_zone_nameを指定することもできます。地域の定義はその地域の省略形が存在するか、もしくは、使われてきたかを確認し、もしそうであれば、適切な意味として使われます。適切な意味とは、確定した値を持つタイムスタンプが使われている意味、その当時は使われていなかったかもしれないが、後に即座に使われる意味、もしくは、その時の後にのみ使われる最も古い意味です。この挙動は歴史的に変化した意味を持つ省略形を扱う際には不可欠です。また、省略形が現れない地域名に関して省略形を定義することもできます。この省略形を使うことは地域名を書き出すことと全く同じです。

ヒント

タイムゾーンの定義の参照が必要になる過程よりはるかに安価であるため、UTCからのオフセットが今までに一度も変わっていない省略形を扱う場合は、単純な整数型のoffsetを使う方が好ましいでしょう。

@INCLUDE構文により、.../share/timezonesets/ディレクトリ内の他のファイルを含有することができます。深さに制限がありますが、入れ子に含有することができます。

@OVERRIDE構文は、ファイル内の続く項目が既存の項目（典型的には、インクルードされたファイルから得られた項目）を上書きできることを示します。これがないと、同一時間帯省略形の定義が競合した場合にエラーとみなされます。

未変更のインストレーションでは、Defaultファイルに、ほとんど全世界の競合しない時間帯省略形をすべて記載しています。さらにAustraliaおよびIndiaファイルがこれらの地区向けに提供されています。これらのファイルはDefaultファイルの先頭で含有されています。必要に応じて省略形の追加・変更を行ってください。

参考のため、標準のインストレーションにはAfrica.txt、America.txtなども含まれています。これらにはIANA時間帯データベースに従って使用されている時間帯省略形に関する情報がすべて含まれています。これらのファイル内にある時間帯定義を必要に応じてコピーペーストして独自の設定ファイルを編集することができます。これらのファイル名にドットが入っていますので、timezone_abbreviations設定から直接参照できないことに注意してください。

注記

時間帯省略形を読み込む時にエラーが発生した場合、新しい値は適用されず、古い値がそのまま残ります。データベースの起動時にエラーが起きた場合は、起動に失敗します。

注意

設定ファイル内で定義される時間帯省略形は、PostgreSQLに組み込み済みの時間帯以外の意味も変更します。たとえば、Australia設定ファイルではSAT(南オーストラリア標準時間)を定義しています。このファイルが有効な場合、SATは土曜の省略形として認識されなくなります。

注意

.../share/timezonesets/を変更する時にバックアップを責任を持って行ってください。このディレクトリは通常のデータベースダンプに含まれません。

B.5. POSIX 時間帯の指定

PostgreSQLはTZ環境変数を使ったPOSIX標準ルールに沿って記述された時間帯指定を受け入れることができます。POSIX時間帯の指定は複雑な実世界の時間帯の歴史を扱うには不足しているところもありますが、それを利用する理由があることもあります。

POSIX時間帯の指定には以下の形式があります。

```
STD offset [ DST [ dstoffset ] [ , rule ] ]
```

(可読性のためにフィールド間にスペースを表示していますが、実際にはスペースは使用されません。) フィールドは以下の通りです。

- STDは標準時間に使用されるゾーンの省略形です。
- offsetはUTCから標準時間のオフセットです。
- DSTは夏時間に使用されるゾーンの省略形です。このフィールドと以下のフィールドが省略された場合、時間帯は夏時間のルールを適用しない固定されたUTCからのオフセットを使用します。
- dstoffsetはUTCからの夏時間のオフセットです。このフィールドは通常は省略されます。このため、デフォルトでは標準時間の offsetより1時間短くなりますが、これは通常は正しい動作です。
- 以下に記載するように、ruleは夏時間が有効な場合のルールを定義します。

この構文では、ゾーンの省略形はESTのような文字列か、<UTC-05>のような角括弧で囲った任意の文字列にすることができます。ここで与えられた省略形は出力にのみ、中でも一部のタイムスタンプの出力フォーマットにのみ使われることに注意してください。タイムスタンプの入力で認識される時間帯の省略形はB.4の中で説明されているように決定されます。

オフセットのフィールドはUTCからの差を時間、オプションで分、秒で指定します。オフセットはhh[:mm[:ss]]のフォーマットで、オプションで先頭に符号をつけることができます(+ もしくは -)。正の符号はグリニッジよりも西の時間帯に使用されます。(これは他のPostgreSQLで使われているISO-8601の規定とは反対であることに注意してください。) hhは1桁もしくは2桁です。mmとssを使う場合は2桁でなければなりません。

サマータイム変換のruleには以下のフォーマットがあります。

```
dstdate [ / dsttime ] , stddate [ / stdtime ]
```

(前述の通り、実際にはスペースを含めるべきではありません) 夏時間の開始時刻は、dstdateとdsttimeフィールドが定義し、標準時間の開始時刻はstddateとstdtimeで定義します。(特に赤道より南の時間帯では前者は後者より年の後半になることもあります。) 日付フィールドには以下のような形式があります。

n

単純な整数は年の日を示し、0から364、閏年の場合は365までを数えます。

Jn

この形式ではnは1から365までを数え、2月29日は存在したとしても数えません。(このように、2月29日の変換が発生する場合はこの方法では指定できません。しかし、2月以降は、うるう年でもそうでなくとも同じ数になります。このため、この形式は特定のある日に変換する場合、通常、単純な整数型の形式を利用するよりも有用です。)

Mm.n.d

この形式は同じ月の同じ曜日にいつも発生する変換を指定します。mは1から12までの月を指定します。nはnで指定された週のd番目の日を指定します。nは数字の1から5で、5の場合はその月の最後の週を意味します(4番目か5番目の週になる可能性があります)。dは数字の0から6で、0は日曜日を指します。例えば、M3.2.0は「3月の第2日曜日」を意味します。

> **注記**
>
> M形式は多くの一般的な夏時間の変換法を記述するのに十分です。しかし、夏時間変換法の変化を扱う変数は無いため、実際には、過去のデータを名前付き時間帯(IANA時間帯のデータベースにある)で配置するためには、過去のタイムスタンプを変換する必要があります。

変換ルール中の時間フィールドは符号を含めることができない点を除いて、先に記載したオフセットのフィールドと同じ形式を持っています。これらのフィールドは他の時間への変換が発生した時の現在のローカル時間を定義します。省略された場合、デフォルトは02:00:00です。

夏時間の省略形が与えられているが移行ruleフィールドが省略されている場合、代替の動作には2020年のアメリカ合衆国の習慣と照合されるM3.2.0,M11.1.0(3月の第2日曜日に夏時間に切り替わり、11月の第1日曜日に戻ります。両方の移行はその時進んでいる時間の午前2時に行われます)が使用されます。この規則は、2007年より前の年の正しいアメリカ合衆国移行日を示していないことに注意してください。

例えば、CET-1CEST,M3.5.0,M10.5.0/3は(2020年時点の)パリの現時点の時計方法を表しています。この指定では、標準時間はCETという略語を持ち、UTCより1時間(東)進んでいます。また、夏時間には、CESTという略語を持ち、暗黙的にUTCより2時間進んでいます。夏時間は3月の最終日曜のAM2時に始まり、10月の最終日曜日の3AM CESTに終わります。

4つの時間帯名、EST5EDT、CST6CDT、MST7MDT、PST8PDTはPOSIXゾーンの指定に見えます。しかし、(歴史的な理由で)IANA時間帯データベースにこれらの名前が記録されているため、実際には名前付き時間帯として

扱われます。これの実際の影響は、明白なPOSIX仕様が提供されない場合でも、これらのゾーン名が有効な歴史的なアメリカ合衆国の夏時間の変換を提供することです。

ゾーンの省略形は妥当性をチェックされていないため、POSIX形式の時間帯指定はスペルミスしやすいことに注意してください。例えば、SET TIMEZONE TO FOOBAR0は動作しますが、実質的にシステムはUTCの特殊な省略形を使用します。

B.6. 単位の歴史

標準SQLでは、「「日付時刻リテラル」定義の中で、「日付時刻の値」はグレゴリオ暦に従った日付と時間の自然法則に則る」と明記されています。PostgreSQLは標準SQLの指針に従い、グレゴリオ暦が使われる以前の年に対してもグレゴリオ暦で日付を数えます。この規則は先発グレゴリオ暦として知られています。

ユリウス暦は、紀元前45年にユリウス・カエサル（Julius Caesar）によって広められたものです。西欧でグレゴリオ暦への移行が開始された1582年まで一般的に使用されていました。ユリウス暦では、太陽年は365日+1/4日=365.25日と概算されます。この暦では、128年で約1日のずれが生じます。

ローマ教皇グレゴリウス13世（Gregory XIII）はトレントの公会議（Council of Trent）の勧告に従って累積していた暦のずれを修正しました。グレゴリオ暦では、太陽年は365+97/400日=365.2425日と近似されます。したがって、グレゴリオ暦で太陽年が1日ずれるにはおよそ3,300年を要します。

365+97/400という近似は、下記の規則に従って400年間に97回のうるう年を設けることによって得られています。

4で割り切れる年を、うるう年にする。
ただし、100で割り切れる年は、うるう年にしない。
ただし、400で割り切れる年は、結局うるう年とする。

したがって、1700、1800、1900、2100、2200はうるう年ではありませんが、2000、2400はうるう年です。それに比べ、古いユリウス暦では4で割り切れる年のみがうるう年でした。

1582年2月の教皇勅書は、1582年の10月から10日間除外することを命じ、したがって10月4日の翌日を10月15日としました。この慣行はイタリア、ポーランド、ポルトガル、スペインで遵守されました。他のカトリックの国々もすぐ後に追従しましたが、プロテスタントの国々は変更を嫌がり、ギリシャ正教を信奉する国々は20世紀の初めまで変更を行いませんでした。1752年に大英帝国とその自治領（現在のアメリカ合衆国を含む）でもその改革は行われました。したがって、1752年9月2日の次は1752年9月14日となっています。このような理由から、Unixシステムでcalプログラムを実行すると、下記のような結果になります。

```
$ cal 9 1752
   September 1752
 S  M Tu  W Th  F  S
       1  2 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
```

しかし、当然、この暦は大英帝国とその自治領でのみ有効なものであり、他の場所ではそうではありません。様々な場所で様々な時代に使われていた実際の暦を追いかけることは難しく、また、混乱することになるで

しょうから、PostgreSQLは追いかけることはせず、この方法が歴史的に正確でなくても日付すべてでグレゴリオ暦の規則に従います。

グレゴリオ暦が開発されるより前から、様々な暦が世界の多くの地域で開発されました。例えば、中国暦のルーツは紀元前14世紀まで遡ることができますし、伝説では、黄帝が紀元前2637年にこの暦を発明したとされています。中華人民共和国では、公的な目的ではグレゴリオ暦を使用していますが、祝祭日の決定には中国暦を使用します。

ユリウス日は別の種類の暦であり、名前が似ていて混乱しますが、ユリウス暦とは関係ありません。ユリウス日は、フランスの学者Joseph Justus Scaliger（1540-1609）によって発明され、おそらくこの語源は彼の父であるイタリアの学者、Julius Caesar Scaliger（1484-1558）からの引用と考えられます。ユリウス日システムでは、JD 0（よくいわゆるユリウス日と呼ばれます）から始まる日は連番です。JD 0はユリウス暦の紀元前4713年1月1日、またはグレゴリオ暦の紀元前4714年11月24日に対応します。ユリウス日の数え方は、天文学者が夜間の観測にラベルを付けるためにより良く使用されました。このため、深夜0時から深夜0時までではなく、UTC正午から次のUTC正午までが1日でした。JD 0は紀元前4714年11月24日のUTC正午から紀元前4714年11月25日のUTC正午までの24時間であると明示されています。

PostgreSQLは日付の入出力においてユリウス日記法を（および、一部の日付時間間隔の計算においてユリウス日を）サポートしますが、正午から正午までという微妙な日付の数え方を守っていません。PostgreSQLは深夜0時から深夜0時までとしてユリウス日を扱います。

付録C SQLキーワード

表 C.1に標準SQLとPostgreSQL 13.1においてキーワードとされているすべてのトークンを示します。この背景となる情報は4.1.1にあります。（スペースの都合上、標準SQLのより新しい2つのバージョンと歴史的な比較のためのSQL-92のみを含めています。これらと他の標準の中間のバージョンの違いはわずかです。）

SQLは予約と未予約キーワードを区別します。標準に従うと予約キーワードのみが真のキーワードとなり、その予約キーワードは識別子として使用することはできません。未予約キーワードはある特定の文脈で特別な意味を持ち、また、その他の文脈では識別子として使用することができます。ほとんどの未予約キーワードは、SQLで規定された組み込みのテーブル名と関数名です。未予約キーワードの本質的な存在意義は、ある文脈においては前もって定義された意味があることを宣言することだけです。

PostgreSQLの構文解析の過程では、少々複雑になります。複数の異なったトークンのクラスがあり、それらは識別子としてまったく使用できないものから、普通の識別子と比較して、パーサ内で特別なステータスを持たないものまであります。（後者は一般的にはSQLで規定された関数です。）PostgreSQLでは予約キーワードは完全に予約されたものではなく、列ラベルとして使用することができます（例：CHECKは予約キーワードですが、SELECT 55 AS CHECKのようにすることができます）。

表 C.1のPostgreSQLの列では、パーサがはっきりと理解しているキーワードではあるが、列名やテーブル名としても使用できるものを「未予約」としています。キーワードの中には、どちらかといえば未予約であるが、関数名や型名として使用できないものもあり、そのように注記しています。（このようなキーワードのほとんどは、特殊な構文をもった組み込み済みの関数やデータ型を表しています。この関数または型は使用することができますが、ユーザによって定義し直すことはできません。）一方、「予約」とされるものは列名やテーブル名として使用できないトークンです。予約キーワードの中には関数名やデータ型として使用できるものもあります。この情報も以下の表に示しています。何も注記がなければ、予約キーワードは「AS」の列ラベル名としてしか使用することができません。

一般的な規則として、以下に示すキーワードのいずれかを識別子として含むコマンドで、おかしなパーサエラーが発生した場合、その識別子を引用符でくくって問題が解決するかどうか確認してください。

表 C.1を見る前に、PostgreSQLにおいて予約されていないキーワードが、その単語に関連する機能を実装していないということを意味していないことを理解しておいてください。逆に、キーワードがあるということも機能が存在することを意味していません。

表C.1 SQLキーワード

キーワード	PostgreSQL	SQL:2016	SQL:2011	SQL-92
A		未予約	未予約	
ABORT	未予約			
ABS		予約	予約	
ABSENT		未予約	未予約	
ABSOLUTE	未予約	未予約	未予約	予約
ACCESS	未予約			
ACCORDING		未予約	未予約	
ACOS		予約		
ACTION	未予約	未予約	未予約	予約

キーワード	PostgreSQL	SQL:2016	SQL:2011	SQL-92
ADA		未予約	未予約	未予約
ADD	未予約	未予約	未予約	予約
ADMIN	未予約	未予約	未予約	
AFTER	未予約	未予約	未予約	
AGGREGATE	未予約			
ALL	予約	予約	予約	予約
ALLOCATE		予約	予約	予約
ALSO	未予約			
ALTER	未予約	予約	予約	予約
ALWAYS	未予約	未予約	未予約	
ANALYSE	予約			
ANALYZE	予約			
AND	予約	予約	予約	予約
ANY	予約	予約	予約	予約
ARE		予約	予約	予約
ARRAY	予約	予約	予約	
ARRAY_AGG		予約	予約	
ARRAY_MAX_CARDINALITY		予約	予約	
AS	予約	予約	予約	予約
ASC	予約	未予約	未予約	予約
ASENSITIVE		予約	予約	
ASIN		予約		
ASSERTION	未予約	未予約	未予約	予約
ASSIGNMENT	未予約	未予約	未予約	
ASYMMETRIC	予約	予約	予約	
AT	未予約	予約	予約	予約
ATAN		予約		
ATOMIC		予約	予約	
ATTACH	未予約			
ATTRIBUTE	未予約	未予約	未予約	
ATTRIBUTES		未予約	未予約	
AUTHORIZATION	予約(関数または型として使用可)	予約	予約	予約
AVG		予約	予約	予約
BACKWARD	未予約			

キーワード	PostgreSQL	SQL:2016	SQL:2011	SQL-92
BASE64		未予約	未予約	
BEFORE	未予約	未予約	未予約	
BEGIN	未予約	予約	予約	予約
BEGIN_FRAME		予約	予約	
BEGIN_PARTITION		予約	予約	
BERNOULLI		未予約	未予約	
BETWEEN	未予約(関数または型として使用不可)	予約	予約	予約
BIGINT	未予約(関数または型として使用不可)	予約	予約	
BINARY	予約(関数または型として使用可)	予約	予約	
BIT	未予約(関数または型として使用不可)			予約
BIT_LENGTH				予約
BLOB		予約	予約	
BLOCKED		未予約	未予約	
BOM		未予約	未予約	
BOOLEAN	未予約(関数または型として使用不可)	予約	予約	
BOTH	予約	予約	予約	予約
BREADTH		未予約	未予約	
BY	未予約	予約	予約	予約
C		未予約	未予約	未予約
CACHE	未予約			
CALL	未予約	予約	予約	
CALLED	未予約	予約	予約	
CARDINALITY		予約	予約	
CASCADE	未予約	未予約	未予約	予約
CASCADED	未予約	予約	予約	予約
CASE	予約	予約	予約	予約
CAST	予約	予約	予約	予約
CATALOG	未予約	未予約	未予約	予約
CATALOG_NAME		未予約	未予約	未予約
CEIL		予約	予約	

キーワード	PostgreSQL	SQL:2016	SQL:2011	SQL-92
CEILING		予約	予約	
CHAIN	未予約	未予約	未予約	
CHAINING		未予約		
CHAR	未予約(関数または型として使用不可)	予約	予約	予約
CHARACTER	未予約(関数または型として使用不可)	予約	予約	予約
CHARACTERISTICS	未予約	未予約	未予約	
CHARACTERS		未予約	未予約	
CHARACTER_LENGTH		予約	予約	予約
CHARACTER_SET_CATALOG		未予約	未予約	未予約
CHARACTER_SET_NAME		未予約	未予約	未予約
CHARACTER_SET_SCHEMA		未予約	未予約	未予約
CHAR_LENGTH		予約	予約	予約
CHECK	予約	予約	予約	予約
CHECKPOINT	未予約			
CLASS	未予約			
CLASSIFIER		予約		
CLASS_ORIGIN		未予約	未予約	未予約
CLOB		予約	予約	
CLOSE	未予約	予約	予約	予約
CLUSTER	未予約			
COALESCE	未予約(関数または型として使用不可)	予約	予約	予約
COBOL		未予約	未予約	未予約
COLLATE	予約	予約	予約	予約
COLLATION	予約(関数または型として使用可)	未予約	未予約	予約
COLLATION_CATALOG		未予約	未予約	未予約
COLLATION_NAME		未予約	未予約	未予約
COLLATION_SCHEMA		未予約	未予約	未予約
COLLECT		予約	予約	
COLUMN	予約	予約	予約	予約
COLUMNS	未予約	未予約	未予約	
COLUMN_NAME		未予約	未予約	未予約

キーワード	PostgreSQL	SQL:2016	SQL:2011	SQL-92
COMMAND_FUNCTION		未予約	未予約	未予約
COMMAND_FUNCTION_CODE		未予約	未予約	
COMMENT	未予約			
COMMENTS	未予約			
COMMIT	未予約	予約	予約	予約
COMMITTED	未予約	未予約	未予約	未予約
CONCURRENTLY	予約(関数または型として使用可)			
CONDITION		予約	予約	
CONDITIONAL		未予約		
CONDITION_NUMBER		未予約	未予約	未予約
CONFIGURATION	未予約			
CONFLICT	未予約			
CONNECT		予約	予約	予約
CONNECTION	未予約	未予約	未予約	予約
CONNECTION_NAME		未予約	未予約	未予約
CONSTRAINT	予約	予約	予約	予約
CONSTRAINTS	未予約	未予約	未予約	予約
CONSTRAINT_CATALOG		未予約	未予約	未予約
CONSTRAINT_NAME		未予約	未予約	未予約
CONSTRAINT_SCHEMA		未予約	未予約	未予約
CONSTRUCTOR		未予約	未予約	
CONTAINS		予約	予約	
CONTENT	未予約	未予約	未予約	
CONTINUE	未予約	未予約	未予約	予約
CONTROL		未予約	未予約	
CONVERSION	未予約			
CONVERT		予約	予約	予約
COPY	未予約	予約		
CORR		予約	予約	
CORRESPONDING		予約	予約	予約
COS		予約		
COSH		予約		
COST	未予約			
COUNT		予約	予約	予約

キーワード	PostgreSQL	SQL:2016	SQL:2011	SQL-92
COVAR_POP		予約	予約	
COVAR_SAMP		予約	予約	
CREATE	予約	予約	予約	予約
CROSS	予約(関数または型として使用可)	予約	予約	予約
CSV	未予約			
CUBE	未予約	予約	予約	
CUME_DIST		予約	予約	
CURRENT	未予約	予約	予約	予約
CURRENT_CATALOG	予約	予約	予約	
CURRENT_DATE	予約	予約	予約	予約
CURRENT_DEFAULT_TRANSFORM_GROUP		予約	予約	
CURRENT_PATH		予約	予約	
CURRENT_ROLE	予約	予約	予約	
CURRENT_ROW		予約	予約	
CURRENT_SCHEMA	予約(関数または型として使用可)	予約	予約	
CURRENT_TIME	予約	予約	予約	予約
CURRENT_TIMESTAMP	予約	予約	予約	予約
CURRENT_TRANSFORM_GROUP_FOR_TYPE		予約	予約	
CURRENT_USER	予約	予約	予約	予約
CURSOR	未予約	予約	予約	予約
CURSOR_NAME		未予約	未予約	未予約
CYCLE	未予約	予約	予約	
DATA	未予約	未予約	未予約	未予約
DATABASE	未予約			
DATALINK		予約	予約	
DATE		予約	予約	予約
DATETIME_INTERVAL_CODE		未予約	未予約	未予約
DATETIME_INTERVAL_PRECISION		未予約	未予約	未予約
DAY	未予約	予約	予約	予約
DB		未予約	未予約	
DEALLOCATE	未予約	予約	予約	予約
DEC	未予約(関数または型として使用不可)	予約	予約	予約

キーワード	PostgreSQL	SQL:2016	SQL:2011	SQL-92
DECFLOAT		予約		
DECIMAL	未予約(関数または型として使用不可)	予約	予約	予約
DECLARE	未予約	予約	予約	予約
DEFAULT	予約	予約	予約	予約
DEFAULTS	未予約	未予約	未予約	
DEFERRABLE	予約	未予約	未予約	予約
DEFERRED	未予約	未予約	未予約	予約
DEFINE		予約		
DEFINED		未予約	未予約	
DEFINER	未予約	未予約	未予約	
DEGREE		未予約	未予約	
DELETE	未予約	予約	予約	予約
DELIMITER	未予約			
DELIMITERS	未予約			
DENSE_RANK		予約	予約	
DEPENDS	未予約			
DEPTH		未予約	未予約	
DEREF		予約	予約	
DERIVED		未予約	未予約	
DESC	予約	未予約	未予約	予約
DESCRIBE		予約	予約	予約
DESCRIPTOR		未予約	未予約	予約
DETACH	未予約			
DETERMINISTIC		予約	予約	
DIAGNOSTICS		未予約	未予約	予約
DICTIONARY	未予約			
DISABLE	未予約			
DISCARD	未予約			
DISCONNECT		予約	予約	予約
DISPATCH		未予約	未予約	
DISTINCT	予約	予約	予約	予約
DLNEWCOPY		予約	予約	
DLPREVIOUSCOPY		予約	予約	
DLURLCOMPLETE		予約	予約	

キーワード	PostgreSQL	SQL:2016	SQL:2011	SQL-92
DLURLCOMPLETEONLY		予約	予約	
DLURLCOMPLETEWRITE		予約	予約	
DLURLPATH		予約	予約	
DLURLPATHONLY		予約	予約	
DLURLPATHWRITE		予約	予約	
DLURLSCHEME		予約	予約	
DLURLSERVER		予約	予約	
DLVALUE		予約	予約	
DO	予約			
DOCUMENT	未予約	未予約	未予約	
DOMAIN	未予約	未予約	未予約	予約
DOUBLE	未予約	予約	予約	予約
DROP	未予約	予約	予約	予約
DYNAMIC		予約	予約	
DYNAMIC_FUNCTION		未予約	未予約	未予約
DYNAMIC_FUNCTION_CODE		未予約	未予約	
EACH	未予約	予約	予約	
ELEMENT		予約	予約	
ELSE	予約	予約	予約	予約
EMPTY		予約	未予約	
ENABLE	未予約			
ENCODING	未予約	未予約	未予約	
ENCRYPTED	未予約			
END	予約	予約	予約	予約
END-EXEC		予約	予約	予約
END_FRAME		予約	予約	
END_PARTITION		予約	予約	
ENFORCED		未予約	未予約	
ENUM	未予約			
EQUALS		予約	予約	
ERROR		未予約		
ESCAPE	未予約	予約	予約	予約
EVENT	未予約			
EVERY		予約	予約	

キーワード	PostgreSQL	SQL:2016	SQL:2011	SQL-92
EXCEPT	予約	予約	予約	予約
EXCEPTION				予約
EXCLUDE	未予約	未予約	未予約	
EXCLUDING	未予約	未予約	未予約	
EXCLUSIVE	未予約			
EXEC		予約	予約	予約
EXECUTE	未予約	予約	予約	予約
EXISTS	未予約(関数または型として使用不可)	予約	予約	予約
EXP		予約	予約	
EXPLAIN	未予約			
EXPRESSION	未予約	未予約	未予約	
EXTENSION	未予約			
EXTERNAL	未予約	予約	予約	予約
EXTRACT	未予約(関数または型として使用不可)	予約	予約	予約
FALSE	予約	予約	予約	予約
FAMILY	未予約			
FETCH	予約	予約	予約	予約
FILE		未予約	未予約	
FILTER	未予約	予約	予約	
FINAL		未予約	未予約	
FINISH		未予約		
FIRST	未予約	未予約	未予約	予約
FIRST_VALUE		予約	予約	
FLAG		未予約	未予約	
FLOAT	未予約(関数または型として使用不可)	予約	予約	予約
FLOOR		予約	予約	
FOLLOWING	未予約	未予約	未予約	
FOR	予約	予約	予約	予約
FORCE	未予約			
FOREIGN	予約	予約	予約	予約
FORMAT		未予約		
FORTRAN		未予約	未予約	未予約

キーワード	PostgreSQL	SQL:2016	SQL:2011	SQL-92
FORWARD	未予約			
FOUND		未予約	未予約	予約
FRAME_ROW		予約	予約	
FREE		予約	予約	
FREEZE	予約(関数または型として使用可)			
FROM	予約	予約	予約	予約
FS		未予約	未予約	
FULFILL		未予約		
FULL	予約(関数または型として使用可)	予約	予約	予約
FUNCTION	未予約	予約	予約	
FUNCTIONS	未予約			
FUSION		予約	予約	
G		未予約	未予約	
GENERAL		未予約	未予約	
GENERATED	未予約	未予約	未予約	
GET		予約	予約	予約
GLOBAL	未予約	予約	予約	予約
GO		未予約	未予約	予約
GOTO		未予約	未予約	予約
GRANT	予約	予約	予約	予約
GRANTED	未予約	未予約	未予約	
GREATEST	未予約(関数または型として使用不可)			
GROUP	予約	予約	予約	予約
GROUPING	未予約(関数または型として使用不可)	予約	予約	
GROUPS	未予約	予約	予約	
HANDLER	未予約			
HAVING	予約	予約	予約	予約
HEADER	未予約			
HEX		未予約	未予約	
HIERARCHY		未予約	未予約	
HOLD	未予約	予約	予約	

キーワード	PostgreSQL	SQL:2016	SQL:2011	SQL-92
HOUR	未予約	予約	予約	予約
ID		未予約	未予約	
IDENTITY	未予約	予約	予約	予約
IF	未予約			
IGNORE		未予約	未予約	
ILIKE	予約(関数または型として使用可)			
IMMEDIATE	未予約	未予約	未予約	予約
IMMEDIATELY		未予約	未予約	
IMMUTABLE	未予約			
IMPLEMENTATION		未予約	未予約	
IMPLICIT	未予約			
IMPORT	未予約	予約	予約	
IN	予約	予約	予約	予約
INCLUDE	未予約			
INCLUDING	未予約	未予約	未予約	
INCREMENT	未予約	未予約	未予約	
INDENT		未予約	未予約	
INDEX	未予約			
INDEXES	未予約			
INDICATOR		予約	予約	予約
INHERIT	未予約			
INHERITS	未予約			
INITIAL		予約		
INITIALLY	予約	未予約	未予約	予約
INLINE	未予約			
INNER	予約(関数または型として使用可)	予約	予約	予約
INOUT	未予約(関数または型として使用不可)	予約	予約	
INPUT	未予約	未予約	未予約	予約
INSENSITIVE	未予約	予約	予約	予約
INSERT	未予約	予約	予約	予約
INSTANCE		未予約	未予約	
INSTANTIABLE		未予約	未予約	

キーワード	PostgreSQL	SQL:2016	SQL:2011	SQL-92
INSTEAD	未予約	未予約	未予約	
INT	未予約(関数または型として使用不可)	予約	予約	予約
INTEGER	未予約(関数または型として使用不可)	予約	予約	予約
INTEGRITY		未予約	未予約	
INTERSECT	予約	予約	予約	予約
INTERSECTION		予約	予約	
INTERVAL	未予約(関数または型として使用不可)	予約	予約	予約
INTO	予約	予約	予約	予約
INVOKER	未予約	未予約	未予約	
IS	予約(関数または型として使用可)	予約	予約	予約
ISNULL	予約(関数または型として使用可)			
ISOLATION	未予約	未予約	未予約	予約
JOIN	予約(関数または型として使用可)	予約	予約	予約
JSON		未予約		
JSON_ARRAY		予約		
JSON_ARRAYAGG		予約		
JSON_EXISTS		予約		
JSON_OBJECT		予約		
JSON_OBJECTAGG		予約		
JSON_QUERY		予約		
JSON_TABLE		予約		
JSON_TABLE_PRIMITIVE		予約		
JSON_VALUE		予約		
K		未予約	未予約	
KEEP		未予約		
KEY	未予約	未予約	未予約	予約
KEYS		未予約		
KEY_MEMBER		未予約	未予約	
KEY_TYPE		未予約	未予約	
LABEL	未予約			

キーワード	PostgreSQL	SQL:2016	SQL:2011	SQL-92
LAG		予約	予約	
LANGUAGE	未予約	予約	予約	予約
LARGE	未予約	予約	予約	
LAST	未予約	未予約	未予約	予約
LAST_VALUE		予約	予約	
LATERAL	予約	予約	予約	
LEAD		予約	予約	
LEADING	予約	予約	予約	予約
LEAKPROOF	未予約			
LEAST	未予約(関数または型として使用不可)			
LEFT	予約(関数または型として使用可)	予約	予約	予約
LENGTH		未予約	未予約	未予約
LEVEL	未予約	未予約	未予約	予約
LIBRARY		未予約	未予約	
LIKE	予約(関数または型として使用可)	予約	予約	予約
LIKE_REGEX		予約	予約	
LIMIT	予約	未予約	未予約	
LINK		未予約	未予約	
LISTAGG		予約		
LISTEN	未予約			
LN		予約	予約	
LOAD	未予約			
LOCAL	未予約	予約	予約	予約
LOCALTIME	予約	予約	予約	
LOCALTIMESTAMP	予約	予約	予約	
LOCATION	未予約	未予約	未予約	
LOCATOR		未予約	未予約	
LOCK	未予約			
LOCKED	未予約			
LOG		予約		
LOG10		予約		
LOGGED	未予約			

キーワード	PostgreSQL	SQL:2016	SQL:2011	SQL-92
LOWER		予約	予約	予約
M		未予約	未予約	
MAP		未予約	未予約	
MAPPING	未予約	未予約	未予約	
MATCH	未予約	予約	予約	予約
MATCHED		未予約	未予約	
MATCHES		予約		
MATCH_NUMBER		予約		
MATCH_RECOGNIZE		予約		
MATERIALIZED	未予約			
MAX		予約	予約	予約
MAXVALUE	未予約	未予約	未予約	
MEASURES		予約		
MEMBER		予約	予約	
MERGE		予約	予約	
MESSAGE_LENGTH		未予約	未予約	未予約
MESSAGE_OCTET_LENGTH		未予約	未予約	未予約
MESSAGE_TEXT		未予約	未予約	未予約
METHOD	未予約	予約	予約	
MIN		予約	予約	予約
MINUTE	未予約	予約	予約	予約
MINVALUE	未予約	未予約	未予約	
MOD		予約	予約	
MODE	未予約			
MODIFIES		予約	予約	
MODULE		予約	予約	予約
MONTH	未予約	予約	予約	予約
MORE		未予約	未予約	未予約
MOVE	未予約			
MULTISET		予約	予約	
MUMPS		未予約	未予約	未予約
NAME	未予約	未予約	未予約	未予約
NAMES	未予約	未予約	未予約	予約
NAMESPACE		未予約	未予約	

キーワード	PostgreSQL	SQL:2016	SQL:2011	SQL-92
NATIONAL	未予約(関数または型として使用不可)	予約	予約	予約
NATURAL	予約(関数または型として使用可)	予約	予約	予約
NCHAR	未予約(関数または型として使用不可)	予約	予約	予約
NCLOB		予約	予約	
NESTED		未予約		
NESTING		未予約	未予約	
NEW	未予約	予約	予約	
NEXT	未予約	未予約	未予約	予約
NFC	未予約	未予約	未予約	
NFD	未予約	未予約	未予約	
NFKC	未予約	未予約	未予約	
NFKD	未予約	未予約	未予約	
NIL		未予約	未予約	
NO	未予約	予約	予約	予約
NONE	未予約(関数または型として使用不可)	予約	予約	
NORMALIZE	未予約(関数または型として使用不可)	予約	予約	
NORMALIZED	未予約	未予約	未予約	
NOT	予約	予約	予約	予約
NOTHING	未予約			
NOTIFY	未予約			
NOTNULL	予約(関数または型として使用可)			
NOWAIT	未予約			
NTH_VALUE		予約	予約	
NTILE		予約	予約	
NULL	予約	予約	予約	予約
NULLABLE		未予約	未予約	未予約
NULLIF	未予約(関数または型として使用不可)	予約	予約	予約
NULLS	未予約	未予約	未予約	
NUMBER		未予約	未予約	未予約

キーワード	PostgreSQL	SQL:2016	SQL:2011	SQL-92
NUMERIC	未予約(関数または型として使用不可)	予約	予約	予約
OBJECT	未予約	未予約	未予約	
OCCURRENCES_REGEX		予約	予約	
OCTETS		未予約	未予約	
OCTET_LENGTH		予約	予約	予約
OF	未予約	予約	予約	予約
OFF	未予約	未予約	未予約	
OFFSET	予約	予約	予約	
OIDS	未予約			
OLD	未予約	予約	予約	
OMIT		予約		
ON	予約	予約	予約	予約
ONE		予約		
ONLY	予約	予約	予約	予約
OPEN		予約	予約	予約
OPERATOR	未予約			
OPTION	未予約	未予約	未予約	予約
OPTIONS	未予約	未予約	未予約	
OR	予約	予約	予約	予約
ORDER	予約	予約	予約	予約
ORDERING		未予約	未予約	
ORDINALITY	未予約	未予約	未予約	
OTHERS	未予約	未予約	未予約	
OUT	未予約(関数または型として使用不可)	予約	予約	
OUTER	予約(関数または型として使用可)	予約	予約	予約
OUTPUT		未予約	未予約	予約
OVER	未予約	予約	予約	
OVERFLOW		未予約		
OVERLAPS	予約(関数または型として使用可)	予約	予約	予約
OVERLAY	未予約(関数または型として使用不可)	予約	予約	

キーワード	PostgreSQL	SQL:2016	SQL:2011	SQL-92
OVERRIDING	未予約	未予約	未予約	
OWNED	未予約			
OWNER	未予約			
P		未予約	未予約	
PAD		未予約	未予約	予約
PARALLEL	未予約			
PARAMETER		予約	予約	
PARAMETER_MODE		未予約	未予約	
PARAMETER_NAME		未予約	未予約	
PARAMETER_ORDINAL_POSITION		未予約	未予約	
PARAMETER_SPECIFIC_CATALOG		未予約	未予約	
PARAMETER_SPECIFIC_NAME		未予約	未予約	
PARAMETER_SPECIFIC_SCHEMA		未予約	未予約	
PARSER	未予約			
PARTIAL	未予約	未予約	未予約	予約
PARTITION	未予約	予約	予約	
PASCAL		未予約	未予約	未予約
PASS		未予約		
PASSING	未予約	未予約	未予約	
PASSTHROUGH		未予約	未予約	
PASSWORD	未予約			
PAST		未予約		
PATH		未予約	未予約	
PATTERN		予約		
PER		予約		
PERCENT		予約	予約	
PERCENTILE_CONT		予約	予約	
PERCENTILE_DISC		予約	予約	
PERCENT_RANK		予約	予約	
PERIOD		予約	予約	
PERMISSION		未予約	未予約	
PERMUTE		予約		
PLACING	予約	未予約	未予約	
PLAN		未予約		

キーワード	PostgreSQL	SQL:2016	SQL:2011	SQL-92
PLANS	未予約			
PLI		未予約	未予約	未予約
POLICY	未予約			
PORTION		予約	予約	
POSITION	未予約(関数または型として使用不可)	予約	予約	予約
POSITION_REGEX		予約	予約	
POWER		予約	予約	
PRECEDES		予約	予約	
PRECEDING	未予約	未予約	未予約	
PRECISION	未予約(関数または型として使用不可)	予約	予約	予約
PREPARE	未予約	予約	予約	予約
PREPARED	未予約			
PRESERVE	未予約	未予約	未予約	予約
PRIMARY	予約	予約	予約	予約
PRIOR	未予約	未予約	未予約	予約
PRIVATE		未予約		
PRIVILEGES	未予約	未予約	未予約	予約
PROCEDURAL	未予約			
PROCEDURE	未予約	予約	予約	予約
PROCEDURES	未予約			
PROGRAM	未予約			
PRUNE		未予約		
PTF		予約		
PUBLIC		未予約	未予約	予約
PUBLICATION	未予約			
QUOTE	未予約			
QUOTES		未予約		
RANGE	未予約	予約	予約	
RANK		予約	予約	
READ	未予約	未予約	未予約	予約
READS		予約	予約	
REAL	未予約(関数または型として使用不可)	予約	予約	予約

キーワード	PostgreSQL	SQL:2016	SQL:2011	SQL-92
REASSIGN	未予約			
RECHECK	未予約			
RECOVERY		未予約	未予約	
RECURSIVE	未予約	予約	予約	
REF	未予約	予約	予約	
REFERENCES	予約	予約	予約	予約
REFERENCING	未予約	予約	予約	
REFRESH	未予約			
REGR_AVGX		予約	予約	
REGR_AVGY		予約	予約	
REGR_COUNT		予約	予約	
REGR_INTERCEPT		予約	予約	
REGR_R2		予約	予約	
REGR_SLOPE		予約	予約	
REGR_SXX		予約	予約	
REGR_SXY		予約	予約	
REGR_SYY		予約	予約	
REINDEX	未予約			
RELATIVE	未予約	未予約	未予約	予約
RELEASE	未予約	予約	予約	
RENAME	未予約			
REPEATABLE	未予約	未予約	未予約	未予約
REPLACE	未予約			
REPLICA	未予約			
REQUIRING		未予約	未予約	
RESET	未予約			
RESPECT		未予約	未予約	
RESTART	未予約	未予約	未予約	
RESTORE		未予約	未予約	
RESTRICT	未予約	未予約	未予約	予約
RESULT		予約	予約	
RETURN		予約	予約	
RETURNED_CARDINALITY		未予約	未予約	
RETURNED_LENGTH		未予約	未予約	未予約

キーワード	PostgreSQL	SQL:2016	SQL:2011	SQL-92
RETURNED_OCTET_LENGTH		未予約	未予約	未予約
RETURNED_SQLSTATE		未予約	未予約	未予約
RETURNING	予約	未予約	未予約	
RETURNS	未予約	予約	予約	
REVOKE	未予約	予約	予約	予約
RIGHT	予約(関数または型として使用可)	予約	予約	予約
ROLE	未予約	未予約	未予約	
ROLLBACK	未予約	予約	予約	予約
ROLLUP	未予約	予約	予約	
ROUTINE	未予約	未予約	未予約	
ROUTINES	未予約			
ROUTINE_CATALOG		未予約	未予約	
ROUTINE_NAME		未予約	未予約	
ROUTINE_SCHEMA		未予約	未予約	
ROW	未予約(関数または型として使用不可)	予約	予約	
ROWS	未予約	予約	予約	予約
ROW_COUNT		未予約	未予約	未予約
ROW_NUMBER		予約	予約	
RULE	未予約			
RUNNING		予約		
SAVEPOINT	未予約	予約	予約	
SCALAR		未予約		
SCALE		未予約	未予約	未予約
SCHEMA	未予約	未予約	未予約	予約
SCHEMAS	未予約			
SCHEMA_NAME		未予約	未予約	未予約
SCOPE		予約	予約	
SCOPE_CATALOG		未予約	未予約	
SCOPE_NAME		未予約	未予約	
SCOPE_SCHEMA		未予約	未予約	
SCROLL	未予約	予約	予約	予約
SEARCH	未予約	予約	予約	
SECOND	未予約	予約	予約	予約

キーワード	PostgreSQL	SQL:2016	SQL:2011	SQL-92
SECTION		未予約	未予約	予約
SECURITY	未予約	未予約	未予約	
SEEK		予約		
SELECT	予約	予約	予約	予約
SELECTIVE		未予約	未予約	
SELF		未予約	未予約	
SENSITIVE		予約	予約	
SEQUENCE	未予約	未予約	未予約	
SEQUENCES	未予約			
SERIALIZABLE	未予約	未予約	未予約	未予約
SERVER	未予約	未予約	未予約	
SERVER_NAME		未予約	未予約	未予約
SESSION	未予約	未予約	未予約	予約
SESSION_USER	予約	予約	予約	予約
SET	未予約	予約	予約	予約
SETOF	未予約(関数または型として使用不可)			
SETS	未予約	未予約	未予約	
SHARE	未予約			
SHOW	未予約	予約		
SIMILAR	予約(関数または型として使用可)	予約	予約	
SIMPLE	未予約	未予約	未予約	
SIN		予約		
SINH		予約		
SIZE		未予約	未予約	予約
SKIP	未予約	予約		
SMALLINT	未予約(関数または型として使用不可)	予約	予約	予約
SNAPSHOT	未予約			
SOME	予約	予約	予約	予約
SOURCE		未予約	未予約	
SPACE		未予約	未予約	予約
SPECIFIC		予約	予約	
SPECIFICTYPE		予約	予約	

キーワード	PostgreSQL	SQL:2016	SQL:2011	SQL-92
SPECIFIC_NAME		未予約	未予約	
SQL	未予約	予約	予約	予約
SQLCODE				予約
SQLERROR				予約
SQLEXCEPTION		予約	予約	
SQLSTATE		予約	予約	予約
SQLWARNING		予約	予約	
SQRT		予約	予約	
STABLE	未予約			
STANDALONE	未予約	未予約	未予約	
START	未予約	予約	予約	
STATE		未予約	未予約	
STATEMENT	未予約	未予約	未予約	
STATIC		予約	予約	
STATISTICS	未予約			
STDDEV_POP		予約	予約	
STDDEV_SAMP		予約	予約	
STDIN	未予約			
STDOUT	未予約			
STORAGE	未予約			
STORED	未予約			
STRICT	未予約			
STRING		未予約		
STRIP	未予約	未予約	未予約	
STRUCTURE		未予約	未予約	
STYLE		未予約	未予約	
SUBCLASS_ORIGIN		未予約	未予約	未予約
SUBMULTISET		予約	予約	
SUBSCRIPTION	未予約			
SUBSET		予約		
SUBSTRING	未予約(関数または型として使用不可)	予約	予約	予約
SUBSTRING_REGEX		予約	予約	
SUCCEEDS		予約	予約	
SUM		予約	予約	予約

キーワード	PostgreSQL	SQL:2016	SQL:2011	SQL-92
SUPPORT	未予約			
SYMMETRIC	予約	予約	予約	
SYSID	未予約			
SYSTEM	未予約	予約	予約	
SYSTEM_TIME		予約	予約	
SYSTEM_USER		予約	予約	予約
T		未予約	未予約	
TABLE	予約	予約	予約	予約
TABLES	未予約			
TABLESAMPLE	予約(関数または型として使用可)	予約	予約	
TABLESPACE	未予約			
TABLE_NAME		未予約	未予約	未予約
TAN		予約		
TANH		予約		
TEMP	未予約			
TEMPLATE	未予約			
TEMPORARY	未予約	未予約	未予約	予約
TEXT	未予約			
THEN	予約	予約	予約	予約
THROUGH		未予約		
TIES	未予約	未予約	未予約	
TIME	未予約(関数または型として使用不可)	予約	予約	予約
TIMESTAMP	未予約(関数または型として使用不可)	予約	予約	予約
TIMEZONE_HOUR		予約	予約	予約
TIMEZONE_MINUTE		予約	予約	予約
TO	予約	予約	予約	予約
TOKEN		未予約	未予約	
TOP_LEVEL_COUNT		未予約	未予約	
TRAILING	予約	予約	予約	予約
TRANSACTION	未予約	未予約	未予約	予約
TRANSACTIONS_COMMITTED		未予約	未予約	
TRANSACTIONS_ROLLED_BACK		未予約	未予約	

キーワード	PostgreSQL	SQL:2016	SQL:2011	SQL-92
TRANSACTION_ACTIVE		未予約	未予約	
TRANSFORM	未予約	未予約	未予約	
TRANSFORMS		未予約	未予約	
TRANSLATE		予約	予約	予約
TRANSLATE_REGEX		予約	予約	
TRANSLATION		予約	予約	予約
TREAT	未予約(関数または型として使用不可)	予約	予約	
TRIGGER	未予約	予約	予約	
TRIGGER_CATALOG		未予約	未予約	
TRIGGER_NAME		未予約	未予約	
TRIGGER_SCHEMA		未予約	未予約	
TRIM	未予約(関数または型として使用不可)	予約	予約	予約
TRIM_ARRAY		予約	予約	
TRUE	予約	予約	予約	予約
TRUNCATE	未予約	予約	予約	
TRUSTED	未予約			
TYPE	未予約	未予約	未予約	未予約
TYPES	未予約			
UESCAPE	未予約	予約	予約	
UNBOUNDED	未予約	未予約	未予約	
UNCOMMITTED	未予約	未予約	未予約	未予約
UNCONDITIONAL		未予約		
UNDER		未予約	未予約	
UNENCRYPTED	未予約			
UNION	予約	予約	予約	予約
UNIQUE	予約	予約	予約	予約
UNKNOWN	未予約	予約	予約	予約
UNLINK		未予約	未予約	
UNLISTEN	未予約			
UNLOGGED	未予約			
UNMATCHED		予約		
UNNAMED		未予約	未予約	未予約
UNNEST		予約	予約	

キーワード	PostgreSQL	SQL:2016	SQL:2011	SQL-92
UNTIL	未予約			
UNTYPED		未予約	未予約	
UPDATE	未予約	予約	予約	予約
UPPER		予約	予約	予約
URI		未予約	未予約	
USAGE		未予約	未予約	予約
USER	予約	予約	予約	予約
USER_DEFINED_TYPE_CATALOG		未予約	未予約	
USER_DEFINED_TYPE_CODE		未予約	未予約	
USER_DEFINED_TYPE_NAME		未予約	未予約	
USER_DEFINED_TYPE_SCHEMA		未予約	未予約	
USING	予約	予約	予約	予約
UTF16		未予約		
UTF32		未予約		
UTF8		未予約		
VACUUM	未予約			
VALID	未予約	未予約	未予約	
VALIDATE	未予約			
VALIDATOR	未予約			
VALUE	未予約	予約	予約	予約
VALUES	未予約(関数または型として使用不可)	予約	予約	予約
VALUE_OF		予約	予約	
VARBINARY		予約	予約	
VARCHAR	未予約(関数または型として使用不可)	予約	予約	予約
VARIADIC	予約			
VARYING	未予約	予約	予約	予約
VAR_POP		予約	予約	
VAR_SAMP		予約	予約	
VERBOSE	予約(関数または型として使用可)			
VERSION	未予約	未予約	未予約	
VERSIONING		予約	予約	
VIEW	未予約	未予約	未予約	予約

キーワード	PostgreSQL	SQL:2016	SQL:2011	SQL-92
VIEWS	未予約			
VOLATILE	未予約			
WHEN	予約	予約	予約	予約
WHENEVER		予約	予約	予約
WHERE	予約	予約	予約	予約
WHITESPACE	未予約	未予約	未予約	
WIDTH_BUCKET		予約	予約	
WINDOW	予約	予約	予約	
WITH	予約	予約	予約	予約
WITHIN	未予約	予約	予約	
WITHOUT	未予約	予約	予約	
WORK	未予約	未予約	未予約	予約
WRAPPER	未予約	未予約	未予約	
WRITE	未予約	未予約	未予約	予約
XML	未予約	予約	予約	
XMLAGG		予約	予約	
XMLATTRIBUTES	未予約(関数または型として使用不可)	予約	予約	
XMLBINARY		予約	予約	
XMLCAST		予約	予約	
XMLCOMMENT		予約	予約	
XMLCONCAT	未予約(関数または型として使用不可)	予約	予約	
XMLDECLARATION		未予約	未予約	
XMLDOCUMENT		予約	予約	
XMLELEMENT	未予約(関数または型として使用不可)	予約	予約	
XMLEXISTS	未予約(関数または型として使用不可)	予約	予約	
XMLFOREST	未予約(関数または型として使用不可)	予約	予約	
XMLITERATE		予約	予約	
XMLNAMESPACES	未予約(関数または型として使用不可)	予約	予約	
XMLPARSE	未予約(関数または型として使用不可)	予約	予約	

キーワード	PostgreSQL	SQL:2016	SQL:2011	SQL-92
XMLPI	未予約(関数または型として使用不可)	予約	予約	
XMLQUERY		予約	予約	
XMLROOT	未予約(関数または型として使用不可)			
XMLSCHEMA		未予約	未予約	
XMLSERIALIZE	未予約(関数または型として使用不可)	予約	予約	
XMLTABLE	未予約(関数または型として使用不可)	予約	予約	
XMLTEXT		予約	予約	
XMLVALIDATE		予約	予約	
YEAR	未予約	予約	予約	予約
YES	未予約	未予約	未予約	
ZONE	未予約	未予約	未予約	予約

付録D SQLへの準拠

本節では、PostgreSQLがどの程度現在の標準SQLに準拠しているかについて、概要を説明します。以下の情報は互換性についての完全な説明ではありません。しかし、ユーザにとって十分適切かつ有用な詳細を主な話題としてここで示しています。

標準SQLの公式な名称は、ISO/IEC 9075「Database Language SQL」です。この標準SQLの改訂バージョンは不定期にリリースされています。最も最近の改訂は2016年に行われました。この2016年版はISO/IEC 9075:2016、もしくは単にSQL:2016と呼ばれています。この前のバージョンはSQL:2011、SQL:2008、SQL:2006、SQL:2003、SQL:1999とSQL-92です。それぞれ前のバージョンを置き換えたバージョンですので、より昔のバージョンへの適合性についての主張には公的な利点がありません。PostgreSQLの開発では、伝統的な機能もしくは共通の考えと矛盾しないように、標準SQLの最新の公式バージョンに準拠させようとしています。標準SQLで必須とされた機能の多くは、多少構文や機能に違いはあるものの、サポートされています。さらに適合性を高めることが将来のリリースで期待されています。

SQL-92では、適合性について、Entry、Intermediate、Fullという3つの機能セットを定義しました。標準SQLの準拠をうたっているデータベース管理システムのほとんどは、Entryレベルのみに適合しています。IntermediateもしくはFullレベルの機能の全体的なセットは、非常に膨大になり過ぎるか、もしくは旧来の動作と競合するからです。

SQL:1999から、標準SQLは、SQL-92で見受けられた3レベルに機能を非効率的に分散させるのではなく、個々の機能を大規模な集合として定義されるようになりました。こうした機能の大規模なサブセットを、全てのSQL準拠の実装が提供しなければならない「コア」機能として表しています。残りの機能は完全に省略可能です。

SQL:2003版以降の標準はまた、数個の部品に分かれています。それぞれ省略した名前を持ちます。この部品に連続した番号が付いていないことに注意してください。

- ISO/IEC 9075-1 Framework (SQL/Framework)
- ISO/IEC 9075-2 Foundation (SQL/Foundation)
- ISO/IEC 9075-3 Call Level Interface (SQL/CLI)
- ISO/IEC 9075-4 Persistent Stored Modules (SQL/PSM)
- ISO/IEC 9075-9 Management of External Data (SQL/MED)
- ISO/IEC 9075-10 Object Language Bindings (SQL/OLB)
- ISO/IEC 9075-11 Information and Definition Schemas (SQL/Schemata)
- ISO/IEC 9075-13 Routines and Types using the Java Language (SQL/JRT)
- ISO/IEC 9075-14 XML-related specifications (SQL/XML)

- ISO/IEC 9075-15 Multi-dimensional arrays (SQL/MDA)

PostgreSQLのコア部分は 1、2、9、11、および14番の部分に対応しています。第3部分はODBCドライバを、 そして第13部分はPL/Javaプラグインを網羅していますが、これらコンポーネントに対する正確な適合性は検証されていません。4、10および15番の部分は現時点でPostgreSQLに実装されていません。

PostgreSQLはSQL:2016の主な機能のほとんどをサポートします。完全なコアの互換性に必要な177の必須機能の内、PostgreSQLは少なくとも170個に適合します。さらに省略可能な機能を数多くサポートします。本書の執筆時点で、SQL:2016のコアに完全に適合したデータベース管理システムのバージョンはないということを、ここで言及しておくのは価値のあることかもしれません。

以下の2つの節では、PostgreSQLでサポートされているこれらの機能をリストし、その後にPostgreSQLでまだサポートされていないSQL:2016で定義された機能をリストしています。これら2つのリストはおおよそのものです。サポート対象であるとリストされている機能が些細な部分で準拠していない可能性があり、サポートされていないある機能の大部分が、実際には実装されている可能性があります。本書の主要な部分では、 動作するものとしないものについての最も正確な情報を常に伝えます。

> **注記**
>
> ハイフンを含む機能コードはサブ機能です。したがって、特定のサブ機能がサポートされない場合、 他のサブ機能がサポートされていてもそのメイン機能はサポートされない機能としてリストされています。

D.1. サポートされている機能

識別子	コアか	説明	コメント
B012		埋め込みC	
B021		直接SQL	
E011	コア	数値データ型	
E011-01	コア	NTEGERおよびSMALLINTデータ型	
E011-02	コア	REAL、DOUBLE PRECISION、FLOATデータ型	
E011-03	コア	DECIMALおよびNUMERICデータ型	
E011-04	コア	算術演算子	
E011-05	コア	数値比較	
E011-06	コア	数値データ型間の暗黙キャスト	
E021	コア	文字データ型	
E021-01	コア	CHARACTERデータ型	
E021-02	コア	CHARACTER VARYINGデータ型	
E021-03	コア	文字リテラル	

識別子	コアか	説明	コメント
E021-04	コア	CHARACTER_LENGTH関数	数える前にCHARACTER値の最後の空白を除去します
E021-05	コア	OCTET_LENGTH関数	
E021-06	コア	SUBSTRING関数	
E021-07	コア	文字の連結	
E021-08	コア	UPPERおよびLOWER関数	
E021-09	コア	TRIM関数	
E021-10	コア	文字列型間の暗黙的キャスト	
E021-11	コア	POSITION関数	
E021-12	コア	文字の比較	
E031	コア	識別子	
E031-01	コア	制限付き識別子	
E031-02	コア	小文字の識別子	
E031-03	コア	末尾のアンダースコア	
E051	コア	基本問い合わせ仕様	
E051-01	コア	SELECT DISTINCT	
E051-02	コア	GROUP BY句	
E051-04	コア	選択リスト中にない列を含むことができるGROUP BY	
E051-05	コア	再命名できる選択リスト項目	
E051-06	コア	HAVING句	
E051-07	コア	選択リスト内の修飾付き*	
E051-08	コア	FROM句内の相関名	
E051-09	コア	FROM句内の列名の変更	
E061	コア	基本述語および検索条件	
E061-01	コア	比較述語	
E061-02	コア	BETWEEN述語	
E061-03	コア	値のリストが付いたIN述語	
E061-04	コア	LIKE述語	
E061-05	コア	LIKE述語 ESCAPE句	
E061-06	コア	NULL述語	
E061-07	コア	修飾付き比較述語	
E061-08	コア	EXISTS述語	
E061-09	コア	比較述語内の副問い合わせ	
E061-11	コア	IN述語内の副問い合わせ	

識別子	コアか	説明	コメント
E061-12	コア	修飾付き比較述語内の副問い合わせ	
E061-13	コア	相関副問い合わせ	
E061-14	コア	検索条件	
E071	コア	基本問い合わせ式	
E071-01	コア	UNION DISTINCTテーブル演算子	
E071-02	コア	UNION ALLテーブル演算子	
E071-03	コア	EXCEPT DISTINCTテーブル演算子	
E071-05	コア	正確に同一のデータ型を持つ必要がないテーブル演算子経由の列の結合	
E071-06	コア	副問い合わせ内のテーブル演算子	
E081	コア	基本権限	
E081-01	コア	SELECT権限	
E081-02	コア	DELETE権限	
E081-03	コア	テーブルレベルのINSERT権限	
E081-04	コア	テーブルレベルのUPDATE権限	
E081-05	コア	列レベルのUPDATE権限	
E081-06	コア	テーブルレベルのREFERENCES権限	
E081-07	コア	列レベルのREFERENCES権限	
E081-08	コア	WITH GRANT OPTION	
E081-09	コア	USAGE権限	
E081-10	コア	EXECUTE権限	
E091	コア	集合関数	
E091-01	コア	AVG	
E091-02	コア	COUNT	
E091-03	コア	MAX	
E091-04	コア	MIN	
E091-05	コア	SUM	
E091-06	コア	ALL修飾子	
E091-07	コア	DISTINCT修飾子	
E101	コア	基本データ操作	
E101-01	コア	INSERT文	
E101-03	コア	探索UPDATE文	
E101-04	コア	探索DELETE文	
E111	コア	単一行SELECT文	
E121	コア	基本カーソルサポート	

識別子	コアか	説明	コメント
E121-01	コア	DECLARE CURSOR	
E121-02	コア	選択リスト内にある必要がないORDER BY列	
E121-03	コア	ORDER BY句内の値式	
E121-04	コア	OPEN文	
E121-06	コア	位置指定UPDATE文	
E121-07	コア	位置指定DELETE文	
E121-08	コア	CLOSE文	
E121-10	コア	FETCH文、暗黙的なNEXT	
E121-17	コア	WITH HOLDカーソル	
E131	コア	NULL値のサポート(値の変わりとなるNULL)	
E141	コア	基本整合性制約	
E141-01	コア	NOT NULL制約	
E141-02	コア	NOT NULL列のUNIQUE制約	
E141-03	コア	PRIMARY KEY制約	
E141-04	コア	参照削除動作、参照更新動作の両方でNO ACTIONをデフォルトで持つ基本FOREIGN KEY制約	
E141-06	コア	CHECK制約	
E141-07	コア	列デフォルト	
E141-08	コア	PRIMARY KEYから推定されるNOT NULL	
E141-10	コア	任意の順序で指定できる外部キー中の名前	
E151	コア	トランザクションサポート	
E151-01	コア	COMMIT文	
E151-02	コア	ROLLBACK文	
E152	コア	基本SET TRANSACTION文	
E152-01	コア	SET TRANSACTION文: ISOLATION LEVEL SERIALIZABLE句	
E152-02	コア	SET TRANSACTION文: READ ONLYおよびREAD WRITE句	
E153	コア	副問い合せのある更新可能な問い合わせ	
E161	コア	二重のマイナス記号から始まるSQLコメント	
E171	コア	SQLSTATEサポート	
E182	コア	ホスト言語バインド	
F021	コア	基本情報スキーマ	
F021-01	コア	COLUMNSビュー	
F021-02	コア	TABLESビュー	
F021-03	コア	VIEWSビュー	

識別子	コアか	説明	コメント
F021-04	コア	TABLE_CONSTRAINTSビュー	
F021-05	コア	REFERENTIAL_CONSTRAINTSビュー	
F021-06	コア	CHECK_CONSTRAINTSビュー	
F031	コア	基本スキーマ操作	
F031-01	コア	永続基礎テーブルを作成するCREATE TABLE文	
F031-02	コア	CREATE VIEW文	
F031-03	コア	GRANT文	
F031-04	コア	ALTER TABLE文: ADD COLUMN句	
F031-13	コア	DROP TABLE文: RESTRICT句	
F031-16	コア	DROP VIEW文: RESTRICT句	
F031-19	コア	REVOKE文: RESTRICT句	
F032		CASCADE削除動作	
F033		ALTER TABLE文: DROP COLUMN句	
F034		拡張REVOKE文	
F034-01		スキーマオブジェクトの所有者以外が実行するREVOKE文	
F034-02		REVOKE文: GRANT OPTION FOR句	
F034-03		被譲与者がWITH GRANT OPTIONを持つ権限を削除するREVOKE文	
F041	コア	基本結合テーブル	
F041-01	コア	内部結合(INNERキーワードは不要)	
F041-02	コア	INNERキーワード	
F041-03	コア	LEFT OUTER JOIN	
F041-04	コア	RIGHT OUTER JOIN	
F041-05	コア	入れ子にできる外部結合	
F041-07	コア	内部結合内でも使用できる左または右外部結合の内部テーブル	
F041-08	コア	すべての比較演算子のサポート(単なる=は除く)	
F051	コア	基本日付時刻	
F051-01	コア	DATEデータ型(DATEリテラルのサポートを含む)	
F051-02	コア	少なくとも0の小数秒精度を持つTIMEデータ型(TIMEリテラルのサポートを含む)	
F051-03	コア	少なくとも0と6の小数秒精度を持つTIMESTAMPデータ型(TIMESTAMPリテラルのサポートを含む)	
F051-04	コア	DATE、TIME、TIMESTAMPデータ型に対する比較述語	
F051-05	コア	日付時刻型と文字列型間の明示的なCAST	

識別子	コアか	説明	コメント
F051-06	コア	CURRENT_DATE	
F051-07	コア	LOCALTIME	
F051-08	コア	LOCALTIMESTAMP	
F052		時間間隔および日付時刻計算	
F053		OVERLAPS述語	
F081	コア	ビューのUNIONおよびEXCEPT	
F111		SERIALIZABLE以外の隔離レベル	
F111-01		READ UNCOMMITTED隔離レベル	
F111-02		READ COMMITTED隔離レベル	
F111-03		REPEATABLE READ隔離レベル	
F131	コア	グループ化操作	
F131-01	コア	グループ化されたビューを使用する問い合わせで提供されるWHERE、GROUP BY、HAVING句	
F131-02	コア	グループ化されたビューを使用する問い合わせで提供される複数テーブル	
F131-03	コア	グループ化されたビューを使用する問い合わせで提供される集合関数	
F131-04	コア	GROUP BY、HAVING句、グループ化されたビューを持つ副問い合わせ	
F131-05	コア	GROUP BY、HAVING句、グループ化されたビューを持つ単一行SELECT	
F171		ユーザ毎の複数スキーマ	
F181	コア	複数モジュールサポート	
F191		参照削除動作	
F200		TRUNCATE TABLE文	
F201	コア	CAST関数	
F202		TRUNCATE TABLE: identity column restartオプション	
F221	コア	明示的なデフォルト	
F222		INSERT文: DEFAULT VALUES句	
F231		権限テーブル	
F231-01		TABLE_PRIVILEGESビュー	
F231-02		COLUMN_PRIVILEGESビュー	
F231-03		USAGE_PRIVILEGESビュー	
F251		ドメインサポート	
F261	コア	CASE式	

識別子	コアか	説明	コメント
F261-01	コア	単純CASE	
F261-02	コア	探索CASE	
F261-03	コア	NULLIF	
F261-04	コア	COALESCE	
F262		拡張CASE式	
F271		複合文字リテラル	
F281		LIKE強化	
F302		INTERSECTテーブル演算子	
F302-01		INTERSECT DISTINCTテーブル演算子	
F302-02		INTERSECT ALLテーブル演算子	
F304		EXCEPT ALLテーブル演算子	
F311	コア	スキーマ定義文	
F311-01	コア	CREATE SCHEMA	
F311-02	コア	永続基礎テーブル用のCREATE TABLE	
F311-03	コア	CREATE VIEW	
F311-04	コア	CREATE VIEW: WITH CHECK OPTION	
F311-05	コア	GRANT文	
F321		ユーザ認証	
F361		副プログラムのサポート	
F381		拡張スキーマ操作	
F381-01		ALTER TABLE文: ALTER COLUMN句	
F381-02		ALTER TABLE文: ADD CONSTRAINT句	
F381-03		ALTER TABLE文: DROP CONSTRAINT句	
F382		列データ型変更	
F383		Set column not null句	
F384		Drop identity property句	
F385		Drop column generation expression句	
F386		Set identity column generation句	
F391		長い識別子	
F392		識別子Unicodeエスケープ	
F393		リテラル内のUnicodeエスケープ	
F394		省略可能な標準フォーム指定	
F401		拡張結合テーブル	
F401-01		NATURAL JOIN	

識別子	コアか	説明	コメント
F401-02		FULL OUTER JOIN	
F401-04		CROSS JOIN	
F402		LOB、配列、複数集合と結合する名前付きの列	
F411		時間帯指定	リテラル解釈に関する違いあり
F421		各国文字	
F431		読み取りのみのスクロール可能なカーソル	
F431-01		明示的なNEXTを持つFETCH	
F431-02		FETCH FIRST	
F431-03		FETCH LAST	
F431-04		FETCH PRIOR	
F431-05		FETCH ABSOLUTE	
F431-06		FETCH RELATIVE	
F441		拡張集合関数のサポート	
F442		集合関数内の混在する列参照	
F471	コア	スカラ副問い合わせ値	
F481	コア	拡張NULL述語	
F491		制約管理	
F501	コア	機能と準拠ビュー	
F501-01	コア	SQL_FEATURESビュー	
F501-02	コア	SQL_SIZINGビュー	
F502		高度ドキュメントテーブル	
F531		一時テーブル	
F555		高度な秒精度	
F561		完全な値式	
F571		真値試験	
F591		派生テーブル	
F611		指示子データ型	
F641		行、テーブルコンストラクタ	
F651		カタログ名修飾子	
F661		単純テーブル	
F672		遡及的検査制約	
F690		照合サポート	しかし文字セットサポートはありません
F692		拡張照合サポート	

識別子	コアか	説明	コメント
F701		参照更新動作	
F711		ドメインのALTER	
F731		列のINSERT権限	
F751		ビューCHECK拡張	
F761		セッション管理	
F762		CURRENT_CATALOG	
F763		CURRENT_SCHEMA	
F771		接続管理	
F781		自己参照操作	
F791		Insensitiveカーソル	
F801		完全な集合関数	
F850		<問い合わせ式>における最上位<order by句>	
F851		副問い合わせにおける<order by句>	
F852		ビューにおける最上位<order by句>	
F855		<問い合わせ式>における入れ子状の<order by句>	
F856		<問い合わせ式>における入れ子状の<fetch first句>	
F857		<問い合わせ式>における最上位<fetch first句>	
F858		副問い合わせにおける<fetch first句>	
F859		ビューにおける最上位<fetch first句>	
F860		<fetch first句>における<fetch first row count>	
F861		<問い合わせ式>における最上位<result offset句>	
F862		副問い合わせにおける<result offset句>	
F863		<問い合わせ式>における入れ子状の<result offset句>	
F864		ビューにおける最上位<result offset句>	
F865		<result offset句>における<offset row count>	
F867		FETCH FIRST句: WITH TIESオプション	
S071		関数内のSQLパスおよび型名の解決	
S092		ユーザ定義型の配列	
S095		問い合わせによる配列の構築	
S096		省略可能な配列境界	
S098		ARRAY_AGG	
S111		問い合わせ式内のONLY	
S201		配列に関するSQL呼出しルーチン	
S201-01		配列パラメータ	

識別子	コアか	説明	コメント
S201-02		関数の結果型としての配列	
S211		ユーザ定義キャスト関数	
S301		拡張UNNEST	
T031		BOOLEANデータ型	
T071		BIGINTデータ型	
T121		問い合わせ式内のWITH(RECURSIVEを除く)	
T122		副問い合わせ内のWITH(RECURSIVEを除く)	
T131		再帰問い合わせ	
T132		副問い合わせ内の再帰問い合わせ	
T141		SIMILAR述語	
T151		DISTINCT述語	
T152		否定付きDISTINCT述語	
T171		テーブル定義内のLIKE句	
T172		テーブル定義内のAS副問い合わせ句	
T173		テーブル定義内の拡張LIKE句	
T174		識別列	
T177		連番生成子サポート: 簡単なリスタートオプション	
T178		識別列: 簡単なリスタートオプション	
T191		参照動作のRESTRICT	
T201		参照制約向けの比較可能データ型	
T211-01		基礎テーブルのUPDATE、INSERT、DELETEで駆動されるトリガ	
T211-02		BEFOREトリガ	
T211-03		AFTERトリガ	
T211-04		FOR EACH ROWトリガ	
T211-05		トリガ呼び出し前に真になる検索条件の指定機能	
T211-07		TRIGGER権限	
T212		拡張トリガ機能	
T213		INSTEAD OFトリガ	
T231		Sensitiveカーソル	
T241		START TRANSACTION文	
T261		連鎖トランザクション	
T271		セーブポイント	
T281		列粒度のSELECT権限	
T285		拡張派生列名	

識別子	コアか	説明	コメント
T312		OVERLAY関数	
T321-01	コア	オーバーロードがないユーザ定義関数	
T321-02	コア	オーバーロードがないユーザ定義ストアドプロシージャ	
T321-03	コア	関数呼び出し	
T321-04	コア	CALL文	
T321-06	コア	ROUTINESビュー	
T321-07	コア	PARAMETERSビュー	
T323		外部ルーチンに対する明示的なセキュリティ	
T325		修飾付きSQLパラメータ参照	
T331		基本ロール	
T341		SQLから呼び出される関数、プロシージャのオーバーロード	
T351		囲みSQLコメント(/*...*/コメント)	
T431		拡張グループ化機能	
T432		入れ子および連結GROUPING SETS	
T433		複数引数のGROUPING関数	
T441		ABSおよびMOD関数	
T461		対称BETWEEN述語	
T491		LATERAL派生テーブル	
T501		拡張EXISTS述語	
T521		CALL文における名前付き引数	
T523		SQL呼び出しプロシージャのINOUTパラメータに対するデフォルト値	
T524		CALL文以外でのルーチン呼び出しにおける名前付き引数	
T525		SQL呼び出し関数のパラメータに対するデフォルト値	
T551		デフォルト構文用の省略可能なキーワード	
T581		正規表現部分文字列関数	
T591		NULLになる可能性のある列のUNIQUE制約	
T611		基本OLAP演算	
T612		高度OLAP演算	
T613		サンプリング	
T614		NTILE関数	
T615		LEADおよびLAG関数	
T617		FIRST_VALUEおよびLAST_VALUE関数	
T620		WINDOW句: GROUPSオプション	
T621		拡張数値関数	

識別子	コアか	説明	コメント
T622		三角関数	
T623		一般の対数関数	
T624		常用対数関数	
T631	コア	リスト要素内のIN述語	
T651		SQLルーチン内のSQLスキーマ文	
T653		外部ルーチン内のSQLスキーマ文	
T655		循環依存ルーチン	
T831		SQL/JSONパス言語: 厳密モード	
T832		SQL/JSONパス言語: 項目メソッド	
T833		SQL/JSONパス言語: 複数の添字	
T834		SQL/JSONパス言語: ワイルドカードメンバアクセサ	
T835		SQL/JSONパス言語: フィルター式	
T836		SQL/JSONパス言語: starts with述語	
T837		SQL/JSONパス言語: regex_like述語	
X010		XML型	
X011		XML型の配列	
X014		XML型の属性	
X016		永続XML値	
X020		XMLConcat	
X031		XMLElement	
X032		XMLForest	
X034		XMLAgg	
X035		XMLAgg: ORDER BYオプション	
X036		XMLComment	
X037		XMLPI	
X040		基本テーブル対応付け	
X041		基本テーブル対応付け: NULLがない	
X042		基本テーブル対応付け: NULLをnilとして扱う	
X043		基本テーブル対応付け: テーブルをフォレストとして扱う	
X044		基本テーブル対応付け: テーブルを要素として扱う	
X045		基本テーブル対応付け: 対象名前空間あり	
X046		基本テーブル対応付け: データ対応付け	
X047		基本テーブル対応付け: メタデータ対応付け	
X048		基本テーブル対応付け: バイナリ列のBASE64符号化	

識別子	コアか	説明	コメント
X049		基本テーブル対応付け: バイナリ列のHEX符号化	
X050		高度テーブル対応付け	
X051		高度テーブル対応付け: NULLがない	
X052		高度テーブル対応付け: NULLをnilとして扱う	
X053		高度テーブル対応付け: テーブルをフォレストとして扱う	
X054		高度テーブル対応付け: テーブルを要素として扱う	
X055		高度テーブル対応付け: 対象名前空間	
X056		高度テーブル対応付け: データ対応付け	
X057		高度テーブル対応付け: メタデータ対応付け	
X058		高度テーブル対応付け: バイナリ列のBASE64符号化	
X059		高度テーブル対応付け: バイナリ列のHEX符号化	
X060		XMLParse: 文字列入力およびCONTENTオプション	
X061		XMLParse: 文字列入力およびDOCUMENTオプション	
X070		XMLSerialize: 文字列シリアル化およびCONTENTオプション	
X071		XMLSerialize: 文字列シリアル化およびDOCUMENTオプション	
X072		XMLSerialize: 文字列シリアル化	
X090		XML文書述語	
X120		SQLルーチンにおけるXMLパラメータ	
X121		外部ルーチンにおけるXMLパラメータ	
X221		値により(BY VALUE)XMLを渡す機構	
X301		XMLTable: 派生列リストオプション	
X302		XMLTable: 序数列オプション	
X303		XMLTable: 列デフォルトオプション	
X304		XMLTable: コンテキスト項目を渡す	XML DOCUMENTでなければなりません
X400		名前と識別子の対応付け	
X410		列のデータ型の変更: XML型	

D.2. サポートされていない機能

以下のSQL:2016で定義されている機能は本リリースのPostgreSQLでは実装されていません。たまに同等の機能が実装されていることがあります。

識別子	コアか	説明	コメント
B011		埋め込みAda	
B013		埋め込みCOBOL	
B014		埋め込みFortran	
B015		埋め込みMUMPS	
B016		埋め込みPascal	
B017		埋め込みPL/I	
B031		基本動的SQL	
B032		拡張動的SQL	
B032-01		<describe入力文>	
B033		型のないSQLから呼び出す関数の引数	
B034		カーソル属性の動的指定	
B035		非拡張記述子名	
B041		埋め込みSQL例外宣言への拡張	
B051		高度な実行権	
B111		モジュール言語 Ada	
B112		モジュール言語 C	
B113		モジュール言語 COBOL	
B114		モジュール言語 Fortran	
B115		モジュール言語 MUMPS	
B116		モジュール言語 Pascal	
B117		モジュール言語 PL/I	
B121		ルーチン言語 Ada	
B122		ルーチン言語 C	
B123		ルーチン言語 COBOL	
B124		ルーチン言語 Fortran	
B125		ルーチン言語 MUMPS	
B126		ルーチン言語 Pascal	
B127		ルーチン言語 PL/I	
B128		ルーチン言語 SQL	
B200		多相型テーブル関数(PTF)	
B201		2つ以上のPTFジェネリックテーブルパラメータ	
B202		PTFコパーティション	
B203		2つ以上のパーティション指定	
B204		PRUNE WHEN EMPTY	

識別子	コアか	説明	コメント
B205		パススルー列	
B206		PTF記述子パラメータ	
B207		パーティションの外積	
B208		PTFコンポーネントプロシージャインタフェース	
B209		PTF拡張名	
B211		モジュール言語 Ada: VARCHARおよびNUMERICのサポート	
B221		ルーチン言語 Ada: VARCHARおよびNUMERICのサポート	
F054		DATE型優先リストにおけるTIMESTAMP	
F121		基本診断管理	
F121-01		GET DIAGNOSTICS文	
F121-02		SET TRANSACTION文: DIAGNOSTICS SIZE句	
F122		拡張診断管理	
F123		すべての診断	
F263		単純CASE式におけるコンマ区切り述語	
F291		UNIQUE述語	
F301		問い合わせ式内のCORRESPONDING	
F312		MERGE文	INSERT ... ON CONFLICT DO UPDATE を検討してください
F313		拡張MERGE文	
F314		DELETE分岐を持つMERGE文	
F341		使用方法テーブル	ROUTINE_*_USAGE テーブルはありません
F403		分割結合されたテーブル	
F404		共通の列名に対する範囲変数	
F451		文字セット定義	
F461		名前付き文字セット	
F492		省略可能なテーブル制約の強制	
F521		表明	
F671		CHECK内の副問い合わせ	意図的な省略
F673		CHECK制約内のSQLデータルーチン呼び出しの読み取り	
F693		SQLセッション、クライアントモジュールの照合	
F695		翻訳サポート	
F696		追加用翻訳ドキュメント	

識別子	コアか	説明	コメント
F721		遅延可能制約	外部キーおよび一意キーのみ
F741		参照MATCH型	部分一致は未実装
F812	コア	基本フラグ付け	
F813		拡張フラグ付け	
F821		ローカルなテーブル参照	
F831		完全なカーソル更新	
F831-01		更新可能なスクロール可能カーソル	
F831-02		更新可能な順序付けカーソル	
F841		LIKE_REGEX述語	
F842		OCCURRENCES_REGEX関数	
F843		POSITION_REGEX関数	
F844		SUBSTRING_REGEX関数	
F845		TRANSLATE_REGEX関数	
F846		正規表現演算子における8進形式サポート	
F847		値式正規表現	
F866		FETCH FIRST句: PERCENTオプション	
R010		行パターン認識: FROM句	
R020		行パターン認識: WINDOW句	
R030		行パターン認識: 完全集約サポート	
S011	コア	Distinctデータ型	
S011-01	コア	USER_DEFINED_TYPESビュー	
S023		基本構造化型	
S024		拡張構造化型	
S025		終端構造化型	
S026		自己参照構造化型	
S027		メソッド名を指定したメソッドの作成	
S028		交換可能UDTオプションリスト	
S041		基本参照型	
S043		拡張参照型	
S051		型のテーブルの作成	一部サポート
S081		副テーブル	
S091		基本配列サポート	一部サポート
S091-01		組み込みデータ型の配列	
S091-02		個別型の配列	

識別子	コアか	説明	コメント
S091-03		配列の式	
S094		参照型の配列	
S097		配列要素の代入	
S151		型述語	
S161		副型の扱い	
S162		参照用の副型の扱い	
S202		複数集合に関するSQL呼出しルーチン	
S231		構造化型の位置付け子	
S232		配列の位置付け子	
S233		複数集合の位置付け子	
S241		変換関数	
S242		変換文の変更	
S251		ユーザ定義の順序	
S261		特定型メソッド	
S271		基本複数集合サポート	
S272		ユーザ定義型の複数集合	
S274		参照型の複数集合	
S275		高度複数集合サポート	
S281		入れ子の照合型	
S291		行全体に対する一意性制約	
S401		配列型に基づく個別型	
S402		個別型に基づく個別型	
S403		ARRAY_MAX_CARDINALITY	
S404		TRIM_ARRAY	
T011		情報スキーマ内のタイムスタンプ	
T021		BINARYおよびVARBINARYデータ型	
T022		BINARYおよびVARBINARYデータ型の高度サポート	
T023		複合2進リテラル	
T024		2進リテラルにおける空白	
T041		基本LOBデータ型サポート	
T041-01		BLOBデータ型	
T041-02		CLOBデータ型	
T041-03		LOBデータ型に対するPOSITION、LENGTH、LOWER、TRIM、UPPER、SUBSTRING関数	
T041-04		LOBデータ型の連結	

識別子	コアか	説明	コメント
T041-05		LOB位置付け子: 保持不可能	
T042		拡張LOBデータ型サポート	
T043		乗数T	
T044		乗数P	
T051		行型	
T053		すべてのフィールドを参照するための明示的な別名	
T061		UCSサポート	
T076		DECFLOATデータ型	
T101		拡張nullabilityの決定	
T111		更新可能な結合、和集合、列	
T175		生成列	ほとんどサポート
T176		連番生成子サポート	NEXT VALUE FORを除いてサポート
T180		システムバージョン付きテーブル	
T181		Application-time periodテーブル	
T211		基本トリガ機能	
T211-06		トリガと制約間の相互作用についての実行時規則のサポート	
T211-08		同一イベントに対する複数のトリガは、カタログ内で作成された順番に実行される	意図的な省略
T251		SET TRANSACTION文: LOCALオプション	
T272		拡張セーブポイント管理	
T301		関数従属性	一部サポート
T321	コア	基本SQL呼び出しルーチン	
T321-05	コア	RETURN文	
T322		宣言されたデータ型属性	
T324		SQLルーチンに対する明示的なセキュリティ	
T326		テーブル関数	
T332		拡張ロール	ほとんどサポート
T434		GROUP BY DISTINCT	
T471		結果集合戻り値	
T472		DESCRIBE CURSOR	
T495		データ変更と取り出しの組み合わせ	異なる構文
T502		Period述語	
T511		トランザクション数	

識別子	コアか	説明	コメント
T522		SQL呼び出しプロシージャのINパラメータに対するデフォルト値	呼び出しでのDEFAULTキーワードを除いてサポート
T561		保持可能な位置付け子	
T571		配列を返すSQL呼び出し関数	
T572		複数集合を返すSQL呼び出し関数	
T601		ローカルなカーソル参照	
T616		LEADおよびLAG関数用のnull処理オプション	
T618		NTH_VALUE関数	関数自体はありますが、一部のオプションがありません
T619		入れ子状ウィンドウ関数	
T625		LISTAGG	
T641		複数列の代入	構文の一部のみサポート
T652		SQLルーチン内の動的SQL文	
T654		外部ルーチン内の動的SQL文	
T811		基本SQL/JSON構築関数	
T812		SQL/JSON: JSON_OBJECTAGG	
T813		SQL/JSON: ORDER BY付きのJSON_ARRAYAGG	
T814		JSON_OBJECTやJSON_OBJECTAGGでのコロン	
T821		基本SQL/JSON問い合わせ演算子	
T822		SQL/JSON: IS JSON WITH UNIQUE KEYS述語	
T823		SQL/JSON: PASSING句	
T824		JSON_TABLE: 特定のPLAN句	
T825		SQL/JSON: ON EMPTYとON ERROR句	
T826		ON ERRORまたはON EMPTY句での一般値式	
T827		JSON_TABLE: 兄弟のNESTED COLUMNS句	
T828		JSON_QUERY	
T829		JSON_QUERY: 配列ラッパオプション	
T830		SQL/JSON構築関数内での一意キーの強制	
T838		JSON_TABLE: PLAN DEFAULT句	
T839		文字列への/からのdatetimeの書式化されたキャスト	
M001		データリンク	
M002		SQL/CLI経由のデータリンク	

識別子	コアか	説明	コメント
M003		埋め込みSQL経由のデータリンク	
M004		外部データサポート	一部サポート
M005		外部スキーマサポート	
M006		GetSQLStringルーチン	
M007		TransmitRequest	
M009		GetOptsおよびGetStatisticsルーチン	
M010		外部データラッパサポート	異なるAPI
M011		Ada経由のデータリンク	
M012		C経由のデータリンク	
M013		COBOL経由のデータリンク	
M014		Fortran経由のデータリンク	
M015		M経由のデータリンク	
M016		Pascal経由のデータリンク	
M017		PL/I経由のデータリンク	
M018		Adaにおける外部データラッパインタフェース処理	
M019		Cにおける外部データラッパインタフェース処理	異なるAPI
M020		COBOLにおける外部データラッパインタフェース処理	
M021		Fortranにおける外部データラッパインタフェース処理	
M022		MUMPSにおける外部データラッパインタフェース処理	
M023		Pascalにおける外部データラッパインタフェース処理	
M024		PL/Iにおける外部データラッパインタフェース処理	
M030		SQLサーバ外部データサポート	
M031		外部データラッパ汎用ルーチン	
X012		XML型の複数集合	
X013		XML型の個別型	
X015		XML型のフィールド	
X025		XMLCast	
X030		XMLDocument	
X038		XMLText	
X065		XMLParse: BLOB入力およびCONTENTオプション	
X066		XMLParse: BLOB入力およびDOCUMENTオプション	
X068		XMLSerialize: BOM	
X069		XMLSerialize: INDENT	
X073		XMLSerialize: BLOBシリアル化およびCONTENTオプション	

識別子	コアか	説明	コメント
X074		XMLSerialize: BLOBシリアル化およびDOCUMENTオプション	
X075		XMLSerialize: BLOBシリアル化	
X076		XMLSerialize: VERSION	
X077		XMLSerialize: 明示的ENCODINGオプション	
X078		XMLSerialize: 明示的XML宣言	
X080		XML発行における名前空間	
X081		問い合わせレベルのXML名前空間宣言	
X082		DMLにおけるXML名前空間宣言	
X083		DDLにおけるXML名前空間宣言	
X084		複合文におけるXML名前空間宣言	
X085		事前定義の名前空間接頭辞	
X086		XMLTableにおけるXML名前空間宣言	
X091		XMLコンテンツ述語	
X096		XMLExists	XPathのみ
X100		XML用のホスト言語サポート: CONTENTオプション	
X101		XML用のホスト言語サポート: DOCUMENTオプション	
X110		XML用のホスト言語サポート: VARCHAR対応付け	
X111		XML用のホスト言語サポート: CLOB対応付け	
X112		XML用のホスト言語サポート: BLOB対応付け	
X113		XML用のホスト言語サポート: STRIP WHITESPACEオプション	
X114		XML用のホスト言語サポート: PRESERVE WHITESPACEオプション	
X131		問い合わせレベルのXMLBINARY句	
X132		DMLにおけるXMLBINARY句	
X133		DDLにおけるXMLBINARY句	
X134		複合文におけるXMLBINARY句	
X135		副問い合わせにおけるXMLBINARY句	
X141		IS VALID述語: データ駆動ケース	
X142		IS VALID述語: ACCORDING TO句	
X143		IS VALID述語: ELEMENT句	
X144		IS VALID述語: スキーマ位置	
X145		検査制約外のIS VALID述語	
X151		DOCUMENTオプション付きIS VALID述語	

識別子	コアか	説明	コメント
X152		CONTENTオプション付きIS VALID述語	
X153		SEQUENCEオプション付きIS VALID述語	
X155		IS VALID述語: ELEMENT句のないNAMESPACE	
X157		IS VALID述語: ELEMENT句付きのNO NAMESPACE	
X160		登録XMLスキーマに対する基本情報スキーマ	
X161		登録XMLスキーマに対する高度情報スキーマ	
X170		XMLにおけるnull取り扱いオプション	
X171		NIL ON NO CONTENTオプション	
X181		XML(DOCUMENT(UNTYPED))型	
X182		XML(DOCUMENT(ANY))型	
X190		XML(SEQUENCE)型	
X191		XML(DOCUMENT(XMLSCHEMA))型	
X192		XML(CONTENT(XMLSCHEMA))型	
X200		XMLQuery	
X201		XMLQuery: RETURNING CONTENT	
X202		XMLQuery: RETURNING SEQUENCE	
X203		XMLQuery: コンテキスト項目を渡す	
X204		XMLQuery: XQuery変数の初期化	
X205		XMLQuery: EMPTY ON EMPTYオプション	
X206		XMLQuery: NULL ON EMPTYオプション	
X211		XML 1.1サポート	
X222		参照により(BY REF)XMLを渡す機構	パーサはBY REFを受け付けますが無視します。常にBY VALUEで渡されます
X231		XML(CONTENT(UNTYPED))型	
X232		XML(CONTENT(ANY))型	
X241		XML発行におけるRETURNING CONTENT	
X242		XML発行におけるRETURNING SEQUENCE	
X251		XML(DOCUMENT(UNTYPED))型の永続的なXML値	
X252		XML(DOCUMENT(ANY))型の永続的なXML値	
X253		XML(CONTENT(UNTYPED))型の永続的なXML値	
X254		XML(CONTENT(ANY))型の永続的なXML値	
X255		XML(SEQUENCE)型の永続的なXML値	
X256		XML(DOCUMENT(XMLSCHEMA))型の永続的なXML値	

識別子	コアか	説明	コメント
X257		XML(CONTENT(XMLSCHEMA))型の永続的なXML値	
X260		XML型: ELEMENT句	
X261		XML型: ELEMENT句のないNAMESPACE	
X263		XML型: ELEMENT句付きNO NAMESPACE	
X264		XML型: スキーマ位置	
X271		XMLValidate: データ駆動ケース	
X272		XMLValidate: ACCORDING TO句	
X273		XMLValidate: ELEMENT句	
X274		XMLValidate: スキーマ位置	
X281		XMLValidate DOCUMENTオプション付き	
X282		XMLValidate CONTENTオプション付き	
X283		XMLValidate SEQUENCEオプション付き	
X284		XMLValidate: ELEMENT句のないNAMESPACE	
X286		XMLValidate: ELEMENT句付きNO NAMESPACE	
X300		XMLTable	XPath 1.0のみ
X305		XMLTable: XQuery変数の初期化	

D.3. XMLの制限とSQL/XMLへの適合

SQL:2006でISO/IEC 9075-14 (SQL/XML)のXML関連の仕様についての重要な改定が導入されました。PostgreSQLのXMLデータ型と関連する関数の実装は、いくつか新しい版から取り入れつつ、主として2003以前の版に従っていました。特に:

- 現在の標準は、型付けされていないかXMLスキーマで型付けされている変数で「document」や「content」を格納するXMLのデータ型の一群や、任意のXML内容の断片を格納するXML(SEQUENCE)型を提供しますが、PostgreSQLは「document」か「content」だけを格納できる単一のxml型を提供します。標準の「sequence」型と同等のものはありません。
- PostgreSQLはSQL:2006で導入された二つの関数を提供しますが、それらに対して標準で指定されているXML QueryではなくXPath 1.0言語を使う変形としてです。

本章では遭遇するであろういくつかの結果の違いを示します。

D.3.1. 問い合わせはXPath 1.0に限定される

PostgreSQL固有の関数xpath()とxpath_exists()はXPath言語を使ってXML文書に問い合わせます。PostgreSQLは、公式にはXQuery言語を使う標準関数のXMLEXISTSとXMLTABLEについても、XPathのみという変形を提供しています。これら全ての関数についてPostgreSQLは、XPath 1.0のみを提供する、libxml2ライブラリに依存しています。

XQuery言語とXPathバージョン2.0以降との間には強い関連があり、両方で構文として有効で正常に実行できる全ての式は同じ結果を生成します（数字を含む式の参照や事前定義された要素の参照について細かな例外があり、それらをXQueryは対応する文字で置換しますが、XPathはそのままにします）。しかし、これら言語とXPath 1.0との間には、このような関連はありません。XPath 1.0はより古い言語であって多くの点で異なります。

認識すべき二つの種類の制限事項があります。SQL標準で指定される関数に対してXQueryでなくXPathであるという制限と、標準関数でもPostgreSQL固有関数でもXPathがバージョン1.0であるという制限です。

D.3.1.1. XQueryがXPathである制限

XPathに含まれないXQueryの機能:

- XQueryの式は、全てのXPathでできる値に加えて、新たなXMLノードを生成して返すことができます。XPathは原子型（数値、文字列など）の値を作成して返すことができますが、XMLノードは入力として式に与えられる文書にあらかじめ存在するものしか返せません。
- XQueryは構造に対する反復、並び替え、グループ化の制御ができます。
- XQueryでは局所関数を宣言して使用することができます。

最近のXPathバージョンはこれらをカバーする機能（関数形式の`for-each`と`sort`、無名関数、文字列からノードを作る`parse-xml`など）を提供し始めていますが、これら機能はXPath 3.0より前では提供されません。

D.3.1.2. XPathが1.0であることによる制限

XQueryとXPath 2.0以降に慣れた開発者にとって、XPath 1.0には以下の対処すべき違いがあります。

- XQuery/XPath式の基本的な型で、XMLノードや原子値、それらの両方を複数含むことができるsequenceがXPath 1.0には存在しません。1.0の式はノードセット（0個以上のXMLノードを含む）か単独の原子値のみ生成できます。
- 任意の要素群を任意の順序で含めることができるXQuery/XPathのシーケンスと違い、XPath 1.0のノードセットは順序保証がなく、集合のように、同じ要素が複数出現することを許しません。

> **注記**
>
> libxml2ライブラリは常に入力文書での順序に連動した同じ順序でPostgreSQLにノードセット返すように見えます。ライブラリのドキュメントはこの振る舞いを保証していませんし、XPath 1.0式はこれを制御できません。

- XQuery/XPathがXML Schemaで定義されたすべての型と、これらの型に対する多数の演算子や関数を提供する一方、XPath 1.0ではノードセットと3つの原子型`boolean`、`double`、`string`のみが使えます。
- XPath 1.0には条件演算子がありません。`if ( hat ) then hat/@size else "no hat"`といったXQuery/XPathの式は、XPath 1.0では実現できません。

- XPath 1.0には文字列の順序比較演算子がありません。"cat" < "dog"も"cat" > "dog"も、どちらも2つのNaNの数値比較であるため、偽になります。対照的に=と!=は文字列を文字列として比較します。

- XPath 1.0では、XQuery/XPathで定義されているところの値比較と一般比較の区別が曖昧です。sale/@hatsize = 7とsale/@customer = "alice"は、共に実体のある定量的な比較であり、与えられた属性に対する値を伴うsaleがあるなら真ですが、sale/@taxable = false()はノードセット全体の有効なブール値との値比較です。taxable属性を持つsaleが全く無い場合のみ真になります。

- XQuery/XPathデータモデルでは、ドキュメントノードはドキュメント形式（すなわちコメントと外側の処理指示だけを伴う厳密に一つだけのトップレベル要素）かコンテキスト形式（これらの制約が緩められたもの）のいずれかを持つことができます。これに対してXPath 1.0ではルートノードはドキュメント形式のみです。このことは、PostgreSQLのXPathに基づくどの関数に対してもコンテキスト要素として渡されるxml値がドキュメント形式でなければならない理由の一つです。

ここに挙げたものは違いの全てではありません。XQueryと2.0以降のXPathには、XPath 1.0互換モードがあり、このモードで摘要されるW3Cの関数ライブラリの変更点[1]と言語の変更点[2]のリストは、より完成された（しかし未だ完全ではない）違いの説明を提供します。この互換モードは新しい言語を正確にXPath 1.0と等しくできるわけではありません。

D.3.1.3. SQLとXMLのデータ型および値のマッピング

SQL:2006以降では、標準SQLデータ型とXMLスキーマ型の間の両方向の変換が正確に明記されています。しかしながら、その規則はXQuery/XPathの型と意味を用いて説明されていて、XPath1.0の異なるデータモデルへの直接の適用は含まれません。

PostgreSQLがSQLデータの値からXML（xmlelementで）、あるいは、XMLからSQL（xmltableの出力列で）に変換するとき、特別扱いされる一部の場合を除いて、PostgreSQLは単純にXMLデータ型のXPath 1.0文字列形式がSQLデータ型のテキスト入力形式として有効であると想定し、逆向きの変換でも同様です。この規則は多くのデータ型に対して実装が単純という長所を持ち、標準で明記された変換と似た結果になります。

他システムとの相互運用性が重要なところでは、一部のデータ型に対して標準の変換を実現するために明示的に（9.8にあるような）データ型整形関数を使うことがおそらく必要です。

D.3.2. その他の実装の制限

本節はlibxml2ライブラリ固有の制限ではないけれども、PostgreSQLの現在の実装で適用される制限について述べます。

D.3.2.1. 引き渡し機構はBY VALUEのみ対応

SQL標準は、SQLからXML関数にXML引数を渡す、あるいは結果を受け取るときに適用される2つの引き渡し方式を定義しています。BY REFでは特定のXML値がそのノードIDを保持し、BY VALUEではXML内容が渡さ

[1] https://www.w3.org/TR/2010/REC-xpath-functions-20101214/#xpath1-compatibility

[2] https://www.w3.org/TR/xpath20/#id-backwards-compatibility

れてノードIDは保持されません。方式は、パラメータリストの手前にそれらすべてのデフォルトとして、あるいは、各パラメータの後ろにデフォルトを上書きするものとして、指定することができます。

違いを例を挙げて示します。もしxがXML値であるなら、SQL:2006環境でのこれら2つの問合わせは、それぞれtrueとfalseを返すでしょう。

```
SELECT XMLQUERY('$a is $b' PASSING BY REF x AS a, x AS b NULL ON EMPTY);
SELECT XMLQUERY('$a is $b' PASSING BY VALUE x AS a, x AS b NULL ON EMPTY);
```

PostgreSQLは、XMLEXISTSやXMLTABLE構築でBY VALUEやBY REFを受け付けますが、無視します。xmlデータ型は連続した文字列表現を持ちますので、保持されるノードIDは無く、引き渡しは実際には常にBY VALUEです。

D.3.2.2. 問い合わせに名前付きパラメータは渡せない

XPathベースの関数はXPath式のコンテキスト要素として働くようにパラメータを渡すことをサポートしていますが、名前付きパラメータの式にできるように追加値を渡すことはサポートしていません。

D.3.2.3. XML(SEQUENCE)型は無い

PostgreSQLのxmlデータ型はDOCUMENTまたはCONTENT形式でのみ値を保持できます。XQuery/XPath式コンテキストの要素は単独のXMLノードか原子値でなければなりません。しかし、XPath 1.0ではさらにXMLノードのみに制限していて、加えてCONTENTが可能なノードタイプを持ちません。結果として、整形式のDOCUMENTのみが、PostgreSQLでXPathコンテキストの要素として提供されるXML値の形式です。

付録E リリースノート

このリリースノートには、PostgreSQLの各リリースでなされた重要な変更点が記載されています。また、主要機能や移行に関する問題点も最初に記載されています。リリースノートにはごく一部のユーザにしか影響しない変更点や内部的なものであるためユーザには隠れている変更点は記載されていません。例えば、オプティマイザはほぼすべてのリリースで改良されていますが、この改良はユーザにとっては通常、ただ問い合わせが高速になったと感じるものです。

各リリースにおける変更点の全一覧は、各リリースのGitログを参照することで入手できます。pgsql-committersメーリングリスト[1]にもすべてのソースコードの変更点が記録されています。また、特定のファイルに対する変更点を表示するWebインタフェース[2]が存在します。

各項目の後に記述した名前は、その項目の主な開発者の名前です。もちろんすべての変更はコミュニティによる議論やパッチレビューが行われていますので、各項目は本当はコミュニティによるものと言えます。

E.1. リリース 13.1

リリース日: 2020-11-12

このリリースは13.0に対し、様々な不具合を修正したものです。13メジャーリリースにおける新機能については、E.2を参照してください。

E.1.1. バージョン13.1への移行

13.Xからの移行ではダンプ/リストアは不要です。

E.1.2. 変更点

- インデックス式とマテリアライズドビューの問い合わせの中での`DECLARE CURSOR ... WITH HOLD`と遅延トリガの発動を防止しました。(Noah Misch)

  これは本質的に「セキュリティ制限された操作」のサンドボックス機構の抜け穴です。テンポラリでないSQLオブジェクトの作成権限を持つ攻撃者は、任意のSQLコードをスーパーユーザとして実行するように抜け穴を拡大できました。

  PostgreSQLプロジェクトは本問題を報告してくれたEtienne Stalmansに感謝します。(CVE-2020-25695)

- pg_dump、pg_restore、clusterdb、reindexdb、および、vacuumdbにおける、複雑な接続文字列のパラメータの使用方法を修正しました。(Tom Lane)

  pg_dumpとpg_restoreの`-d`パラメータや、列記した他のプログラムの`--maintenance-db`パラメータは、単一のデータベース名ではなく複数の接続パラメータを含む「接続文字列」とすることができます。並列

[1] https://www.postgresql.org/list/pgsql-committers/

[2] https://git.postgresql.org/gitweb/?p=postgresql.git;a=summary

処理や複数データベースの処理など、これらのプログラムが追加の接続を初期化する必要がある場合、この接続文字列は無視されて、追加接続には基本の接続パラメータ(データベース名、ホスト、ポート、ユーザ名)だけが使われました。接続文字列にデフォルトでないSSLやGSSのパラメータなど他に何らか重要な情報が含まれている場合に、この動作は接続失敗をもたらす可能性がありました。さらに悪いことには、接続は成功するけれども意図通りに暗号化されなかったり、意図した接続パラメータが防ぐはずだった中間者攻撃に脆弱になったりするかもしれませんでした。(CVE-2020-25694)

- psqlの\connectコマンドが接続パラメータを再利用するとき、以前の接続文字列から上書きされていないパラメータが確実に再利用されるようにしました。(Tom Lane)

  これにより、デフォルトでないSSLやGSSのオプションなど、関連するパラメータの脱落により再接続に失敗するかもしれないケースを回避します。さらに悪いことに、再接続は成功したけれども意図通りに暗号化されなかったり、意図した接続パラメータで防ぐはずの中間者攻撃に脆弱になったりするかもしれませんでした。ユーザが意図的に一部の接続パラメータを上書きするのでpsqlの振る舞いはより複雑ですが、これは前述のpg_dump等に対するものと大筋で同じ問題です。(CVE-2020-25694)

- psqlの\gsetコマンドが特別扱いされる変数を書き換えるのを防止しました。(Noah Misch)

  接頭辞を伴わない\gsetはサーバが示した変数を何であれ上書きします。したがって、安全でないサーバがPROMPT1などの特別扱いされる変数を設定して、ユーザのセッションで任意シェルコードを実行できるようにする可能性がありました。

  PostgreSQLプロジェクトは本問題を報告してくれたNick Cleatonに感謝します。(CVE-2020-25696)

- レプリケーションプロトコルの意図せぬ破壊を修正しました。(Álvaro Herrera)

  walsenderはSTART_REPLICATIONに対するコマンド完了イベントを2つ報告しました。これはドキュメント記載されておらず、明らかに意図せぬものです。それで、私たちは最近の13.0の変更が二重イベントを削除したことを知らせていませんでした。しかしながら、walreceiverが一部のコードパスで余分なイベントを必要とすることが判明しました。最も現実的な修正は、余分なイベントをプロトコルの一部と決定し、余分なイベント発生を復活させることです。

- チェックポイント時にSLRUディレクトリが適切にfsyncされるようにしました。(Thomas Munro)

  これはオペレーティングシステムのクラッシュ後に起こりうるデータ損失を防止します。

- BYPASSRLS属性を持ったユーザに対するALTER ROLEを修正しました。(Tom Lane, Stephen Frost)

  BYPASSRLS属性はスーパーユーザのみ変更することができますが、パスワード変更などの他のALTER ROLE操作は通常のパーミッション検査に基づき可能とすべきです。これまでのコードでは誤って、このようなロールへの全ての変更をスーパーユーザからのみに制限していました。

- 子テーブルがあるときにALTER TABLE ONLY ... DROP EXPRESSIONを禁止しました。(Peter Eisentraut)

  現在の実装ではこの場合について正しく扱うことができないため、当面は単にこれを禁止します。

- ALTER TABLE ONLY ... ENABLE/DISABLE TRIGGERが子テーブルに再帰しないようにしました。(Álvaro Herrera)

  これまではONLYフラグが無視されていました。

- LOCK TABLEが自己参照ビューで動作するようにしました。(Tom Lane)

これまでは無限再帰を問題とするエラーを投げていましたが、この場合を禁止する必要はないと思われます。

- REINDEX CONCURRENTLYを経てもインデックスに関する統計情報を維持するようにしました。(Michael Paquier, Fabrízio de Royes Mello)

  同時実行でないインデックス再作成では既にこのような統計情報を維持するようになっていました。

- REINDEX CONCURRENTLYからの誤った進行報告を修正しました。(Matthias van de Meent, Michael Paquier)

- ルールや更新可能ビューを通して依存する列が更新されたときに、GENERATED列が確実に更新されるようにしました。(Tom Lane)

  この修正は、このような場合において起こりうる列指定トリガの発動失敗にも対処しています。

- 照合順序に依存したパーティション境界の式でのエラーを修正しました。(Tom Lane)

- テキスト配列のハッシュ化に対応しました。(Peter Eisentraut)

  配列要素型が照合順序を適用できる場合に配列のハッシュ作成に失敗していました。特に、これはテキスト配列の列をパーティションキーとしたハッシュパーティショニングの使用を妨げました。

- 異なる型の間で日付時刻を比較する際の内部オーバーフローを防止しました。(Nikita Glukhov, Alexander Korotkov, Tom Lane)

  これまでは、dateがtimestampとして有効な範囲より過去の場合、dateとtimestampとの比較はエラーになりました。また、関連した稀な場合として、タイムゾーンのローテーションに際して限界値に近いtimestamp値のオーバーフローもありました。

- to_date()とto_timestamp()でのマイナス年からBC日付への一つ違いの変換を修正しました。(Dar Alathar-Yemen, Tom Lane)

  また、マイナス年と明示的な「BC」マークとの組み合わせに対して、打ち消してADを生成するように調整しました。

- jsonpathの.datetime()メソッドがISO 8601形式のタイムスタンプを受け入れできるようにしました。(Nikita Glukhov)

  これはSQLで要求されていることではありませんが、to_json()関数がJavascriptの互換性のために、このタイムスタンプ形式を生成することから適切と考えられます。

- archive_modeがalwaysに設定されているとき、スタンバイサーバがWALタイムラインヒストリファイルを確実にアーカイブするようにしました。(Grigory Smolkin, Fujii Masao)

  この見落としにより、その後のPITRリカバリの試みが失敗する可能性がありました。

- kqueue()を使うプラットフォームにて早すぎるpostmasterダウンの検出での境界ケースを修正しました。(Thomas Munro)

- ソートキーが揮発的な式であるときの誤ったインクリメンタルソートのプランの生成を回避しました。(James Coleman)

- GEQOのプラン作成時にパーティション同士の結合を検討したときに起こりうるクラッシュを修正しました。(Tom Lane)
- TOASTの圧縮の展開で起こりうる無限ループまたは壊れた出力データを修正しました。(Tom Lane)
- クリーンアップだけをするVACUUMのときのBツリーインデックスで項目数のカウントを修正しました。(Peter Geoghegan)
- データがBRINインデックスに挿入される前に確実にTOASTから戻すようにしました。(Tomas Vondra)

  インデックスのエントリに行外のTOASTポインタを含むことはサポートされませんが、BRINはこれを考慮していませんでした。これにより、「missing chunk number 0 for toast value NNN（TOAST値NNNのためのチャンク番号0がありません）」のようなエラーが生じる可能性がありました。（既存のインデックスでこのようなエラーに遭遇したなら、修正するのにREINDEXを行えばよいです。）
- インデックスが付加された列を持つときに動作するように、バッファ付きGiSTインデックスの構築を修正しました。(Pavel Borisov)
- pg_hba_file_rulesビューでgetnameinfo()の移植性に欠ける使用法を修正しました。(Tom Lane)

  FreeBSD 11では、あるいは他のプラットフォームでも、このビューのaddressとnetmask列はこの誤りのために常にnullでした。
- パラレルワーカを開始するときにdebug_query_stringがNULLである場合のクラッシュを回避しました。(Noah Misch)
- BEFORE ROW UPDATEトリガが、削除されたか「見つからない」列を持つテーブルの「old」行を返すときの失敗を回避しました。(Amit Langote, Tom Lane)

  この更新を抑制する方法は、「見つからない」列がNULLとして読まれることで、クラッシュや予期せぬCHECK制約エラー、あるいは、誤ったRETURNING出力をもたらす可能性がありました。（ある列が定数ですが非NULLのデフォルト値でALTER TABLE ADD COLUMNにて追加された場合、その列は「見つからない」ものとなります。）削除された列も同様に問題を引き起こすおそれがありました。
- インクリメンタルソートの計画に対するEXPLAINの出力を、XML出力モードで正しい入れ子のタグを持つように修正しました。(Daniel Gustafsson)
- 共有メモリのキューを通して非常に大きな内容を転送するときの不要なエラーを回避しました。(Markus Wanner)
- SQL言語関数でいくつかの場合における結果データ型の強制の欠落を修正しました。(Tom Lane)

  関わるデータ型によっては、これにより誤った結果やクラッシュをもたらす可能性がありました。
- JITコード生成でのテンプレート関数属性の誤った扱いを修正しました。(Andres Freund)

  これはs390xでクラッシュを引き起こすことが示されていて、他のプラットフォームでも他の問題がある可能性があります。
- PPCでcompare_exchangeとfetch_addの操作についてコード生成を改善しました。(Noah Misch)
- RLSポリシーに関してリレーションキャッシュのメモリリークを修正しました。(Tom Lane)
- index_get_partition()で境界条件的な場合のメモリリークを修正しました。(Justin Pryzby)

- SIGHUP処理が新たなGUC変数値は再起動なしには適用できないと判断するときの小さなメモリリークを修正しました。(Tom Lane)
- PL/pgsqlのCALL処理でメモリリークを修正しました。(Pavel Stehule, Tom Lane)
- Windowsのlibpqで、プロセスごとに一度WSAStartup()を呼び出し、WSACleanup()は全く呼び出さないようにしました。(Tom Lane, Alexander Lakhin)

  これまでは、libpqは接続開始時にWSAStartup()を呼び出し、接続をクリーンアップするときにWSACleanup()を呼び出していました。しかしながら、WSACleanup()の呼び出しは他のプログラムの操作と干渉する可能性があることがわかりました。特にstdoutへの期待された出力に稀に失敗することが観測されています。この呼び出しを省略しても悪影響はないとみられるため、そのようにしました。(これはプログラムがデータベース接続を連続して行うときのDLLロードとアンロードの繰り返しによる性能問題も取り除きます。)
- WindowsでのecpgライブラリのスレッドごとのDLL初期化ロジックを修正しました。(Tom Lane, Alexander Lakhin)

  マルチスレッド化されたecpgアプリケーションは、誤ったロックにより稀に誤動作を起こす可能性がありました。
- ecpgのB'...'およびX'...'リテラルに対する誤った処理を修正しました。(Shenhao Wang)
- Windowsで、psqlがバッククォートコマンドの出力をバイナリモードでなくテキストモードで読むようにしました。(Tom Lane)

  これにより改行を適切に扱えます。
- pg_dumpが拡張の設定テーブルの列ごとの情報を確実に収集するようにしました。(Fabrízio de Royes Mello, Tom Lane)

  これに失敗することで、--insertsを指定したときにクラッシュしたり、COPYを使用してテーブルのデータを再ロードするときに、(通常は正しいのですが)不十分な指定のCOPYコマンドになったりしました。
- pg_upgradeが対象クラスタにテーブルスペースディレクトリが既に存在するかを検査するようにしました。(Bruce Momjian)
- contrib/pgcryptoの潜在的なメモリリークを修正しました。(Michael Paquier)
- contrib/pgcryptoでの起こりそうにない失敗ケースに対する検査を追加しました。(Daniel Gustafsson)
- 最近に追加されたtimetzのテストケースを修正して、アメリカが夏時間を守っていないときにも動作するようにしました。(Tom Lane)
- タイムゾーンデータファイルをtzdata release 2020dに更新しました。フィジー、モロッコ、パレスチナ、カナダのユーコン、マッコーリー島、ケーシー基地(南極)の夏時間法の変更と、フランス、ハンガリー、モナコ、パレスチナの歴史的修正が含まれます。
- タイムゾーンライブラリのコピーをIANA tzcode release 2020dに同期しました。(Tom Lane)

  これはアップストリームのzicのデフォルト出力オプションの「fat」から「slim」への変更を吸収します。V13より前のブランチでは引き続き「fat」モードを選択するため、これは単に見た目を整えるためのものです。この変更はstrftime()が、それが失敗しない限り、errnoを変更しないことも保証します。

E.2. リリース13

リリース日: 2020-09-24

E.2.1. 概要

PostgreSQL 13には以下をはじめとする多数の新機能と拡張が含まれます。

- Bツリーインデックス項目の重複除去による省スペース化と性能向上
- 集約やパーティションテーブルを使う問い合わせの性能改善
- 拡張統計情報を使ったときのより良い問い合わせの計画作成
- 並列化されたインデックスのバキューム
- インクリメンタルソート

PostgreSQL 13の上記の項目とその他の機能は次節でより詳しく説明されます。

E.2.2. バージョン13への移行

以前のリリースからデータを移行したい時は、どのリリースについても、pg_dumpallを利用したダンプとリストア、あるいはpg_upgradeやロジカルレプリケーションの使用が必要です。新たなメジャーバージョンへの移行に関する一般的な情報については18.6を参照してください。

バージョン13には、以前のバージョンとの互換性に影響するかもしれない多数の変更点が含まれています。以下の非互換性に注意してください。

- `SIMILAR TO ... ESCAPE NULL`がNULLを返すように変更しました。(Tom Lane)

  この新たな振る舞いはSQL仕様に適合します。これまではNULLのESCAPE値はデフォルトのエスケープ文字列(バックスラッシュ文字)を使うという意味になっていました。この変更は、substring(`text FROM pattern ESCAPE text`)にも適用されます。変更されていない元の関数を維持することで古いビューではこれまでの振る舞いが保たれます。

- `json[b]_to_tsvector()`がstringオプションのスペリングを完全に検査するようにしました。(Dominik Czarnota)

- デフォルトでないeffective_io_concurrency値が同時実行性に作用する方式を変更しました。

  これまでは、この値は同時要求数の設定に先立って調整が行われました。これからは、この値が直接的に使われます。古い値から新たな値への変換は以下を使ってできます。

  ```
  SELECT round(sum(OLDVALUE / n::float)) AS newvalue FROM generate_series(1, OLDVALUE) s(n);
  ```

- pg_stat_sslおよびpg_stat_gssapiシステムビューで補助プロセスの表示を防止しました。(Euler Taveira)

これらのビューをpg_stat_activityと結合して補助プロセスも参照しようとする問い合わせでは左結合を使う必要があるでしょう。

- 一貫性を改善するため様々な待機イベントの名前を変更しました。(Fujii Masao, Tom Lane)
- ALTER FOREIGN TABLE ... RENAME COLUMNをより適切なコマンドタグを返すように修正しました。(Fujii Masao)

  これまではALTER TABLEを返しましたが、これからはALTER FOREIGN TABLEを返します。
- ALTER MATERIALIZED VIEW ... RENAME COLUMNをより適切なコマンドタグを返すように修正しました。(Fujii Masao)

  これまではALTER TABLEを返しましたが、これからはALTER MATERIALIZED VIEWを返します。
- 設定パラメータwal_keep_segmentsをwal_keep_sizeに名前変更しました。(Fujii Masao)

  これはどれだけWALをスタンバイサーバのために保持するかを決定します。従来パラメータでのファイル数ではなく、メガバイトで指定されます。これまでwal_keep_segmentsを使っていたなら、以下の式でほぼ同等の設定が得られます。

```
wal_keep_size = wal_keep_segments * wal_segment_size (典型的には 16MB)
```

- PostgreSQL 8.0より前の構文を使った演算子クラスの定義のサポートを廃止しました。(Daniel Gustafsson)
- PostgreSQL 7.3より前の構文を使った外部キー制約の定義のサポートを廃止しました。(Daniel Gustafsson)
- PostgreSQL 7.3より前のサーバで使われるopaque疑似型のサポートを廃止しました。(Daniel Gustafsson)
- パッケージ化されていない(9.1より前の)拡張のアップグレードのサポートを廃止しました。(Tom Lane)

  CREATE EXTENSIONのFROMオプションはもはやサポートされなくなります。未だパッケージ化されていな拡張を使っているインストレーションは、PostgreSQL 13にバージョンアップする前にそれら拡張をパッケージ化されたバージョンにアップグレードすべきです。
- タイムゾーンデータベースでposixrulesファイルのサポートを廃止しました。(Tom Lane)

  IANAのタイムゾーングループはこの機能を廃止していて、今後の数年間でシステムのタイムゾーンデータベースから徐々に消えていくことになります。タイムゾーンデータの更新による予期せぬ振る舞いの変化が現れるのを避けるため、私たちはバージョン13でPostgreSQLによる本機能のサポートを廃止しました。これは、明示的な夏時間ルールが欠けているPOSIXスタイルのタイムゾーン仕様の動作にだけ影響します。以前は、独自のposixrulesファイルを導入することで変換規則を決定できましたが、これからは固定の動作となります。影響をうけるインストレーションに対する推奨される修正は、地理的なタイムゾーン名を使うようにすることです。
- ltreeで、lqueryのパターンが、例えば*{2}.*{3}という中括弧付きの隣接するアスタリスクを含む場合、それを適切に*{5}と解釈するようにしました。(Nikita Glukhov)

- pageinspectのbt_metap()が、オーバーフローする可能性の低い、より適切なデータ型を返すように修正しました。(Peter Geoghegan)

E.2.3. 変更点

以下にPostgreSQL 13と前メジャーリリースとの詳細な変更点を記載します。

E.2.3.1. サーバ

E.2.3.1.1. パーティショニング

- より多くの場合にパーティションの除去が行われるようにしました。(Yuzuko Hosoya, Amit Langote, Álvaro Herrera)
- より多くの場合にパーティション同士の結合が行われるようにしました。(Ashutosh Bapat, Etsuro Fujita, Amit Langote, Tom Lane)

  例えば、パーティションの境界が完全には一致していないパーティションテーブルの間においても、パーティション同士の結合が行えるようになります。
- パーティションテーブルで行単位のBEFOREトリガに対応しました。(Álvaro Herrera)

  ただし、このようなトリガでどのパーティションが宛先であるかを変更することはできません。
- パーティションテーブルをパブリケーションによってロジカルレプリケーション対象にできるようにしました。(Amit Langote)

  これまではパーティションを個々にレプリケーションする必要がありました。これからは明示的にパーティションテーブルをパブリッシュすることができて、これにより属する全てのパーティションが自動的にパブリッシュされます。パーティションの追加/削除をすると、同様にパブリケーションの追加や削除が行われます。CREATE PUBLICATIONのオプションpublish_via_partition_rootは、パーティションに対する変更をパブリッシュするのに、それら自身の変更とするか親の変更とするかを制御します。
- サブスクライバでパーティションテーブルにロジカルレプリケーションできるようにしました。(Amit Langote)

  これまでは、サブスクライバはパーティション化されていないテーブルに限って行を受け入れることができました。
- 行全体の変数(すなわちtable.*) をパーティションの式で使えるようにしました。(Amit Langote)

E.2.3.1.2. インデックス

- Bツリーインデックスで重複をより効率的に格納するようにしました。(Anastasia Lubennikova, Peter Geoghegan)

  これは重複するキーを一度だけ格納することにより、低カーディナリティ列の効率的なBツリーインデックス作成を可能にします。pg_upgradeでアップグレードしたユーザは、既存インデックスで本機能を使うためにREINDEXを行う必要があります。

- box列に対するGiSTおよびSP-GiSTインデックスが、ORDER BY box <-> pointの問い合わせに対応しました。(Nikita Glukhov)
- GINインデックスがtsquery検索での!(否定)句をより効率的に処理できるようにしました。(Nikita Glukhov, Alexander Korotkov, Tom Lane, Julien Rouhaud)
- インデックスの演算子クラスがパラメータを取れるようにしました。(Nikita Glukhov)
- CREATE INDEXでGiSTのシグネチャの長さと整数範囲の最大数を指定できるようにしました。(Nikita Glukhov)

  4および8バイトの整数配列、tsvector、pg_trgm、ltree、および、hstoreの列に対して作成されたインデックスは、デフォルト値を使うのではなく、GiSTインデックスパラメータを制御できるようになりました。
- デフォルトでない照合順序を使うインデックスをテーブルのユニークまたは主キー制約として追加できないようにしました。(Tom Lane)

  インデックスの照合順序は元になる列の照合順序と一致しなければなりませんが、これまではALTER TABLEでこの検査ができていませんでした。

E.2.3.1.3. オプティマイザ

- 含有/一致の演算子に対して、オプティマイザの選択度の見積を改善しました。(Tom Lane)
- 拡張統計情報に対する統計対象を設定できるようにしました。(Tomas Vondra)

  これは新たなコマンドオプションALTER STATISTICS ... SET STATISTICSで制御されます。これまではより汎用的な統計対象の設定に基づいて計算されました。
- 一つの問い合わせに複数の拡張統計情報オブジェクトを使用できるようにしました。(Tomas Vondra)
- OR句およびIN/ANYの定数リストに対して拡張統計情報オブジェクトを使用できるようにしました。(Pierre Ducroquet, Tomas Vondra)
- 定数に評価できる場合には、FROM句で関数をプルアップ(インライン化)できるようにしました。(Alexander Kuzmenkov, Aleksandr Parfenov)

E.2.3.1.4. 性能全般

- インクリメンタルソートを実装しました。(James Coleman, Alexander Korotkov, Tomas Vondra)

  中間問い合わせ結果が必要なソート順序の1つまたは複数の先行キーによってソートされることが分かっている場合で、行が同じ先行キーを持つバッチでソートされる場合、残りのキーのみを考慮して追加のソートを実行できます。

  これは必要に応じてenable_incremental_sortを使用して制御できます。
- inet値のソート性能を改善しました。(Brandur Leach)

- ハッシュ集約が大きな集約結果セットにディスクストレージを使うことができるようになりました。(Jeff Davis)

  これまではwork_mem以上のメモリを使うと予測される場合、ハッシュ集約は避けられました。これからはそうであってもハッシュ集約のプランが選択できるようになります。ハッシュテーブルはwork_mem×hash_mem_multiplierを超える場合にはディスクに書き出されます。

  この動作は、一度ハッシュ集約が選ばれると作られるハッシュテーブルがいかに大きくとも(プランナが見積を誤ると巨大になり得ます)メモリ上の保持されるという従来の動作よりも、通常はより望ましいです。必要であれば、hash_mem_multiplierを増やすことで従来動作と似た振る舞いが得られます。

- 自動バキュームで更新、削除のみならず挿入もバキューム処理の契機とできるようにしました。(Laurenz Albe, Darafei Praliaskouski)

  これまでは挿入のみの処理は、自動アナライズを引き起こしても、削除すべきデッドタプルは存在しないことを理由として、自動バキュームを引き起こしませんでした。しかしながらバキュームの走査は、インデックスオンリースキャンの効率を改善するページ全可視ビットの設定などの、他の有益な副次効果を持ちます。また、挿入のみのテーブルが定期的なバキュームを受けられるようすることは、古いタプルを「凍結する」仕事を分散するために役立ち、テーブル全体が周回防止の閾値に達したときに突然一度に大量の凍結処理が実行されることがなくなります。

  必要に応じて、この振る舞いは新たなパラメータautovacuum_vacuum_insert_thresholdとautovacuum_vacuum_insert_scale_factor、あるいは、同等のテーブルストレージオプションで調整できます。

- メンテナンス操作におけるI/Oの並列性を制御するmaintenance_io_concurrencyパラメータを追加しました。(Thomas Munro)

- wal_levelがminimalの場合に、リレーションを作成または書き直しするトランザクションのときにWAL書き込みを省略できるようにしました。(Kyotaro Horiguchi)

  wal_skip_thresholdよりも大きいリレーションは、WALを生成するのではなく、そのファイルをfsyncさせます。これまではCOPY操作に限ってこれが行われていましたが、実装にバグがあり、クラッシュリカバリに際してデータ損失が生じるおそれがありました。

- 多くのテーブルスペースが使われているときにDROP DATABASEコマンドをリプレイするときの性能を改善しました。(Fujii Masao)

- 巨大なリレーションのTRUNCATEの性能を改善しました。(Kirk Jamison)

- TOASTされた値の先行するバイトの取得を改善しました。(Binguo Bao, Andrey Borodin)

  これまでは、いくらかの先行バイトのみが必要と分かっているときでも、圧縮された行外のTOAST値は完全に取得されました。これからは、結果の生成に必要なだけのデータのみが取得されます。

- LISTEN/NOTIFYの性能を改善しました。(Martijn van Oosterhout, Tom Lane)

- 整数からテキストへの変換速度を向上させました。(David Fetter)

- 多数のSQL文を含む、問い合わせ文字列や拡張のスクリプトに対するメモリ使用を減らしました。(Amit Langote)

E.2.3.1.5. モニタリング

- EXPLAIN、auto_explain、autovacuum、および、pg_stat_statementsがWAL使用の統計を追跡できるようにしました。(Kirill Bychik, Julien Rouhaud)
- 全てのSQL文ではなく、SQL文のサンプルをログ記録できるようにしました。(Adrien Nayrat)

  log_min_duration_sample以上の時間を要する文がlog_statement_sample_rateの比率でログ記録されます。
- csvlogとオプションでlog_line_prefixのログ出力にバックエンドタイプを追加しました。(Peter Eisentraut)
- 準備された文のパラメータのログ出力制御を改善しました。(Alexey Bashtanov, Álvaro Herrera)

  GUC設定のlog_parameter_max_lengthでエラーでない文のログ記録時のパラメータ値の出力の最大長を制御し、また、log_parameter_max_length_on_errorはエラーを伴う文のログ記録について同様に制御します。これまでは準備された文のパラメータはエラー時にはログ記録されませんでした。
- 関数呼び出しのバックトレースをエラー後に記録できるようにしました。(Peter Eisentraut, Álvaro Herrera)

  新たなパラメータbacktrace_functionsはどのC関数がエラーにバックトレースを生成すべきかを指定します。
- バキュームのバッファカウンタをオーバーフロー回避のため64ビット幅にしました。(Álvaro Herrera)

E.2.3.1.6. システムビュー

- pg_stat_activityにパラレルワーカーのリーダープロセスを報告するleader_pidを追加しました。(Julien Rouhaud)
- ストリーミングベースバックアップの進捗を報告するシステムビューpg_stat_progress_basebackupを追加しました。(Fujii Masao)
- ANALYZEの進捗を報告するシステムビューpg_stat_progress_analyzeを追加しました。(Álvaro Herrera, Tatsuro Yamada, Vinayak Pokale)
- 共有メモリの使用を表示するシステムビューpg_shmem_allocationsを追加しました。(Andres Freund, Robert Haas)
- 内部のSLRUキャッシュを観測するシステムビューpg_stat_slruを追加しました。(Tomas Vondra)
- track_activity_query_sizeが1MBまで設定できるようにしました。(Vyacheslav Makarov)

  以前の最大は100kBでした。

E.2.3.1.7. 待機イベント

- posix_fallocate()でDSMセグメントを作成する間に待機イベントを報告するようにしました。(Thomas Munro)

- コストに基づくバキューム遅延について報告する待機イベントVacuumDelayを追加しました。(Justin Pryzby)
- WALのアーカイブとリカバリ停止に対する待機イベントを追加しました。(Fujii Masao)

  新たなイベントはBackupWaitWalArchiveとRecoveryPauseです。
- リカバリの衝突を監視する待機イベントRecoveryConflictSnapshotおよびRecoveryConflictTablespaceを追加しました。(Masahiko Sawada)
- BSDベースのシステムで待機イベントの性能を改善しました。(Thomas Munro)

E.2.3.1.8. 認証

- スーパーユーザのみがssl_passphrase_command設定を参照できるようにしました。(Insung Moon)

  これはセキュリティ上の予防策として変更されました。
- 暗号化接続でのサーバのデフォルトの最小TLSバージョンを1.0から1.2に変更しました。(Peter Eisentraut)

  この選択はssl_min_protocol_versionで制御できます。

E.2.3.1.9. サーバ設定

- 読み取り専用トランザクションモードで許可されるユーティリティコマンドについての規定を厳格化しました。(Robert Haas)

  また、この変更はパラレルクエリで実行できるユーティリティコマンドの数を増やします。
- allow_system_table_modsをサーバ起動後に変更できるようにしました。(Peter Eisentraut)
- allow_system_table_modsが設定されているとき、非スーパーユーザがシステムテーブル変更できないようにしました。(Peter Eisentraut)

  これまではサーバ起動時にallow_system_table_modsが設定されていると、非スーパーユーザがシステムテーブルにINSERT/UPDATE/DELETEコマンドを発行できました。
- WindowsでUnixドメインソケットをサポートしました。(Peter Eisentraut)

E.2.3.2. ストリーミングレプリケーションとリカバリ

- ストリーミングレプリケーション設定をリロードで変更できるようにしました。(Sergei Kornilov)

  これまではprimary_conninfoとprimary_slot_nameの変更にはサーバ再起動が必要でした。
- WALレシーバが、永続スロットが指定されていないときに、一時的なレプリケーションスロットを使用できるようにしました。(Peter Eisentraut, Sergei Kornilov)

  この振る舞いはwal_receiver_create_temp_slotを使って有効にできます。

- レプリケーションスロットに対するWALストレージをmax_slot_wal_keep_sizeで制限できるようにしました。(Kyotaro Horiguchi)

  この値の超過を要するレプリケーションスロットは無効と印付けされます。

- スタンバイの昇格ですべての停止要求を取り消しできるようにしました。(Fujii Masao)

  これまではスタンバイが停止状態のときには昇格できませんでした。

- リカバリが指定されたリカバリターゲットに達しなかった場合、エラーを生成するようにしました。(Leif Gunnar Erlandsen, Peter Eisentraut)

  これまではたとえターゲットに達していなくとも、WALの最後に達したならスタンバイは昇格していました。

- ロジカルデコーディングでディスクに書き出す前に使われるメモリを制御できるようにしました。(Tomas Vondra, Dilip Kumar, Amit Kapila)

  これはlogical_decoding_work_memで制御されます。

- たとえ不正なページがWALで参照されたとしても、リカバリを継続できるようにしました。(Fujii Masao)

  これはignore_invalid_pagesを使って有効にできます。

E.2.3.3. ユーティリティコマンド

- VACUUMがテーブルのインデックスをパラレルに処理できるようにしました。(Masahiko Sawada, Amit Kapila)

  新たなPARALLELオプションでこれを制御します。

- FETCH FIRSTで最後の結果行と一致する行を追加的に返すWITH TIESを使えるようにしました。(Surafel Temesgen)

- EXPLAINのBUFFER出力でプラン作成時のバッファ使用を報告するようにしました。(Julien Rouhaud)

- CREATE TABLE LIKEがCHECK制約のNO INHERITの属性を作成するテーブルに伝搬するようにしました。(Ildar Musin, Chris Travers)

- パーティションテーブルでLOCK TABLEを使うとき、子テーブルで権限の検査をしないようにしました。(Amit Langote)

- 識別列への挿入でOVERRIDING USER VALUEに対応しました。(Dean Rasheed)

- 列からGENERATED属性を削除できるALTER TABLE ... DROP EXPRESSIONを追加しました。(Peter Eisentraut)

- 複数ステップのALTER TABLEコマンドのバグを修正しました。(Tom Lane)

  これからは、(インデックス作成などの)派生動作が既存列には実行されないということで、IF NOT EXISTS句が期待通り動作するようになります。また、関連する動作を一つのALTER TABLEに連結しているいくつかの場合について、以前は動作しなかったものが動作するようになりました。

- ビューの列を名前変更できるALTER VIEW構文を追加しました。(Fujii Masao)

  ビューの列の名前変更は既に可能でしたが、ALTER TABLE RENAME COLUMNと書く必要があり、混乱するものでした。

- 元となる型のTOAST属性とサポート関数を変更するALTER TYPEオプションを追加しました。(Tomas Vondra, Tom Lane)

- CREATE DATABASEのLOCALEオプションを追加しました。(Peter Eisentraut)

  これは既存オプションのLC_COLLATEとLC_CTYPEを一つのオプションにまとめたものです。

- DROP DATABASEが対象データベースを使用しているセッションを切断して、データベース削除に成功できるようにしました。(Pavel Stehule, Amit Kapila)

  これはFORCEオプションで可能になります。

- C言語の更新トリガがどの列が更新されたかを知ることができる構造体メンバtg_updatedcolsを追加しました。(Peter Eisentraut)

E.2.3.4. 日付型

- 互換性のある引数を必要とする関数での使用のため、いくつか多様データ型を追加しました。(Pavel Stehule)

  新たなデータ型はanycompatible、anycompatiblearray、anycompatiblenonarray、および、anycompatiblerangeです。

- FullTransactionIdを表現するSQLのデータ型xid8を追加しました。(Thomas Munro)

  既存のxidデータ型は4バイトしかないため、トランザクションエポックを提供できません。

- 照合順序オブジェクトのOIDを表示するための、データ型regcollationと関連する関数を追加しました。(Julien Rouhaud)

- 照合順序のバージョン識別子として、一部の場合にglibcバージョンを使うようにしました。(Thomas Munro)

  glibcバージョンが変更された場合、照合順序に依存したインデックスが壊れる可能性について警告が発行されるようになります。

- Windowsで照合順序バージョンに対応しました。(Thomas Munro)

- ROW式でメンバを接尾記法で展開できるようにしました。(Tom Lane)

  例えば(ROW(4, 5.0)).f1は4を返すようになります。

E.2.3.5. 関数

- 改良されたNULLの処理を備えたjsonb_set()の別バージョンを追加しました。(Andrew Dunstan)

新たな関数jsonb_set_lax()は、要求にしたがってNULLの新たな値を、指定キーをJSONのnullに設定する、キーを削除する、例外を上げる、変更されていないjsonb値を返す、のいずれかで処理します。

- jsonpathの.datetime()メソッドを追加しました。(Nikita Glukhov, Teodor Sigaev, Oleg Bartunov, Alexander Korotkov)

  この関数はJSON値をタイムスタンプに変換できます。そのタイムスタンプはjsonpath式で処理が可能です。この変更では、タイムゾーンを意識した出力に対応したjsonpath関数も追加されます。

- ユニコード文字列を標準化するSQL関数NORMALIZE()と、標準化を検査するIS NORMALIZEDを追加しました。(Peter Eisentraut)

- pg_lsnに対するmin()およびmax()集約を追加しました。(Fabrízio de Royes Mello)

  これらは特に監視の問い合わせに有用です。

- ユニコードエスケープに対応しました。例えば、E'\unnnn'やU&'\nnnn'で、データベースエンコーディングがUTF-8でないときであってもデータベースエンコーディングで有効な任意の文字を指定できます。(Tom Lane)

- to_date()とto_timestamp()が英語以外の月/日の名前を認識できるようにしました。(Juan José Santamaría Flecha, Tom Lane)

  認識される名前は同じ書式パターンにおけるto_char()による出力と同じです。

- 1から6の小数秒桁数の入力や出力を指定する日付時刻の書式パターンFF1–FF6を追加しました。(Alexander Korotkov, Nikita Glukhov, Teodor Sigaev, Oleg Bartunov)

  これらのパターンはto_char()、to_timestamp()、および、jsonpathの.datetime()で使用できます。

- SQL標準のSSSSの別名として日付時刻の書式パターンSSSSSを追加しました。(Nikita Glukhov, Alexander Korotkov)

- バージョン4のUUIDを生成する関数gen_random_uuid()を追加しました。(Peter Eisentraut)

  これまでUUID生成関数は拡張モジュールのuuid-osspとpgcryptoでのみ利用できました。

- 最大公約数(gcd)と最小公倍数(lcm)の関数を追加しました。(Vik Fearing)

- numeric型の平方根(sqrt)および自然対数(ln)関数の性能と正確性を改善しました。(Dean Rasheed)

- 完全な精度のnumeric値を表現するのに必要となる、小数点の右の桁数を返す関数min_scale()を追加しました。(Pavel Stehule)

- 末尾のゼロを除いてnumeric値のスケールを減らす関数trim_scale()を追加しました。(Pavel Stehule)

- 距離演算子の交換を追加しました。(Nikita Glukhov)

  例えばこれまでは、point <-> lineのみに対応していましたが、これからは、line <-> pointも同様に動作します。

- 全てのトランザクションID関数のxid8版を作成しました。(Thomas Munro)

後方互換性のため、古いxidベースの関数は未だ存在します。

- get_bit()とset_bit()がbytea値の最初の256MBを超えてビット設定できるようにしました。(Movead Li)
- 勧告的ロック関数が一部の並列操作で利用できるようにしました。(Tom Lane)
- 拡張におけるオブジェクトの依存性を削除できるようにしました。(Álvaro Herrera)

  対応するオブジェクトは関数、マテリアライズドビュー、インデックスまたはトリガーです。構文はALTER .. NO DEPENDS ONです。

E.2.3.6. PL/pgSQL

- 単純なPL/pgSQL式の性能を改善しました。(Tom Lane, Amit Langote)
- 不変式を使うPL/pgSQL関数の性能を改善しました。(Konstantin Knizhnik)

E.2.3.7. クライアントインタフェース

- libpqクライアントが暗号化接続でチャンネルバインディングを要求できるようにしました。(Jeff Davis)

  libpq接続パラメータchannel_bindingを使用して、TLS接続の相手先にユーザのパスワードを知っていることを証明させます。これは中間者攻撃を防止します。

- 暗号化接続で許容されるTLSの最小および最大バージョンを制御するlibpqの接続パラメータを追加しました。(Daniel Gustafsson)

  この設定項目はssl_min_protocol_versionとssl_max_protocol_versionです。デフォルトでは最小TLSバージョンは1.2です（これは以前のリリースからの振る舞いの変更になります）。

- クライアント証明書のロック解除にパスワードを使用できるようにしました。(Craig Ringer, Andrew Dunstan)

  libpqのsslpassword接続パラメータで有効になります。

- libpqでDERでエンコードされたクライアント証明書を使用できるようにしました。(Craig Ringer, Andrew Dunstan)

- ecpgのEXEC SQL elif指示子が正しく動作するように修正しました。(Tom Lane)

  これまでこれはifdefに続くendifと同じように動作していたため、同じif構造内の手前の分岐で成功しても、elif分岐や続く分岐の展開を止めませんでした。

E.2.3.8. クライアントアプリケーション

E.2.3.8.1. psql

- トランザクション状態(%x)がpsqlのデフォルトプロンプトに加わりました。(Vik Fearing)

- psqlの第二プロンプトを第一プロンプトと同じ幅だけの空白にできるようにしました。(Thomas Munro)

  これはPROMPT2を%wに設定することで実現できます。

- psqlの\gと\gxコマンドで単一コマンドの間について\pset出力オプションを変更できるようにしました。(Tom Lane)

  この機能は\g (expand=on)といった構文を可能にします。\gxも同様です。

- 演算子クラスと演算子族を表示するpsqlコマンドを追加しました。(Sergey Cherkashin, Nikita Glukhov, Alexander Korotkov)

  新たなコマンドは\dAc、\dAf、\dAo、および、\dApです。

- psqlの\dt+と関連のコマンドでテーブルの永続性を表示するようにしました。(David Fetter)

  冗長モードでテーブル/インデックス/ビューはオブジェクトがpermanentか、temporaryか、あるいは、unloggedかを表示します。

- psqlでTOASTテーブルに対する\dの出力を改善しました。(Justin Pryzby)

- psqlの\eコマンド後の再表示を修正しました。(Tom Lane)

  エディタを終了時に問い合わせがセミコロンや\gで終了していない場合、これからは問い合わせのバッファ内容が表示されなくなります。

- psqlに\warnコマンドを追加しました。(David Fetter)

  これは\echoと似ていますが、文字列が標準出力ではなく標準エラーに送られます。

- PostgreSQLホームページをコマンドライン--helpの出力に加えました。(Peter Eisentraut)

E.2.3.8.2. pgbench

- pgbenchが「accounts」テーブルをパーティション分割できるようにしました。(Fabien Coelho)

  これはパーティショニングのパフォーマンステストを可能します。

- pgbenchコマンド\asetを追加しました。\gsetのように動作しますが、複数の問い合わせに対応します。(Fabien Coelho)

- pgbenchが初期データをクライアント側ではなくサーバ側で生成できるようにしました。(Fabien Coelho)

- pgbenchがオプション--show-scriptでスクリプト内容を表示できるようにしました。(Fabien Coelho)

E.2.3.9. サーバアプリケーション

- ベースバックアップに対してバックアップマニフェストを生成して、検証するようにしました。(Robert Haas)

  新たなツールpg_verifybackupは、バックアップを検証できます。

- pg_basebackupがデフォルトで合計バックアップサイズを見積りするようにしました。(Fujii Masao)

  この計算によりpg_stat_progress_basebackupで進捗を表示できるようになります。必要でなければ、--no-estimate-sizeオプションを使って無効化できます。これまでは--progressオプションが使われていた場合だけ、この計算が行われました。

- pg_rewindにスタンバイを設定するオプションを追加しました。(Paul Guo, Jimmy Yih, Ashwin Agrawal)

  これはpg_basebackupの--write-recovery-confに相当します。

- pg_rewindが必要なWALを取得するために対象クラスタのrestore_commandを使えるようにしました。(Alexey Kondratov)

  これは-c/--restore-target-walオプションを使って有効になります。

- pg_rewindが巻き戻しをする前にクラッシュリカバリを自動的に実行するようにしました。(Paul Guo, Jimmy Yih, Ashwin Agrawal)

  これは--no-ensure-shutdownを使って無効にできます。

- pg_waldumpで報告されるPREPARE TRANSACTION関連の情報を増やしました。(Fujii Masao)

- pg_waldumpにエラー以外の出力を抑制するオプション--quietを追加しました。(Andres Freund, Robert Haas)

- pg_dumpに外部サーバからデータをダンプするオプション--include-foreign-dataを追加しました。(Luis Carril)

- vacuumdbから実行されるバキュームコマンドをパラレルモードで実行できるようにしました。(Masahiko Sawada)

  これは新たな--parallelオプションで有効になります。

- reindexdbがパラレルで動作できるようにしました。(Julien Rouhaud)

  新たな--jobsオプションでパラレルモードが有効になります。

- dropdbが対象データベースを使用しているセッションを切断して、削除に成功できるようにしました。(Pavel Stehule)

  これは-fオプションで有効になります。

- createuserから--adduserと--no-adduserを除きました。(Alexander Lakhin)

  これらに対応するより望ましい以前からあるオプションは--superuserと--no-superuserです。

- pg_upgradeの実行時に、デフォルトの--new-bindir設定としてpg_upgradeプログラムのディレクトリを使うようにしました。(Daniel Gustafsson)

E.2.3.10. 文書

- 文書に用語集を追加しました。(Corey Huinker, Jürgen Purtz, Roger Harkavy, Álvaro Herrera)

- 関数と演算子の情報を含む表をよりわかりやすい書式に変更しました。(Tom Lane)
- DocBook 4.5を使うようにアップグレードしました。(Peter Eisentraut)

E.2.3.11. ソースコード

- Visual Studio 2019でのビルドに対応しました。(Haribabu Kommi)
- MSYS2でのビルドに対応しました。(Peter Eisentraut)
- Power PCコンパイラむけにcompare_exchangeとfetch_addのアセンブラ言語コードを追加しました。(Noah Misch)
- 全文検索で使われるSnowball stemmer辞書を更新しました。(Panagiotis Mavrogiorgos)

  これはギリシャ語の語幹処理を追加し、デンマーク語とフランス語の語幹処理を改善します。
- Windows 2000のサポートを廃止しました。(Michael Paquier)
- 非ELFのBSDシステムのサポートを廃止しました。(Peter Eisentraut)
- Pythonバージョン2.5.X以前のサポートを廃止しました。(Peter Eisentraut)
- OpenSSL 0.9.8と1.0.0のサポートを廃止しました。(Michael Paquier)
- configureのオプション`--disable-float8-byval`および`--disable-float4-byval`を廃止しました。(Peter Eisentraut)

  これらは一部のバージョン0のC関数との互換性のために必要とされていましたが、それらももはやサポートされません。
- 問い合わせ文字列をプランナフック関数に渡すようにしました。(Pascal Legrand, Julien Rouhaud)
- `TRUNCATE`コマンドのフックを追加しました。(Yuli Khodorkovskiy)
- TLS初期化のフックを追加しました。(Andrew Dunstan)
- 事前定義されたUnixドメインソケットのディレクトリ無しにビルドできるようにしました。(Peter Eisentraut)
- UnixプラットフォームでSysVリソースキーが衝突する可能性を減らしました。(Tom Lane)
- 機微な情報を含んだメモリを確実に削除するため、OSの関数を使うようにしました。(Peter Eisentraut)

  これは例えばメモリ内に格納されたパスワードの消去に使用されます。
- Cヘッダファイルの互換性をテストするheaderscheckスクリプトを追加しました。(Tom Lane)
- 内部リストをセルの連鎖ではなく配列で実装しました。(Tom Lane)

  これは多数のオブジェクトにアクセスする問い合わせの性能を改善します。
- TS_execute()に対するAPIを変更しました。(Tom Lane, Pavel Borisov)

これからはTS_executeコールバックは3値(yes/no/maybe)のロジックを提供しなければなりません。NOT問い合わせを正確に計算することは、これからはデフォルトになります。

E.2.3.12. 追加モジュール

- 拡張にtrustedと指定できるようにしました。(Tom Lane)

  このような拡張は、スーパーユーザでなくとも、データベースレベルのCREATE権限を持つユーザであればデータベースにインストールできます。また、この変更でpg_pltemplateシステムカタログが削除されました。

- 非スーパーユーザがpostgres_fdwの外部サーバにパスワード無しで接続できるようにしました。(Craig Ringer)

  具体的には、スーパーユーザがユーザマッピングに対してpassword_requiredをfalseに設定できるようにしました。それでも、非スーパーユーザがスーパーユーザの権限を使って外部サーバに接続するのを防止するように注意しなければなりません。

- postgres_fdwが証明書認証を使用できるようにしました。(Craig Ringer)

  異なるユーザが異なる証明書を使えます。

- sepgsqlがTRUNCATEコマンドの使用を制御できるようにしました。(Yuli Khodorkovskiy)

- SQLのbooleanとPL/Perlのbooleanを相互に変換する、拡張bool_plperlを追加しました。(Ivan Panchenko)

- pg_stat_statementsがSELECT ... FOR UPDATEコマンドをFOR UPDATEの無いものと区別して扱うようにしました。(Andrew Gierth, Vik Fearing)

- pg_stat_statementsがオプションでSQL文のプラン作成時間を追跡できるようにしました。(Julien Rouhaud, Pascal Legrand, Thomas Munro, Fujii Masao)

  これまでは実行時間のみが追跡されました。

- ltreeのlquery構文がNOT (!)をより論理的に処理するように作り直しました。(Filip Rembialkowski, Tom Lane, Nikita Glukhov)

  「*」の無い問い合わせで一致回数の範囲({})を使えるようにしました。

- ltree、lquery、および、ltxtquery型のバイナリI/Oのサポートを追加しました。(Nino Floris)

- dict_intに整数の符号を無視するオプションを追加しました。(Jeff Janes)

- adminpackにファイルにfsyncを行うことができる関数pg_file_sync()を追加しました。(Fujii Masao)

- pageinspectにt_infomask/t_infomask2値を人間に読みやすい書式で出力する関数を追加しました。(Craig Ringer, Sawada Masahiko, Michael Paquier)

- Bツリーインデックスの重複除去処理の列をpageinspectの出力に追加しました。(Peter Geoghegan)

E.2.4. 謝辞

以下の人々（アルファベット順）はパッチ作者、コミッター、レビューア、テスターあるいは問題の報告者として本リリースに貢献しました。

Abhijit Menon-Sen
Adam Lee
Adam Scott
Adé Heyward
Adrien Nayrat
Ahsan Hadi
Alastair McKinley
Aleksandr Parfenov
Alex Aktsipetrov
Alex Macy
Alex Shulgin
Alexander Korotkov
Alexander Kukushkin
Alexander Kuzmenkov
Alexander Lakhin
Alexey Bashtanov
Alexey Kondratov
Álvaro Herrera
Amit Kapila
Amit Khandekar
Amit Langote
Amul Sul
Anastasia Lubennikova
Andreas Joseph Krogh
Andreas Karlsson
Andreas Kunert
Andreas Seltenreich
Andrei Zubkov
Andres Freund
Andrew Bille
Andrew Dunstan
Andrew Gierth
Andrey Borodin
Andrey Klychkov
Andrey Lepikhov
Anna Akenteva
Anna Endo
Anthony Nowocien
Anton Vlasov
Antonin Houska
Ants Aasma

Arne Roland
Arnold Müller
Arseny Sher
Arthur Nascimento
Arthur Zakirov
Ashutosh Bapat
Ashutosh Sharma
Ashwin Agrawal
Asif Rehman
Asim Praveen
Atsushi Torikoshi
Augustinas Jokubauskas
Austin Drenski
Basil Bourque
Beena Emerson
Ben Cornett
Benjie Gillam
Benoît Lobréau
Bernd Helmle
Bharath Rupireddy
Bhargav Kamineni
Binguo Bao
Brad DeJong
Brandur Leach
Brent Bates
Brian Williams
Bruce Momjian
Cameron Ezell
Cary Huang
Chapman Flack
Charles Offenbacher
Chen Huajun
Chenyang Lu
Chris Bandy
Chris Travers
Christoph Berg
Christophe Courtois
Corey Huinker
Craig Ringer
Cuiping Lin
Dagfinn Ilmari Mannsåker
Daniel Fiori
Daniel Gustafsson
Daniel Vérité
Daniel Westermann
Darafei Praliaskouski

Daryl Waycott
Dave Cramer
David Christensen
David Fetter
David G. Johnston
David Gilman
David Harper
David Rowley
David Steele
David Zhang
Davinder Singh
Dean Rasheed
Denis Stuchalin
Dent John
Didier Gautheron
Dilip Kumar
Dmitry Belyavsky
Dmitry Dolgov
Dmitry Ivanov
Dmitry Telpt
Dmitry Uspenskiy
Dominik Czarnota
Dongming Liu
Ed Morley
Edmund Horner
Emre Hasegeli
Eric Gillum
Erik Rijkers
Erwin Brandstetter
Ethan Waldo
Etsuro Fujita
Eugen Konkov
Euler Taveira
Fabien Coelho
Fabrízio de Royes Mello
Felix Lechner
Filip Janus
Filip Rembialkowski
Frank Gagnepain
Georgios Kokolatos
Gilles Darold
Greg Nancarrow
Grigory Smolkin
Guancheng Luo
Guillaume Lelarge
Hadi Moshayedi

Haiying Tang
Hamid Akhtar
Hans Buschmann
Hao Wu
Haribabu Kommi
Haruka Takatsuka
Heath Lord
Heikki Linnakangas
Himanshu Upadhyaya
Hironobu Suzuki
Hugh McMaster
Hugh Ranalli
Hugh Wang
Ian Barwick
Ibrar Ahmed
Ildar Musin
Insung Moon
Ireneusz Pluta
Isaac Morland
Ivan Kartyshov
Ivan Panchenko
Ivan Sergio Borgonovo
Jaime Casanova
James Coleman
James Gray
James Hunter
James Inform
James Lucas
Jan Mussler
Jaroslav Sivy
Jeevan Chalke
Jeevan Ladhe
Jeff Davis
Jeff Janes
Jehan-Guillaume de Rorthais
Jeremy Evans
Jeremy Schneider
Jeremy Smith
Jerry Sievers
Jesper Pedersen
Jesse Kinkead
Jesse Zhang
Jian Zhang
Jie Zhang
Jim Nasby
Jimmy Yih

Jobin Augustine
Joe Conway
John Hsu
John Naylor
Jon Jensen
Jonathan Katz
Jorge Gustavo Rocha
Josef Šimánek
Joseph Nahmias
Juan José Santamaría Flecha
Julian Backes
Julien Rouhaud
Jürgen Purtz
Justin King
Justin Pryzby
Karl O. Pinc
Keisuke Kuroda
Keith Fiske
Kelly Min
Ken Tanzer
Kirill Bychik
Kirk Jamison
Konstantin Knizhnik
Kuntal Ghosh
Kyle Kingsbury
Kyotaro Horiguchi
Lars Kanis
Laurenz Albe
Leif Gunnar Erlandsen
Li Japin
Liudmila Mantrova
Lucas Viecelli
Luis M. Carril
Lukáš Sobotka
Maciek Sakrejda
Magnus Hagander
Mahadevan Ramachandran
Mahendra Singh Thalor
Manuel Rigger
Marc Munro
Marcos David
Marina Polyakova
Mark Dilger
Mark Wong
Marko Tiikkaja
Markus Winand

Marti Raudsepp
Martijn van Oosterhout
Masahiko Sawada
Masahiro Ikeda
Masao Fujii
Mateusz Guzik
Matt Jibson
Matteo Beccati
Maxence Ahlouche
Melanie Plageman
Michael Banck
Michael Luo
Michael Meskes
Michael Paquier
Michail Nikolaev
Mike Palmiotto
Mithun Cy
Movead Li
Nathan Bossart
Nazli Ugur Koyluoglu
Neha Sharma
Nicola Contu
Nicolás Alvarez
Nikhil Sontakke
Nikita Glukhov
Nikolay Shaplov
Nino Floris
Noah Misch
Noriyoshi Shinoda
Oleg Bartunov
Oleg Samoilov
Oleksii Kliukin
Ondrej Jirman
Panagiotis Mavrogiorgos
Pascal Legrand
Patrick McHardy
Paul Guo
Paul Jungwirth
Paul Ramsey
Paul Sivash
Paul Spencer
Pavan Deolasee
Pavel Borisov
Pavel Luzanov
Pavel Stehule
Pavel Suderevsky

Peifeng Qiu
Pengzhou Tang
Peter Billen
Peter Eisentraut
Peter Geoghegan
Peter Smith
Petr Fedorov
Petr Jelínek
Phil Bayer
Philip Semanchuk
Philippe Beaudoin
Pierre Ducroquet
Pierre Giraud
Piotr Gabriel Kosinski
Piotr Wlodarczyk
Prabhat Sahu
Quan Zongliang
Quentin Rameau
Rafael Castro
Rafia Sabih
Raj Mohite
Rajkumar Raghuwanshi
Ramanarayana M
Ranier Vilela
Rares Salcudean
Raúl Marín Rodríguez
Raymond Martin
Reijo Suhonen
Richard Guo
Robert Ford
Robert Haas
Robert Kahlert
Robert Treat
Robin Abbi
Robins Tharakan
Roger Harkavy
Roman Peshkurov
Rui DeSousa
Rui Hai Jiang
Rushabh Lathia
Ryan Lambert
Ryohei Takahashi
Scott Ribe
Sean Farrell
Sehrope Sarkuni
Sergei Agalakov

Sergei Kornilov
Sergey Cherkashin
Shawn Debnath
Shawn Wang
Shay Rojansky
Shenhao Wang
Simon Riggs
Slawomir Chodnicki
Soumyadeep Chakraborty
Stéphane Lorek
Stephen Frost
Steve Rogerson
Steven Winfield
Surafel Temesgen
Suraj Kharage
Takanori Asaba
Takao Fujii
Takayuki Tsunakawa
Takuma Hoshiai
Tatsuhito Kasahara
Tatsuo Ishii
Tatsuro Yamada
Taylor Vesely
Teodor Sigaev
Tham Nguyen
Thibaut Madelaine
Thom Brown
Thomas Kellerer
Thomas Munro
Tiago Anastacio
Tim Clarke
Tim Möhlmann
Tom Ellis
Tom Gottfried
Tom Lane
Tomas Vondra
Tuomas Leikola
Tushar Ahuja
Victor Wagner
Victor Yegorov
Vignesh C
Vik Fearing
Vinay Banakar
Vladimir Leskov
Vladimir Sitnikov
Vyacheslav Makarov

Vyacheslav Shablistyy
Will Leinweber
William Crowell
Wyatt Alt
Yang Xiao
Yaroslav Schekin
Yi Huang
Yigong Hu
Yoann La Cancellera
Yoshikazu Imai
Yu Kimura
Yugo Nagata
Yuli Khodorkovskiy
Yusuke Egashira
Yuya Watari
Yuzuko Hosoya
ZhenHua Cai

E.3. Prior Releases

以前のリリースブランチのリリースノートはhttps://www.postgresql.org/docs/release/にあります。

付録F 追加で提供されるモジュール

この付録と次の付録にはPostgreSQL配布物のcontribディレクトリにあるモジュールに関する情報があります。ここには、移植用のツール、解析ユーティリティ、限定した利用者を対象にしていること、または、主ソースツリーに含めるには実験的すぎることが主な理由でPostgreSQLのコアシステムにはないプラグイン機能が含まれます。これはその有用性を妨げるものではありません。

この付録では、contribにあるエクステンションやその他のサーバプラグインモジュールを説明します。付録Gは、ユーティリティプログラムをカバーしています。

ソース配布から構築する場合、"world"を対象に構築しない限り、これらのモジュールは自動的に構築されません(ステップ 2参照)。次のコマンドを実行することで、これらすべてを構築しインストールすることができます。

```
make
make install
```

選択した1つのモジュールのみを構築しインストールするには、そのモジュールのディレクトリで同じコマンドを行ってください。多くのモジュールにはリグレッションテストがあり、インストール前であれば以下のコマンドで、

```
make check
```

PostgreSQLサーバが動いている状態であれば以下のコマンドで実行できます。

```
make installcheck
```

PostgreSQLのパッケージ化されたバージョンを使用している場合は通常、例えばpostgresql-contribのような別途副パッケージとしてこれらのモジュールが利用可能です。

多くのモジュールは新しいユーザ定義関数、演算子、型を提供します。こうしたモジュールの1つを使用できるようにするためには、コードをインストールした後に、新しいSQLオブジェクトをデータベースサーバに登録する必要があります。これはCREATE EXTENSIONコマンドを実行することで行われます。新しいデータベースでは、以下のように簡単に行うことができます。

```
CREATE EXTENSION module_name;
```

このコマンドは現在のデータベースの中にのみ新しいSQLオブジェクトを登録します。このため、そのモジュールの機能を利用可能にさせたいデータベース毎にこのコマンドを実行しなければなりません。その拡張が今後作成されるデータベースにデフォルトでコピーされるようにtemplate1データベースに対して実行する方法もあります。

これら全てのモジュールについてCREATE EXTENSIONはデータベーススーパーユーザによって実行される必要がありますが、モジュールが「trusted」と見なされる場合は現在のデータベースに対してCREATE権限を持つユーザであれば誰でも実行することができます。信頼されているモジュールは、以降の節でそのように識別されています。一般的に信頼されているモジュールとは、データベース外の機能へのアクセスを提供できないモジュールのことです。

多くのモジュールはユーザが選択したスキーマ内にそのオブジェクトをインストールすることができます。これを行うためにはCREATE EXTENSIONコマンドにSCHEMA schema_nameを追加してください。デフォルトでは、オブジェクトは現在の作成対象スキーマ内に格納され、そのスキーマのデフォルトはpublicです。

しかしながら、いくつかのモジュールはこの意味での「エクステンション」ではなく、例えばshared_preload_librariesといった他の方法でサーバにロードされることに注意してください。各モジュールの詳細はドキュメントを参照してください。

F.1. adminpack

adminpackは、pgAdminやその他の管理・運用ツールがサーバログファイルの遠隔管理を行うなどの、追加的な機能を提供できるようにするための数多くのサポート関数を提供します。デフォルトでは、この関数の使用はすべてスーパーユーザに限定されていますが、GRANTコマンドを使用して他のユーザーに許可されている場合があります。

表 F.1に示す関数はサーバをホスティングしているマシン上のファイルに対して書き込みアクセスを提供します。(表 9.95の関数も参照してください。そちらは読み取り専用アクセスを提供します。) ユーザがスーパーユーザか、関数に応じたpg_read_server_files、またはpg_write_server_filesロールのいずれかのロールを与えられていない限り、データベースクラスタディレクトリ内のファイルにのみアクセス可能です。ただし、相対パスと絶対パスのどちらも利用できます。

表F.1 adminpack関数

関数 説明
pg_catalog.pg_file_write (filename text, data text, append boolean) → bigint テキストファイルに書き込む、または追記する
pg_catalog.pg_file_sync (filename text) → void ファイルまたはディレクトリをディスクにフラッシュする
pg_catalog.pg_file_rename (oldname text, newname text [, archivename text]) → boolean ファイル名を変更する
pg_catalog.pg_file_unlink (filename text) → boolean ファイルを削除する
pg_catalog.pg_logdir_ls () → setof record log_directoryディレクトリ内のログファイルの一覧を表示する

pg_file_writeは指定されたdataをfilenameで指定されたファイルに書き込みます appendが偽であれば、ファイルは既に存在していてはいけません。appendが真であれば、ファイルが既に存在していても構いません。その場合、追記されます。書き込んだバイト数を返します。

pg_file_syncはfilenameで指定されたファイルまたはディレクトリをfsyncします。(例えば、指定されたファイルが存在しないなど)失敗するとエラーを発生します。data_sync_retryはこの関数には影響しませんので、データベースファイルのフラッシュの失敗であってもPANICレベルのエラーは起こらないことに注意してください。

pg_file_renameはファイルの名前を変更します。archivenameが省略されたり、NULLである場合は、単純にoldnameをnewname(既に存在していてはいけません)に変更します。archivenameが指定されていれば、まずnewnameをarchivename(既に存在していてはいけません)に変更し、それからoldnameをnewnameに変更します。第2段階の名前の変更が失敗した場合には、エラーを報告する前にarchivenameをnewnameに戻そうとします。成功した場合には真を、元のファイルが存在しなかったり、書き込みできなかった場合には偽を返します。その他の場合にはエラーを発生します。

pg_file_unlinkは指定されたファイルを削除します。成功した場合には真を、指定されたファイルが存在しなかったり、unlink()の呼出しが失敗した場合には偽を返します。その他の場合にはエラーを発生します。

pg_logdir_lsはlog_directoryディレクトリ内にあるログファイルすべての開始時のタイムスタンプとパス名を返します。この関数を使うには、log_filenameパラメータはデフォルト設定(postgresql-%Y-%m-%d_%H%M%S.log)でなければなりません。

F.2. amcheck

amcheckモジュールは、リレーションの構造の論理的な一貫性を検査する機能を提供します。構造が適正であると見なされれば、エラーは報告されません。

この関数は、特定のリレーションの構造表現における様々な不変量を検査します。インデックスの走査や、その他の重要な操作を担うアクセスメソッド関数の正しさは、これらの不変量を常に保つことに依存しています。たとえば、ある関数は、とりわけすべてのB-Treeページの中の項目が「論理的な」順序になっていることを検査します。(たとえばtextのB-Treeインデックスでは、インデックスタプルは語句の照合順になっていなければなりません。) その特定の不変量が何らかの理由で保たれなければ、該当するページで二分探索が不正なインデックス走査をもたらし、SQL問い合わせに誤った答えを返すことになるでしょう。

検証は、インデックス走査自身で使われるのと同じ手続きを用いて行われます。その手続きは、ユーザ定義演算子クラスのコードかもしれません。たとえば、B-Treeインデックスの検査は、一つ以上のB-Treeサポート関数1ルーチンを用いる比較に依存しています。演算子クラスサポート関数の詳細については37.16.3をご覧ください。

amcheck関数はスーパーユーザだけが使用できます。

F.2.1. 関数

bt_index_check(index regclass, heapallindexed boolean) returns void

bt_index_checkは対象となるB-Treeインデックスが、様々な不変量を維持していることをテストします。例を示します。

```
test=# SELECT bt_index_check(index => c.oid, heapallindexed => i.indisunique),
               c.relname,
               c.relpages
```

```
FROM pg_index i
JOIN pg_opclass op ON i.indclass[0] = op.oid
JOIN pg_am am ON op.opcmethod = am.oid
JOIN pg_class c ON i.indexrelid = c.oid
JOIN pg_namespace n ON c.relnamespace = n.oid
WHERE am.amname = 'btree' AND n.nspname = 'pg_catalog'
-- Don't check temp tables, which may be from another session:
AND c.relpersistence != 't'
-- Function may throw an error when this is omitted:
AND c.relkind = 'i' AND i.indisready AND i.indisvalid
ORDER BY c.relpages DESC LIMIT 10;
 bt_index_check |             relname             | relpages
----------------+---------------------------------+----------
                | pg_depend_reference_index       |       43
                | pg_depend_depender_index        |       40
                | pg_proc_proname_args_nsp_index  |       31
                | pg_description_o_c_o_index      |       21
                | pg_attribute_relid_attnam_index |       14
                | pg_proc_oid_index               |       10
                | pg_attribute_relid_attnum_index |        9
                | pg_amproc_fam_proc_index        |        5
                | pg_amop_opr_fam_index           |        5
                | pg_amop_fam_strat_index         |        5
(10 rows)
```

この例では、データベース「test」中のもっとも大きな10個のカタログインデックスの検証を行うセッションを示しています。インデックスタプルに対応するヒープタプルの存在の検証が、ユニークインデックスであるインデックスの一部に対して要求されています。エラーは出ていないので、テストしたすべてのインデックスは論理的に一貫していることがわかります。当然のことながら、この問い合わせは、検証可能なデータベース中のすべてのインデックスに対してbt_index_checkを呼び出すように変更できます。

bt_index_checkは、ターゲットとなるインデックスと、それが所属するヒープリレーションに対してAccessShareLockを獲得します。このロックモードは、単純なSELECT文がリレーションに対して獲得するのと同じものです。bt_index_checkは、子／親関係に渡る不変量を検査しませんが、heapallindexedがtrueの場合には、インデックス中のインデックスタプルに対応するすべてのヒープタプルの存在が検証されます、実行中のプロダクション環境で定期的、軽量なデータ破損検査が必要な場合、bt_index_checkを使うのが、検査の完全性と、アプリケーションの性能と稼働への影響を限定することとの間の最良のトレードオフになることがしばしばあります。

bt_index_parent_check(index regclass, heapallindexed boolean, rootdescend boolean) returns void

bt_index_parent_checkは、ターゲットとなるB-Treeインデックスが様々な不変量を保っていることを検査します。オプションとして、heapallindexed引数がtrueの場合、インデックスに対応して存在すべきすべてのヒープタプルの存在を検証します。省略可能な引数rootdescendがtrueであれば、各タプルに対するルートページから新しく探索することで、検証はリーフレベルのタプルを再び見つけます。bt_index_parent_checkにより実施される検査は、bt_index_checkにより実施される検査のスーパーセットになっています。bt_index_parent_checkは、bt_index_checkの更なる完璧版であると考えること

ができます。つまり、bt_index_checkと違ってbt_index_parent_checkは、インデックス構造中のダウンリンクに漏れがないことを含め、親／子関係に渡る不変量も検査します。bt_index_parent_checkは、論理的な非一貫性やその他の問題を発見した場合、一般的な習慣に従ってエラーを報告します。

bt_index_parent_checkは、ターゲットインデックスにShareLockを獲得することを必要とします。（ShareLockはヒープリレーションにも獲得されます。）このロックは、INSERT、UPDATE、DELETEが並行してデータ更新することを防ぎます。また、このロックはVACUUMその他のユーティリティコマンドによって、背後にあるリレーションが同時に処理されることを防ぎます。この関数は実行中のみロックを保持し、トランザクション全体に保持するのではないことに注意してください。

bt_index_parent_checkによる追加の検査では、様々な病的な事象を検出する可能性があります。これらの現象は、チェック対象のインデックスが使用している間違った実装がされたB-Tree演算子クラスによるものや、もしかしたら関連するB-Treeインデックスアクセスメソッドのコード中のまだ見つかっていないバグによるものなのかもしれません。bt_index_checkと違って、bt_index_parent_checkは、ホットスタンバイモードが有効な場合（すなわち、読み出し専用物理レプリカ）では使用できません。

ヒント

bt_index_checkとbt_index_parent_checkは両方とも、DEBUG1とDEBUG2の深刻度レベルで検証プロセスに関するログメッセージを出力します。このメッセージは、PostgreSQL開発者にとって興味のあるかもしれない検証プロセスに関する詳細な情報を提供します。検証が実際に非一貫性を検出する追加の状況を提供しますので、上級ユーザにもこの情報は役立つかもしれません。検証問い合わせを実行する前に対話式のpsqlセッションで

```
SET client_min_messages = DEBUG1;
```

を実行すると、扱いやすい程度の詳細で検証の進行状況に関するメッセージを表示します。

F.2.2. オプションheapallindexed検証

検証関数のheapallindexed引数がtrueならば、ターゲットのインデックスリレーションと関連付けられたテーブルに対して追加の検証フェーズが実施されます。これは「ダミー」のCREATE INDEX操作から構成され、インメモリ上の一時的なサマリー構造（これは必要に応じて基礎的な最初の検証フェーズで構築されます）に対する仮想的な新しいインデックスタプルがすべて存在することをチェックします。サマリー構造はターゲットのインデックスで見つかったすべてのタプルに対して「指紋採取(fingerprint)」を行います。heapallindexed検証の背後にある高レベルの原理は、新しいインデックスが既存のインデックスと等しいこと、ターゲットインデックスが既存の構造中に見つかったエントリーのみを含むことです。

追加のheapallindexedフェーズは大きなオーバーヘッドをもたらします。典型的には、検証に数倍時間かかるようになります。しかし、取得されたリレーションレベルのロックに対して、heapallindexed検証が実施されるときに変更はありません。

サマライズ構造は、maintenance_work_memによってその大きさが制限されます。インデックス中に存在すべきヒープタプルの非一貫性の検出失敗の確率が2%を超えないことを保証するために、タプルごとに約2バイトのメモリが必要です。タプルごとに利用可能なメモリが少ないほど、非一貫性を見逃す可能性が徐々に増

えていきます。この手法によって検証のオーバーヘッドを大幅に減らせる一方、とりわけ検証を日常的な保守作業として行うシステムでは、問題を検出できる確率が少し減少するだけです。失われた、あるいは不正なタプルは次の検証の機会に検出されます。

F.2.3. amcheckを効果的に使う

amcheckは、データページチェックサムが検知できないような、様々なタイプの障害モードを効果的に検知できます。以下のようなものがあります。

- 演算子クラスの正しくない実装によって引き起こされる構造の非一貫性。

  オペレーティングシステムの照合順の比較ルールの変更による問題も含まれます。textのような照合可能な型のデータの比較は、不変でなければならず（B-Treeインデックスの走査のための、すべての比較が不変でなければならないのと同じことです）、それはオペレーティングシステムの照合順が決して変化してはいけないことを意味します。まれであるとは言え、オペレーティングシステムの照合順ルールの更新は、これらの問題を引き起こします。もっと普通に起こることとしては、マスターサーバとスタンバイサーバの照合順の違いが関与することです。これは、使用されているオペレーティングシステムのメジャーバージョンに一貫性がないことによります。そうした一貫性の欠如は一般的にスタンバイサーバでのみ起こるので、通常スタンバイサーバでのみ検出されます。

  そうした問題が起きても、影響を受けた照合順を使って順序付けられた個々のインデックスには影響ないかもしれません。これは単純に、振る舞いにおける一貫性のなさにかかわらずインデックスされた値は同じ絶対的な順になるからです。PostgreSQLがオペレーティングシステムのロケールと照合順をどう使用するかについての更なる詳細については、23.1と23.2をご覧ください。

- インデックスとインデックス付されたヒープリレーションの間の構造的な非一貫性（heapallindexed検証が実施される場合）

  通常の操作においてはインデックスとそのヒープリレーションの間にはクロスチェックはありません。ヒープの破壊による症状は些細なものかもしれません。

- 依拠するPostgreSQLのアクセスメソッド、あるいはソート、トランザクション管理コードにおける、潜在的なまだ見つかっていないバグによる破損。

  新規、あるいは提案中の PostgreSQLの機能が、論理的な非一貫性をもたらしかねないかどうか全般的にテストする際に、インデックスの構造的な一貫性の自動検証が役立ちます。テーブル構造と、関連する可視性およびトランザクション状態情報の検証も同じような役割を果たします。わかりやすいテスト戦略の一つは、関数標準の回帰テストを実行中に、amcheckを継続的に呼び出すことです。テストの実行に関する詳細は、32.1をご覧ください。

- 単にチェックサムが有効になっていないファイルシステムあるいはストレージサブシステムの障害。

  amcheckは、ブロックをアクセスする際に共有バッファがヒットした場合、検査時に共有メモリバッファ上に表現されたページを調査します。結果として、amcheckは、検査時にファイルシステムから読み込んだデータを調査するとは限りません。チェックサムが有効な場合、amcheckは壊れたブロックをバッファに読み込んだ際にチェックサム障害によるエラーを報告するかもしれません。

- 欠陥のあるRAMあるいは、広範囲に渡るメモリサブシステムによる破損。

PostgreSQLは、訂正可能なメモリーエラーからは身を守らないので、業界標準のエラー訂正コード(ECC)、あるいはもっと優れた保護機構を使ったRAMを使って運用する前提となっています。しかし、ECCメモリは典型的には単一ビットエラーに対してのみ耐性があり、メモリ破損に起因する障害に対して完全な保護を提供すると考えるべきではありません。

heapallindexed検証が実施されると、一般に1ビットエラーを検出する可能性が非常に高くなります。これは、バイナリー致を厳密にテストすることと、またヒープ内のインデックス付けされたアトリビュートをテストすることによります。

一般的に、amcheckは破損の存在を証明することはできますが、破損がないことを証明することはできません。

F.2.4. 破損の修復

amcheckが報告するエラーが関与する破損で、偽陽性のものはありません。amcheckは、定義により発生してはならないはずの条件下で発生したエラーを報告するので、amcheckの報告するエラーを注意深く解析することがしばしば求められます。

amcheckが検出した問題を修復する一般的な方法はありません。不変条件違反の根本的な原因の説明が求められます。amcheckが検出した破損の診断には、pageinspectが有用な役割を担うかもしれません。REINDEXは破損の修復には効果的ではないかもしれません。

F.3. auth_delay

auth_delayはパスワードの総当たり攻撃をより難しくするために認証エラーの報告を行う前にわずかにサーバを停止させます。これはDoS攻撃を防ぐためのものでは無いことに注意してください。認証エラーを待たせ、コネクションスロットを消費させるため、DoS攻撃の影響を増長させるかもしれません。

この機能を有効にするためにはpostgresql.confの shared_preload_librariesよりモジュールをロードする必要があります。

F.3.1. 設定パラメータ

auth_delay.milliseconds (int)

指定されたミリ秒数認証エラーを返す前に待機します。デフォルトは0です。

これらのパラメータをpostgresql.confファイルに設定する必要があります。典型的な使用方法は以下のようになります。

```
# postgresql.conf
shared_preload_libraries = 'auth_delay'

auth_delay.milliseconds = '500'
```

F.3.2. 作者

KaiGai Kohei <kaigai@ak.jp.nec.com>

F.4. auto_explain

auto_explainモジュールは、手動でEXPLAINの実行を必要とせず、自動的に遅い文の実行計画をログ記録する手段を提供します。大きなアプリケーションにおける最適化されていない問い合わせを追跡するのに特に有用です。

このモジュールはSQLでアクセスできる関数を提供しません。使用するには、サーバに単に読み込ませます。ある個別のセッションに読み込ませることができます。

```
LOAD 'auto_explain';
```

（実行するためにはスーパーユーザでなければなりません。）より一般的な使用方法は、postgresql.confのsession_preload_librariesまたはshared_preload_librariesにauto_explainを含めて、特定のまたはすべてのセッションで事前にロードしておくことです。すると、想定外に低速な問い合わせを発生時に何も行うことなく追跡することができます。当然ながらこのためのオーバーヘッドという代償があります。

F.4.1. 設定パラメータ

auto_explainの動作を制御するいくつかの設定パラメータが存在します。デフォルトの動作は何もしないことなので、なんらかの結果を望むのであれば少なくともauto_explain.log_min_durationを設定しなければならないことに注意してください。

auto_explain.log_min_duration (integer)

auto_explain.log_min_durationは、文の実行計画がログに記録されるようになる、ミリ秒単位の最小の文実行時間です。これを0にすれば、すべての計画が記録されます。-1（デフォルト）は計画の記録を無効にします。例えば、250msに設定すると、250ms以上実行する文すべてが記録されます。スーパーユーザのみがこの設定を変更することができます。

auto_explain.log_analyze (boolean)

auto_explain.log_analyzeは、実行計画のログが取得されたときに出力されるものとして、単にEXPLAIN出力ではなく、EXPLAIN ANALYZE出力を行います。このパラメータはデフォルトで無効です。スーパーユーザのみがこの設定を変更できます。

> **注記**
>
> このパラメータが有効の場合、計画ノードごとの時間的調整は事実上ログされるまで如何に時間が掛かろうと、全ての実行文に対して引き起こります。極端に性能上のマイナスの影響が起こり得ます。auto_explain.log_timingを無効にすれば、得られる情報が少なくなるという犠牲を払って、性能の損失が改善されます。

auto_explain.log_buffers (boolean)

auto_explain.log_buffersは実行計画のログを取得するときに、バッファ使用統計を出力するかどうかを制御します。EXPLAINのBUFFERSオプションと同じです。auto_explain.log_analyzeパラメータが設定されていなければ、このパラメータは効果がありません。このパラメータはデフォルトで無効です。スーパーユーザのみがこの設定を変更することができます。

auto_explain.log_wal (boolean)

auto_explain.log_walは実行計画のログを取得するときに、WAL使用統計を出力するかどうかを制御します。EXPLAINのWALオプションと同じです。auto_explain.log_analyzeパラメータが設定されていなければ、このパラメータは効果がありません。このパラメータはデフォルトで無効です。スーパーユーザのみがこの設定を変更することができます。

auto_explain.log_timing (boolean)

auto_explain.log_timingは、実行計画のログが取得されたときに、ノード毎の時間的調整情報を出力するかどうかを制御します。EXPLAINのTIMINGオプションと同じです。システムクロックを繰り返し読み出すことによるオーバーヘッドのため、システムの中には問い合わせが非常に遅くなるものがありますので、実際の行数のみ必要で正確な時刻は必要でない場合にはこのパラメータを無効にすると役に立つかも知れません。auto_explain.log_analyzeが設定されていなければ、このパラメータは効果がありません。デフォルトで有効です。スーパーユーザのみがこの設定を変更することができます。

auto_explain.log_triggers (boolean)

auto_explain.log_triggersにより、実行計画のログを記録するときに、トリガ実行の統計を含めるようになります。auto_explain.log_analyzeパラメータが設定されていなければ、このパラメータは効果がありません。このパラメータはデフォルトで無効です。スーパーユーザのみがこの設定を変更することができます。

auto_explain.log_verbose (boolean)

auto_explain.log_verboseは、実行計画のログが取得されたときに、冗長な詳細が出力されるかどうかを制御します。EXPLAINのVERBOSEオプションと同じです。このパラメータはデフォルトで無効です。スーパーユーザのみがこの設定を変更できます。

auto_explain.log_settings (boolean)

auto_explain.log_settingsは、実行計画が記録される時に修正された設定オプションに関する情報を表示するかどうかを制御します。問い合わせ計画に影響し、組み込みのデフォルトの値と異なる値であるオプションだけが出力に含まれます。このパラメータはデフォルトで無効です。スーパーユーザのみがこの設定を変更できます。

auto_explain.log_format (enum)

auto_explain.log_formatは使用するEXPLAIN出力書式を選びます。許容される値はtext、xml、json、yamlです。デフォルトはtextです。スーパーユーザのみがこの設定を変更することができます。

auto_explain.log_level (enum)

auto_explain.log_levelは、auto_explainが問い合わせ計画を記録するログレベルを選択します。有効な値はDEBUG5、DEBUG4、DEBUG3、DEBUG2、DEBUG1、INFO、NOTICE、WARNING、LOGです。デフォルトはLOGです。スーパーユーザのみがこの設定を変更できます。

auto_explain.log_nested_statements (boolean)

auto_explain.log_nested_statementsにより、入れ子状の文(関数内から実行される文)を考慮して記録するようになります。無効ならば、最上位の問い合わせ計画のみが記録されます。このパラメータはデフォルトで無効です。スーパーユーザのみがこの設定を変更することができます。

auto_explain.sample_rate (real)

auto_explain.sample_rateにより、auto_explainは各セッションで一部の文の実行計画のみをログに記録するようになります。デフォルトは1で、すべての問い合わせの実行計画をログに記録します。入れ子になった文の場合には、実行計画がすべてログに記録されるか、全くされないかのどちらかです。スーパーユーザのみがこの設定を変更できます。

スーパーユーザは自身のセッション内でその場で変更できますが、通常の使用では、これらパラメータはpostgresql.confに設定しなければなりません。典型的な使用方法は以下のようになります。

```
# postgresql.conf
session_preload_libraries = 'auto_explain'

auto_explain.log_min_duration = '3s'
```

F.4.2. 例

```
postgres=# LOAD 'auto_explain';
postgres=# SET auto_explain.log_min_duration = 0;
postgres=# SET auto_explain.log_analyze = true;
postgres=# SELECT count(*)
           FROM pg_class, pg_index
           WHERE oid = indrelid AND indisunique;
```

これにより、以下のようなログ出力が作成されます。

```
LOG:  duration: 3.651 ms  plan:
  Query Text: SELECT count(*)
              FROM pg_class, pg_index
              WHERE oid = indrelid AND indisunique;
  Aggregate  (cost=16.79..16.80 rows=1 width=0) (actual time=3.626..3.627 rows=1 loops=1)
    ->  Hash Join  (cost=4.17..16.55 rows=92 width=0) (actual time=3.349..3.594 rows=92 loops=1)
          Hash Cond: (pg_class.oid = pg_index.indrelid)
          ->  Seq Scan on pg_class  (cost=0.00..9.55 rows=255 width=4) (actual time=0.016..0.140
 rows=255 loops=1)
          ->  Hash  (cost=3.02..3.02 rows=92 width=4) (actual time=3.238..3.238 rows=92 loops=1)
                Buckets: 1024  Batches: 1  Memory Usage: 4kB
                ->  Seq Scan on pg_index  (cost=0.00..3.02 rows=92 width=4) (actual
 time=0.008..3.187 rows=92 loops=1)
                      Filter: indisunique
```

F.4.3. 作者

板垣 貴裕 <itagaki.takahiro.at.oss.ntt.co.jp>

F.5. bloom

bloomは、ブルームフィルター[1]によるインデックスのアクセスメソッドを提供します。

ブルームフィルターは、空間効率の良いデータ構造で、ある要素が集合のメンバーかどうかをテストするのに用いられます。インデックスのアクセスメソッドとして使用する場合、インデックス作成時に大きさが決まるシグネチャーを使って、条件を満たさないタプルを高速に除外することができます。

シグネチャーはインデックス化された属性を非可逆的に表現するもので、その性質上、偽陽性の結果を出すことがあります。すなわち、集合の中にない要素が、集合の中にあると報告するかもしれません。ですから、インデックスの検索結果は、ヒープエントリ中の実際の属性値を使って、必ず再検査しなければなりません。シグネチャーが大きくなれば偽陽性の可能性が下がるので不必要なヒープの検索は減りますが、もちろんそうなるとインデックスが大きくなるので、スキャンが遅くなります。

この種のインデックスは、テーブルに多数の属性があり、その任意の組み合わせを検索する問い合わせを実行するときにもっとも有効です。伝統的なbtreeインデックスはブルームインデックスよりも高速ですが、 可能なすべての問い合わせをサポートするためには多数のbtreeインデックスが必要なのに対し、ブルームインデックスでは、たった一つのブルームインデックスだけで事足ります。しかし、ブルームインデックスでは等価検索だけをサポートすることに注意してください。btreeインデックスでは、等価だけでなく、範囲検索も実行できます。

F.5.1. パラメータ

bloomインデックスは、WITH句中の以下のパラメータを受け付けます。

length

ビット単位の個々のシグネチャー(インデックスエントリー)の長さ。16の倍数に近い値に丸められます。デフォルトは80ビットで、最大値は4096です。

col1 — col32

各インデックスカラムに対して生成するビット数。各々のパラメータ名は、管理対象のインデックスカラムの番号です。デフォルトは2ビットで、最大値は4095です。実際には使用されないインデックスカラムについてのパラメータは無視されます。

F.5.2. Examples

ブルームインデックスの作成例です。

[1] https://en.wikipedia.org/wiki/Bloom_filter

```
CREATE INDEX bloomidx ON tbloom USING bloom (i1,i2,i3)
       WITH (length=80, col1=2, col2=2, col3=4);
```

このインデックスは80ビット長のシグネチャーで作成され、属性i1とi2は2ビットに、i3は4ビットにマップされます。length、col1、col2指定はデフォルト値を使っているので、省略しても構いません。

より完全なブルームインデックスの定義と使用法を示します。比較のために、これと同等のbtreeインデックスも併せて示します。ブルームインデックスはbtreeインデックスよりもかなり小さく、また、より良い性能を発揮できるかもしれません。

```
=# CREATE TABLE tbloom AS
   SELECT
     (random() * 1000000)::int as i1,
     (random() * 1000000)::int as i2,
     (random() * 1000000)::int as i3,
     (random() * 1000000)::int as i4,
     (random() * 1000000)::int as i5,
     (random() * 1000000)::int as i6
   FROM
  generate_series(1,10000000);
SELECT 10000000
```

これだけ大きなテーブルに対するシーケンシャルスキャンは長い時間がかかります。

```
=# EXPLAIN ANALYZE SELECT * FROM tbloom WHERE i2 = 898732 AND i5 = 123451;
                                                  QUERY PLAN


-------------------------------------------------------------------
-----------------------------------
 Seq Scan on tbloom  (cost=0.00..2137.14 rows=3 width=24) (actual time=15.480..15.480 rows=0
 loops=1)
   Filter: ((i2 = 898732) AND (i5 = 123451))
   Rows Removed by Filter: 100000
 Planning Time: 0.340 ms
 Execution Time: 15.501 ms
(5 rows)
```

たとえbtreeインデックスが定義されていたとしても、結果はまだシーケンシャルスキャンです。

```
=# CREATE INDEX btreeidx ON tbloom (i1, i2, i3, i4, i5, i6);
CREATE INDEX
=# SELECT pg_size_pretty(pg_relation_size('btreeidx'));
 pg_size_pretty
----------------
 3976 kB
(1 row)
```

```
=# EXPLAIN ANALYZE SELECT * FROM tbloom WHERE i2 = 898732 AND i5 = 123451;
                                                  QUERY PLAN

-----------------------------------------------------------------------
-----------------------------------
 Seq Scan on tbloom  (cost=0.00..2137.00 rows=2 width=24) (actual time=12.604..12.604 rows=0
 loops=1)
   Filter: ((i2 = 898732) AND (i5 = 123451))
   Rows Removed by Filter: 100000
 Planning Time: 0.155 ms
 Execution Time: 12.617 ms
(5 rows)
```

そのテーブルにブルームインデックスが定義されていれば、btreeよりもこの種の検索をうまく扱います。

```
=# CREATE INDEX bloomidx ON tbloom USING bloom (i1, i2, i3, i4, i5, i6);
CREATE INDEX
=# SELECT pg_size_pretty(pg_relation_size('bloomidx'));
 pg_size_pretty
----------------
 1584 kB
(1 row)
=# EXPLAIN ANALYZE SELECT * FROM tbloom WHERE i2 = 898732 AND i5 = 123451;
                                                        QUERY PLAN

-----------------------------------------------------------------------
-------------------------------------------------
 Bitmap Heap Scan on tbloom  (cost=1792.00..1799.69 rows=2 width=24) (actual time=0.384..0.384
 rows=0 loops=1)
   Recheck Cond: ((i2 = 898732) AND (i5 = 123451))
   Rows Removed by Index Recheck: 26
   Heap Blocks: exact=26
   ->  Bitmap Index Scan on bloomidx  (cost=0.00..1792.00 rows=2 width=0) (actual
 time=0.350..0.350 rows=26 loops=1)
         Index Cond: ((i2 = 898732) AND (i5 = 123451))
 Planning Time: 0.122 ms
 Execution Time: 0.407 ms
(8 rows)
```

btree検索の主要な問題は、検索条件が、先頭（そしてそれに続く）インデックスカラムを使用しないときに、効率が悪くなってしまうことです。btreeでは各々のカラムに対して別々のインデックスを作るのが良い戦略です。するとプランはこのような選択をします。

```
=# CREATE INDEX btreeidx1 ON tbloom (i1);
CREATE INDEX
=# CREATE INDEX btreeidx2 ON tbloom (i2);
```

```
CREATE INDEX
=# CREATE INDEX btreeidx3 ON tbloom (i3);
CREATE INDEX
=# CREATE INDEX btreeidx4 ON tbloom (i4);
CREATE INDEX
=# CREATE INDEX btreeidx5 ON tbloom (i5);
CREATE INDEX
=# CREATE INDEX btreeidx6 ON tbloom (i6);
CREATE INDEX
=# EXPLAIN ANALYZE SELECT * FROM tbloom WHERE i2 = 898732 AND i5 = 123451;
                                                        QUERY PLAN


------------------------------------------------------------------
-----------------------------------------------------
 Bitmap Heap Scan on tbloom  (cost=24.34..32.03 rows=2 width=24) (actual time=0.032..0.033 rows=0
 loops=1)
   Recheck Cond: ((i5 = 123451) AND (i2 = 898732))
   ->  BitmapAnd  (cost=24.34..24.34 rows=2 width=0) (actual time=0.029..0.030 rows=0 loops=1)
         ->  Bitmap Index Scan on btreeidx5  (cost=0.00..12.04 rows=500 width=0) (actual
 time=0.029..0.029 rows=0 loops=1)
               Index Cond: (i5 = 123451)
         ->  Bitmap Index Scan on btreeidx2  (cost=0.00..12.04 rows=500 width=0) (never executed)
               Index Cond: (i2 = 898732)
 Planning Time: 0.537 ms
 Execution Time: 0.064 ms
(9 rows)
```

個別のインデックスのどれかを使うよりもこの問い合わせはずっと高速に実行できますが、インデックスのサイズにペナルティーを払わなければなりません。各々の単一カラムのbtreeインデックスは、2MBになります。ですから、全体で必要なスペースは12MBです。ブルームインデックスで使用するスペースの8倍以上です。

F.5.3. 演算子クラスインタフェース

ブルームインデックスの演算子クラスには、インデックス対象のデータ型に対するハッシュ関数と、検索のための等価演算子だけが必要です。この例では、textデータ型に対する演算子クラスの定義を示します。

```
CREATE OPERATOR CLASS text_ops
DEFAULT FOR TYPE text USING bloom AS
    OPERATOR    1   =(text, text),
    FUNCTION    1   hashtext(text);
```

F.5.4. 制限事項

- このモジュールには、int4とtextに対する演算子クラスだけが含まれています。

- =演算子だけが検索ではサポートされています。しかし、配列の和、積演算のサポートを将来追加することは可能です。
- bloomアクセスメソッドはUNIQUEをサポートしていません。
- bloomアクセスメソッドはNULL値の検索をサポートしていません。

F.5.5. 作者

Teodor Sigaev <teodor@postgrespro.ru>, Postgres Professional, Moscow, Russia

Alexander Korotkov <a.korotkov@postgrespro.ru>, Postgres Professional, Moscow, Russia

Oleg Bartunov <obartunov@postgrespro.ru>, Postgres Professional, Moscow, Russia

F.6. btree_gin

btree_ginは、次に列挙するデータ型に対しB-treeと同等な動作を実装するGIN演算子クラスを提供します。データ型は、int2、int4、int8、float4、float8、timestamp with time zone、timestamp without time zone、time with time zone、time without time zone、date、interval、oid、money、"char"、varchar、text、bytea、bit、varbit、macaddr、macaddr8、inet、cidr、uuid、name、bool、bpcharおよびすべてのenum型です。

一般的に、これらの演算子クラスは同等な標準B-treeインデックス方式を性能的に凌駕する物ではなく、標準B-treeコードの1つの重要機能である一意性強要の能力を欠いています。しかしながら、GINの試験、およびその他のGIN演算子クラスの開発の基礎として便利です。同時に、GINインデックス化可能列およびB-treeインデックス化可能列双方を試験する問い合わせに対し、ビットマップを介してANDを取り一体化されるべき2つの別々のインデックスを作成するよりも、これらの演算子クラスの1つを使用する複数列GINインデックスを作成するほうがより効率的です。

このモジュールは「trusted」と見なされます。つまり、現在のデータベースに対してCREATE権限を持つ非スーパーユーザがインストールできます。

F.6.1. 使用例

```
CREATE TABLE test (a int4);

-- インデックスの作成
CREATE INDEX testidx ON test USING GIN (a);

-- 問い合わせ
SELECT * FROM test WHERE a < 10;
```

F.6.2. 作者

Teodor Sigaev（<teodor@stack.net>）および Oleg Bartunov（<oleg@sai.msu.su>）。追加情報はhttp://www.sai.msu.su/~megera/oddmuse/index.cgi/Ginを参照ください。

F.7. btree_gist

btree_gistは、次に列挙するデータ型に対しB-treeと同等な動作を実装するGiSTインデックス演算子クラスを提供します。データ型は、int2、int4、int8、float4、float8、numeric、timestamp with time zone、timestamp without time zone、time with time zone、time without time zone、date、interval、oid、money、char、varchar、text、bytea、bit、varbit、macaddr、macaddr8、inet、cidr、uuid、およびすべてのenum型です。

一般的に、これらの演算子クラスは同等な標準B-treeインデックス方式を性能的に凌駕する物ではなく、標準B-treeコードの1つの重要機能である一意性強要の能力を欠いています。しかしながら、以下で述べるようにB-treeインデックスにはない特徴をいくつか備えています。また、これらの演算子クラスは、GiSTでのみインデックス可能なデータ型の列もあれば、単純なデータ型の列もあるような複数列のGiSTインデックスが必要な場合に便利です。最後に、GiSTの試験、およびその他のGiST演算子クラスの開発の基礎として便利です。

典型的なB-tree検索演算子に加えて、btree_gistは<>(「等しくない」)に対してもインデックスのサポートを提供します。これは、後で述べるような排他制約と組み合わせると便利でしょう。

また、自然な距離のあるデータ型には、btree_gistは距離演算子<->を定義し、この演算子を使った最近接検索へのGiSTインデックスのサポートを提供します。距離演算子はint2、int4、int8、float4、float8、timestamp with time zone、timestamp without time zone、time without time zone、date、interval、oid、moneyに提供されます。

このモジュールは「trusted」と見なされます。つまり、現在のデータベースに対してCREATE権限を持つ非スーパーユーザがインストールできます。

F.7.1. 使用例

btreeの代わりにbtree_gistを使った簡単な例

```
CREATE TABLE test (a int4);

-- インデックスの作成
CREATE INDEX testidx ON test USING GIST (a);

-- 問い合わせ
SELECT * FROM test WHERE a < 10;

-- 最近接検索: "42"に一番近い10個のエントリを見つける
SELECT *, a <-> 42 AS dist FROM test ORDER BY a <-> 42 LIMIT 10;
```

動物園の一つの檻に1種類の動物しかいないというルールを強制するために排他制約を使います。

```
=> CREATE TABLE zoo (
  cage   INTEGER,
  animal TEXT,
```

```
  EXCLUDE USING GIST (cage WITH =, animal WITH <>)
);

=> INSERT INTO zoo VALUES(123, 'zebra');
INSERT 0 1
=> INSERT INTO zoo VALUES(123, 'zebra');
INSERT 0 1
=> INSERT INTO zoo VALUES(123, 'lion');
ERROR:  conflicting key value violates exclusion constraint "zoo_cage_animal_excl"
DETAIL:  Key (cage, animal)=(123, lion) conflicts with existing key (cage, animal)=(123, zebra).
=> INSERT INTO zoo VALUES(124, 'lion');
INSERT 0 1
```

F.7.2. 作者

Teodor Sigaev (<teodor@stack.net>)、Oleg Bartunov (<oleg@sai.msu.su>)、Janko Richter (<jankorichter@yahoo.de>)、およびPaul Jungwirth (<pj@illuminatedcomputing.com>)。追加情報はhttp://www.sai.msu.su/~megera/postgres/gist/を参照ください。

F.8. citext

citextモジュールは、大文字小文字の区別がない文字列型を提供します。これは値の比較の際、基本的に内部でlowerを呼び出します。この他はほぼtextと同様に動作します。

> **ヒント**
>
> このモジュールの代わりに非決定論的照合順序(23.2.2.4参照)を使うことを検討してください。大文字小文字を区別しない比較、アクセントを区別しない比較、その他の組み合わせに対して使えますし、より多くのユニコードの特別な場合を正しく扱います。

このモジュールは「trusted」と見なされます。つまり、現在のデータベースに対してCREATE権限を持つ非スーパーユーザがインストールできます。

F.8.1. 原理

PostgreSQLにおいて大文字小文字の区別のない比較を行う標準的な手法は、値を比べる際に以下のようにlower関数を使用することでした。例です。

```
SELECT * FROM tab WHERE lower(col) = LOWER(?);
```

これはまあまあ動作しますが、数多くの欠点があります。

- 作成するSQL文を冗長にします。また常に列と問い合わせの値両方にlowerを使用することを忘れないようにしなければなりません。
- lowerを使用して関数インデックスを作成していない限り、インデックスを使用しません。
- UNIQUEまたはPRIMARY KEYとして列を宣言するのであれば、暗黙的に生成されるインデックスは大文字小文字を区別します。このため、大文字小文字を区別しない検索では使えず、また、大文字小文字を区別しない一意性を強制させられません。

citextデータ型によりSQL問い合わせ内のlower呼び出しを省くことができます。さらに、大文字小文字の区別がない主キーを実現できます。citextはtextと同様にロケールも考慮します。つまり大文字と小文字のマッチングは、LC_CTYPEデータベース設定の規則に依存します。ここでも、この動作はlowerを使用した問い合わせと同一です。しかしこのデータ型により、ロケールの考慮は透過的に行われますので、問い合わせで特殊なことを行うことを覚えておく必要はありません。

F.8.2. 使用方法

簡単な例を示します。

```
CREATE TABLE users (
    nick CITEXT PRIMARY KEY,
    pass TEXT   NOT NULL
);

INSERT INTO users VALUES ( 'larry',  sha256(random()::text::bytea) );
INSERT INTO users VALUES ( 'Tom',    sha256(random()::text::bytea) );
INSERT INTO users VALUES ( 'Damian', sha256(random()::text::bytea) );
INSERT INTO users VALUES ( 'NEAL',   sha256(random()::text::bytea) );
INSERT INTO users VALUES ( 'Bjø rn',  sha256(random()::text::bytea) );

SELECT * FROM users WHERE nick = 'Larry';
```

SELECT文は、nick列がlarryに設定され、問い合わせがLarryに対してであっても、1つのタプルを返します。

F.8.3. 文字列比較の動作

citextはそれぞれの文字列を（lowerが呼ばれますが）小文字に変換して結果を普通に比較します。よって、例えばlowerで小文字にした場合に同じ結果となるような2つの文字列が等しいとみなされます。

大文字小文字の区別のない照合をできる限り正確にエミュレートするために、数多くのcitext独自版の各種文字列処理演算子と関数があります。したがって、例えば正規表現演算子~および~*は、citextに適用する時に同じ動作を提供します。これら両方は大文字小文字を区別することなくマッチします。!~や!~*だけではなくLIKE演算子、~~、~~*、!~~、!~~*でも同じことが言えます。もし大文字小文字を区別して比較したい場合は、演算子の引数をtextにキャストすることができます。

引数がcitextであれば、同様にして以下の関数は大文字小文字を区別しない一致を実行します。

- regexp_match()
- regexp_matches()
- regexp_replace()
- regexp_split_to_array()
- regexp_split_to_table()
- replace()
- split_part()
- strpos()
- translate()

正規表現関数（RegExp関数）では、大文字小文字を区別して一致させたい場合「c」フラグを付けて、強制的に大文字小文字を区別して一致させることができます。そうしないと、大文字小文字を区別させたい場合にはこれらの関数のいずれかを使用する前段階でtextにキャストしなければなりません。

F.8.4. 制限

- citextの大文字小文字を区別しない動作は使用するデータベースのLC_CTYPEに依存します。どのように値を比較するかは、データベースが作成されたときに決定されます。Unicode標準の定義という観点では、真に大文字小文字の区別がないわけではありません。実質的に何を意味しているかというと、使用している照合が十分なものであれば、citextによる比較も十分なものになるはずです。しかしデータベースに様々な言語でデータを格納している場合は、ある言語のユーザは照合が他の言語用のものであった場合想定外の問い合わせ結果を得るかもしれません。

- PostgreSQL 9.1では、COLLATE指定をcitext列もしくはデータ値に付け加えることができます。現状では、citext演算子は大文字小文字を含んだ文字列を比較する際に、デフォルトではないCOLLATE指定を重んじます。しかし、最初の小文字変換はデータベースのLC_CTYPE設定にしたがって、常に実行されます（つまり、COLLATE "default"が指定されたようになります）これは、両方のステップが入力されたCOLLATE指定に従うように、将来のリリースにおいて変更されるでしょう。

- 演算子関数およびB-tree比較関数でデータの複製を作成しそれを比較のために小文字に変換しなければなりませんので、citextはtextほど効率的ではありません。また、textだけがB-Tree重複排除をサポートできます。しかし大文字小文字の区別をしない一致をさせるためにlowerを使用する場合よりかなり効率的です。

- citextは、ある文脈では大文字小文字の区別を行い、またある文脈では大文字小文字の区別を行わない比較をする必要がある場合、あまり役に立ちません。標準的な解法はtext型を使用し、大文字小文字を区別する比較が必要であれば手作業でlower関数を使用することです。これは大文字小文字を区別しない比較の必要性がまれであれば、問題なく動作します。大文字小文字を区別しない比較がほとんどで、大文字小文字を区別する比較の必要性がまれである場合は、データをcitextとして格納し、大文字小文字を区別する比較の際にその列を明示的にtextにキャストすることを検討してください。どちらの場合でも、2種類の検索の両方を高速にするために2つのインデックスを作成しなければならないでしょう。

- citext演算子を含んだスキーマは、現在のsearch_path(典型的にはpublic)に 存在しなければいけません。もし無い場合は通常の大文字小文字が区別されるtext比較が代わりに呼び出されます。
- 比較のために文字列を小文字にする方法は、例えば、大文字1つに対応する小文字が2つある場合等、ユニコードの特別な場合を正しく扱えないことがあります。ユニコードはこの理由で大文字小文字の対応関係と大文字小文字の畳み込みを区別します。正しくその場合を扱うには、citextの代わりに非決定論的照合順序を使ってください。

F.8.5. 作者

David E. Wheeler <david@kineticode.com>

Donald Fraserによるcitextモジュール原本からのヒント

F.9. cube

本モジュールは、多次元立方体を表すためのcubeデータ型を実装します。(訳注:以下cubeを立方体と訳しますが、ここでのcubeが指しているものは、厳密には「(超)立方体」ではありません。正確には、それぞれの「(超)面」がある座標軸に対して垂直な「(超)直方体」です。)

このモジュールは「trusted」と見なされます。つまり、現在のデータベースに対してCREATE権限を持つ非スーパーユーザがインストールできます。

F.9.1. 構文

表 F.2はcube型で有効な外部表現を示します。x、yなどは浮動小数点数を意味します。

表F.2 cubeの外部表現

外部構文	意味
x	1次元の点。(すなわち、幅ゼロの一次元間隔)
(x)	同上
x1,x2,...,xn	n次元空間の点。内部的には体積0の立方体として表されます。
(x1,x2,...,xn)	同上
(x),(y)	xからyまで(またはその逆)の1次元の間隔。順序は関係ありません。
[(x),(y)]	同上
(x1,...,xn),(y1,...,yn)	対角の組み合わせで表されるn次元の立方体。
[(x1,...,xn),(y1,...,yn)]	同上

立方体の対角の入力順序は関係ありません。統一的な「左下 — 右上」という内部表現を作成するために必要ならば、cube関数は自動的に値を交換します。角が一致する場合、cube型には無駄を省くために、「点である」フラグを加えた一つの角のみ格納されます。

空白文字は入力時に無視されます。このため、[(x),(y)]は[(x), (y)]と同じです。

F.9.2. 精度

値は内部的に64ビットの浮動小数点数値として格納されます。つまり、有効桁が16より大きい数値は切り詰められることを意味します。

F.9.3. 使用方法

表 F.3は、cube型に提供されている専用の演算子を示します。

表F.3 cubeの演算子

演算子 説明
cube && cube → boolean 立方体が重なるか。
cube @> cube → boolean 1番目の立方体は2番目のものを含むか。
cube <@ cube → boolean 1番目の立方体は2番目のものに含まれるか。
cube -> integer → float8 立方体の(1から数えた)n次座標を取得します。
cube ~> integer → float8 以下のように立方体のn次座標を取得します。n = 2 * k - 1 はk次元の下界を、n = 2 * k はk次元の上界を意味します。負のnは、対応する正の座標の正負反転した値を示します。この演算子はKNN-GiSTのサポートのために設計されています。
cube <-> cube → float8 2つの立方体のユークリッド距離を計算します。
cube <#> cube → float8 2つの立方体のタクシー(L-1計量)距離を計算します。
cube <=> cube → float8 2つの立方体のチェビシェフ(L-無限大計量)距離を計算します。

(PostgreSQL 8.2以前では、包含演算子@>と<@はそれぞれ@と~と呼ばれていました。これらの名前はまだ利用できますが、廃止予定であり、最終的にはなくなります。古い名前がコアの幾何データ型が従う規約と反対であることに注意してください。)

上記の演算子に加えて、cube型では表 9.1にある通常の比較演算子が利用可能です。これらの演算子は、まず最初の座標を比較し、それらが同一の場合は2番目の座標を比較し、と続けます。それらは、主にcube型のためのB-treeインデックス演算子クラスをサポートするために存在し、例えばcube型の列にUNIQUE制約をつけたい場合に便利です。

cubeモジュールは、cube型の値用にGiSTインデックス演算子クラスも提供します。cube型GiSTインデックスは、WHERE句内にて=、&&、@>、<@演算子を用いて値を検索するために使用することができます。

加えて、cube型GiSTインデックスは、ORDER BY句内にて<->、<#>、<=>のメトリック演算子を用いて近傍値を発見するために使用することができます。例えば、3次元の点である(0.5, 0.5, 0.5)の最近傍点は、以下のように効率よく発見できます。

```
SELECT c FROM test ORDER BY c <-> cube(array[0.5,0.5,0.5]) LIMIT 1;
```

~>演算子でもこの方法で、選択された座標によってソートされた最初のいくつかの値を効率よく探査するために使用できます。例えば、1番目の座標(左下隅)によって昇順に並べ替えられた最初のいくつかの立方体を取得するために、以下のような問い合わせを使用することができます。

```
SELECT c FROM test ORDER BY c ~> 1 LIMIT 5;
```

そして、右上の1番目の座標によって昇順に並び替えられた2次元の立方体を取得するために、以下のような問い合わせを使用することができます。

```
SELECT c FROM test ORDER BY c ~> 3 DESC LIMIT 5;
```

表 F.4は有効な関数を示します。

表F.4 cubeの関数

関数 説明 例
cube (float8) → cube 同じ座標をもつ、1次元の立方体を作成します。 cube(1) → (1)
cube (float8, float8) → cube 1次元の立方体を作成します。 cube(1,2) → (1),(2)
cube (float8[]) → cube 配列で定義される座標を使用した体積0の立方体を作成します。 cube(ARRAY[1,2,3]) → (1, 2, 3)
cube (float8[], float8[]) → cube 2つの配列で定義される右上および左下座標を持つ立方体を作成します。配列長は同じでなければなりません。 cube(ARRAY[1,2], ARRAY[3,4]) → (1, 2),(3, 4)
cube (cube, float8) → cube 既存の立方体に次元を加え、新しい座標の同じ値の端点をもつ立方体を新しく作成します。これは計算した値で部品を追加しながら立方体を構築する場合に有用です。 cube('(1,2),(3,4)'::cube, 5) → (1, 2, 5),(3, 4, 5)
cube (cube, float8, float8) → cube 既存の立方体に次元を加えた立方体を新しく作成します。これは計算した値で部品を追加しながら立方体を構築する場合に有用です。 cube('(1,2),(3,4)'::cube, 5, 6) → (1, 2, 5),(3, 4, 6)
cube_dim (cube) → integer

関数 説明 例
立方体の次元数を返します。 cube_dim('(1,2),(3,4)') → 2
cube_ll_coord (cube, integer) → float8 立方体の左下隅のn次座標の値を返します。 cube_ll_coord('(1,2),(3,4)', 2) → 2
cube_ur_coord (cube, integer) → float8 立方体の右上隅のn次座標の値を返します。 cube_ur_coord('(1,2),(3,4)', 2) → 4
cube_is_point (cube) → boolean 立方体が点、つまり立方体が定義する2つの隅が同一の場合真を返します。 cube_is_point(cube(1,1)) → t
cube_distance (cube, cube) → float8 2つの立方体間の距離を返します。両方の立方体が点の場合、これは通常の距離測定関数です。 cube_distance('(1,2)', '(3,4)') → 2.8284271247461903
cube_subset (cube, integer[]) → cube 配列内の次元インデックスの一覧を使用して、既存の立方体から新しい立方体を作成します。単一次元の端点を展開するために使用したり、次元を除去したり、希望通りの順序に並び替えたりすることができます。 cube_subset(cube('(1,3,5),(6,7,8)'), ARRAY[2]) → (3),(7) cube_subset(cube('(1,3,5),(6,7,8)'), ARRAY[3,2,1,1]) → (5, 3, 1, 1),(8, 7, 6, 6)
cube_union (cube, cube) → cube 2つの立方体の和集合を作成します。 cube_union('(1,2)', '(3,4)') → (1, 2),(3, 4)
cube_inter (cube, cube) → cube 2つの立方体の共通部分を作成します。 cube_inter('(1,2)', '(3,4)') → (3, 4),(1, 2)
cube_enlarge (c cube, r double, n integer) → cube 最小でn次元において指定した径rで立方体のサイズを増加させます。径が負の場合、立方体は縮小されます。定義済のすべての次元は径rだけ変わります。左下座標をrだけ減少し、右上座標をrだけ増加します。左下座標が対応する右上座標よりも増加する場合(これはr < 0の場合にのみ発生します)、両方の座標はその平均値に設定されます。nが定義済の次元より多く、かつ、立方体が拡大される(r > 0)場合、n次元すべてを作成するために余分な次元が追加されます。余分な座標には、初期値として0が使用されます。この関数は、近傍点を見つけるための点を囲む外接矩形を作成する際に有用です。 cube_enlarge('(1,2),(3,4)', 0.5, 3) → (0.5, 1.5, -0.5),(3.5, 4.5, 0.5)

F.9.4. デフォルト

```
select cube_union('(0,5,2),(2,3,1)', '0');
cube_union
-------------------
(0, 0, 0),(2, 5, 2)
```

```
(1 row)
```

この和集合および以下の共通集合は一般常識と矛盾しないと思います。

```
select cube_inter('(0,-1),(1,1)', '(-2),(2)');
cube_inter
-------------
(0, 0),(1, 0)
(1 row)
```

次元が異なる立方体の二項演算すべてにおいて、より低い次元の方がデカルト投影、つまり、文字列表現で省略された座標に0を持つものになると仮定します。上の例は以下と同じです。

```
cube_union('(0,5,2),(2,3,1)','(0,0,0),(0,0,0)');
cube_inter('(0,-1),(1,1)','(-2,0),(2,0)');
```

以下の包含の述部は点構文を使用しますが、実際内部的には第2引数は矩形として表されます。この構文により、別の点用の型や(box,point)という述部用の関数を定義することが不要になります。

```
select cube_contains('(0,0),(1,1)', '0.5,0.5');
cube_contains
--------------
t
(1 row)
```

F.9.5. 注釈

使用例については、sql/cube.sqlリグレッションテストを参照してください。

破壊防止のために立方体の次元数に100までという制限を行いました。これはcubedata.hで設定されており、必要に応じて多少大きくすることができます。

F.9.6. クレジット

原作者: Gene Selkov, Jr. <selkovjr@mcs.anl.gov>, Mathematics and Computer Science Division, Argonne National Laboratory.

GiST (http://gist.cs.berkeley.edu/)の要点(gist)を説明してくださったJoe Hellerstein博士 (https://dsf.berkeley.edu/jmh/) に感謝します。また、Illustra用の例の作者である彼の以前の生徒Andy Dongに感謝します。また、自分の世界を作成できるようにし、静かに生活できるようにしてもらった、過去から現在までのすべてのPostgres開発者に感謝します。データベース研究を長年誠実にサポートしてくれたArgonne LabとU.S. Department of Energyにも感謝します。

2002年8月/9月にBruno Wolff III <bruno@wolff.to>による小規模な改修がこのパッケージになされました。この改修には、単精度から倍精度への精度の変更といくつかの関数の追加が含まれます。

2006年7月にJoshua Reich <josh@root.net>による改修がさらになされました。この改修にはcube(float8[], float8[])が含まれ、また、古いV0プロトコルからV1呼び出しプロトコルを使用するようコードが整理されました。

F.10. dblink

dblinkは、データベースセッション内から他のPostgreSQLデータベースへの接続をサポートするモジュールです。

postgres_fdwも参照して下さい。より新しく標準に対する互換性の高い基盤を使ってほぼ同じ機能を提供しています。

dblink_connect

dblink_connect — リモートデータベースへの永続的な接続を開きます

概要

```
dblink_connect(text connstr) returns text
dblink_connect(text connname, text connstr) returns text
```

説明

dblink_connect()はリモートのPostgreSQLデータベースへの接続を確立します。接続先のサーバとデータベースは標準のlibpq接続文字列を通して識別されます。省略可能ですが、名前を接続に割り当てることも可能です。複数の名前付きの接続を一度に開くことができますが、無名の接続は同時に1つしか許されません。接続は、閉ざされるまで、または、データベースセッションが終わるまで永続します。

接続文字列は同時に既存の外部サーバ名であっても構いません。外部サーバを定義する場合、外部データラッパーdblink_fdwを使用することを推奨します。後述の例とCREATE SERVER、CREATE USER MAPPINGを参照してください。

引数

connname

接続に使用する名前です。省略した場合、既存の無名の接続を閉ざし、無名の接続を開きます。

connstr

例えばhostaddr=127.0.0.1 port=5432 dbname=mydb user=postgres password=mypasswdといったlibpq形式の接続情報文字列です。詳細は33.1.1を参照してください。もしくは外部サーバ名です。

戻り値

状態を返します。これは常にOKです（何らかのエラーが起きるとこの関数は戻らずエラーとなるためです）。

注釈

信頼できないユーザが、安全なスキーマ使用パターンを適用していないデータベースへアクセスする際には、セッション開始時にsearch_pathから、第三者が書き込みができるスキーマを削除してください。これはたとえばconnstrに値-csearch_path=を設定することで可能となります。このような配慮は、dblinkに限ったことではありません。任意のSQLコマンドを実行するすべてのインタフェースに当てはまります。

スーパーユーザのみがパスワード認証がない接続を作成するためにdblink_connectを使用することができます。スーパーユーザ以外でこの機能が必要ならばdblink_connect_uを代わりに使用してください。

他のdblink関数内で接続情報文字列が混乱する危険が発生しますので、等号記号を含む接続名を選択することは勧めません。

例

```
SELECT dblink_connect('dbname=postgres options=-csearch_path=');
 dblink_connect
----------------
 OK
(1 row)

SELECT dblink_connect('myconn', 'dbname=postgres options=-csearch_path=');
 dblink_connect
----------------
 OK
(1 row)


-- 外部データラッパー（FOREIGN DATA WRAPPER）の機能
-- 注意：これが正常に機能するにはローカル接続にパスワード認証が必須です。
--       さもないと、dblink_connect():から以下のエラーを受け取ります。
--       ----------------------------------------------------------------------
--       ERROR:  password is required
--       DETAIL:  Non-superuser cannot connect if the server does not request a password.
--       HINT:  Target server's authentication method must be changed.

CREATE SERVER fdtest FOREIGN DATA WRAPPER dblink_fdw OPTIONS (hostaddr '127.0.0.1', dbname
 'contrib_regression');

CREATE USER regress_dblink_user WITH PASSWORD 'secret';
CREATE USER MAPPING FOR regress_dblink_user SERVER fdtest OPTIONS (user 'regress_dblink_user',
 password 'secret');
GRANT USAGE ON FOREIGN SERVER fdtest TO regress_dblink_user;
GRANT SELECT ON TABLE foo TO regress_dblink_user;

\set ORIGINAL_USER :USER
\c - regress_dblink_user
SELECT dblink_connect('myconn', 'fdtest');
 dblink_connect
----------------
 OK
(1 row)
```

```
SELECT * FROM dblink('myconn', 'SELECT * FROM foo') AS t(a int, b text, c text[]);
 a  | b |       c
----+---+---------------
  0 | a | {a0,b0,c0}
  1 | b | {a1,b1,c1}
  2 | c | {a2,b2,c2}
  3 | d | {a3,b3,c3}
  4 | e | {a4,b4,c4}
  5 | f | {a5,b5,c5}
  6 | g | {a6,b6,c6}
  7 | h | {a7,b7,c7}
  8 | i | {a8,b8,c8}
  9 | j | {a9,b9,c9}
 10 | k | {a10,b10,c10}
(11 rows)

\c - :ORIGINAL_USER
REVOKE USAGE ON FOREIGN SERVER fdtest FROM regress_dblink_user;
REVOKE SELECT ON TABLE foo FROM regress_dblink_user;
DROP USER MAPPING FOR regress_dblink_user SERVER fdtest;
DROP USER regress_dblink_user;
DROP SERVER fdtest;
```

dblink_connect_u

dblink_connect_u — リモートデータベースへの永続的な危険な接続を開きます

概要

```
dblink_connect_u(text connstr) returns text
dblink_connect_u(text connname, text connstr) returns text
```

説明

dblink_connect_u()は、非スーパーユーザが任意の認証方式を使用して接続することができる点を除き、dblink_connect()と同じです。

リモートサーバがパスワードを含まない認証方式を選択していた場合、セッションがローカルなPostgreSQLサーバを実行するユーザから構成されたものとなりますので、なりすましやその後の権限昇格が起こり得ます。また、リモートサーバがパスワードを要求したとしても、サーバ側のユーザに属する~/.pgpassファイルなどサーバの環境から提供されるパスワードになる可能性があります。これはなりすましの危険性だけでなく、信頼できないリモートサーバへのパスワードが漏れる可能性を引き起こします。このためdblink_connect_u()はまず、スーパーユーザ以外から呼び出すことができないように、PUBLICからすべての権限を取り除いた形でインストールされます。状況によっては、信頼できるとみなされた特定のユーザにdblink_connect_u()のEXECUTE権限を与えることが適切となる場合があります。しかしこれは注意して行わなければなりません。また、サーバのユーザに属する~/.pgpassファイルにはすべて、ホスト名としてワイルドカードを指定した項目をまったく含めないことを推奨します。

この他の情報はdblink_connect()を参照してください。

dblink_disconnect

dblink_disconnect — リモートデータベースへの永続的な接続を閉ざします

概要

```
dblink_disconnect() returns text
dblink_disconnect(text connname) returns text
```

説明

dblink_disconnect()はdblink_connect()で開かれた既存の接続を閉ざします。引数がない構文では無名の接続を閉ざします。

引数

connname

閉ざす名前付き接続の名前です。

戻り値

状態を返します。これは常にOKです（何らかのエラーが起きるとこの関数は戻らずエラーとなるためです）。

例

```
SELECT dblink_disconnect();
 dblink_disconnect
-------------------
 OK
(1 row)

SELECT dblink_disconnect('myconn');
 dblink_disconnect
-------------------
 OK
(1 row)
```

dblink

dblink — リモートデータベースで問い合わせを実行します

概要

```
dblink(text connname, text sql [, bool fail_on_error]) returns setof record
dblink(text connstr, text sql [, bool fail_on_error]) returns setof record
dblink(text sql [, bool fail_on_error]) returns setof record
```

説明

dblinkはリモートデータベースで問い合わせ（通常はSELECTですが行を返す任意のSQLコマンドを行うことができます）を実行します。

2つのtext型の引数が与えられた場合、一番目の引数はまず永続接続の名前を検索するために使われます。もし見つかれば、コマンドがその接続上で実行されます。見つからなければ、一番目の引数はdblink_connect用の接続情報文字列として扱われ、このコマンド実行時と同様に指定された接続が開きます。

引数

connname

使用する接続の名前です。無名の接続を使用する場合はこのパラメータを省略します。

connstr

上でdblink_connectで説明した接続情報文字列です。

sql

例えばselect * from fooといった、リモートデータベースで実行させるSQL問い合わせです。

fail_on_error

真（省略時のデフォルト）の場合、接続のリモート側で発生したエラーによりローカル側でもエラーが発生します。偽の場合リモート側のエラーはローカル側にはNOTICEとして報告され、この関数は行を返しません。

戻り値

この関数は問い合わせにより生成された行を返します。dblinkは任意の問い合わせで使用することができますので、これは特定の列集合を指定するのではなく、record型を返すものと宣言されています。これは呼び出し元の問い合わせで想定列集合を指定しなければならないことを意味します。さもないとPostgreSQLは何が想定されているかわかりません。以下に例を示します。

```
SELECT *
    FROM dblink('dbname=mydb options=-csearch_path=',
```

```
                'select proname, prosrc from pg_proc')
      AS t1(proname name, prosrc text)
    WHERE proname LIKE 'bytea%';
```

FROM句の「別名」部分は関数が返す列名とその型を指定しなければなりません。（別名内の列名の指定はSQL標準の構文ですが、列の型の指定はPostgreSQLの拡張です。）これによりシステムは、関数を実行する前に、*がどのように展開されるか、WHERE句内のpronameが何を参照するかを理解します。実行時、リモートデータベースから返る実際の問い合わせの結果がFROM句で示された列数と異なる場合エラーが発生します。しかし、列名は一致する必要はありません。また、dblinkは正確な型一致も強制しません。返されるデータ文字列がFROM句で宣言された列型の有効な入力である限り成功します。

注釈

前もって判明している問い合わせをdblinkで使用する簡便な方法はビューを作成することです。これにより問い合わせの度に列型の情報を記載することなく、ビュー内に隠すことができます。以下に例を示します。

```
CREATE VIEW myremote_pg_proc AS
  SELECT *
    FROM dblink('dbname=postgres options=-csearch_path=',
                'select proname, prosrc from pg_proc')
    AS t1(proname name, prosrc text);

SELECT * FROM myremote_pg_proc WHERE proname LIKE 'bytea%';
```

例

```
SELECT * FROM dblink('dbname=postgres options=-csearch_path=',
                     'select proname, prosrc from pg_proc')
  AS t1(proname name, prosrc text) WHERE proname LIKE 'bytea%';
  proname   |   prosrc
------------+------------
 byteacat   | byteacat
 byteaeq    | byteaeq
 bytealt    | bytealt
 byteale    | byteale
 byteagt    | byteagt
 byteage    | byteage
 byteane    | byteane
 byteacmp   | byteacmp
 bytealike  | bytealike
 byteanlike | byteanlike
 byteain    | byteain
 byteaout   | byteaout
(12 rows)

SELECT dblink_connect('dbname=postgres options=-csearch_path=');
```

```
 dblink_connect
----------------
 OK
(1 row)

SELECT * FROM dblink('select proname, prosrc from pg_proc')
  AS t1(proname name, prosrc text) WHERE proname LIKE 'bytea%';
  proname   |   prosrc
------------+------------
 byteacat   | byteacat
 byteaeq    | byteaeq
 bytealt    | bytealt
 byteale    | byteale
 byteagt    | byteagt
 byteage    | byteage
 byteane    | byteane
 byteacmp   | byteacmp
 bytealike  | bytealike
 byteanlike | byteanlike
 byteain    | byteain
 byteaout   | byteaout
(12 rows)

SELECT dblink_connect('myconn', 'dbname=regression options=-csearch_path=');
 dblink_connect
----------------
 OK
(1 row)

SELECT * FROM dblink('myconn', 'select proname, prosrc from pg_proc')
  AS t1(proname name, prosrc text) WHERE proname LIKE 'bytea%';
  proname   |   prosrc
------------+------------
 bytearecv  | bytearecv
 byteasend  | byteasend
 byteale    | byteale
 byteagt    | byteagt
 byteage    | byteage
 byteane    | byteane
 byteacmp   | byteacmp
 bytealike  | bytealike
 byteanlike | byteanlike
 byteacat   | byteacat
 byteaeq    | byteaeq
 bytealt    | bytealt
 byteain    | byteain
```

```
 byteaout   | byteaout
(14 rows)
```

dblink_exec

dblink_exec — リモートデータベースでコマンドを実行します

概要

```
dblink_exec(text connname, text sql [, bool fail_on_error]) returns text
dblink_exec(text connstr, text sql [, bool fail_on_error]) returns text
dblink_exec(text sql [, bool fail_on_error]) returns text
```

説明

dblink_execはリモートデータベースでコマンド（つまり行を返さない任意のSQL文）を実行します。

2つのtext型の引数が与えられた場合、一番目の引数はまず永続接続の名前を検索するために使われます。もし見つかれば、コマンドがその接続上で実行されます。見つからなければ、一番目の引数はdblink_connect用の接続情報文字列として扱われ、このコマンド実行時と同様に指定された接続が開きます。

引数

connname

使用する接続の名前です。無名の接続を使用する場合はこのパラメータを省略します。

connstr

上でdblink_connectで説明した接続情報文字列です。

sql

例えばinsert into foo values(0, 'a', '{"a0","b0","c0"}')といった、リモートデータベースで実行させるSQL問い合わせです。

fail_on_error

真（省略時のデフォルト）の場合、接続のリモート側で発生したエラーによりローカル側でもエラーが発生します。偽の場合リモート側のエラーはローカル側にはNOTICEとして報告され、この関数の戻り値はERRORになります。

戻り値

状態、つまりコマンドの状態またはERRORを返します。

例

```
SELECT dblink_connect('dbname=dblink_test_standby');
 dblink_connect
----------------
 OK
(1 row)

SELECT dblink_exec('insert into foo values(21, ''z'', ''{"a0","b0","c0"}'');');
   dblink_exec
-----------------
 INSERT 943366 1
(1 row)

SELECT dblink_connect('myconn', 'dbname=regression');
 dblink_connect
----------------
 OK
(1 row)

SELECT dblink_exec('myconn', 'insert into foo values(21, ''z'', ''{"a0","b0","c0"}'');');
   dblink_exec
------------------
 INSERT 6432584 1
(1 row)

SELECT dblink_exec('myconn', 'insert into pg_class values (''foo'')',false);
NOTICE:  sql error
DETAIL:  ERROR:  null value in column "relnamespace" violates not-null constraint

 dblink_exec
-------------
 ERROR
(1 row)
```

dblink_open

dblink_open — リモートデータベースでカーソルを開きます

概要

```
dblink_open(text cursorname, text sql [, bool fail_on_error]) returns text
dblink_open(text connname, text cursorname, text sql [, bool fail_on_error]) returns text
```

説明

dblink_open()はリモートデータベースでカーソルを開きます。その後カーソルをdblink_fetch()とdblink_close()で操作することができます。

引数

connname

使用する接続の名前です。無名の接続を使用する場合はこのパラメータを省略します。

cursorname

このカーソルに割り当てる名前です。

sql

例えばselect * from pg_classといった、リモートデータベースで実行させたいSELECT文です。

fail_on_error

真（省略時のデフォルト）の場合、接続のリモート側で発生したエラーによりローカル側でもエラーが発生します。偽の場合リモート側のエラーはローカル側にはNOTICEとして報告され、この関数の戻り値はERRORになります。

戻り値

状態、つまりOKまたはERRORを返します。

注釈

カーソルはトランザクション内でのみ持続することができますので、リモート側がまだトランザクションの内部でない場合、dblink_openはリモート側で明示的なトランザクションブロックを開始（BEGIN）します。このトランザクションは対応するdblink_closeが実行された時に同様に閉ざされます。dblink_openとdblink_closeの間にdblink_execを使用してデータを変更した場合、エラーが発生するこ

とに注意してください。また、dblink_closeの前にdblink_disconnectを使用すると、トランザクションがアボートしますので変更が失われることに注意してください。

例

```
SELECT dblink_connect('dbname=postgres options=-csearch_path=');
 dblink_connect
----------------
 OK
(1 row)

SELECT dblink_open('foo', 'select proname, prosrc from pg_proc');
 dblink_open
-------------
 OK
(1 row)
```

dblink_fetch

dblink_fetch — リモートデータベースで開いているカーソルから行を取り出します

概要

```
dblink_fetch(text cursorname, int howmany [, bool fail_on_error]) returns setof record
dblink_fetch(text connname, text cursorname, int howmany [, bool fail_on_error]) returns setof
 record
```

説明

dblink_fetchはdblink_openによりあらかじめ確立したカーソルから行を取り出します。

引数

connname

使用する接続の名前です。無名の接続を使用する場合はこのパラメータを省略します。

cursorname

行を取り出すカーソルの名前です。

howmany

受け取る行の最大数です。カーソルの現在位置から次のhowmany行を取り出し、カーソルの位置を前方に移動します。カーソルが終端まで達すると、これ以上の行は生成されません。

fail_on_error

真（省略時のデフォルト）の場合、接続のリモート側で発生したエラーによりローカル側でもエラーが発生します。偽の場合リモート側のエラーはローカル側にはNOTICEとして報告され、この関数は行を返しません。

戻り値

この関数はカーソルから取り出された行を返します。この関数を使用するためには、dblinkで説明したように、想定する列集合を指定する必要があります。

注釈

リモートカーソルから返る実際の列数とFROM句で指定された列数と異なる場合エラーが発生します。この場合リモート側のカーソルは、エラーが発生しなかった場合と同じ行数分位置が変わります。リモート側のFETCHが完了した後にローカル側でこの他のエラーが発生した場合も同じです。

例

```
SELECT dblink_connect('dbname=postgres options=-csearch_path=');
 dblink_connect
----------------
 OK
(1 row)

SELECT dblink_open('foo', 'select proname, prosrc from pg_proc where proname like ''bytea%''');
 dblink_open
-------------
 OK
(1 row)

SELECT * FROM dblink_fetch('foo', 5) AS (funcname name, source text);
 funcname |  source
----------+----------
 byteacat | byteacat
 byteacmp | byteacmp
 byteaeq  | byteaeq
 byteage  | byteage
 byteagt  | byteagt
(5 rows)

SELECT * FROM dblink_fetch('foo', 5) AS (funcname name, source text);
 funcname  |  source
-----------+-----------
 byteain   | byteain
 byteale   | byteale
 bytealike | bytealike
 bytealt   | bytealt
 byteane   | byteane
(5 rows)

SELECT * FROM dblink_fetch('foo', 5) AS (funcname name, source text);
  funcname  |   source
------------+------------
 byteanlike | byteanlike
 byteaout   | byteaout
(2 rows)

SELECT * FROM dblink_fetch('foo', 5) AS (funcname name, source text);
 funcname | source
----------+--------
(0 rows)
```

dblink_close

dblink_close — リモートデータベースでカーソルを閉ざします

概要

```
dblink_close(text cursorname [, bool fail_on_error]) returns text
dblink_close(text connname, text cursorname [, bool fail_on_error]) returns text
```

説明

dblink_closeは前もってdblink_openで開かれたカーソルを閉ざします。

引数

connname

使用する接続の名前です。無名の接続を使用する場合はこのパラメータを省略します。

cursorname

閉ざすカーソルの名前です。

fail_on_error

真（省略時のデフォルト）の場合、接続のリモート側で発生したエラーによりローカル側でもエラーが発生します。偽の場合リモート側のエラーはローカル側にはNOTICEとして報告され、この関数の戻り値はERRORになります。

戻り値

状態、つまりOKまたはERRORを返します。

注釈

dblink_openが明示的なトランザクションブロックを開始し、これが接続上で最後まで開き続けているカーソルであった場合、dblink_closeは対応するCOMMITを発行します。

例

```
SELECT dblink_connect('dbname=postgres options=-csearch_path=');
 dblink_connect
```

```
----------------
 OK
(1 row)

SELECT dblink_open('foo', 'select proname, prosrc from pg_proc');
 dblink_open
-------------
 OK
(1 row)

SELECT dblink_close('foo');
 dblink_close
--------------
 OK
(1 row)
```

dblink_get_connections

dblink_get_connections — 接続中の名前付きdblink接続すべての名前を返します

概要

```
dblink_get_connections() returns text[]
```

説明

dblink_get_connectionsは、接続中の名前付きのdblink接続すべての名前を配列として返します。

戻り値

接続名のテキスト型配列を返します。なければNULLです。

例

```
SELECT dblink_get_connections();
```

dblink_error_message

dblink_error_message — 名前付き接続上の最後のエラーメッセージを入手します

概要

```
dblink_error_message(text connname) returns text
```

説明

dblink_error_messageは指定された接続における、最後のリモートエラーメッセージを取り出します。

引数

connname

使用する接続名です。

戻り値

最後のエラーメッセージを返します。接続においてエラーが存在しなかった場合はOKを返します。

注釈

非同期問い合わせがdblink_send_queryで開始された場合、接続に伴うエラーメッセージは、
サーバの応答メッセージが消費されるまで更新されないかもしれません。典型的にはこのことは、
dblink_error_messageに先立ってdblink_is_busyあるいはdblink_get_resultを呼び出すべきであること
を意味します。そうすることによって非同期問い合わせによって生成されたエラーが見えるようになります。

例

```
SELECT dblink_error_message('dtest1');
```

dblink_send_query

dblink_send_query — リモートデータベースに非同期問い合わせを送信します

概要

```
dblink_send_query(text connname, text sql) returns int
```

説明

dblink_send_queryは非同期に、つまり、結果をすぐに待機することなく実行する問い合わせを送信します。接続上で進行中の非同期問い合わせが存在してはなりません。

非同期問い合わせの登録が成功した後、dblink_is_busyを使用して完了状況を検査することができます。そして最後に、dblink_get_resultを使用して結果を収集します。また、dblink_cancel_queryを使用して実行中の非同期問い合わせを取り消すことができます。

引数

connname

使用する接続名です。

sql

例えばselect * from pg_classといった、リモートデータベースで実行させたいSQL文です。

戻り値

問い合わせの登録に成功した場合1を返します。失敗した場合は0を返します。

例

```
SELECT dblink_send_query('dtest1', 'SELECT * FROM foo WHERE f1 < 3');
```

dblink_is_busy

dblink_is_busy — 接続において非同期問い合わせが実行中か検査します

概要

```
dblink_is_busy(text connname) returns int
```

説明

dblink_is_busyは非同期問い合わせが進行中かどうか試験します。

引数

connname

検査対象の接続名です。

戻り値

接続において進行中だった場合は1、さもなくば0を返します。この関数が0を返した場合、
dblink_get_resultがブロックされないことが保証されます。

例

```
SELECT dblink_is_busy('dtest1');
```

dblink_get_notify

dblink_get_notify — 接続上の非同期通知を取り出します

概要

```
dblink_get_notify() returns setof (notify_name text, be_pid int, extra text)
dblink_get_notify(text connname) returns setof (notify_name text, be_pid int, extra text)
```

説明

dblink_get_notifyは名前の付いていない接続、または、もし指定されて名前が付いている接続いずれからも通知を取り出します。dblink経由で通知を受け取るには、dblink_execを使用してLISTENを最初に発行しなければなりません。詳細はLISTENとNOTIFYを参照ください。

引数

connname

通知を受け取る名前つきの接続の名前

戻り値

setof (notify_name text, be_pid int, extra text)または存在しない場合は空集合を返します。

例

```
SELECT dblink_exec('LISTEN virtual');
 dblink_exec
-------------
 LISTEN
(1 row)

SELECT * FROM dblink_get_notify();
 notify_name | be_pid | extra
-------------+--------+-------
(0 rows)

NOTIFY virtual;
NOTIFY

SELECT * FROM dblink_get_notify();
```

```
 notify_name | be_pid | extra
-------------+--------+-------
 virtual     |   1229 |
(1 row)
```

dblink_get_result

dblink_get_result — 非同期問い合わせの結果を入手します

概要

```
dblink_get_result(text connname [, bool fail_on_error]) returns setof record
```

説明

dblink_get_resultは、事前にdblink_send_queryで送信された非同期問い合わせの結果を収集します。問い合わせがまだ完了していなかった場合、dblink_get_resultは終わるまで待機します。

引数

connname

使用する接続名です。

fail_on_error

真（省略時のデフォルト）の場合、接続のリモート側で発生したエラーによりローカル側でもエラーが発生します。偽の場合リモート側のエラーはローカル側にはNOTICEとして報告され、この関数は行を返しません。

戻り値

非同期問い合わせ（行を返すSQL文の場合）について、この関数は問い合わせで生成された行を返します。この関数を使用するためには、上のdblinkで説明したように想定する列集合を指定する必要があります。

非同期コマンド（行を返さないSQL文の場合）について、この関数はコマンド状態文字列からなるテキスト列を1つ持つ1行を返します。この場合も呼び出し元のFROM句で結果が単一のテキスト列を持つことを指定する必要があります。

注釈

dblink_send_queryが1を返した場合にこの関数を呼び出さなければなりません。接続を再度利用できるようになる前に、送信した問い合わせに対し一度呼び出されなければなりません。もう一度実行すると空の結果集合を得ることになります。

dblink_send_queryとdblink_get_resultを使う場合には、dblinkはリモート側の問い合わせ結果をローカルの問い合わせ処理に渡す前にすべて取り込みます。問い合わせが大量の行を返す場合、ローカルセッションで一時的なメモリ膨張が起こるかも知れません。そのような問い合わせはdblink_openでカーソルとし

て開き、それから一度に管理可能な行数を取り出す方が良いでしょう。あるいは、普通のdblink()を使って下さい。大きな結果集合をディスクにスプールすることでメモリ膨張を回避します。

例

```
contrib_regression=# SELECT dblink_connect('dtest1', 'dbname=contrib_regression');
 dblink_connect
----------------
 OK
(1 row)

contrib_regression=# SELECT * FROM
contrib_regression-# dblink_send_query('dtest1', 'select * from foo where f1 < 3') AS t1;
 t1
----
  1
(1 row)

contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2 text, f3 text[]);
 f1 | f2 |     f3
----+----+------------
  0 | a  | {a0,b0,c0}
  1 | b  | {a1,b1,c1}
  2 | c  | {a2,b2,c2}
(3 rows)

contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2 text, f3 text[]);
 f1 | f2 | f3
----+----+----
(0 rows)

contrib_regression=# SELECT * FROM
contrib_regression-# dblink_send_query('dtest1', 'select * from foo where f1 < 3; select * from
 foo where f1 > 6') AS t1;
 t1
----
  1
(1 row)

contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2 text, f3 text[]);
 f1 | f2 |     f3
----+----+------------
  0 | a  | {a0,b0,c0}
  1 | b  | {a1,b1,c1}
  2 | c  | {a2,b2,c2}
```

```
(3 rows)

contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2 text, f3 text[]);
 f1 | f2 |      f3
----+----+---------------
  7 | h  | {a7,b7,c7}
  8 | i  | {a8,b8,c8}
  9 | j  | {a9,b9,c9}
 10 | k  | {a10,b10,c10}
(4 rows)

contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2 text, f3 text[]);
 f1 | f2 | f3
----+----+----
(0 rows)
```

dblink_cancel_query

dblink_cancel_query — 名前付き接続上の実行中の問い合わせをすべて取り消します

概要

```
dblink_cancel_query(text connname) returns text
```

説明

dblink_cancel_queryは名前付き接続上で進行中の問い合わせをすべて取り消そうとします。これは成功するとは限らないことに注意してください（例えばリモート問い合わせがすでに終わっているかもしれないからです）。取り消し要求は単に問い合わせがすぐに失敗する可能性を大きくするだけです。例えば、dblink_get_resultを呼び出すなど、通常の問い合わせ手順を完了させる必要があります。

引数

connname

使用する接続名です。

戻り値

取り消し要求が送信された場合OKを、さもなくば失敗についてのエラーメッセージテキストを返します。

例

```
SELECT dblink_cancel_query('dtest1');
```

dblink_get_pkey

dblink_get_pkey — リレーションの主キーフィールドの位置とフィールド名を返します

概要

```
dblink_get_pkey(text relname) returns setof dblink_pkey_results
```

説明

dblink_get_pkeyは、ローカルデータベース内のリレーションの主キーに関する情報を提供します。これはリモートデータベースに送信する問い合わせを生成する際に役に立つことがあります。

引数

relname

例えばfooやmyschema.mytabといった、ローカル側のリレーションの名前です。例えば"FooBar"のように名前に大文字小文字が混在する場合や特殊文字が含まれる場合は二重引用符で括ってください。引用符がないと文字列は小文字に変換されます。

戻り値

主キー毎に1行を返します。リレーションが主キーを持たない場合は行は返されません。結果の行型は以下のように定義されます。

```
CREATE TYPE dblink_pkey_results AS (position int, colname text);
```

position列は単に1からNを返します。それは、主キー内にあるフィールドの数で、テーブルの列内にある数ではありません。

例

```
CREATE TABLE foobar (
    f1 int,
    f2 int,
    f3 int,
    PRIMARY KEY (f1, f2, f3)
);
CREATE TABLE

SELECT * FROM dblink_get_pkey('foobar');
```

```
 position | colname
----------+---------
        1 | f1
        2 | f2
        3 | f3
(3 rows)
```

dblink_build_sql_insert

dblink_build_sql_insert — ローカル側のタプルを使用し、主キーフィールドの値を別の提供される値に置き換えてINSERT文を構築します

概要

```
dblink_build_sql_insert(text relname,
                        int2vector primary_key_attnums,
                        integer num_primary_key_atts,
                        text[] src_pk_att_vals_array,
                        text[] tgt_pk_att_vals_array) returns text
```

説明

dblink_build_sql_insertはローカル側のテーブルの一部を選択した複製をリモートデータベースに行う場合に有用になる可能性があります。これは主キーによりローカルテーブルから行を選択し、その主キー値を最後の引数で与えた値に置き換えて、行を複製するINSERT SQLコマンドを構築します。（行をそのまま複製する場合は、単に最後の2つの引数に同じ値を指定してください。）

引数

relname

例えばfooやmyschema.mytabといったローカル側のリレーションの名前です。例えば"FooBar"のように名前に大文字小文字が混在する場合や特殊文字が含まれる場合は二重引用符で括ってください。引用符がないと文字列は小文字に変換されます。

primary_key_attnums

例えば1 2といった、主キーフィールドの属性番号（1始まり）です。

num_primary_key_atts

主キーフィールドの個数です。

src_pk_att_vals_array

ローカルタプルを検索するために使用される主キーフィールドの値です。各フィールドはテキスト形式で表されます。これらの主キーの値を持つ行がローカル側に存在しない場合はエラーが発生します。

tgt_pk_att_vals_array

最終的なINSERTコマンドにおいて置き換えられる主キーフィールドの値です。各フィールドはテキスト形式で表されます。

戻り値

要求したSQL文をテキストとして返します。

注釈

PostgreSQL 9.0の段階で、primary_key_attnumsの中の属性数は、SELECT * FROM relname内の列の位置に対応する、論理的列数として翻訳されます。以前のバージョンは物理的な列の位置として数を翻訳しました。テーブルの存続期間中に、表示された列の左側のどんな列でも削除されると差異が生じます。

例

```
SELECT dblink_build_sql_insert('foo', '1 2', 2, '{"1", "a"}', '{"1", "b''a"}');
             dblink_build_sql_insert
-------------------------------------------------
 INSERT INTO foo(f1,f2,f3) VALUES('1','b''a','1')
(1 row)
```

dblink_build_sql_delete

dblink_build_sql_delete — 主キーフィールドの値として提供された値を使用したDELETE文を構築します

概要

```
dblink_build_sql_delete(text relname,
                        int2vector primary_key_attnums,
                        integer num_primary_key_atts,
                        text[] tgt_pk_att_vals_array) returns text
```

説明

dblink_build_sql_deleteはローカル側のテーブルの一部を選択した複製をリモートデータベースに行う場合に有用になる可能性があります。これは指定した主キーの値を持つ行を削除するDELETE SQLコマンドを構築します。

引数

relname

例えばfooやmyschema.mytabといったローカル側のリレーションの名前です。例えば"FooBar"のように名前に大文字小文字が混在する場合や特殊文字が含まれる場合は二重引用符で括ってください。引用符がないと文字列は小文字に変換されます。

primary_key_attnums

例えば1 2といった、主キーフィールドの属性番号（1始まり）です。

num_primary_key_atts

主キーフィールドの個数です。

tgt_pk_att_vals_array

最終的なDELETEコマンドにおいて使用される主キーフィールドの値です。各フィールドはテキスト形式で表されます。

戻り値

要求したSQL文をテキストとして返します。

注釈

PostgreSQL 9.0の段階で、primary_key_attnumsの中の属性数は、SELECT * FROM relname内の列の位置に対応する、論理的列数として翻訳されます。以前のバージョンは物理的な列の位置として数を翻訳しました。テーブルの存続期間中に、表示された列の左側のどんな列でも削除されると差異が生じます。

例

```
SELECT dblink_build_sql_delete('"MyFoo"', '1 2', 2, '{"1", "b"}');
           dblink_build_sql_delete
---------------------------------------------
 DELETE FROM "MyFoo" WHERE f1='1' AND f2='b'
(1 row)
```

dblink_build_sql_update

dblink_build_sql_update — 主キーフィールドの値として提供された値を使用したUPDATE文を構築します

概要

```
dblink_build_sql_update(text relname,
                        int2vector primary_key_attnums,
                        integer num_primary_key_atts,
                        text[] src_pk_att_vals_array,
                        text[] tgt_pk_att_vals_array) returns text
```

説明

dblink_build_sql_updateはローカル側のテーブルの一部を選択した複製をリモートデータベースに行う場合に有用になる可能性があります。これは主キーによりローカルテーブルから行を選択し、その主キー値を最後の引数で与えた値に置き換えて、行を複製するUPDATE SQLコマンドを構築します。(行をそのまま複製する場合は、単に最後の2つの引数に同じ値を指定してください。) このUPDATEコマンドは常に行のすべてのフィールドを代入します。この関数とdblink_build_sql_insertの主な違いは、対象の行がリモート側のテーブルにすでに存在すると仮定している点です。

引数

relname

例えばfooやmyschema.mytabといったローカル側のリレーションの名前です。例えば"FooBar"のように名前に大文字小文字が混在する場合や特殊文字が含まれる場合は二重引用符で括ってください。引用符がないと文字列は小文字に変換されます。

primary_key_attnums

例えば1 2といった、主キーフィールドの属性番号(1始まり)です。

num_primary_key_atts

主キーフィールドの個数です。

src_pk_att_vals_array

ローカルタプルを検索するために使用される主キーフィールドの値です。各フィールドはテキスト形式で表されます。これらの主キーの値を持つ行がローカル側に存在しない場合はエラーが発生します。

tgt_pk_att_vals_array

最終的なUPDATEコマンドにおいて置き換えられる主キーフィールドの値です。各フィールドはテキスト形式で表されます。

戻り値

要求したSQL文をテキストとして返します。

注釈

PostgreSQL 9.0の段階で、primary_key_attnumsの中の属性数は、SELECT * FROM relname内の列の位置に対応する、論理的列数として翻訳されます。以前のバージョンは物理的な列の位置として数を翻訳しました。テーブルの存続期間中に、表示された列の左側のどんな列でも削除されると差異が生じます。

例

```
SELECT dblink_build_sql_update('foo', '1 2', 2, '{"1", "a"}', '{"1", "b"}');
                   dblink_build_sql_update
-------------------------------------------------------------
 UPDATE foo SET f1='1',f2='b',f3='1' WHERE f1='1' AND f2='b'
(1 row)
```

F.11. dict_int

dict_intは、全文検索用の辞書テンプレートの追加例です。この辞書例の目的は、検索性能に大きく影響する一意な単語数の急激な増大を防ぎながら、こうした数のインデックス付けを行うことができるように、整数（符号付きおよび符号無）のインデックス付けを制御することです。

このモジュールは「trusted」と見なされます。つまり、現在のデータベースに対してCREATE権限を持つ非スーパーユーザがインストールできます。

F.11.1. 設定

この辞書は3つのオプションを受け付けます。

- maxlenパラメータは整数型の単語で許される最大桁数を指定します。デフォルト値は6です。
- rejectlongパラメータは、桁数を超える整数を切り詰めるか無視するかを指定します。rejectlongがfalse（デフォルト）ならば、辞書は整数の先頭のmaxlen桁を返します。rejectlongがtrueならば、辞書は桁数を超えた整数をストップワードとして扱います。このためインデックス付けされません。これはまた、こうした整数を検索することができないことを意味します。
- absvalパラメータは、先頭の「+」または「-」符号を整数型の単語から削除するかどうかを指定します。デフォルトはfalseです。trueの場合、maxlenが適用される前に符号は削除されます。

F.11.2. 使用方法

dict_int拡張機能をインストールすると、intdict_templateテキスト検索テンプレートとこれに基づき、そのデフォルト値でintdict辞書が作成されます。以下のようにパラメータを変更することができます。

```
mydb# ALTER TEXT SEARCH DICTIONARY intdict (MAXLEN = 4, REJECTLONG = true);
ALTER TEXT SEARCH DICTIONARY
```

または、このテンプレートを基に新しい辞書を作成してください。

辞書を試験するためには以下を試してください。

```
mydb# select ts_lexize('intdict', '12345678');
 ts_lexize
-----------
 {123456}
```

しかし、現実世界で使用する場合は、第12章で説明されるテキスト検索設定内にこれを含むようになるでしょう。以下のようになります。

```
ALTER TEXT SEARCH CONFIGURATION english
    ALTER MAPPING FOR int, uint WITH intdict;
```

F.12. dict_xsyn

dict_xsyn（拡張類義語辞書）は全文検索用の辞書テンプレートの追加例です。この種類の辞書は、単語を類義語の集まりに置き換え、その類義語のいずれかを使用して単語を検索できるようにします。

F.12.1. 設定

dict_xsyn辞書は以下のオプションを受け付けます。

- matchorigは辞書で元の単語が受け付けられるか否かを制御します。デフォルトはtrueです。
- matchsynonymsは類義語が辞書で受け付けられるか否かを制御します。デフォルトはfalseです。
- keeporigは元の単語が辞書出力に含められるか否かを制御します。デフォルトはtrueです。
- keepsynonymsは類義語が辞書出力に含められるか否かを制御します。デフォルトはtrueです。
- rulesは、類義語リストを含むファイルのベース名です。このファイルは$SHAREDIR/tsearch_data/（$SHAREDIRはPostgreSQLインストレーションの共有データ用ディレクトリを示します）に格納しなければなりません。この名前は.rulesで終わらなければなりません（これはrulesパラメータには含まれません）。

rulesファイルは以下の書式です。

- 各行は、行の先頭で与えられる1つの単語に対する類義語の集まりを表します。類義語は以下のように空白文字で区切られます。

  ```
  word syn1 syn2 syn3
  ```

- シャープ記号(#)はコメント区切り記号です。行の任意の位置に記載することができます。行の残りの部分は飛ばされます。

例として$SHAREDIR/tsearch_data/にインストールされるxsyn_sample.rulesを参照してください。

F.12.2. 使用方法

dict_xsyn拡張機能をインストールすると、xsyn_templateテキスト検索テンプレートが作成され、それに基づき、デフォルトのパラメータを持ったxsyn辞書が作成されます。例えば以下のように、パラメータを変更することができます。

```
mydb# ALTER TEXT SEARCH DICTIONARY xsyn (RULES='my_rules', KEEPORIG=false);
ALTER TEXT SEARCH DICTIONARY
```

またこのテンプレートに基づいた新しい辞書を作成することもできます。

辞書を試験するためには以下を試してください。

```
mydb=# SELECT ts_lexize('xsyn', 'word');
      ts_lexize
-----------------------
 {syn1,syn2,syn3}

mydb# ALTER TEXT SEARCH DICTIONARY xsyn (RULES='my_rules', KEEPORIG=true);
ALTER TEXT SEARCH DICTIONARY

mydb=# SELECT ts_lexize('xsyn', 'word');
      ts_lexize
-----------------------
 {word,syn1,syn2,syn3}

mydb# ALTER TEXT SEARCH DICTIONARY xsyn (RULES='my_rules', KEEPORIG=false, MATCHSYNONYMS=true);
ALTER TEXT SEARCH DICTIONARY

mydb=# SELECT ts_lexize('xsyn', 'syn1');
      ts_lexize
-----------------------
 {syn1,syn2,syn3}

mydb# ALTER TEXT SEARCH DICTIONARY xsyn (RULES='my_rules', KEEPORIG=true, MATCHORIG=false,
 KEEPSYNONYMS=false);
ALTER TEXT SEARCH DICTIONARY

mydb=# SELECT ts_lexize('xsyn', 'syn1');
      ts_lexize
-----------------------
```

```
 {word}
```

現実世界で使用する場合は、第12章で説明されるテキスト検索設定内にこれを含むようになるでしょう。以下のようになります。

```
ALTER TEXT SEARCH CONFIGURATION english
    ALTER MAPPING FOR word, asciiword WITH xsyn, english_stem;
```

F.13. earthdistance

earthdistanceは地表面上の大圏距離を計算する、2つの異なる方式を提供します。最初に説明する方式はcubeモジュールに依存します。2番目の方式は、座標系として緯度経度を使用した、組み込みのpointデータ型を基にしたものです。

このモジュールでは地球は完全な球体であると仮定します。(この精度が不十分な場合は、PostGIS[1]プロジェクトを参照することを勧めます。)

cubeモジュールはearthdistanceをインストールする前にインストールしなければなりません(一つのコマンドで両方をインストールするためにCREATE EXTENSIONのCASCADEオプションを使うこともできますが)。

> **注意**
>
> earthdistanceとcubeは同じスキーマにインストールし、そのスキーマは信頼できないユーザにCREATE権限を許可していないし、今後も許可することのないものとすることを強く勧めます。さもないと、earthdistanceのスキーマが悪意のあるユーザにより定義されたオブジェクトを含んでいた場合に、インストール時のセキュリティ問題になります。さらに、インストール後にearthdistanceの関数を使う時には、サーチパス全体には信頼するスキーマだけが含まれるようにすべきです。

F.13.1. cubeを基にした地表距離

地球中心からのx、y、z距離をあらわす3次元を使用した点(両隅が同じ)であるcubeとして、データは格納されます。cube上にearthドメインが提供されます。これには、値がこれら制限を満たすか、また値が理論的に実際の地表面に近いかどうかの整合性検査を含みます。

地球の半径はearth()関数から入手されます。この単位はメートルです。しかしこの1つの関数を変更することで、何らかの他の単位を使用するようにしたり、より適切と考える別の半径を使用したりするようにこのモジュールを変更することができます。

このパッケージは天文学データベースへの応用もあります。天文学者はおそらく距離が度単位になるように、earth()が180/pi()の半径を返すものと変更したいでしょう。

緯度経度(度単位)の入力をサポート、緯度経度の出力をサポート、2点間の大圏距離を計算、インデックス検索に使用可能な簡単に外接矩形を指定するための関数が提供されます。

[1] https://postgis.net/

提供されている関数は表 F.5に示されています。

表F.5 cubeを基にしたearthdistanceの関数

関数 説明
earth () → float8 地球の想定半径を返します。
sec_to_gc (float8) → float8 地表の2点間の通常の直線（割線）距離を大圏距離に変換します。
gc_to_sec (float8) → float8 地表の2点間の大圏距離を通常の直線（割線）距離に変換します。
ll_to_earth (float8, float8) → earth 度単位で指定された緯度（第1引数）と経度（第2引数）に対する地表位置を返します。
latitude (earth) → float8 地表上の点の緯度を度単位で返します。
longitude (earth) → float8 地表上の点の経度を度単位で返します。
earth_distance (earth, earth) → float8 地表上の2点間の大圏距離を返します。
earth_box (earth, float8) → cube 位置から指定した大圏距離内の点に対するcubeの@>演算子を使用するインデックス検索に適した矩形を返します。矩形内の点の一部は指定した大圏距離の外部にあります。このため、earth_distanceを使用した第2の検査を問い合わせに含めなければなりません。

F.13.2. pointを基にした地表距離

このモジュールの第2部分はpoint型の値として地球上の位置を表現することに依存します。ここで第1要素は経度を度単位で、第2要素は緯度を度単位で表現していると見なします。直感的に経度はX軸、緯度はY軸という考えがより合うため、点は(経度, 緯度)として見なされますが、逆には見なされません。

表 F.6に示されている1つの演算子が提供されます。

表F.6 pointを基にしたearthdistanceの演算子

演算子 説明
point <@> point → float8 法定マイル単位の地表の2点間の距離を計算します。

このモジュールのcubeを基にした場合と異なり、ここでの単位はコード内に固定で記載されることに注意してください。earth()関数を変更しても、この演算子の結果には影響しません。

緯度経度表現の1つの欠点は、極近辺と経度±180度近辺の限界条件に注意する必要があることです。cubeを基にした表現ではこうした不連続性を防止できます。

F.14. file_fdw

file_fdwモジュールは、サーバのファイルシステムにあるデータファイルにアクセスするのに使用できる外部データラッパfile_fdwを提供します。サーバのファイルにアクセスしたり、サーバ上のプログラムを実行して出力を読み取ったりできます。データファイルはCOPY FROMで読むことのできるフォーマットでなければなりません。詳細は COPY を参照してください。データファイルへのアクセスは現時点では読み取り専用です。

このラッパーで作成された外部テーブルには以下のオプションを設定することができます。

filename

読み取るファイルを指定します。相対パスはデータディレクトリからの相対パスです。filenameかprogramのどちらかを指定できますが、両方は指定できません。

program

実行するコマンドを指定します。このコマンドの標準出力をCOPY FROM PROGRAMが使用されたかのように読み込みます。programかfilenameのどちらかを指定できますが、両方は指定できません。

format

データフォーマットを指定するもので、COPYのFORMATオプションと同じです。

header

データがヘッダ行を持つか指定するもので、COPYのHEADERオプションと同じです。

delimiter

データの区切り文字を指定するもので、COPYのDELIMITERオプションと同じです。

quote

データの引用符文字を指定するもので、COPYのQUOTEオプションと同じです。

escape

データのエスケープ文字を指定するもので、COPYのESCAPEオプションと同じです。

null

データのNULL文字列を指定するもので、COPYのNULLオプションと同じです。

encoding

データのエンコーディングを指定するもので、COPYのENCODINGオプションと同じです。

COPYではHEADERといったオプションを対応する値なしで指定できるのに対して、外部データラッパの構文では全ての場合において値を指定する必要がある点に注意してください。通常の値なしで指定されるCOPYオプションを有効にするには、そのようなオプションはすべてbooleanであるため、代わりにTRUEを渡すことができます。

このラッパーを使って作られた外部テーブルのカラムは、以下のオプションを持つことができます。

force_not_null

これはbooleanオプションです。真の場合は、このカラムの値はNULL文字列(これはテーブルレベルのnullオプションです)と比較されません。これは、COPYのFORCE_NOT_NULLオプションに列名を指定するのと同じ効果があります。

force_null

これはbooleanオプションです。真の場合、NULL文字列と一致するこのカラムの値は、たとえ引用符で括られていたとしてもNULLと返されます。このオプションがなければ、NULL文字列と一致する引用符で括られていない値のみがNULLと返されます。これは、COPYのFORCE_NULLオプションに列名を指定するのと同じ効果があります。

COPYのFORCE_QUOTEオプションはfile_fdwでは現在サポートされていません。

これらのオプションは外部テーブルまたはその列にのみ指定可能で、file_fdw外部データラッパやそれを使用するサーバ、ユーザマッピングのオプションには指定できません。

どのファイルが読み込まれ、どのプログラムが実行されるかをコントロールできるのは一定のユーザのみであるべきというセキュリティ上の理由から、テーブルレベルのオプションを変更するにはスーパーユーザであるか、デフォルトロールのpg_read_server_files(ファイル名を使う)やpg_execute_server_program(プログラムを使う)の権限を持っていることが必要です。原則としては非スーパーユーザはその他のオプションを変更することを許されてもよいのですが、現時点ではサポートされていません。

programオプションが指定されたとき、オプションの文字列がシェルによって実行されることに注意してください。信頼できないソースをコマンド引数に渡す場合、シェルにとって特別な意味を持つ可能性のある文字を取り除くかエスケープするように注意する必要があります。セキュリティ上の理由から、固定のコマンド文字列を使用するか、少なくともユーザー入力を渡さないようにすることをお勧めします。

file_fdwを使用する外部テーブルでは、EXPLAINは読み込むファイルの名前又は実行しているプログラムを表示します。COSTS OFFが指定されない場合は(バイト単位の)ファイルサイズも表示されます。

例F.1 PostgreSQL CSV ログ用の外部テーブル作成

file_fdwの明確な用途の一つはPostgreSQLの活動ログをテーブルとして検索できるようにすることです。これを実現するには、ここではpglog.csvと呼ぶCSVファイルにログを記録している必要があります。まず、file_fdwを拡張機能としてインストールします。

```
CREATE EXTENSION file_fdw;
```

続いて外部サーバを作成します。

```
CREATE SERVER pglog FOREIGN DATA WRAPPER file_fdw;
```

これで外部テーブルを作成する準備ができました。CREATE FOREIGN TABLEコマンドを使って、テーブルのカラム、CSVファイル名とそのフォーマットを定義する必要があるでしょう。

```
CREATE FOREIGN TABLE pglog (
  log_time timestamp(3) with time zone,
  user_name text,
```

```
  database_name text,
  process_id integer,
  connection_from text,
  session_id text,
  session_line_num bigint,
  command_tag text,
  session_start_time timestamp with time zone,
  virtual_transaction_id text,
  transaction_id bigint,
  error_severity text,
  sql_state_code text,
  message text,
  detail text,
  hint text,
  internal_query text,
  internal_query_pos integer,
  context text,
  query text,
  query_pos integer,
  location text,
  application_name text,
  backend_type text
) SERVER pglog
OPTIONS ( filename 'log/pglog.csv', format 'csv' );
```

これで全てです。もうあなたはログに直接検索を実行することができます。実運用においては、もちろんログローテーションを処理する方法を定義する必要があるでしょう。

F.15. fuzzystrmatch

fuzzystrmatchモジュールは、文字列間の類似度や相違度を決める複数の関数を提供します。

> **注意**
>
> 現時点で、soundex、metaphone、dmetaphone、dmetaphone_altは（UTF-8のような）マルチバイト符号化方式では充分に動作しません。

このモジュールは「trusted」と見なされます。つまり、現在のデータベースに対してCREATE権限を持つ非スーパーユーザがインストールできます。

F.15.1. Soundex

Soundexシステムは、同一コードに変換することで似ているように見える名称を一致させる手法です。これは、1880年、1900年、1910年の米国国勢調査で初めて使用されました。Soundexは非英語圏の名称では特に有用なものではないことに注意してください。

fuzzystrmatchはSoundexコードを使用して動作する2つの関数を提供します。

```
soundex(text) returns text
difference(text, text) returns int
```

soundex関数は文字列をSoundexコードに変換します。difference関数は2つの文字列をそのSoundexコードに変換し、コード位置が一致する個数を報告します。Soundexコードは4文字からなりますので、結果は0から4までの範囲になります。0はまったく一致しないことを、4は完全に一致することを示します。(したがってこの関数の名前は間違っています。similarityの方がより優れた名前だったかもしれません。)

以下に使用例をいくつか示します。

```
SELECT soundex('hello world!');

SELECT soundex('Anne'), soundex('Ann'), difference('Anne', 'Ann');
SELECT soundex('Anne'), soundex('Andrew'), difference('Anne', 'Andrew');
SELECT soundex('Anne'), soundex('Margaret'), difference('Anne', 'Margaret');

CREATE TABLE s (nm text);

INSERT INTO s VALUES ('john');
INSERT INTO s VALUES ('joan');
INSERT INTO s VALUES ('wobbly');
INSERT INTO s VALUES ('jack');

SELECT * FROM s WHERE soundex(nm) = soundex('john');

SELECT * FROM s WHERE difference(s.nm, 'john') > 2;
```

F.15.2. レーベンシュタイン

この関数は2つの文字列間のレーベンシュタイン距離を計算します。

```
levenshtein(text source, text target, int ins_cost, int del_cost, int sub_cost) returns int
levenshtein(text source, text target) returns int
levenshtein_less_equal(text source, text target, int ins_cost, int del_cost, int sub_cost, int
 max_d) returns int
levenshtein_less_equal(text source, text target, int max_d) returns int
```

sourceおよびtargetは255文字までの任意の非NULL文字列を取ることができます。コストパラメータはそれぞれ、文字の挿入、削除、置換に負わせる文字数を指定します。この関数の2番目のバージョンのようにコストパラメータを省略することができます。この場合デフォルトですべて1になります。

levenshtein_less_equalは小さな距離だけを問題にする場合についてのlevenshtein関数の高速化版です。実際の距離がmax_d以下の場合、levenshtein_less_equalは正しい値を返しますが、そうでなければ、max_dより大きい何らかの値を返します。max_dが負の場合は、levenshteinと同じ動作になります。

以下に例を示します。

```
test=# SELECT levenshtein('GUMBO', 'GAMBOL');
 levenshtein
-------------
           2
(1 row)

test=# SELECT levenshtein('GUMBO', 'GAMBOL', 2, 1, 1);
 levenshtein
-------------
           3
(1 row)

test=# SELECT levenshtein_less_equal('extensive', 'exhaustive', 2);
 levenshtein_less_equal
------------------------
                      3
(1 row)

test=# SELECT levenshtein_less_equal('extensive', 'exhaustive', 4);
 levenshtein_less_equal
------------------------
                      4
(1 row)
```

F.15.3. Metaphone

Metaphoneは、Soundex同様、入力文字に対する対応するコードを構築するという考えに基づいたものです。2つの文字列が同一コードを持つ場合、類似とみなされます。

以下の関数は入力文字列に対するmetaphoneコードを計算します。

```
metaphone(text source, int max_output_length) returns text
```

sourceは255文字までの非NULL文字列を取ることができます。max_output_lengthは出力metaphoneコードの最大長を設定します。出力は長すぎるとこの長さに切り詰められます。

以下に例を示します。

```
test=# SELECT metaphone('GUMBO', 4);
 metaphone
-----------
 KM
```

```
(1 row)
```

F.15.4. Double Metaphone

Double Metaphoneシステムは与えられた入力文字列に対する、「primary」と「alternate」という2つの「似たように見える」文字列を計算します。ほとんどの場合、これらは同じですが、英語以外の名称では特に発音に応じて多少異なる場合があります。以下の関数はprimaryコードとalternateコードを計算します。

```
dmetaphone(text source) returns text
dmetaphone_alt(text source) returns text
```

入力文字列長に関する制限はありません。

以下に例を示します。

```
test=# SELECT dmetaphone('gumbo');
 dmetaphone
------------
 KMP
(1 row)
```

F.16. hstore

本モジュールはキー、値の組み合わせの集合を単一のPostgreSQL値に格納するためのhstoreデータ型を実装します。あまり厳密に検査されない属性を多く持つ行や半構造化データなど、多くの状況で有用になる可能性があります。キーと値は単純なテキスト文字列です。

このモジュールは「trusted」と見なされます。つまり、現在のデータベースに対してCREATE権限を持つ非スーパーユーザがインストールできます。

F.16.1. hstoreの外部表現

入力および出力で使用されるhstore値のテキスト表現はカンマで区切られた、ゼロ以上のkey => valueという組み合わせを含みます。以下に例を示します。

```
k => v
foo => bar, baz => whatever
"1-a" => "anything at all"
```

組み合わせの順序は重要ではありません（出力時に再現されないこともあります）。組み合わせ間や=>記号の前後の空白文字は無視されます。キーや値が空白文字、カンマ、=、>を含む場合は二重引用符でくくります。キーや値に二重引用符やバックスラッシュを含めるには、バックスラッシュでエスケープしてください。

hstore内の各キーは一意です。重複するキーを持つhstoreを宣言すると、hstoreには1つしか保存されません。またどちらが残るかは保証されません。

```
SELECT 'a=>1,a=>2'::hstore;
  hstore
----------
 "a"=>"1"
```

値はSQLのNULLを取ることができます(キーは不可)。以下に例を示します。

```
key => NULL
```

NULLキーワードは大文字小文字の区別をしません。nullを普通の文字列「NULL」として扱うためには二重引用符でくくってください。

> **注記**
>
> 入力として使用される場合hstoreテキスト書式は、前もって必要な引用符付けやエスケープ処理を適用することに注意してください。パラメータとしてhstoreリテラルを渡す場合、追加処理は必要ありません。しかし、引用符付けしたリテラル定数として渡す場合には、単一引用符および(standard_conforming_strings設定パラメータに依存しますが)バックスラッシュ文字をすべて正しくエスケープしなければなりません。文字列定数の取り扱いについては4.1.2.1を参照してください。

出力の場合、厳密に必要がない場合であっても、常にキーと値は二重引用符でくくられます。

F.16.2. hstoreの演算子と関数

hstoreモジュールで提供される演算子を表 F.7に、関数を表 F.8に示します。

表F.7 hstoreの演算子

演算子 説明 例
hstore -> text → text 与えられたキーに対応する値を、存在しなければNULLを返します。 'a=>x, b=>y'::hstore -> 'a' → x
hstore -> text[] → text[] 与えられたキーに対応する値を、存在しなければNULLを返します。 'a=>x, b=>y, c=>z'::hstore -> ARRAY['c','a'] → {"z","x"}
hstore || hstore → hstore 2つのhstoreを連結します。 'a=>b, c=>d'::hstore || 'c=>x, d=>q'::hstore → "a"=>"b", "c"=>"x", "d"=>"q"

演算子 説明 例
hstore ? text → boolean hstoreがキーを含むか。 'a=>1'::hstore ? 'a' → t
hstore ?& text[] → boolean hstoreが指定したキーをすべて含むか。 'a=>1,b=>2'::hstore ?& ARRAY['a','b'] → t
hstore ?| text[] → boolean hstoreが指定したキーのいずれかを含むか。 'a=>1,b=>2'::hstore ?| ARRAY['b','c'] → t
hstore @> hstore → boolean 左辺は右辺を含むか。 'a=>b, b=>1, c=>NULL'::hstore @> 'b=>1' → t
hstore <@ hstore → boolean 左辺は右辺に含まれるか。 'a=>c'::hstore <@ 'a=>b, b=>1, c=>NULL' → f
hstore - text → hstore 左辺からキーを削除します。 'a=>1, b=>2, c=>3'::hstore - 'b'::text → "a"=>"1", "c"=>"3"
hstore - text[] → hstore 左辺からキー(複数)を削除します。 'a=>1, b=>2, c=>3'::hstore - ARRAY['a','b'] → "c"=>"3"
hstore - hstore → hstore 右辺の組み合わせに一致する組み合わせを左辺から削除します。 'a=>1, b=>2, c=>3'::hstore - 'a=>4, b=>2'::hstore → "a"=>"1", "c"=>"3"
anyelement #= hstore → anyelement 左辺(複合型でなければなりません)のフィールドをhstoreの対応する値で置換します。 ROW(1,3) #= 'f1=>11'::hstore → (11,3)
%% hstore → text[] hstoreをキーと値が交互に並んだ配列に変換します。 %% 'a=>foo, b=>bar'::hstore → {a,foo,b,bar}
%# hstore → text[] hstoreをキーと値の2次元配列に変換します。 %# 'a=>foo, b=>bar'::hstore → {{a,foo},{b,bar}}

注記

PostgreSQL 8.2より前では、包含演算子@>と<@はそれぞれ@と~と呼ばれていました。これらの名前はまだ利用できますが、廃止予定であり、最終的にはなくなります。古い名前がコアの幾何データ型が従う規約と反対であることに注意してください。

表F.8 hstoreの関数

関数 説明 例
hstore (record) → hstore レコードまたは行からhstoreを生成します。 hstore(ROW(1,2)) → "f1"=>"1", "f2"=>"2"
hstore (text[]) → hstore 配列からhstoreを生成します。配列はキー、値の配列でも2次元の配列でも構いません。 hstore(ARRAY['a','1','b','2']) → "a"=>"1", "b"=>"2" hstore(ARRAY[['c','3'],['d','4']]) → "c"=>"3", "d"=>"4"
hstore (text[], text[]) → hstore キー、値で分けた配列からhstoreを作成します。 hstore(ARRAY['a','b'], ARRAY['1','2']) → "a"=>"1", "b"=>"2"
hstore (text, text) → hstore hstore型の単一項目を作成します。 hstore('a', 'b') → "a"=>"b"
akeys (hstore) → text[] hstoreのキーを配列として取り出します。 akeys('a=>1,b=>2') → {a,b}
skeys (hstore) → setof text hstoreのキーを集合として取り出します。 skeys('a=>1,b=>2') → a b
avals (hstore) → text[] hstoreの値を配列として取り出します。 avals('a=>1,b=>2') → {1,2}
svals (hstore) → setof text hstoreの値を集合として取り出します。 svals('a=>1,b=>2') → 1 2
hstore_to_array (hstore) → text[] hstoreのキーと値を、キーと値が交互に並んだ配列として取り出します。 hstore_to_array('a=>1,b=>2') → {a,1,b,2}
hstore_to_matrix (hstore) → text[] hstoreのキーと値を、2次元の配列として取り出します。 hstore_to_matrix('a=>1,b=>2') → {{a,1},{b,2}}
hstore_to_json (hstore) → json

関数

説明

例

非nullの値をすべてJSON文字列に変換しながら、hstoreをjson値に変換します。

この関数はhstore値がjsonにキャストされるときに暗黙的に使用されます。

hstore_to_json('"a key"=>1, b=>t, c=>null, d=>12345, e=>012345, f=>1.234, g=>2.345e+4') → {"a key": "1", "b": "t", "c": null, "d": "12345", "e": "012345", "f": "1.234", "g": "2.345e+4"}

hstore_to_jsonb (hstore) → jsonb

非nullの値をすべてJSON文字列に変換しながら、hstoreをjsonb値に変換します。

この関数はhstore値がjsonbにキャストされるときに暗黙的に使用されます。

hstore_to_jsonb('"a key"=>1, b=>t, c=>null, d=>12345, e=>012345, f=>1.234, g=>2.345e+4') → {"a key": "1", "b": "t", "c": null, "d": "12345", "e": "012345", "f": "1.234", "g": "2.345e+4"}

hstore_to_json_loose (hstore) → json

hstoreをjson値に変換します。ですが、数値およびブール値を識別しようとするため、その2つはJSON中では引用符が付きません。

hstore_to_json_loose('"a key"=>1, b=>t, c=>null, d=>12345, e=>012345, f=>1.234, g=>2.345e+4') → {"a key": 1, "b": true, "c": null, "d": 12345, "e": "012345", "f": 1.234, "g": 2.345e+4}

hstore_to_jsonb_loose (hstore) → jsonb

hstoreをjsonb値に変換します。ですが、数値およびブール値を識別しようとするため、その2つはJSON中では引用符が付きません。

hstore_to_jsonb_loose('"a key"=>1, b=>t, c=>null, d=>12345, e=>012345, f=>1.234, g=>2.345e+4') → {"a key": 1, "b": true, "c": null, "d": 12345, "e": "012345", "f": 1.234, "g": 2.345e+4}

slice (hstore, text[]) → hstore

指定されたキーだけを含むhstoreの部分集合を取り出します。

slice('a=>1,b=>2,c=>3'::hstore, ARRAY['b','c','x']) → "b"=>"2", "c"=>"3"

each (hstore) → setof record (key text, value text)

hstoreのキーと値をレコードの集合として取り出します。

select * from each('a=>1,b=>2') →

```
key | value
-----+-------
 a   | 1
 b   | 2
```

exist (hstore, text) → boolean

hstoreがキーを含むか。

exist('a=>1', 'a') → t

defined (hstore, text) → boolean

hstoreがキーに対して非NULLの値を含むか。

defined('a=>NULL', 'a') → f

delete (hstore, text) → hstore

キーに一致する組み合わせを削除します。

delete('a=>1,b=>2', 'b') → "a"=>"1"

delete (hstore, text[]) → hstore

関数 説明 例
キー(複数)に一致する組み合わせを削除します。 delete('a=>1,b=>2,c=>3', ARRAY['a','b']) → "c"=>"3"
delete (hstore, hstore) → hstore 第2引数内の組み合わせと一致する組み合わせを削除します。 delete('a=>1,b=>2', 'a=>4,b=>2'::hstore) → "a"=>"1"
populate_record (anyelement, hstore) → anyelement 左辺(複合型でなければなりません)のフィールドをhstoreの対応する値で置換します。 populate_record(ROW(1,2), 'f1=>42'::hstore) → (42,2)

F.16.3. インデックス

hstoreは@>、?、?&および?|演算子向けのGiSTおよびGINインデックスをサポートします。以下に例を示します。

```
CREATE INDEX hidx ON testhstore USING GIST (h);

CREATE INDEX hidx ON testhstore USING GIN (h);
```

gist_hstore_ops GiST演算子クラスはキー/値の集合をビットマップ署名として近似します。オプションの整数パラメータsiglenは、署名の長さをバイト単位で決定します。デフォルトの署名の長さは16バイトです。署名の長さの有効な値は1から2024バイトまでです。長い署名では、インデックスはより大きくなってしまいますが、(インデックスのより小さな部分とより少ないヒープページを走査することで)検索がより正確になります。

署名の長さが32バイトのインデックスを作成する例

```
CREATE INDEX hidx ON testhstore USING GIST (h gist_hstore_ops(siglen=32));
```

hstoreはまた、=演算子向けにbtreeまたはhashインデックスをサポートします。これによりhstoreの列をUNIQUEと宣言すること、また、GROUP BY、ORDER BY、DISTINCTの式で使用することができます。hstore値のソート順序付けはあまり有用ではありません。しかしこれらのインデックスは同値検索の際に有用になるかもしれません。=比較用のインデックスを以下のように作成します。

```
CREATE INDEX hidx ON testhstore USING BTREE (h);

CREATE INDEX hidx ON testhstore USING HASH (h);
```

F.16.4. 例

キーを追加、または、既存のキーを新しい値で更新します。

```
UPDATE tab SET h = h || hstore('c', '3');
```

キーを削除します。

```
UPDATE tab SET h = delete(h, 'k1');
```

recordをhstoreに変換します。

```
CREATE TABLE test (col1 integer, col2 text, col3 text);
INSERT INTO test VALUES (123, 'foo', 'bar');

SELECT hstore(t) FROM test AS t;
                   hstore
--------------------------------------------
 "col1"=>"123", "col2"=>"foo", "col3"=>"bar"
(1 row)
```

hstoreを事前に定義されたrecord型に変換します。

```
CREATE TABLE test (col1 integer, col2 text, col3 text);

SELECT * FROM populate_record(null::test,
                              '"col1"=>"456", "col2"=>"zzz"');
 col1 | col2 | col3
------+------+------
  456 | zzz  |
(1 row)
```

hstoreの値を使用して既存のレコードを変更します。

```
CREATE TABLE test (col1 integer, col2 text, col3 text);
INSERT INTO test VALUES (123, 'foo', 'bar');

SELECT (r).* FROM (SELECT t #= '"col3"=>"baz"' AS r FROM test t) s;
 col1 | col2 | col3
------+------+------
  123 | foo  | baz
(1 row)
```

F.16.5. 統計情報

内在する自由度のため、hstore型は異なるキーを多く含むことができます。有効なキーを検査することはアプリケーション側の作業です。以下の例では、キー検査および統計情報の入手に関する複数の技法を示します。

簡単な例を示します。

```
SELECT * FROM each('aaa=>bq, b=>NULL, ""=>1');
```

テーブルを使用する例です。

```
SELECT (each(h)).key, (each(h)).value INTO stat FROM testhstore;
```

オンライン統計値です。

```
SELECT key, count(*) FROM
  (SELECT (each(h)).key FROM testhstore) AS stat
  GROUP BY key
  ORDER BY count DESC, key;
    key    | count
-----------+-------
 line      |   883
 query     |   207
 pos       |   203
 node      |   202
 space     |   197
 status    |   195
 public    |   194
 title     |   190
 org       |   189
.................
```

F.16.6. 互換性

PostgreSQL 9.0からhstoreの内部表現はこれまでから変更されました。(ダンプ内で使用される)テキスト表現には変更がありませんので、ダンプ/リストアによる更新の妨げにはなりません。

バイナリによる更新の際、新しいコードで古い書式のデータを認識させることにより、上位互換が保持されます。これには、新しいコードによりまだ変更されていないデータを処理する際に、性能の劣化を多少伴います。以下のようにUPDATE文を実行することによりテーブル列内のすべての値を強制的に更新することができます。

```
UPDATE tablename SET hstorecol = hstorecol || '';
```

上を行う他の方法を以下に示します。

```
ALTER TABLE tablename ALTER hstorecol TYPE hstore USING hstorecol || '';
```

ALTER TABLEによる方法はテーブルに対して排他ロックを必要とします。しかし、古いバージョンの行でテーブルが膨張することはありません。

F.16.7. 変換

PL/Perl言語やPL/Python言語向けにhstore型の変換を実装した追加の拡張が入手可能です。PL/Perl向けの拡張は、信頼されたPL/Perlに対してはhstore_plperlという名前で、信頼されないものに対してはhstore_plperluという名前です。関数を作成するときにこの変換をインストールして指定していれば、hstoreの値はPerlのハッシュにマップされます。PL/Python向けの拡張はhstore_plpythonu、hstore_plpython2u、hstore_plpython3uという名前です(PL/Pythonの命名規約については45.1を参照してください)。この拡張を使うとhstoreの値はPythonの辞書型にマップされます。

> **注意**
>
> 変換の拡張はhstoreと同じスキーマにインストールすることを強く勧めます。さもないと、変換の拡張のスキーマが悪意のあるユーザにより定義されたオブジェクトを含んでいた場合に、インストール時のセキュリティ問題になります。

F.16.8. 作者

Oleg Bartunov <oleg@sai.msu.su>, Moscow, Moscow University, Russia

Teodor Sigaev <teodor@sigaev.ru>, Moscow, Delta-Soft Ltd., Russia

追加の改良はAndrew Gierth <andrew@tao11.riddles.org.uk>,United Kingdomによりなされました。

F.17. intagg

intaggモジュールは整数型の集約子と列挙子を提供します。その能力の上位集合を提供する組み込み関数が存在しますので、intaggは現在使われません。しかし、このモジュールは組み込み関数の互換ラッパとして今でもまだ提供されています。

F.17.1. 関数

集約子は、正確に提供する整数のみを含む整数型配列を生成するint_array_aggregate(integer)集約関数です。これは任意の配列型で同じことを行うarray_aggのラッパです。

列挙子は、setof integerを返すint_array_enum(integer[])関数です。これは基本的に上記集約子の反対の操作を行います。指定された整数型配列を行集合に拡張します。これは任意の配列型で同じことを行うunnestのラッパです。

F.17.2. 使用例

多くのデータベースシステムは1対多のテーブルを持ちます。こうしたテーブルは通常、以下のように2つのインデックス用のテーブルの間に存在します。

```
CREATE TABLE left (id INT PRIMARY KEY, ...);
CREATE TABLE right (id INT PRIMARY KEY, ...);
CREATE TABLE one_to_many(left INT REFERENCES left, right INT REFERENCES right);
```

通常以下のように使用されます。

```
SELECT right.* from right JOIN one_to_many ON (right.id = one_to_many.right)
  WHERE one_to_many.left = item;
```

これは、左辺のテーブル内にある項目に対応した、右辺のテーブル内のすべての項目を返します。これはSQLで非常によく使用される式です。

さて、この方法論はone_to_manyテーブル内に非常に多数の項目がある場合に扱いにくくなることがあり得ます。しばしばこうした結合は、インデックススキャンと特定された左辺の項目に対応した右辺のテーブル内の項目をそれぞれ取り出すことになります。非常に動的なシステムでは、できることは多くありません。しかし、ほぼ静的なデータが一部にある場合、集約子を使用して要約テーブルを作成することができます。

```
CREATE TABLE summary AS
  SELECT left, int_array_aggregate(right) AS right
  FROM one_to_many
  GROUP BY left;
```

これは左辺項目毎に1行を持ち、右辺の項目の配列をもつテーブルを作成します。さて、これは配列を使用する何らかの方法がないとかなり使い勝手が悪くなります。これが配列列挙子が存在する理由です。以下を行うことができます。

```
SELECT left, int_array_enum(right) FROM summary WHERE left = item;
```

上のint_array_enumを使用した問い合わせは、以下と同じ結果を生成します。

```
SELECT left, right FROM one_to_many WHERE left = item;
```

違いは、要約テーブルに対する問い合わせはテーブルから1行だけを取り出す必要があるのに対し、直接one_to_manyに問い合わせる場合はインデックススキャンと各項目に対し行を取り出さなければならないという点です。

あるシステムではEXPLAINを行うと8488というコストを持つ問い合わせが329というコストまで減少しました。元の問い合わせはone_to_manyテーブルを含む結合でしたが、以下のように置き換えられました。

```
SELECT right, count(right) FROM
  ( SELECT left, int_array_enum(right) AS right
    FROM summary JOIN (SELECT left FROM left_table WHERE left = item) AS lefts
         ON (summary.left = lefts.left)
  ) AS list
  GROUP BY right
  ORDER BY count DESC;
```

F.18. intarray

intarrayモジュールはNULLのない整数の配列の操作に便利な関数と演算子を多く提供します。また、一部の演算子を使用したインデックス検索をサポートします。

配列にNULL要素が一つでも含まれていれば、これらの操作はすべてエラーを発生します。

これらの操作の多くは一次元配列に対してのみ適当なものです。高次元の入力配列を受け付けますが、データは格納された順の一次元の配列であるかのように扱われます。

このモジュールは「trusted」と見なされます。つまり、現在のデータベースに対してCREATE権限を持つ非スーパーユーザがインストールできます。

F.18.1. intarrayの関数および演算子

intarrayモジュールで提供される関数を表 F.9に、演算子を表 F.10に示します。

表F.9 intarray関数

関数 説明 例
icount (integer[]) → integer 配列内の要素数を返します。 icount('{1,2,3}'::integer[]) → 3
sort (integer[], dir text) → integer[] 昇順または降順に配列をソートします。dirはascまたはdescのいずれかでなければなりません。 sort('{1,3,2}'::integer[], 'desc') → {3,2,1}
sort (integer[]) → integer[] sort_asc (integer[]) → integer[] 昇順にソートします。 sort(array[11,77,44]) → {11,44,77}
sort_desc (integer[]) → integer[] 降順にソートします。 sort_desc(array[11,77,44]) → {77,44,11}
uniq (integer[]) → integer[] 隣接する重複を削除します。 uniq(sort('{1,2,3,2,1}'::integer[])) → {1,2,3}
idx (integer[], item integer) → integer itemに最初に一致する要素番号を、一致するものがなければ0を返します。 idx(array[11,22,33,22,11], 22) → 2
subarray (integer[], start integer, len integer) → integer[] startの位置から始まりlen個の要素の部分配列を取り出します。

関数 説明 例
subarray('{1,2,3,2,1}'::integer[], 2, 3) → {2,3,2}
subarray (integer[], start integer) → integer[] startの位置から始まる部分配列を取り出します。 subarray('{1,2,3,2,1}'::integer[], 2) → {2,3,2,1}
intset (integer) → integer[] 単一要素の配列を作成します。 intset(42) → {42}

表F.10 intarray演算子

演算子 説明
integer[] && integer[] → boolean 配列が重なるか(少なくとも1つの共通要素があるか)。
integer[] @> integer[] → boolean 左辺の配列は右辺の配列を含むか。
integer[] <@ integer[] → boolean 左辺の配列は右辺の配列に含まれるか。
integer[] → integer 配列内の要素数を返します。
integer[] # integer → integer 右辺の引数に最初に一致する要素番号を、一致するものがなければ0を返します。(idx関数と同じです。)
integer[] + integer → integer[] 要素を配列の末尾に追加します。
integer[] + integer[] → integer[] 配列を連結します。
integer[] - integer → integer[] 配列から右辺の引数に一致する項目を削除します。
integer[] - integer[] → integer[] 左辺の配列から右辺の配列要素を削除します。
integer[] | integer → integer[] 引数の結合を計算します。
integer[] | integer[] → integer[] 引数の結合を計算します。
integer[] & integer[] → integer[] 引数の共通部分を計算します。
integer[] @@ query_int → boolean 配列が問い合わせを満たすか。(下記参照)
query_int ~~ integer[] → boolean

演算子 説明
配列が問い合わせを満たすか。(@@の交代演算子)

(PostgreSQL 8.2以前では、包含演算子@>と<@はそれぞれ@と~と呼ばれていました。これらの名前はまだ利用できますが、廃止予定であり、最終的にはなくなります。古い名前はコアの幾何データ型が以前従っていた規約とは反対であることに注意してください。)

演算子&&、@>と<@は、これらはNULLを含まない整数配列のみで動作し、組み込み演算子はどの配列型に対しても動作する点を除き、同じ名前のPostgreSQLの組み込み演算子とそれぞれほぼ等価です。この制限により、多くの場合、組み込み演算子より高速です。

@@および~~演算子は、配列が特化したデータ型query_intで表現される問い合わせを満たすかどうかを試験します。問い合わせは、おそらく&(論理積)、| (論理和)、! (否定)演算子を組み合わせて使用した、配列要素に対して検査される整数値からなります。例えば1&(2|3)という問い合わせは1および、2か3のいずれかを含む配列に一致します。

F.18.2. インデックスサポート

intarrayは&&、@>、<@、@@演算子に関して通常の配列等価性と同様にインデックスサポートを提供します。

2つのパラメータ化されたGiSTインデックス演算子クラスが提供されます。gist__int_ops(デフォルトで使用されます)は小中規模要素数のデータセットに適します。一方、gist__intbig_opsはより大きな署名を使用しますので、大規模データセット(つまり、異なった配列値を多数持つ列)のインデックスにより適しています。実装は組み込みの非可逆圧縮を持ったRD-treeデータ構造を使用します。

gist__int_opsは整数の集合を整数の範囲の配列として近似します。オプションの整数パラメータnumrangesは、一つのインデックスキー内の範囲の最大数を決定します。numrangesのデフォルト値は100です。有効な値は1から253までです。GiSTインデックスキーとしてより大きな値を使うと、インデックスはより大きくなってしまいますが、(インデックスのより小さな部分とより少ないヒープページを走査することで)検索がより正確になります。

gist__intbig_opsは整数の集合をビットマップ署名として近似します。オプションの整数パラメータsiglenは、署名の長さをバイト単位で決定します。デフォルトの署名の長さは16バイトです。署名の長さの有効な値は1から2024バイトまでです。長い署名では、インデックスはより大きくなってしまいますが、(インデックスのより小さな部分とより少ないヒープページを走査することで)検索がより正確になります。

また、同じ演算子をサポートするデフォルトではないGIN演算子クラスgin__int_opsも存在します。

GiSTおよびGINインデックスのどちらを選択するかは、別途説明されるGiSTとGINの相対的な性能特徴に依存します。

F.18.3. 例

```
-- メッセージ(message)は1つ以上の「節(section)」の中にある
```

```
CREATE TABLE message (mid INT PRIMARY KEY, sections INT[], ...);


-- 署名の長さが32バイトの特化したインデックスを作成
CREATE INDEX message_rdtree_idx ON message USING GIST (sections gist__int_ops(siglen=32));


-- 節1 OR 2のメッセージを選択 - OVERLAP演算子
SELECT message.mid FROM message WHERE message.sections && '{1,2}';


-- 節1 AND 2のメッセージを選択 - CONTAINS演算子
SELECT message.mid FROM message WHERE message.sections @> '{1,2}';


-- 同上、QUERY演算子を使用
SELECT message.mid FROM message WHERE message.sections @@ '1&2'::query_int;
```

F.18.4. ベンチマーク

ソースディレクトリ以下のcontrib/intarray/benchにはベンチマーク試験一式があり、インストールされたPostgreSQLサーバで実行できます。(DBD::Pgもインストールされていないといけません。) 以下のように実行します。

```
cd .../contrib/intarray/bench
createdb TEST
psql -c "CREATE EXTENSION intarray" TEST
./create_test.pl | psql TEST
./bench.pl
```

bench.plスクリプトには多くのオプションがあります。これらは引数を付けずに実行すると表示されます。

F.18.5. 作者

Teodor Sigaev (<teodor@sigaev.ru>)とOleg Bartunov (<oleg@sai.msu.su>)によりすべての作業がなされました。さらなる情報についてはhttp://www.sai.msu.su/~megera/postgres/gist/を参照してください。Andrey Oktyabrskiは新しい関数、演算子の追加において素晴らしい作業を行いました。

F.19. isn

isnモジュールは、EAN13、UPC、ISBN (書籍)、ISMN (音楽)、ISSN (連番)という国際的な標準製品番号に従うデータ型を提供します。番号は入力時にハードコードされた接頭辞の一覧に基づいて検証されます。こ

の接頭辞の一覧は出力時に数字にハイフンを付けるのにも使われます。新しい接頭辞が時々追加されますので、接頭辞の一覧は古くなっているかもしれません。このモジュールの将来のバージョンでは、必要なときにユーザが簡単に更新できる一つもしくは複数のテーブルから接頭辞の一覧を取得することが望まれます。しかし、現時点では、一覧はソースコードを修正し再コンパイルすることでしか更新できません。あるいは、接頭辞の検証とハイフン付けのサポートはこのモジュールの将来のバージョンからは外されるかもしれません。

このモジュールは「trusted」と見なされます。つまり、現在のデータベースに対してCREATE権限を持つ非スーパーユーザがインストールできます。

F.19.1. データ型

表 F.11にisnモジュールで提供されるデータ型を示します。

表F.11 isnデータ型

データ型	説明
EAN13	ヨーロッパ統一商品コード。常にEAN13用表示形式で表示されます。
ISBN13	国際標準図書番号。新しいEAN13用表示形式で表示されます。
ISMN13	国際標準楽譜番号。新しいEAN13用表示形式で表示されます。
ISSN13	国際標準逐次刊行物番号。新しいEAN13用表示形式で表示されます。
ISBN	国際標準図書番号。旧式の簡略表示形式で表示されます。
ISMN	国際標準楽譜番号。旧式の簡略表示形式で表示されます。
ISSN	国際標準逐次刊行物番号。旧式の簡略表示形式で表示されます。
UPC	統一商品コード。

数点注意事項があります。

1. ISBN13、ISMN13、ISSN13番号はすべてEAN13数値です。
2. EAN13番号は必ずISBN13、ISMN13、ISSN13のいずれかであるという訳ではありません (一部はそうです)。
3. 一部のISBN13番号はISBNとして表示可能です。
4. 一部のISMN13番号はISMNとして表示可能です。
5. 一部のISSN13番号はISSNとして表示可能です。
6. UPC番号はEAN13番号の部分集合です(基本的にはEAN13から先頭の0の数字を取り除いたものです)。
7. すべてのUPC、ISBN、ISMN、ISSNはEAN13番号として表現可能です。

すべての型は内部的には同一表現(64ビット整数)を使用し、すべて相互交換が可能です。複数の型は、表示書式を制御する、および、番号のある特定の型を表さなければならない入力に対する有効性検査をより強く行えるようにするために提供されています。

ISBN、ISMN、ISSN型では、可能ならば番号の簡略表示形式(ISxN 10)で表示されます。簡略形式に合わない番号ではISxN 13書式で表示されます。EAN13、ISBN13、ISMN13、ISSN13型では常にISxNの長めの形式(EAN13)で表示されます。

F.19.2. キャスト

isnモジュールは以下の型の組合せに関するキャストを提供します。

- ISBN13 <=> EAN13
- ISMN13 <=> EAN13
- ISSN13 <=> EAN13
- ISBN <=> EAN13
- ISMN <=> EAN13
- ISSN <=> EAN13
- UPC <=> EAN13
- ISBN <=> ISBN13
- ISMN <=> ISMN13
- ISSN <=> ISSN13

EAN13から他の型へキャストする時、その値が他の型のドメイン内であるかどうか実行時に検査が行われます。ドメイン内になければエラーが発生します。他のキャストでは単にラベル付けを再実行するだけですので、常に成功します。

F.19.3. 関数と演算子

isnモジュールは標準的な比較演算子とこれらデータ型すべてに対するB-treeおよびハッシュインデックスサポートを提供します。さらに、表 F.12で示される複数の特化した関数も存在します。以下の表ではisnはこのモジュールのデータ型のいずれか1つを意味します。

表F.12 isn Functions

関数 説明
isn_weak (boolean) → boolean weak入力モードを設定し、新しい設定を返します。
isn_weak () → boolean

関数 説明
weakモードの現在の状態を返します。
make_valid (isn) → isn 無効な番号を検証します（無効フラグを消去します）。
is_valid (isn) → boolean 無効フラグの有無を検査します。

*weak*モードは無効なデータをテーブルに挿入できるようにするために使用されます。無効とは間違ったチェックディジットを意味するものであり、番号自体は存在します。

このweakモードを使いたいと考えるのは何故でしょうか。大規模なISBN番号群があり、その内の多くが何らかの理由で間違ったチェックディジットを持つことはあり得ます。（印刷された一覧をスキャンしてOCRした結果番号を間違えた場合、手作業で番号を取り出した場合などがあり得ます。）とにかく、こうした混乱は整理したいことですが、データベース内に番号をすべて取り込んで、より簡単に情報を検査し有効にすることができるように、外部ツールを使用してデータベース内の無効な番号の位置を特定したいと思うかも知れません。例えば、テーブル内の無効な番号をすべて選択したいと思うかも知れません。

weakモードを使用して無効な番号をテーブルに挿入する時、番号は修正されたチェックディジット付きで挿入されますが、最後にイクスクラメーション印（!）付きで、例えば0-11-000322-5!と表示されます。この無効印はis_valid関数を使って検査することができ、また、make_valid関数で消去することができます。

また、番号の最後に!文字を付与することで、weakモードでなくとも無効な番号を強制的に挿入することもできます。

この他の入力における特殊な機能として、チェックディジットとして?を書くことができます。これにより正確なチェックディジットが自動的に挿入されます。

F.19.4. 例

```
---型を直接使う
SELECT isbn('978-0-393-04002-9');
SELECT isbn13('0901690546');
SELECT issn('1436-4522');


--型キャスト
-- 番号が対象の型の範囲として有効な場合にのみean13から他の型へキャストできることに注意
-- そのため、
      次のようなものは上手くいかない: select isbn(ean13('0220356483481'));
-- しかし以下は上手くいく
SELECT upc(ean13('0220356483481'));
SELECT ean13(upc('220356483481'));
```

```
--ISBN番号を保持する列が１つあるテーブルを作成する
CREATE TABLE test (id isbn);
INSERT INTO test VALUES('9780393040029');


--チェックディジットを自動的に計算する('?'を見よ)
INSERT INTO test VALUES('220500896?');
INSERT INTO test VALUES('978055215372?');

SELECT issn('3251231?');
SELECT ismn('979047213542?');


--weakモードを利用する
SELECT isn_weak(true);
INSERT INTO test VALUES('978-0-11-000533-4');
INSERT INTO test VALUES('9780141219307');
INSERT INTO test VALUES('2-205-00876-X');
SELECT isn_weak(false);

SELECT id FROM test WHERE NOT is_valid(id);
UPDATE test SET id = make_valid(id) WHERE id = '2-205-00876-X!';

SELECT * FROM test;

SELECT isbn13(id) FROM test;
```

F.19.5. 参考文献

本モジュールを実装するための情報は以下を含むいくつかのサイトを通して集められました。

- https://www.isbn-international.org/
- https://www.issn.org/
- https://www.ismn-international.org/
- https://www.wikipedia.org/

ハイフン付けに使用した接頭辞も以下から集められました。

- https://www.gs1.org/standards/id-keys
- https://en.wikipedia.org/wiki/List_of_ISBN_identifier_groups
- https://www.isbn-international.org/content/isbn-users-manual

- https://en.wikipedia.org/wiki/International_Standard_Music_Number
- https://www.ismn-international.org/ranges.html

アルゴリズムの作成には注意を払い、公式ISBN、ISMN、ISSNユーザマニュアルで提示されたアルゴリズムに対して念入り過ぎるほど検証されました。

F.19.6. 作者

Germán Méndez Bravo (Kronuz), 2004–2006

本モジュールはGarrett A. Wollmanの`isbn_issn`コードに触発されたものです。

F.20. lo

`lo`モジュールはラージオブジェクト（LOやBLOBとも呼ばれます）保守作業のサポートを提供します。`lo`データ型と`lo_manage`トリガが含まれます。

このモジュールは「trusted」と見なされます。つまり、現在のデータベースに対して`CREATE`権限を持つ非スーパーユーザがインストールできます。

F.20.1. 原理

JDBCドライバにおける問題の1つ（ODBCドライバでもこれは影響します）は、規定ではBLOB（バイナリラージオブジェクト）への参照はテーブル内に格納され、その項目が変更されると、関連するBLOBがデータベースから削除されると想定している点です。

PostgreSQLの立場では、これは起こりません。ラージオブジェクトは独自の権限をもったオブジェクトとして扱われます。テーブル項目はOIDによりラージオブジェクトを参照することはできますが、同じラージオブジェクトOIDを参照するテーブル項目を複数持つことも可能です。このため、システムは、こうした項目を変更または削除したという理由だけでは、ラージオブジェクトを削除しません。

さて、これはPostgreSQL固有のアプリケーションでは問題ありませんが、JDBCやODBCを使用する標準的なコードでは、オブジェクトが削除されず、孤児、つまりどこからも参照されずディスクを消費するだけのオブジェクトになります。

`lo`モジュールによりLO参照列を持つテーブルにトリガを付与して、これを解消することができます。このトリガは基本的には、ラージオブジェクトを参照する値を削除または変更した時常に`lo_unlink`を単に行います。このトリガを使用する時は、単一のデータベースのみがトリガの対象列で参照されるラージオブジェクトを参照することを前提とします。

また、本モジュールは、単に`oid`型のドメインに過ぎない`lo`データ型を提供します。ラージオブジェクトへの参照を持つデータベース列とこの他のOIDを持つデータベース列との間に違いを持たせるために有用です。実

際このトリガを使用するためにlo型を使用する必要はありません。しかし、データベース内のどの列がトリガで管理されているラージオブジェクトを示しているかを保持するために、これを使用することは簡便かもしれません。また、BLOB列でloを使用しない場合、ODBCドライバが混乱してしまうと取りざたされています。

F.20.2. 使用方法

簡単な使用例を示します。

```
CREATE TABLE image (title text, raster lo);

CREATE TRIGGER t_raster BEFORE UPDATE OR DELETE ON image
    FOR EACH ROW EXECUTE FUNCTION lo_manage(raster);
```

一意なラージオブジェクト参照を含む列それぞれに対し、BEFORE UPDATE OR DELETEトリガを作成してください。そして、単一のトリガ引数として列名を指定してください。BEFORE UPDATE OF column_nameを使って列が更新される時にのみ実行するようトリガを制限することもできます。同一テーブル上に複数のlo型の列を持たせる必要がある場合、それぞれに対して別のトリガを作成してください。同一テーブル上の各トリガに別の名前を与えることは忘れないでください。

F.20.3. 制限

- トリガが実行されませんので、テーブル削除により含まれるオブジェクトは孤児化します。DROP TABLEの前にDELETE FROM tableを行うことで防止することができます。

  TRUNCATEも同様の危険があります。

  ラージオブジェクトを孤児化させた、または孤児化させた疑いがある場合は、消去するための手助けとなるvacuumloモジュールを参照してください。lo_manageトリガのバックネットとしてvacuumloを時々実行することを勧めます。

- フロントエンドの中には独自のテーブルを作成するものがあり、その場合、関連するトリガは作成されません。また、ユーザはトリガを作成することを忘れる(または知らない)かもしれません。

F.20.4. 作者

Peter Mount <peter@retep.org.uk>

F.21. ltree

本モジュールは階層ツリーを模擬した構造に格納されたデータのラベルを表現するltreeデータ型を実装します。ラベルツリー全体を検索する高度な機能を提供します。

このモジュールは「trusted」と見なされます。つまり、現在のデータベースに対してCREATE権限を持つ非スーパーユーザがインストールできます。

F.21.1. 定義

ラベルは、アルファベット文字とアンダースコア（例えばCロケールではA-Za-z0-9_文字が許されます。）の並びです。ラベルの長さは256文字未満でなければなりません。

例えば42、Personal_Servicesです。

ラベル経路は、例えばL1.L2.L3のようなドットで区切られた0個以上のラベルの並びであり、階層ツリーのルートから特定のノードまでの経路を表します。ラベル経路の長さは65535ラベルを超えることはできません。

例：'Top.Countries.Europe.Russia'

ltreeモジュールは以下の複数のデータ型を提供します。

- ltreeはラベル経路を格納します。
- lqueryは、ltree値に一致する正規表現のようなパターンを表現します。単一の単語は経路内のラベルに一致します。スター記号（*）は0個以上のラベルに一致します。ドットでつなげることで、ラベル経路全体に一致するパターンを形作ることができます。以下に例を示します。

```
foo         正確にfooというラベル経路に一致します。
*.foo.*     fooというラベルを含むラベル経路すべてに一致します。
*.foo       fooというラベルで終わるラベル経路すべてに一致します。
```

スター記号と単一の単語のどちらも一致可能なラベル数を制限するために量指定を行うことができます。

```
*{n}        正確にn個のラベルに一致します。
*{n,}       少なくともn個のラベルに一致します。
*{n,m}      少なくともn個に一致し、
      多くてもm個を超えないラベルに一致します。
*{,m}       最大m個のラベルに一致します。つまりと次と同じです。*{0,m}
foo{n,m}    少なくともn個に一致し、
      多くてもm個を超えないfooに一致します。
foo{,}      ゼロを含む任意の数のfooに一致します。
```

明示的な量指定子が存在しなければ、スター記号に対するデフォルトは任意の数のラベルに一致(つまり{,})である一方、非スター項目に対するデフォルトは正確に1回(つまり{1})です。

単なる正確な一致以上の一致を行うために、スターでないlquery項目の終端に記述できる複数の修飾子が存在します。

```
@           大文字小文字を区別しない一致。例えばa@はAに一致します。
```

```
*           この接頭辞を持つすべてのラベルに一致。例えばfoo*はfoobarに一致します。
%           最初のアンダースコアで区切られた単語に一致。
```

%の動作は多少複雑です。ラベル全体ではなく単語一致を試みます。例えばfoo_bar%はfoo_bar_bazに一致しますがfoo_barbazに一致しません。*と組み合わせる場合、接頭辞一致が各単語ごとに適用されます。例えばfoo_bar%*はfoo1_bar2_bazに一致しますが、foo1_br2_bazに一致しません。

また、項目のいずれかに一致させるために|(論理和)で区切って、修飾子が付いているかもしれない複数の非スター項目を記述することもできます。さらに、非スターグループ先頭に! (否定)を記述して選択肢のいずれにも一致しないすべてのラベルに一致させることもできます。もしあれば、量指定子はグループの最後になります。これはグループ全体として一致する数を意味します(すなわち、一致するラベルの数、または、選択肢のいずれにも一致しない数です)。

以下に注釈付きのlqueryの例を示します。

```
Top.*{0,2}.sport*@.!football|tennis{1,}.Russ*|Spain
a.  b.     c.      d.                   e.
```

この問い合わせは以下のようなラベルに一致します。

a. Topラベルから始まる。

b. 次いで0から2個のラベルを持つ。

c. 直後にsport接頭辞(大文字小文字の区別無)から始まるラベルを持つ。

d. そして、footballにもtennisにも一致しない1つ以上のラベルを持つ。

e. Russから始まる、または、正確にSpainに一致するラベルで終わる。

- ltxtqueryはltree値に対する全文検索のようなパターンを表します。ltxtquery値は、おそらく最後に@、*、%修飾子を持った単語からなります。修飾子の意味はlqueryと同じです。単語は& (論理積)、| (論理和)、! (否定)、括弧を組み合わせることが可能です。主なlqueryとの違いは、ltxtqueryはラベル経路上の位置を考慮せずに単語に一致することです。

  ltxtqueryの例を示します。

  ```
  Europe & Russia*@ & !Transportation
  ```

  これはEuropeラベルとRussia(大文字小文字の区別無)から始まるラベルを含む経路に一致します。しかし、Transportationラベルを含む経路は一致しません。経路内の単語の位置は重要ではありません。また、%が使用された場合、位置に関係なく、単語をラベル内のアンダースコアで区切られた何らかの単語に一致させることができます。

注意:ltxtqueryではシンボルの間に空白を入れることができますが、ltreeとlqueryではできません。

F.21.2. 演算子と関数

ltree型は、通常の比較演算子=、<>、<、>、<=、>=を持ちます。比較では、ツリーの巡回順でソートされ、ノードの子要素はラベルテキストでソートされます。さらに、表 F.13に示す特殊な演算子が使用可能です。

表F.13 ltree演算子

演算子 説明
ltree @> ltree → boolean 左辺の引数が右辺の祖先要素（か同じ）かどうか。
ltree <@ ltree → boolean 左辺の引数が右辺の子孫要素（か同じ）かどうか。
ltree ~ lquery → boolean lquery ~ ltree → boolean ltreeがlqueryに一致するかどうか。
ltree ? lquery[] → boolean lquery[] ? ltree → boolean ltreeが配列内のいずれかのlqueryに一致するかどうか。
ltree @ ltxtquery → boolean ltxtquery @ ltree → boolean ltreeがltxtqueryに一致するかどうか。
ltree || ltree → ltree ltree経路を連結します。
ltree || text → ltree text || ltree → ltree テキストをltreeに変換し、連結します。
ltree[] @> ltree → boolean ltree <@ ltree[] → boolean 配列にltreeの祖先要素が含まれるかどうか。
ltree[] <@ ltree → boolean ltree @> ltree[] → boolean 配列にltreeの子孫要素が含まれるかどうか。
ltree[] ~ lquery → boolean lquery ~ ltree[] → boolean 配列にlqueryに一致する経路が含まれるかどうか。
ltree[] ? lquery[] → boolean lquery[] ? ltree[] → boolean ltree配列にいずれかのlqueryに一致する経路が含まれるかどうか。
ltree[] @ ltxtquery → boolean ltxtquery @ ltree[] → boolean 配列にltxtqueryに一致する経路が含まれるかどうか。
ltree[] ?@> ltree → ltree ltreeの祖先要素となる配列内の最初の要素を、存在しなければNULLを返します。
ltree[] ?<@ ltree → ltree ltreeの子孫要素となる配列内の最初の要素を、存在しなければNULLを返します。
ltree[] ?~ lquery → ltree

演算子 説明
lqueryに一致する配列内の最初の要素を、存在しなければNULLを返します。
ltree[] ?@ ltxtquery → ltree ltxtqueryに一致する配列内の最初の要素を、存在しなければNULLを返します。

演算子<@、@>、@、~には類似の演算子^<@、^@>、^@、^~があります。後者はインデックスを使用しない点を除き、同一です。後者は試験の際にだけ役に立ちます。

使用可能な関数を表 F.14に示します。

表F.14 ltree関数

関数 説明 例
subltree (ltree, start integer, end integer) → ltree start位置からend-1位置までのltreeの部分経路を返します（位置は0から始まります）。 subltree('Top.Child1.Child2', 1, 2) → Child1
subpath (ltree, offset integer, len integer) → ltree offset位置からlen個のltreeの部分経路を返します。offsetが負の場合、部分経路は経路の終端から数えた位置から始まります。lenが負の場合、経路の終端から指定個のラベルを除きます。 subpath('Top.Child1.Child2', 0, 2) → Top.Child1
subpath (ltree, offset integer) → ltree offset位置から経路の終端までのltreeの部分経路を返します。offsetが負の場合、部分経路は経路の終端から数えた位置から始まります。 subpath('Top.Child1.Child2', 1) → Child1.Child2
nlevel (ltree) → integer 経路内のラベル数を返します。 nlevel('Top.Child1.Child2') → 3
index (a ltree, b ltree) → integer a内でbが最初に出現する位置を、存在しなければ-1を返します。 index('0.1.2.3.5.4.5.6.8.5.6.8', '5.6') → 6
index (a ltree, b ltree, offset integer) → integer a内でbが最初に出現する位置を、存在しなければ-1を返します。検索はoffsetから始まります。負のoffsetは経路終端から-offsetラベルから検索を始めることを意味します。 index('0.1.2.3.5.4.5.6.8.5.6.8', '5.6', -4) → 9
text2ltree (text) → ltree textをltreeにキャストします。
ltree2text (ltree) → text ltreeをtextにキャストします。
lca (ltree [, ltree [, ...]]) → ltree 経路で共通する最長接頭辞を計算します（最大8個の引数をサポートします）。 lca('1.2.3', '1.2.3.4.5.6') → 1.2

関数 説明 例
lca(ltree[])→ltree 配列内の経路で共通する最長接頭辞を計算します。 lca(array['1.2.3'::ltree,'1.2.3.4'])→1.2

F.21.3. インデックス

ltreeは、以下で示された演算子を高速化できる、複数種類のインデックスをサポートします。

- ltreeに対するB-treeインデックス：<、<=、=、>=、>

- ltreeに対するGiSTインデックス(gist_ltree_ops演算子クラス)：<、<=、=、>=、>、@>、<@、@、~、?

  gist_ltree_opsGiST演算子クラスは経路ラベルの集合をビットマップ署名として近似します。オプションの整数パラメータsiglenは、署名の長さをバイト単位で決定します。デフォルトの署名の長さは8バイトです。署名の長さの有効な値は1から2024バイトまでです。長い署名では、インデックスはより大きくなってしまいますが、(インデックスのより小さな部分とより少ないヒープページを走査することで)検索がより正確になります。

  デフォルトの署名の長さが8バイトのインデックスを作成する例。

  ```
  CREATE INDEX path_gist_idx ON test USING GIST (path);
  ```

  署名の長さが100バイトのインデックスを作成する例。

  ```
  CREATE INDEX path_gist_idx ON test USING GIST (path gist_ltree_ops(siglen=100));
  ```

- ltree[]に対するGiSTインデックス：ltree[] <@ ltree、ltree @> ltree[]、@、~、?

  gist__ltree_ops GiST演算子クラスはgist_ltree_opsと同じように動作しますが、署名の長さをパラメータとして取ります。gist__ltree_opsでのsiglenのデフォルトの値は28バイトです。

  デフォルトの署名の長さが28バイトのインデックスを作成する例。

  ```
  CREATE INDEX path_gist_idx ON test USING GIST (array_path);
  ```

  署名の長さが100バイトのインデックスを作成する例。

  ```
  CREATE INDEX path_gist_idx ON test USING GIST (array_path gist__ltree_ops(siglen=100));
  ```

  注意：この種類のインデックスは非可逆です。

F.21.4. 例

この例は、後述のデータを使用します（ソース配布内のcontrib/ltree/ltreetest.sqlファイルでも利用可能です）。

```
CREATE TABLE test (path ltree);
INSERT INTO test VALUES ('Top');
INSERT INTO test VALUES ('Top.Science');
INSERT INTO test VALUES ('Top.Science.Astronomy');
INSERT INTO test VALUES ('Top.Science.Astronomy.Astrophysics');
INSERT INTO test VALUES ('Top.Science.Astronomy.Cosmology');
INSERT INTO test VALUES ('Top.Hobbies');
INSERT INTO test VALUES ('Top.Hobbies.Amateurs_Astronomy');
INSERT INTO test VALUES ('Top.Collections');
INSERT INTO test VALUES ('Top.Collections.Pictures');
INSERT INTO test VALUES ('Top.Collections.Pictures.Astronomy');
INSERT INTO test VALUES ('Top.Collections.Pictures.Astronomy.Stars');
INSERT INTO test VALUES ('Top.Collections.Pictures.Astronomy.Galaxies');
INSERT INTO test VALUES ('Top.Collections.Pictures.Astronomy.Astronauts');
CREATE INDEX path_gist_idx ON test USING GIST (path);
CREATE INDEX path_idx ON test USING BTREE (path);
```

これで、以下の階層を記述するデータが投入されたtestテーブルができます。

```
Top
                     /   |  \
             Science Hobbies Collections
                 /       |             \
        Astronomy   Amateurs_Astronomy Pictures
           /  \                            |
Astrophysics  Cosmology                Astronomy
                                        /  |    \
                                  Galaxies Stars Astronauts
```

継承を行うことができます。

```
ltreetest=> SELECT path FROM test WHERE path <@ 'Top.Science';
                path
----------------------------------
 Top.Science
 Top.Science.Astronomy
 Top.Science.Astronomy.Astrophysics
 Top.Science.Astronomy.Cosmology
(4 rows)
```

経路一致の例をいくつか示します。

```
ltreetest=> SELECT path FROM test WHERE path ~ '*.Astronomy.*';
                     path
-----------------------------------------------
 Top.Science.Astronomy
```

```
 Top.Science.Astronomy.Astrophysics
 Top.Science.Astronomy.Cosmology
 Top.Collections.Pictures.Astronomy
 Top.Collections.Pictures.Astronomy.Stars
 Top.Collections.Pictures.Astronomy.Galaxies
 Top.Collections.Pictures.Astronomy.Astronauts
(7 rows)

ltreetest=> SELECT path FROM test WHERE path ~ '*.!pictures@.Astronomy.*';
                path
-----------------------------------
 Top.Science.Astronomy
 Top.Science.Astronomy.Astrophysics
 Top.Science.Astronomy.Cosmology
(3 rows)
```

全文検索の例をいくつか示します。

```
ltreetest=> SELECT path FROM test WHERE path @ 'Astro*% & !pictures@';
                path
-----------------------------------
 Top.Science.Astronomy
 Top.Science.Astronomy.Astrophysics
 Top.Science.Astronomy.Cosmology
 Top.Hobbies.Amateurs_Astronomy
(4 rows)

ltreetest=> SELECT path FROM test WHERE path @ 'Astro* & !pictures@';
                path
-----------------------------------
 Top.Science.Astronomy
 Top.Science.Astronomy.Astrophysics
 Top.Science.Astronomy.Cosmology
(3 rows)
```

関数を使用した経路構築の例です。

```
ltreetest=> SELECT subpath(path,0,2)||'Space'||subpath(path,2) FROM test WHERE path <@
 'Top.Science.Astronomy';
                 ?column?
-----------------------------------------
 Top.Science.Space.Astronomy
 Top.Science.Space.Astronomy.Astrophysics
 Top.Science.Space.Astronomy.Cosmology
(3 rows)
```

経路内の位置にラベルを挿入するSQL関数を作成することで、これを簡略化することができます。

```
CREATE FUNCTION ins_label(ltree, int, text) RETURNS ltree
    AS 'select subpath($1,0,$2) || $3 || subpath($1,$2);'
    LANGUAGE SQL IMMUTABLE;

ltreetest=> SELECT ins_label(path,2,'Space') FROM test WHERE path <@ 'Top.Science.Astronomy';
                ins_label
-----------------------------------------
 Top.Science.Space.Astronomy
 Top.Science.Space.Astronomy.Astrophysics
 Top.Science.Space.Astronomy.Cosmology
(3 rows)
```

F.21.5. 変換

PL/Python言語向けにltree型の変換を実装した追加の拡張が入手可能です。その拡張は、ltree_plpythonu、ltree_plpython2u、ltree_plpython3uという名前です(PL/Pythonの命名規約については45.1を参照してください)。関数を作成するときにこの変換をインストールして指定していれば、ltreeの値はPythonのリストにマップされます。(しかしながら、その逆は今のところサポートされていません。)

> **注意**
>
> 変換の拡張はltreeと同じスキーマにインストールすることを強く勧めます。さもないと、変換の拡張のスキーマが悪意のあるユーザにより定義されたオブジェクトを含んでいた場合に、インストール時のセキュリティ問題になります。

F.21.6. 作者

開発はすべてTeodor Sigaev (<teodor@stack.net>)とOleg Bartunov (<oleg@sai.msu.su>)によりなされました。さらなる情報についてはhttp://www.sai.msu.su/~megera/postgres/gist/を参照してください。作者は有用な議論を行ったEugeny Rodichevに感謝しています。コメントや不具合報告を歓迎します。

F.22. pageinspect

pageinspectモジュールは、デバッグの際に有用となる低レベルなデータベースページの内容を調べることができる関数を提供します。これらの関数はすべて、スーパーユーザのみが使用することができます。

F.22.1. 一般的な関数

`get_raw_page(relname text, fork text, blkno int) returns bytea`

get_raw_pageは指定された名前付きリレーションの指定されたブロックを読み取り、bytea値としてそのコピーを返します。これにより、単一ブロックの時間的に一貫性を持つコピーを入手することができま

す。forkは、主データフォークでは'main'、空き領域マップでは'fsm'、可視性マップでは'vm'、初期化フォークでは'init'としなければなりません。

get_raw_page(relname text, blkno int) returns bytea

get_raw_pageの簡略形であり、主フォークから読み取ります。get_raw_page(relname, 'main', blkno)と同じです。

page_header(page bytea) returns record

page_headerは、すべてのPostgreSQLヒープとインデックスページで共通するフィールドを表示します。

get_raw_pageで得られたページイメージを引数として渡さなければなりません。以下に例を示します。

```
test=# SELECT * FROM page_header(get_raw_page('pg_class', 0));
    lsn    | checksum | flags  | lower | upper | special | pagesize | version | prune_xid
-----------+----------+--------+-------+-------+---------+----------+---------+-----------
 0/24A1B50 |        0 |      1 |   232 |   368 |    8192 |     8192 |       4 |         0
```

返される列は、PageHeaderData構造体のフィールドに対応します。詳細はsrc/include/storage/bufpage.hを参照してください。

checksumフィールドはページに保存されたチェックサムであり、ページがどういうわけか壊れていれば正しくないでしょう。このインスタンスに対してデータチェックサムが有効になっていなければ、保存されている値には意味がありません。

page_checksum(page bytea, blkno int4) returns smallint

page_checksumは指定されたブロックに位置するかのようにページのチェックサムを計算します。

get_raw_pageで得られたページイメージを引数として渡さなければなりません。以下に例を示します。

```
test=# SELECT page_checksum(get_raw_page('pg_class', 0), 0);
 page_checksum
---------------
         13443
```

チェックサムはブロック番号に依存するので、(難解なデバッグをする場合以外は)対応するブロック番号を渡さなければならないことに注意してください。

この関数で計算されたチェックサムは、page_header関数のchecksum結果フィールドと比較できます。このインスタンスに対してデータチェックサムが有効になっていれば、二つの値は等しくならなければなりません。

fsm_page_contents(page bytea) returns text

fsm_page_contentsは、FSMページの内部ノード構造を表示します。以下に例を示します。

```
test=# SELECT fsm_page_contents(get_raw_page('pg_class', 'fsm', 0));
```

出力は複数行からなる文字列で、各行がページ内のバイナリツリー(二分木)の1ノードを表します。それらのノードのうち、非ゼロのノードのみが出力されます。そのページから返される次のスロットを指し示すための"next(次)"と呼ばれるポインタも出力されます。

FSMページの構造に関する更に詳しい情報は、src/backend/storage/freespace/READMEを参照してください。

F.22.2. ヒープ関数

heap_page_items(page bytea) returns setof record

heap_page_itemsはヒープページ上の行ポインタをすべて表示します。使用中の行ポインタでは、タプルヘッダおよびタプルの生データも表示されます。生ページがコピーされた時点のMVCCスナップショットでタプルが可視かどうかは関係なく、すべてのタプルが表示されます。

get_raw_pageで得られたヒープページイメージを引数として渡さなければなりません。以下に例を示します。

```
test=# SELECT * FROM heap_page_items(get_raw_page('pg_class', 0));
```

返されるフィールドの説明については、src/include/storage/itemid.hおよびsrc/include/access/htup_details.hを参照してください。

heap_tuple_infomask_flags関数を使用すると、ヒープタプルのt_infomaskおよびt_infomask2のフラグビットを展開することができます。

tuple_data_split(rel_oid oid, t_data bytea, t_infomask integer, t_infomask2 integer, t_bits text [, do_detoast bool]) returns bytea[]

tuple_data_splitはバックエンドの内部がするのと同じ方法で、タプルデータを属性に分割します。

```
test=# SELECT tuple_data_split('pg_class'::regclass, t_data, t_infomask, t_infomask2, t_bits)
 FROM heap_page_items(get_raw_page('pg_class', 0));
```

この関数はheap_page_itemsの戻り値の属性と同じ引数で呼び出してください。

do_detoastがtrueの場合、属性は必要に応じて非TOAST化されます。デフォルト値はfalseです。

heap_page_item_attrs(page bytea, rel_oid regclass [, do_detoast bool]) returns setof record

heap_page_item_attrsはheap_page_itemsと同じですが、タプルの生データをdo_detoast(デフォルトはfalse)によって非TOAST化可能な属性の配列として返すところが異なります。

get_raw_pageで取得できるヒープページのイメージを引数として渡してください。以下に例を示します。

```
test=# SELECT * FROM heap_page_item_attrs(get_raw_page('pg_class', 0), 'pg_class'::regclass);
```

heap_tuple_infomask_flags(t_infomask integer, t_infomask2 integer) returns record

heap_tuple_infomask_flagsは、heap_page_itemsから返されるt_infomaskおよびt_infomask2を、フラグ名で構成される人間が見てわかる形式の配列セットにデコードします。このとき、すべてのフラグ用の列が一つ、結合されたフラグ用の列が一つあります。以下に例を示します。

```
test=# SELECT t_ctid, raw_flags, combined_flags
         FROM heap_page_items(get_raw_page('pg_class', 0)),
           LATERAL heap_tuple_infomask_flags(t_infomask, t_infomask2)
         WHERE t_infomask IS NOT NULL OR t_infomask2 IS NOT NULL;
```

この関数はheap_page_itemsの戻り値の属性と同じ引数で呼び出してください。

結合されたフラグは、HEAP_XMIN_FROZENなど、複数のrawビットの値を考慮するソースレベルのマクロから得られたビットを表示します。

返されるフラグ名の説明については、src/include/access/htup_details.hを参照して下さい。

F.22.3. B-tree関数

bt_metap(relname text) returns record

bt_metapはB-treeインデックスのメタページに関する情報を返します。以下に例を示します。

```
test=# SELECT * FROM bt_metap('pg_cast_oid_index');
-[ RECORD 1 ]-----------+-------
magic                   | 340322
version                 | 4
root                    | 1
level                   | 0
fastroot                | 1
fastlevel               | 0
oldest_xact             | 582
last_cleanup_num_tuples | 1000
allequalimage           | f
```

bt_page_stats(relname text, blkno int) returns record

bt_page_statsは、B-treeインデックスの個別のページについての要約情報を返します。以下に例を示します。

```
test=# SELECT * FROM bt_page_stats('pg_cast_oid_index', 1);
-[ RECORD 1 ]-+-----
blkno         | 1
type          | l
live_items    | 224
dead_items    | 0
avg_item_size | 16
page_size     | 8192
```

```
free_size     | 3668
btpo_prev     | 0
btpo_next     | 0
btpo          | 0
btpo_flags    | 3
```

bt_page_items(relname text, blkno int) returns setof record

bt_page_itemsは、B-treeインデックスページ上の全項目についての詳細情報を返します。以下に例を示します。

```
test=# SELECT itemoffset, ctid, itemlen, nulls, vars, data, dead, htid, tids[0:2] AS
 some_tids
        FROM bt_page_items('tenk2_hundred', 5);
 itemoffset |   ctid    | itemlen | nulls | vars |          data           | dead |  htid  |
     some_tids
------------+-----------+---------+-------+------+-------------------------+------+--------
+--------------------
          1 | (16,1)    |      16 | f     | f    | 30 00 00 00 00 00 00 00 |      |        |
          2 | (16,8292) |     616 | f     | f    | 24 00 00 00 00 00 00 00 | f    | (1,6)  |
 {"(1,6)","(10,22)"}
          3 | (16,8292) |     616 | f     | f    | 25 00 00 00 00 00 00 00 | f    | (1,18) |
 {"(1,18)","(4,22)"}
          4 | (16,8292) |     616 | f     | f    | 26 00 00 00 00 00 00 00 | f    | (4,18) |
 {"(4,18)","(6,17)"}
          5 | (16,8292) |     616 | f     | f    | 27 00 00 00 00 00 00 00 | f    | (1,2)  |
 {"(1,2)","(1,19)"}
          6 | (16,8292) |     616 | f     | f    | 28 00 00 00 00 00 00 00 | f    | (2,24) |
 {"(2,24)","(4,11)"}
          7 | (16,8292) |     616 | f     | f    | 29 00 00 00 00 00 00 00 | f    | (2,17) |
 {"(2,17)","(11,2)"}
          8 | (16,8292) |     616 | f     | f    | 2a 00 00 00 00 00 00 00 | f    | (0,25) |
 {"(0,25)","(3,20)"}
          9 | (16,8292) |     616 | f     | f    | 2b 00 00 00 00 00 00 00 | f    | (0,10) |
 {"(0,10)","(0,14)"}
         10 | (16,8292) |     616 | f     | f    | 2c 00 00 00 00 00 00 00 | f    | (1,3)  |
 {"(1,3)","(3,9)"}
         11 | (16,8292) |     616 | f     | f    | 2d 00 00 00 00 00 00 00 | f    | (6,28) |
 {"(6,28)","(11,1)"}
         12 | (16,8292) |     616 | f     | f    | 2e 00 00 00 00 00 00 00 | f    | (0,27) |
 {"(0,27)","(1,13)"}
         13 | (16,8292) |     616 | f     | f    | 2f 00 00 00 00 00 00 00 | f    | (4,17) |
 {"(4,17)","(4,21)"}
(13 rows)
```

これはB-treeのリーフページです。テーブルを指し示すすべてのタプルは、ポスティングリストのタプルになっています（これらはすべて6バイトTIDが合計100個格納されます）。また、itemoffset番号1に

は「ハイキー(high key)」タプルもあります。この例では、各タプルに関するエンコードされた情報を格納するためにctidが使用されていますが、リーフページタプルでは、ヒープTIDが直接ctidフィールドに格納されることがよくあります。tidsはポスティングリストとして格納されるTIDのリストです。

内部ページ(示されていません)では、ctidのブロック番号部分は、「ダウンリンク(downlink)」であり、インデックス内の他のページのブロック番号です。ctidのオフセット部分(2番め)には、存在する列の数など、タプルに関するエンコードされた情報が格納されます(サフィックスの切り捨てによって、不要なサフィックス列が削除されている場合があります)。切り捨てられた列は、「マイナス無限大(minus infinity)」を持つものとして扱われます。

htidは、基礎となるタプル表現に関係なく、タプルのヒープTIDを示します。この値は、ctidと一致する場合もあれば、ポスティングリストのタプルおよび内部ページのタプルが使用する別の表現からデコードされる場合もあります。内部ページのタプルでは、実装レベルのヒープTID列が切り捨てられ、NULL htid値として表されます。

右端以外のページ(btpo_nextフィールドの値が0でないページ)において、最初の項目はページの「ハイキー」であることに注意してください。つまりそのページに現れるすべての項目の上限となるdataになりますが、一方でそのctidフィールドは別のブロックを指していないことを意味します。また、内部ページでは、最初の実データ項目(ハイキーでない最初の項目)は、確実にすべての列が切り捨てられ、そのdataフィールドに実際の値は残りません。しかし、そのような項目でも、そのctidフィールドには有効なダウンリンクが入っています。

B-Treeインデックスの構造についての詳細は63.4.1を参照してください。重複排除とポスティングリストについての詳細は63.4.2を参照してください。

bt_page_items(page bytea) returns setof record

ページをbt_page_itemsにbyteaの値として渡すことも可能です。get_raw_pageで得られたページイメージを引数として渡さなければなりません。なので、最後の例は以下のように書き直すこともできます。

```
test=# SELECT itemoffset, ctid, itemlen, nulls, vars, data, dead, htid, tids[0:2] AS
 some_tids
       FROM bt_page_items(get_raw_page('tenk2_hundred', 5));
 itemoffset |   ctid    | itemlen | nulls | vars |          data           | dead |  htid  |
    some_tids
------------+-----------+---------+-------+------+-------------------------+------+--------
+--------------------
          1 | (16,1)    |      16 | f     | f    | 30 00 00 00 00 00 00 00 |      |        |
          2 | (16,8292) |     616 | f     | f    | 24 00 00 00 00 00 00 00 | f    | (1,6)  |
 {"(1,6)","(10,22)"}
          3 | (16,8292) |     616 | f     | f    | 25 00 00 00 00 00 00 00 | f    | (1,18) |
 {"(1,18)","(4,22)"}
          4 | (16,8292) |     616 | f     | f    | 26 00 00 00 00 00 00 00 | f    | (4,18) |
 {"(4,18)","(6,17)"}
          5 | (16,8292) |     616 | f     | f    | 27 00 00 00 00 00 00 00 | f    | (1,2)  |
 {"(1,2)","(1,19)"}
          6 | (16,8292) |     616 | f     | f    | 28 00 00 00 00 00 00 00 | f    | (2,24) |
 {"(2,24)","(4,11)"}
```

```
         7 | (16,8292) |     616 | f     | f    | 29 00 00 00 00 00 00 00 | f    | (2,17) |
 {"(2,17)","(11,2)"}
         8 | (16,8292) |     616 | f     | f    | 2a 00 00 00 00 00 00 00 | f    | (0,25) |
 {"(0,25)","(3,20)"}
         9 | (16,8292) |     616 | f     | f    | 2b 00 00 00 00 00 00 00 | f    | (0,10) |
 {"(0,10)","(0,14)"}
        10 | (16,8292) |     616 | f     | f    | 2c 00 00 00 00 00 00 00 | f    | (1,3)  |
 {"(1,3)","(3,9)"}
        11 | (16,8292) |     616 | f     | f    | 2d 00 00 00 00 00 00 00 | f    | (6,28) |
 {"(6,28)","(11,1)"}
        12 | (16,8292) |     616 | f     | f    | 2e 00 00 00 00 00 00 00 | f    | (0,27) |
 {"(0,27)","(1,13)"}
        13 | (16,8292) |     616 | f     | f    | 2f 00 00 00 00 00 00 00 | f    | (4,17) |
 {"(4,17)","(4,21)"}
(13 rows)
```

その他の詳細はすべて前の項目で説明したものと同じです。

F.22.4. BRIN関数

brin_page_type(page bytea) returns text

brin_page_typeは指定のBRINインデックスページのページ種別を返します。ページが有効なBRINページでないときはエラーが発生します。以下に例を示します。

```
test=# SELECT brin_page_type(get_raw_page('brinidx', 0));
 brin_page_type
----------------
 meta
```

brin_metapage_info(page bytea) returns record

brin_metapage_infoはBRINインデックスのメタページについて様々な情報を返します。以下に例を示します。

```
test=# SELECT * FROM brin_metapage_info(get_raw_page('brinidx', 0));
   magic    | version | pagesperrange | lastrevmappage
------------+---------+---------------+----------------
 0xA8109CFA |       1 |             4 |              2
```

brin_revmap_data(page bytea) returns setof tid

brin_revmap_dataはBRINインデックスの範囲マップページのタプル識別子のリストを返します。以下に例を示します。

```
test=# SELECT * FROM brin_revmap_data(get_raw_page('brinidx', 2)) LIMIT 5;
```

```
  pages
---------
 (6,137)
 (6,138)
 (6,139)
 (6,140)
 (6,141)
```

brin_page_items(page bytea, index oid) returns setof record

brin_page_itemsはBRINデータページに記録されているデータを返します。以下に例を示します。

```
test=# SELECT * FROM brin_page_items(get_raw_page('brinidx', 5),
                                     'brinidx')
       ORDER BY blknum, attnum LIMIT 6;
 itemoffset | blknum | attnum | allnulls | hasnulls | placeholder |    value
------------+--------+--------+----------+----------+-------------+--------------
        137 |      0 |      1 | t        | f        | f           |
        137 |      0 |      2 | f        | f        | f           | {1 .. 88}
        138 |      4 |      1 | t        | f        | f           |
        138 |      4 |      2 | f        | f        | f           | {89 .. 176}
        139 |      8 |      1 | t        | f        | f           |
        139 |      8 |      2 | f        | f        | f           | {177 .. 264}
```

返される列はBrinMemTuple構造体およびBrinValues構造体のフィールドに対応します。詳しくはsrc/include/access/brin_tuple.hを参照して下さい。

F.22.5. GIN関数

gin_metapage_info(page bytea) returns record

gin_metapage_infoはGINインデックスのメタページに関する情報を返します。以下に例を示します。

```
test=# SELECT * FROM gin_metapage_info(get_raw_page('gin_index', 0));
-[ RECORD 1 ]----+-----------
pending_head     | 4294967295
pending_tail     | 4294967295
tail_free_size   | 0
n_pending_pages  | 0
n_pending_tuples | 0
n_total_pages    | 7
n_entry_pages    | 6
n_data_pages     | 0
n_entries        | 693
version          | 2
```

gin_page_opaque_info(page bytea) returns record

gin_page_opaque_infoはページ種別のようなGINインデックスの不透明な領域についての情報を返します。以下に例を示します。

```
test=# SELECT * FROM gin_page_opaque_info(get_raw_page('gin_index', 2));
 rightlink | maxoff |         flags
-----------+--------+-----------------------
         5 |      0 | {data,leaf,compressed}
(1 row)
```

gin_leafpage_items(page bytea) returns setof record

gin_leafpage_itemsはGINのリーフページに格納されているデータについての情報を返します。以下に例を示します。

```
test=# SELECT first_tid, nbytes, tids[0:5] AS some_tids
        FROM gin_leafpage_items(get_raw_page('gin_test_idx', 2));
 first_tid | nbytes |                        some_tids
-----------+--------+-----------------------------------------------------------
 (8,41)    |    244 | {"(8,41)","(8,43)","(8,44)","(8,45)","(8,46)"}
 (10,45)   |    248 | {"(10,45)","(10,46)","(10,47)","(10,48)","(10,49)"}
 (12,52)   |    248 | {"(12,52)","(12,53)","(12,54)","(12,55)","(12,56)"}
 (14,59)   |    320 | {"(14,59)","(14,60)","(14,61)","(14,62)","(14,63)"}
 (167,16)  |    376 | {"(167,16)","(167,17)","(167,18)","(167,19)","(167,20)"}
 (170,30)  |    376 | {"(170,30)","(170,31)","(170,32)","(170,33)","(170,34)"}
 (173,44)  |    197 | {"(173,44)","(173,45)","(173,46)","(173,47)","(173,48)"}
(7 rows)
```

F.22.6. Hash関数

hash_page_type(page bytea) returns text

hash_page_typeは与えられたHASHインデックスページのページ種別を返します。以下に例を示します。

```
test=# SELECT hash_page_type(get_raw_page('con_hash_index', 0));
 hash_page_type
----------------
 metapage
```

hash_page_stats(page bytea) returns setof record

hash_page_statsはHASHインデックスのバケットページやオーバーフローページに関する情報を返します。以下に例を示します。

```
test=# SELECT * FROM hash_page_stats(get_raw_page('con_hash_index', 1));
```

```
-[ RECORD 1 ]---+-----------
live_items      | 407
dead_items      | 0
page_size       | 8192
free_size       | 8
hasho_prevblkno | 4096
hasho_nextblkno | 8474
hasho_bucket    | 0
hasho_flag      | 66
hasho_page_id   | 65408
```

hash_page_items(page bytea) returns setof record

hash_page_itemsはHASHインデックスページのバケットページやオーバーフローページに保存されたデータに関する情報を返します。以下に例を示します。

```
test=# SELECT * FROM hash_page_items(get_raw_page('con_hash_index', 1)) LIMIT 5;
 itemoffset |   ctid    |    data
------------+-----------+------------
          1 | (899,77)  | 1053474816
          2 | (897,29)  | 1053474816
          3 | (894,207) | 1053474816
          4 | (892,159) | 1053474816
          5 | (890,111) | 1053474816
```

hash_bitmap_info(index oid, blkno int) returns record

hash_bitmap_infoはHASHインデックスの特定のオーバーフローページに対するビットマップページのビットの状態を表示します。以下に例を示します。

```
test=# SELECT * FROM hash_bitmap_info('con_hash_index', 2052);
 bitmapblkno | bitmapbit | bitstatus
-------------+-----------+-----------
          65 |         3 | t
```

hash_metapage_info(page bytea) returns record

hash_metapage_infoはHASHインデックスのメタページに保存された情報を返します。以下に例を示します。

```
test=# SELECT magic, version, ntuples, ffactor, bsize, bmsize, bmshift,
test-#     maxbucket, highmask, lowmask, ovflpoint, firstfree, nmaps, procid,
test-#     regexp_replace(spares::text, '(,0)*}', '}') as spares,
test-#     regexp_replace(mapp::text, '(,0)*}', '}') as mapp
test-# FROM hash_metapage_info(get_raw_page('con_hash_index', 0));
-[ RECORD 1 ]-------------------------------------------------------------------------------
magic     | 105121344
version   | 4
```

```
ntuples   | 500500
ffactor   | 40
bsize     | 8152
bmsize    | 4096
bmshift   | 15
maxbucket | 12512
highmask  | 16383
lowmask   | 8191
ovflpoint | 28
firstfree | 1204
nmaps     | 1
procid    | 450
spares    | {0,0,0,0,0,0,1,1,1,1,1,1,1,1,3,4,4,4,45,55,58,59,508,567,628,704,1193,1202,1204}
mapp      | {65}
```

F.23. passwordcheck

passwordcheckモジュールはCREATE ROLEまたはALTER ROLEによって設定したユーザのパスワードを検査します。パスワードが弱すぎると考えられた場合、パスワードは拒絶されてコマンドはエラーで終わります。

このモジュールを有効にするには、postgresql.conf中のshared_preload_librariesに'$libdir/passwordcheck'を追加して、サーバを再起動してください。

ソースコードの変更により、このモジュールをユーザの用途に適合できます。例えば、パスワード検査のためCrackLib[2]を使用できます。これに必要な作業はMakefile中の 2行のコメントアウトを外してモジュールを再構築することだけです。(ライセンスの理由からCrackLibをデフォルトで組み込むことができません。) CrackLibがなくても、モジュールはパスワードの強度に関する幾つかの単純な規則を強制します。ユーザはこの規則を、適切となるように修正または拡張できます。

注意

暗号化されないパスワードが、ネットワークを送信されること、サーバログに記録されることおよびデータベース管理者に盗聴されることを防ぐために、PostgreSQLはパスワードの一方向暗号化をユーザに提供できます。この機能を使用して、多くのクライアントプログラムはサーバへ送信する前にパスワードを暗号化できます。

一方向暗号化されたパスワードは復号できないので、これがpasswordcheckモジュールの有効性の限界となります。この理由により、高度のセキュリティが要求される場合、passwordcheckは推奨されません。データベース内部に保存したパスワードに依存するよりは、GSSAPIのような外部の認証方法を使用した方が安全です(第20章参照)。

その他の方法として、一方向暗号化されたパスワードを拒否するためにpasswordcheckを修正できます。しかし、この方法ではパスワードが平文のテキストとして送信されるため、ユーザに多大なセキュリティリスクを負担させます。

[2] https://sourceforge.net/projects/cracklib/

F.24. pg_buffercache

pg_buffercacheモジュールは、共有バッファキャッシュで何が起きているかをリアルタイムに確認する方法を提供します。

このモジュールはレコード集合を返すpg_buffercache_pages C関数と、簡単に利用できるようにこの関数を隠蔽するpg_buffercacheビューを提供します。

デフォルトでは、使用はスーパーユーザとpg_monitorロールのメンバに限定されています。GRANTを使って他人にアクセス権を付与できます。

F.24.1. pg_buffercacheビュー

ビューによって公開されている列の定義を表 F.15に示します。

表F.15 pg_buffercacheの列

列 型 説明
bufferid integer 1からshared_buffersまでの範囲で示されるID
relfilenode oid (参照先 pg_class.relfilenode) リレーションのファイルノード番号
reltablespace oid (参照先 pg_tablespace.oid) リレーションのテーブル空間OID
reldatabase oid (参照先pg_database.oid) リレーションのデータベースOID
relforknumber smallint リレーション内のフォーク番号。include/common/relpath.h参照
relblocknumber bigint リレーション内のページ番号
isdirty boolean ダーティページかどうか
usagecount smallint Clock-sweepアクセスカウント
pinning_backends integer このバッファをピン留めしているバックエンドの数

共有キャッシュ内の各バッファに対して、1行が存在します。未使用のバッファは、bufferidを除き、すべてのフィールドがNULLになります。共有システムカタログは、OIDがゼロのデータベースに属するものとして表示されます。

キャッシュはすべてのデータベースで共有されているため、現在のデータベースに属さないリレーションのページも表示されます。これは、一部の行に対して一致するpg_classの結合行が存在しない、間違った結

合をしてしまう可能性すらあることを意味します。pg_classに対して結合しようとする場合、現在のデータベースのOIDまたは0と等しいreldatabaseを持つ行に限定して結合することをお勧めします。

ビューが表示するバッファ状態データのコピーのために、バッファマネージャのロックを取得しません。このため、pg_buffercacheビューへのアクセスは、通常のバッファ処理への影響がより小さくなりますが、バッファすべてに渡る矛盾のない結果を提供しません。しかしながら、各バッファの情報に自己矛盾がないことは保証されます。

F.24.2. サンプル出力

```
regression=# SELECT n.nspname, c.relname, count(*) AS buffers
             FROM pg_buffercache b JOIN pg_class c
             ON b.relfilenode = pg_relation_filenode(c.oid) AND
                b.reldatabase IN (0, (SELECT oid FROM pg_database
                                      WHERE datname = current_database()))
             JOIN pg_namespace n ON n.oid = c.relnamespace
             GROUP BY n.nspname, c.relname
             ORDER BY 3 DESC
             LIMIT 10;

  nspname   |        relname         | buffers
------------+------------------------+---------
 public     | delete_test_table      |     593
 public     | delete_test_table_pkey |     494
 pg_catalog | pg_attribute           |     472
 public     | quad_poly_tbl          |     353
 public     | tenk2                  |     349
 public     | tenk1                  |     349
 public     | gin_test_idx           |     306
 pg_catalog | pg_largeobject         |     206
 public     | gin_test_tbl           |     188
 public     | spgist_text_tbl        |     182
(10 rows)
```

F.24.3. 作者

Mark Kirkwood <markir@paradise.net.nz>

設計協力: Neil Conway <neilc@samurai.com>

デバッグのアドバイス: Tom Lane <tgl@sss.pgh.pa.us>

F.25. pgcrypto

pgcryptoモジュールはPostgreSQL用の暗号関数を提供します。

このモジュールは「trusted」と見なされます。つまり、現在のデータベースに対してCREATE権限を持つ非スーパーユーザがインストールできます。

F.25.1. 汎用ハッシュ関数

F.25.1.1. digest()

```
digest(data text, type text) returns bytea
digest(data bytea, type text) returns bytea
```

与えられたdataのバイナリハッシュを計算します。typeは使用するアルゴリズムです。標準アルゴリズムはmd5、sha1、sha224、sha256、sha384、およびsha512です。pgcryptoがOpenSSL付きで構築された場合、表 F.19で詳解する、より多くのアルゴリズムを利用することができます。

ダイジェストを16進数表記の文字列としたい場合は、結果に対してencode()を使用してください。以下に例を示します。

```
CREATE OR REPLACE FUNCTION sha1(bytea) returns text AS $$
    SELECT encode(digest($1, 'sha1'), 'hex')
$$ LANGUAGE SQL STRICT IMMUTABLE;
```

F.25.1.2. hmac()

```
hmac(data text, key text, type text) returns bytea
hmac(data bytea, key bytea, type text) returns bytea
```

keyをキーとしたdataのハッシュ化MACを計算します。typeはdigest()の場合と同じです。

digest()と似ていますが、ハッシュはキーを知っている場合にのみ再計算できます。これは、誰かがデータを変更し、同時に一致するようにハッシュを変更するという状況を防ぎます。

キーがハッシュブロックサイズより大きい場合、まずハッシュ化され、その結果をキーとして使用します。

F.25.2. パスワードハッシュ化関数

crypt()およびgen_salt()関数は特にパスワードのハッシュ化のために設計されたものです。crypt()がハッシュ処理を行い、gen_salt()はハッシュ処理用のアルゴリズム上のパラメータを準備します。

crypt()アルゴリズムは、以下の点で通常のMD5やSHA1のようなハッシュ処理アルゴリズムと異なります。

1. 低速です。データ量が少ないためパスワード総当たり攻撃に対して頑健にする唯一の方法です。
2. 結果にはソルトというランダムな値が含まれます。このため同じパスワードのユーザでも異なった暗号化パスワードを持ちます。これはアルゴリズムの逆処理に対する追加の防御です。
3. 結果内にアルゴリズムの種類が含まれます。このため異なるアルゴリズムでハッシュ化したパスワードが混在可能です。

4. 一部は適応型です。つまり、コンピュータが高速になったとしても、既存のパスワードとの互換性を損なうことなくアルゴリズムを低速に調整することができます。

crypt()関数がサポートするアルゴリズムを表 F.16に列挙します。

表F.16 crypt()がサポートするアルゴリズム

アルゴリズム	パスワード最大長	適応型かどうか	ソルトビット長	出力長	説明
bf	72	はい	128	60	Blowfishベース、2a版
md5	無制限	いいえ	48	34	MD5ベースの暗号
xdes	8	はい	24	20	拡張DES
des	8	いいえ	12	13	元来のUNIX crypt

F.25.2.1. crypt()

```
crypt(password text, salt text) returns text
```

passwordのcrypt(3)形式のハッシュを計算します。新しいパスワードを保管する時には、gen_salt()を使用して新しいsaltを生成する必要があります。パスワードを検査する時、既存のハッシュ値をsaltとして渡し、結果が格納された値と一致するかどうかを確認します。

新しいパスワードの設定例を以下に示します。

```
UPDATE ... SET pswhash = crypt('new password', gen_salt('md5'));
```

認証の例です。

```
SELECT (pswhash = crypt('entered password', pswhash)) AS pswmatch FROM ... ;
```

入力パスワードが正しければtrueを返します。

F.25.2.2. gen_salt()

```
gen_salt(type text [, iter_count integer ]) returns text
```

crypt()で使用するランダムなソルト文字列を新規に生成します。また、このソルト文字列はcrypt()にどのアルゴリズムを使用するかを通知します。

typeパラメータはハッシュ化アルゴリズムを指定します。受付可能な種類は、des、xdes、md5、bfです。

繰り返し回数を持つアルゴリズムでは、ユーザはiter_countパラメータを使用して繰り返し回数を指定できます。指定する回数を高くすれば、パスワードのハッシュ化にかかる時間が長くなり、それを破るための時間も長くなります。しかし、あまりに多くの回数を指定すると、ハッシュ計算にかかる時間は数年に渡ってしまう可能性があります。これは実用的ではありません。iter_countパラメータを省略した場合、デフォルトの繰

り返し回数が使用されます。iter_countで受け付けられる値はアルゴリズムに依存し、表 F.17に示す通りです。

表F.17 crypt()用の繰り返し回数

アルゴリズム	デフォルト	最小	最大
xdes	725	1	16777215
bf	6	4	31

xdesの場合は他にも、回数が奇数でなければならないという制限があります。

適切な繰り返し回数を選択するために、元々のDES暗号は当時のハードウェアで1秒あたり4個のハッシュを持つことができるように設計されたことを考えてください。毎秒4ハッシュより遅いと、おそらく使い勝手が悪いでしょう。毎秒100ハッシュより速いというのは、十中八九、あまりにも速すぎるでしょう。

ハッシュ化アルゴリズム別に相対的な速度に関する概要を表 F.18にまとめました。この表は、8文字のパスワード内のすべての文字の組合せを取るためにかかる時間を示します。また、すべて小文字の英字のみのパスワードである場合と大文字小文字が混在した英字と数字のパスワードの場合を仮定します。crypt-bfの項では、スラッシュの後の数値はgen_saltのiter_countです。

表F.18 ハッシュアルゴリズムの速度

アルゴリズム	1秒当たりのハッシュ数	[a-z]の場合	[A-Za-z0-9]の場合	md5ハッシュを単位とした持続期間
crypt-bf/8	1792	4年	3927年	100k
crypt-bf/7	3648	2年	1929年	50k
crypt-bf/6	7168	1年	982年	25k
crypt-bf/5	13504	188日	521年	12.5k
crypt-md5	171584	15日	41年	1k
crypt-des	23221568	157.5分	108日	7
sha1	37774272	90分	68日	4
md5(ハッシュ)	150085504	22.5分	17日	1

注意:

- Intel Mobile Core i3のマシンを使用しました。
- crypt-desおよびcrypt-md5アルゴリズムの数値はJohn the Ripper v1.6.38の-test出力から得たものです。
- md5ハッシュの数値はmdcrack 1.2のものです。
- sha1の数値はlcrack-20031130-betaのものです。
- crypt-bfの数は、1000個の8文字パスワードをループする単純なプログラムを使用して得たものです。こうして、異なる回数の速度を示すことができました。参考までに、john -testはcrypt-bf/5で13506 loops/secでした。(結果の差異が非常に小さいことは、pgcryptoにおけるcrypt-bf実装がJohn the Ripperで使用されるものと同じであるという事実と一致します。)

「すべての組み合わせを試行する」ことは現実的な行使ではありません。通常パスワード推定は、普通の単語とその変形の両方を含む辞書を使用して行われます。ですので、いささかなりとも言葉に似たパスワードは上で示した数値よりも速く推定されます。また6文字の単語に似ていないパスワードは推定を免れるかもしれませんし、免れないかもしれません。

F.25.3. PGP暗号化関数

ここで示す関数はOpenPGP（RFC 4880）標準の暗号処理部分を実装します。対称鍵および公開鍵暗号化がサポートされます。

暗号化されたPGPメッセージは次の2つの部品（または*パケット*）から構成されます。

- セッションキーを含むパケット。対称鍵または公開鍵で暗号化されています。
- セッションキーにより暗号化されたデータを含むパケット。

対称鍵（つまりパスワード）で暗号化する場合

1. 与えられたパスワードはString2Key(S2K)アルゴリズムでハッシュ化されます。これはどちらかというとcrypt()アルゴリズムと似て、意図的に低速で、かつランダムなソルトを使用します。しかし、全長のバイナリキーを生成します。
2. 分離したセッションキーが要求された場合、新しいランダムなキーが生成されます。さもなくば、S2Kキーがそのままセッションキーとして使用されます。
3. S2Kキーがそのまま使用される場合、S2K設定のみがセッションキーパケットに格納されます。さもなくば、セッションキーはS2Kキーで暗号化され、セッションキーパケットに格納されます。

公開鍵で暗号化する場合

1. 新しいランダムなセッションキーが生成されます。
2. これは公開鍵を使用して暗号化され、セッションキーパケットに格納されます。

どちらの場合でもデータ暗号化は以下のように処理されます。

1. 省略可能なデータ操作として、圧縮、UTF-8への変換、改行の変換があります。
2. データの前にはランダムなバイト数のブロックが付きます。これはrandom IVを使用する場合と同じです。
3. ランダムな前置ブロックとデータのSHA1ハッシュが後に付けられます。
4. これをすべてセッションキーで暗号化し、データパケットに格納します。

F.25.3.1. pgp_sym_encrypt()

```
pgp_sym_encrypt(data text, psw text [, options text ]) returns bytea
pgp_sym_encrypt_bytea(data bytea, psw text [, options text ]) returns bytea
```

対称PGPキー`psw`で`data`を暗号化します。`options`パラメータには後述のオプション設定を含めることができます。

F.25.3.2. pgp_sym_decrypt()

```
pgp_sym_decrypt(msg bytea, psw text [, options text ]) returns text
pgp_sym_decrypt_bytea(msg bytea, psw text [, options text ]) returns bytea
```

対称鍵で暗号化されたPGPメッセージを復号します。

pgp_sym_decryptでbytea型のデータを復号することはできません。これは無効な文字データの出力を防止するためです。元のテキストのデータをpgp_sym_decrypt_byteaで復号することが正しい方法です。

optionsパラメータには後述のオプション設定を含めることができます。

F.25.3.3. pgp_pub_encrypt()

```
pgp_pub_encrypt(data text, key bytea [, options text ]) returns bytea
pgp_pub_encrypt_bytea(data bytea, key bytea [, options text ]) returns bytea
```

公開PGPキーkeyでdataを暗号化します。この関数に秘密キーを与えるとエラーになります。

optionsパラメータには後述のオプション設定を含めることができます。

F.25.3.4. pgp_pub_decrypt()

```
pgp_pub_decrypt(msg bytea, key bytea [, psw text [, options text ]]) returns text
pgp_pub_decrypt_bytea(msg bytea, key bytea [, psw text [, options text ]]) returns bytea
```

公開鍵で暗号化されたメッセージを復号します。keyは、暗号化に使用した公開鍵に対応する秘密鍵でなければなりません。秘密鍵がパスワードで保護されている場合は、そのパスワードをpswで指定しなければなりません。パスワードはないが、オプションを指定したい場合は空のパスワードを指定する必要があります。

pgp_pub_decryptでbytea型のデータを復号することはできません。これは無効な文字データの出力を防止するためです。元のテキストのデータをpgp_pub_decrypt_byteaで復号することが正しい方法です。

optionsパラメータには後述のオプション設定を含めることができます。

F.25.3.5. pgp_key_id()

```
pgp_key_id(bytea) returns text
```

pgp_key_idはPGP公開鍵または秘密鍵のキーIDを取り出します。暗号化されたメッセージが指定された場合は、データの暗号化に使用されたキーIDを与えます。

2つの特殊なキーIDを返すことがあります。

- SYMKEY

  メッセージは対称鍵で暗号化されました。

- ANYKEY

  メッセージは公開鍵で暗号化されましたが、キーIDが消去されていました。つまり、どれで復号できるかを判定するためにはすべての秘密キーを試行しなければならないことを意味します。pgcrypto自身はこうしたメッセージを生成しません。

異なるキーが同一IDを持つ場合があることに注意してください。これは稀ですが、正常なイベントです。この場合クライアントアプリケーションはどちらが当てはまるかを調べるために、ANYKEYの場合と同様に、それぞれのキーで復号を試行しなければなりません。

F.25.3.6. armor(), dearmor()

```
armor(data bytea [ , keys text[], values text[] ]) returns text
dearmor(data text) returns bytea
```

PGPのASCIIアーマー形式にデータを隠す、または、データを取り出します。ASCIIアーマーは基本的にCRC付きのBASE64という形式で、追加のフォーマットがあります。

keysとvaluesの配列が指定された場合には、各キーと値の対に対してアーマーヘッダがアーマー形式に追加されます。どちらの配列も1次元で、同じ長さでなければなりません。keysとvaluesに非ASCII文字を含めることはできません。

F.25.3.7. pgp_armor_headers

```
pgp_armor_headers(data text, key out text, value out text) returns setof record
```

pgp_armor_headers()はdataからアーマーヘッダを取り出します。戻り値はキーと値の2つの列からなる行の集合です。もしキーや値に非アスキー文字が含まれていれば、UTF-8として扱われます。

F.25.3.8. PGP関数用のオプション

オプションはGnuPGに似せて命名しています。オプションの値は等号記号の後に指定しなければなりません。複数のオプションはカンマで区切ってください。以下に例を示します。

```
pgp_sym_encrypt(data, psw, 'compress-algo=1, cipher-algo=aes256')
```

convert-crlfを除くすべてのオプションは暗号化関数にのみ適用可能です。復号関数はPGPデータからこうしたパラメータを入手します。

もっとも興味深いオプションはおそらくcompress-algoとunicode-modeでしょう。残りはデフォルトで問題ないはずです。

F.25.3.8.1. cipher-algo

使用する暗号アルゴリズム。

値: bf, aes128, aes192, aes256 (OpenSSL-only: 3des, cast5)
デフォルト: aes128
適用範囲: pgp_sym_encrypt, pgp_pub_encrypt

F.25.3.8.2. compress-algo

使用する圧縮アルゴリズム。PostgreSQLがzlib付きで構築されている場合のみ利用可能です。

値:
0 - 非圧縮
1 - ZIP圧縮
2 - ZLIB圧縮 (ZIPにメタデータとブロックCRCを加えたもの)
デフォルト: 0
適用範囲: pgp_sym_encrypt, pgp_pub_encrypt

F.25.3.8.3. compress-level

どの程度圧縮するかです。レベルが大きい程小さくなりますが、低速になります。0は圧縮を無効にします。

値: 0, 1-9
デフォルト: 6
適用範囲: pgp_sym_encrypt, pgp_pub_encrypt

F.25.3.8.4. convert-crlf

暗号化の際に\nを\r\nに、復号の際に\r\n を\n に変換するかどうか。RFC 4880では、テキストデータは改行コードとして\r\n を使用して格納すべきであると規定されています。完全にRFC準拠の動作を行いたければ、これを使用してください。

値: 0, 1
デフォルト: 0
適用範囲: pgp_sym_encrypt, pgp_pub_encrypt, pgp_sym_decrypt, pgp_pub_decrypt

F.25.3.8.5. disable-mdc

データをSHA-1で保護しません。このオプションを使用することが良い唯一の理由は、SHA-1で保護されたパケットがRFC 4880に追加される前の、古いPGP製品との互換性を得るためです。最近のgnupg.orgおよびpgp.comのソフトウェアではこれを正しくサポートしています。

値: 0, 1
デフォルト: 0
適用範囲: pgp_sym_encrypt, pgp_pub_encrypt

F.25.3.8.6. sess-key

分離したセッションキーを使用します。公開鍵暗号では常に分離したセッションキーを使用します。このオプションは対称鍵暗号向けのもので、デフォルトではS2Kキーをそのまま使用します。

値: 0, 1
デフォルト: 0
適用範囲: pgp_sym_encrypt

F.25.3.8.7. s2k-mode

使用するS2Kアルゴリズム。

値:
 0 - ソルト無。危険です!
 1 - ソルト有。固定繰り返し回数。
 3 - 可変繰り返し回数。
デフォルト: 3
適用範囲: pgp_sym_encrypt

F.25.3.8.8. s2k-count

使用するS2Kアルゴリズムで使う繰り返しの回数。1024以上、65011712以下の値でなければなりません。

デフォルト: 65536から253952までの乱数値
適用範囲: s2k-mode=3と指定した時のpgp_sym_encrypt

F.25.3.8.9. s2k-digest-algo

S2K計算で使用するダイジェストアルゴリズム。

値: md5, sha1

デフォルト: sha1
適用範囲: pgp_sym_encrypt

F.25.3.8.10. s2k-cipher-algo

分離したセッションキーの暗号化に使用する暗号。

値: bf, aes, aes128, aes192, aes256
デフォルト: cipher-algoを使用
適用範囲: pgp_sym_encrypt

F.25.3.8.11. unicode-mode

テキストデータをデータベース内部符号化方式からUTF-8に変換して戻すかどうかです。データベースがすでにUTF-8であれば、変換は起こらず、データにUTF-8としてタグが付くのみです。このオプションがないと、何も行われません。

値: 0, 1
デフォルト: 0
適用範囲: pgp_sym_encrypt, pgp_pub_encrypt

F.25.3.9. GnuPGを使用したキーの生成

新しいキーを生成します。

```
gpg --gen-key
```

推奨するキー種類は「DSAとElgamal」です。

RSA暗号化のためには、マスタとしてDSAまたはRSAで署名のみのキーを作成し、そしてgpg --edit-keyを使用してRSAで暗号化された副キーを追加しなければなりません

キーを列挙します。

```
gpg --list-secret-keys
```

ASCIIアーマー形式で公開鍵をエクスポートします。

```
gpg -a --export KEYID > public.key
```

ASCIIアーマー形式の秘密鍵をエクスポートします。

```
gpg -a --export-secret-keys KEYID > secret.key
```

PGP関数にこれらのキーを渡す前にdearmor()を使用する必要があります。バイナリデータを扱うことができる場合、コマンドから-aフラグを省略することができます。

詳細はman gpg、The GNU Privacy Handbook[3]、https://www.gnupg.org/サイトの各種文書を参照してください。

F.25.3.10. PGPコードの制限

- 署名に関するサポートはありません。これはまた、暗号化副キーがマスタキーに属しているかどうか検査しないことを意味します。
- マスタキーとして暗号化キーをサポートしません。一般的にこうした状況は現実的ではありませんので、問題にならないはずです。
- 複数の副キーに関するサポートはありません。よくありますので、これは問題になりそうに見えます。一方、通常のGPG/PGPキーをpgcryptoで使用すべきではありません。使用する状況が多少異なりますので新しく作成してください。

F.25.4. 暗号化そのものを行う関数

これらの関数はデータ全体を暗号化するためだけに実行します。PGP暗号化の持つ先端的な機能はありません。したがって、大きな問題がいくつか存在します。

1. 暗号キーとしてユーザキーをそのまま使用します。
2. 暗号化されたデータが変更されたかどうかを確認するための整合性検査をまったく提供しません。
3. ユーザが、IVをも含め暗号化パラメータ自体をすべて管理していることを想定しています。
4. テキストは扱いません。

このため、PGP暗号化の導入もあり、暗号化のみの関数はあまり使用されません。

```
encrypt(data bytea, key bytea, type text) returns bytea
decrypt(data bytea, key bytea, type text) returns bytea

encrypt_iv(data bytea, key bytea, iv bytea, type text) returns bytea
decrypt_iv(data bytea, key bytea, iv bytea, type text) returns bytea
```

typeで指定した暗号方法を使用してデータの暗号化・復号を行います。type文字列の構文は以下の通りです。

```
algorithm [ - mode ] [ /pad: padding ]
```

[3] https://www.gnupg.org/gph/en/manual.html

ここでalgorithmは以下の1つです。

- bf — Blowfish
- aes — AES (Rijndael-128, -192 or -256)

またmodeは以下の1つです。

- cbc — 次のブロックは前ブロックに依存します（デフォルト）
- ecb — 各ブロックは独自に暗号化されます（試験用途のみ）

paddingは以下の1つです。

- pkcs — データ長に制限はありません（デフォルト）
- none — データは暗号ブロックサイズの倍数でなければなりません

このため、例えば以下は同じです。

```
encrypt(data, 'fooz', 'bf')
encrypt(data, 'fooz', 'bf-cbc/pad:pkcs')
```

encrypt_ivおよびdecrypt_ivでは、ivパラメータはCBCモード用の初期値となります。ECBでは無視されます。正確にブロック長でない場合、切り詰められるか、もしくはゼロで埋められます。このパラメータがない場合、関数のデフォルト値はすべてゼロです。

F.25.5. ランダムデータ関数

```
gen_random_bytes(count integer) returns bytea
```

暗号論的に強いランダムなcountバイトを返します。一度に最大で1024バイトを抽出することができます。ランダム性ジェネレータプールを空にすることを防止するためのものです。

```
gen_random_uuid() returns uuid
```

バージョン4(ランダムな)UUIDを返します。(廃れたものです。この関数は今ではコアのPostgreSQLにも含まれています。)

F.25.6. 注釈

F.25.6.1. 構築

pgcryptoは自身で主PostgreSQLのconfigureスクリプトの検出結果に従って構築します。構築に影響するオプションは--with-zlibと--with-openssl です。

ZLIB付きでコンパイルされた場合、PGP暗号化関数は暗号化前にデータを圧縮することができます。

OpenSSL付きでコンパイルされた場合、より多くのアルゴリズムが利用できるようになります。また、OpenSSLがより最適化されたBIGNUM関数を持つため、公開鍵暗号化関数は高速になります。

表F.19 OpenSSLの有無による機能のまとめ

機能	組み込み	OpenSSL付き
MD5	○	○
SHA1	○	○
SHA224/256/384/512	○	○
この他のダイジェストアルゴリズム	×	○(注1)
Blowfish	○	○
AES	○	○
DES/3DES/CAST5	×	○
暗号化そのもの	○	○
PGP対称暗号化	○	○
PGP公開キー暗号化	○	○

注

1. OpenSSLがサポートする任意のダイジェストアルゴリズムが自動的に選択されます。これは、明示的にサポートされなければならない暗号では使用できません。

F.25.6.2. NULLの扱い

標準SQLの通り、引数のいずれかがNULLの場合、すべての関数はNULLを返します。注意せずに使用すると、これがセキュリティ上の問題になるかもしれません。

F.25.6.3. セキュリティ上の制限

pgcryptoの関数はすべてデータベースサーバ内部で実行されます。これは、pgcryptoとクライアントアプリケーションとの間でやり取りされるデータはすべて平文であることを意味します。したがって、以下を行う必要があります。

1. ローカルまたはSSL接続で接続
2. システム管理者およびデータベース管理者を信頼

これらが不可能であれば、クライアントアプリケーション内で暗号化する方が望まれます。

実装はサイドチャネル攻撃[4]に耐えられません。例えば、pgcrypto復号関数が完了するのに掛かる時間は、一定の長さの暗号文に対して一様ではありません。

[4] https://en.wikipedia.org/wiki/Side-channel_attack

F.25.6.4. 有用な文書

- https://www.gnupg.org/gph/en/manual.html

  GNUプライバシーハンドブック

- https://www.openwall.com/crypt/

  blowfish暗号アルゴリズムの説明

- https://www.iusmentis.com/security/passphrasefaq/

  優れたパスワードの選び方

- http://world.std.com/~reinhold/diceware.html

  パスワード決定に関する面白い考え

- http://www.interhack.net/people/cmcurtin/snake-oil-faq.html

  良い暗号、悪い暗号に関する説明

F.25.6.5. 技術的な参考情報

- https://tools.ietf.org/html/rfc4880

  OpenPGPメッセージフォーマット

- https://tools.ietf.org/html/rfc1321

  MD5 メッセージダイジェストアルゴリズム

- https://tools.ietf.org/html/rfc2104

  HMAC: Keyed-Hashing for Message Authentication.

- https://www.usenix.org/legacy/events/usenix99/provos.html

  DES暗号、MD5暗号、bcryptアルゴリズムの比較

- https://en.wikipedia.org/wiki/Fortuna_(PRNG)

  Fortuna CSPRNGの説明

- https://jlcooke.ca/random/

  Linux用Jean-Luc Cooke Fortunaに基づく/dev/randomドライバ

F.25.7. 作者

Marko Kreen <markokr@gmail.com>

pgcryptoは以下のソースを使用しています。

アルゴリズム	作者	元ソース
DES crypt	David Burren他	FreeBSD libcrypt
MD5 crypt	Poul-Henning Kamp	FreeBSD libcrypt
Blowfish crypt	Solar Designer	www.openwall.com
Blowfish cipher	Simon Tatham	PuTTY
Rijndael cipher	Brian Gladman	OpenBSD sys/crypto
MD5ハッシュとSHA1	WIDE Project	KAME kame/sys/crypto
SHA256/384/512	Aaron D. Gifford	OpenBSD sys/crypto
BIGNUM math	Michael J. Fromberger	dartmouth.edu/~sting/sw/imath

F.26. pg_freespacemap

pg_freespacemapモジュールは、空き領域マップ(FSM)を検査する手法を提供します。pg_freespaceと呼ばれる関数、正確に言うと、二つの多重定義された関数を提供します。これらの関数は、指定されたページ、あるいはリレーションのすべてのページについての、空き領域マップに記録されている値を表示します。

デフォルトでは、使用はスーパーユーザとpg_stat_scan_tablesロールのメンバに限定されています。GRANTを使って他人にアクセス権を付与できます。

F.26.1. 関数

pg_freespace(rel regclass IN, blkno bigint IN) returns int2

FSMを参照して、blknoで指定されたリレーションのページ上の空き領域のサイズを返します。

pg_freespace(rel regclass IN, blkno OUT bigint, avail OUT int2)

FSMを参照して、リレーションの各ページの空き領域のサイズを表示します。リレーションの各ページに対して(blkno bigint, avail int2)が1タプルとなり、これらのタプルのセットが返却されます。

空き領域マップに格納された値は、正確ではありません。これらの値はBLCKSZの1/256(デフォルトBLCKSZでは32バイト)の精度で丸められ、また、タプルが挿入や更新されるのと同時に完全に最新に保たれているというわけではありません。

インデックスでは、ページ内の空き領域ではなく、完全に未使用のページが追跡されます。したがって、その値には意味がなく、単にページが一杯か空かを表します。

注記

注意: バージョン8.4で導入された新しいFSM実装を反映するために、同バージョンでインタフェースが変更されました。

F.26.2. サンプル出力

```
postgres=# SELECT * FROM pg_freespace('foo');
 blkno | avail
-------+-------
     0 |     0
     1 |     0
     2 |     0
     3 |    32
     4 |   704
     5 |   704
     6 |   704
     7 |  1216
     8 |   704
     9 |   704
    10 |   704
    11 |   704
    12 |   704
    13 |   704
    14 |   704
    15 |   704
    16 |   704
    17 |   704
    18 |   704
    19 |  3648
(20 rows)

postgres=# SELECT * FROM pg_freespace('foo', 7);
 pg_freespace
--------------
         1216
(1 row)
```

F.26.3. 作者

オリジナルバージョンは Mark Kirkwood <markir@paradise.net.nz>によるものです。バージョン8.4では、Heikki Linnakangas <heikki@enterprisedb.com>により、新しいFSM実装に合うよう書き直されました。

F.27. pg_prewarm

pg_prewarmはオペレーティングシステムのバッファキャッシュまたはPostgreSQLのバッファキャッシュいずれかにリレーションデータをロードするための便利な方法を提供します。プレウォームはpg_prewarm関数を使って手動で行うこともできますし、pg_prewarmをshared_preload_librariesに含めることで自動でも実行

できます。後者の場合には、システムは、定期的にautoprewarm.blocksという名前のファイルに共有バッファの内容を記録するバックグラウンドワーカーを実行し、再起動後には2つのバックグラウンドワーカー使って同じブロックを再ロードします。

F.27.1. 関数

```
pg_prewarm(regclass, mode text default 'buffer', fork text default 'main',
           first_block int8 default null,
           last_block int8 default null) RETURNS int8
```

第1引数はプレウォーム(あらかじめロード)したいリレーションを指定します。第2引数はプレウォームに使用する方法を指定します。詳細は以下でさらに説明します。第3引数はプレウォームされるリレーションフォークを指定します、これは通常mainです。第4引数は、プレウォームを開始するブロックを指定します(NULLはゼロとみなされます)。第5引数は終了ブロックを指定します(NULL はリレーションの最後のブロックまで指定したとみなされます)。返り値は、プレウォームされたブロック数です。

プレウォームに使用する方法としては以下の3種類が使用可能です。prefetchは、オペレーティングシステムに非同期のプレフェッチをリクエストします。もしオペレーティングシステムやビルド時にプレフェッチをサポートしていない場合はエラーとなります。readは、ブロックの要求された範囲を読み込みます。プレフェッチとは違って、すべてのプラットフォームにサポートするようにビルドされていますが、速度が遅くなります。bufferは、データベースのバッファキャッシュに要求された範囲を読み込みます。

これらの方法のいずれかでもキャッシュ出来るよりも多くのブロックをプレウォームしようとすると注意が必要です。prefetchやreadのようなOSのキャッシュを使用する場合、または PostgreSQLのbufferにキャッシュする場合、高い番号のブロックが読み込まれると低い番号のブロックが追い出されます。プレウォームは、キャッシュに対して特別な保護をしていないので、それが他のシステムにとって可能であるように、それらが読み込まれた直後に、新しいプレウォームによって追い出すことが可能です。逆に、プレウォームはキャッシュから他のデータを追い出すこともあります。これらの理由から、プレウォームはキャッシュが主に空のとき、一般的には起動時にもっとも有用です。

```
autoprewarm_start_worker() RETURNS void
```

自動プレウォームワーカーを起動します。通常これは自動的に行なわれますが、サーバーのスタート時に自動プレウォームが設定されておらず、後でワーカーを起動したい場合に有用です。

```
autoprewarm_dump_now() RETURNS int8
```

直ちにautoprewarm.blocksを更新します。これは、自動プレウォームワーカーが動いていなくて、次の再起動後に自動プレウォームワーカーを動かそうと考えているときに有用かもしれません。戻り値はautoprewarm.blocksに書き込まれたブロック数です。

F.27.2. 設定パラメータ

pg_prewarm.autoprewarm (boolean)

サーバーが自動プレウォームワーカーを起動すべきかどうかを制御します。デフォルトはonです。このパラメータはサーバ起動時のみ設定可能です。

pg_prewarm.autoprewarm_interval (int)

これはautoprewarm.blocksを更新する間隔です。デフォルトは300秒です。0に設定すると、このファイルは定常間隔では更新されず、サーバーが停止する時にだけ更新されます。

F.27.3. 作者

Robert Haas <rhaas@postgresql.org>

F.28. pgrowlocks

pgrowlocksモジュールは、指定したテーブルにおける行ロックの情報を示す関数を提供します。

デフォルトでは、使用は、スーパーユーザ、pg_stat_scan_tablesロールのメンバ、そのテーブルのSELECT権限を持つユーザに限定されています。

F.28.1. 概要

```
pgrowlocks(text) returns setof record
```

パラメータはテーブルの名前です。結果はレコードの集合となり、各レコードはテーブル内のロックされた1行を示します。出力列は表 F.20の通りです。

表F.20 pgrowlocksの出力列

名前	型	説明
locked_row	tid	ロックされた行のタプルID（TID）
locker	xid	ロックを獲得したトランザクションのトランザクションID、もしマルチトランザクションの場合はmultixact ID
multi	boolean	ロックをマルチトランザクションが獲得していた場合は真
xids	xid[]	ロックを獲得しているトランザクションのトランザクションID（マルチトランザクションの場合は複数）
modes	text[]	ロックを獲得しているトランザクションのロックモード（マルチトランザクションの場合は複数）。Key Share, Share、For No Key Update、No Key Update、For Update、Updateの配列。
pids	integer[]	ロックを獲得しているバックエンドのプロセスID（マルチトランザクションの場合は複数）

pgrowlocksは対象テーブルに対してAccessShareLockを獲得し、ロック情報の収集のために1行ずつ行を読み取ります。これは大規模テーブルにおいては高速とは言えません。以下に注意してください:

1. テーブル全体が他から排他ロックされている場合、pgrowlocksはブロックされます。

2. pgrowlocksでは、自己矛盾のないスナップショットを生成することは保証されません。その実行中に、新しい行ロックが獲得されることも、古いロックが解放されることもあり得ます。

pgrowlocksは、ロックされた行の内容は表示しません。同時に行の内容を参照したい場合には、以下のようにして実現することができます:

```
SELECT * FROM accounts AS a, pgrowlocks('accounts') AS p
  WHERE p.locked_row = a.ctid;
```

しかし、こうした問い合わせが非常に非効率であることに注意してください。

F.28.2. サンプル出力

```
=# SELECT * FROM pgrowlocks('t1');
 locked_row | locker | multi | xids  |     modes      |  pids
------------+--------+-------+-------+----------------+--------
 (0,1)      |    609 | f     | {609} | {"For Share"}  | {3161}
 (0,2)      |    609 | f     | {609} | {"For Share"}  | {3161}
 (0,3)      |    607 | f     | {607} | {"For Update"} | {3107}
 (0,4)      |    607 | f     | {607} | {"For Update"} | {3107}
(4 rows)
```

F.28.3. 作者

Tatsuo Ishii

F.29. pg_stat_statements

pg_stat_statementsモジュールは、サーバで実行されたすべてのSQL文のプラン生成時と実行時の統計情報を記録する手段を提供します。

このモジュールは追加の共有メモリを必要とするため、postgresql.confのshared_preload_librariesにpg_stat_statementsを追加してモジュールをロードしなければなりません。このことは、このモジュールを追加もしくは削除するには、サーバを再起動する必要があるということを意味しています。

pg_stat_statementsはロードされると、サーバのデータベース全体に渡って統計情報を記録します。この統計情報にアクセスしたり操作したりするために、このモジュールはビューpg_stat_statementsとユーティリティ関数pg_stat_statements_reset、pg_stat_statementsを提供します。これらは大域的に利用可能ではなく、CREATE EXTENSION pg_stat_statementsで特定のデータベースで可能になります。

F.29.1. pg_stat_statements ビュー

このモジュールにって収集された統計情報は、pg_stat_statementsというビューを通して利用することができます。このビューは、1行に対して、それぞれ個々のデータベースID、ユーザID、および問い合わせIDを含んでいます（モジュールが記録できるSQL文の最大数まで）。ビューの列は、表 F.21に示す通りです。

表F.21 pg_stat_statementsの列

列 型 説明
userid oid（参照先 pg_authid.oid） SQL文を実行したユーザのOID
dbid oid（参照先 pg_database.oid） SQL文が実行されたデータベースのOID
queryid bigint SQL文の解析木から計算された内部ハッシュコード
query text 代表的なSQL文の文字列
plans bigint SQL文がプラン生成された回数（pg_stat_statements.track_planningが有効な場合。無効であればゼロ）
total_plan_time double precision SQL文のプラン生成に費やした総時間（ミリ秒単位）（pg_stat_statements.track_planningが有効な場合。無効であればゼロ）
min_plan_time double precision SQL文のプラン生成に費やした最小時間（ミリ秒単位）（pg_stat_statements.track_planningが有効な場合。無効であればゼロ）
max_plan_time double precision SQL文のプラン生成に費やした最大時間（ミリ秒単位）（pg_stat_statements.track_planningが有効な場合。無効であればゼロ）
mean_plan_time double precision SQL文のプラン生成に費やした平均時間（ミリ秒単位）（pg_stat_statements.track_planningが有効な場合。無効であればゼロ）
stddev_plan_time double precision SQL文のプラン生成に費やした時間の母標準偏差（ミリ秒単位）（pg_stat_statements.track_planningが有効な場合。無効であればゼロ）
calls bigint SQL文が実行された回数
total_exec_time double precision SQL文の実行に費やした総時間（ミリ秒単位）
min_exec_time double precision SQL文の実行に費やした最小時間（ミリ秒単位）
max_exec_time double precision

列 型 説明
SQL文の実行に費やした最大時間（ミリ秒単位）
mean_exec_time double precision SQL文の実行に費やした平均時間（ミリ秒単位）
stddev_exec_time double precision SQL文の実行に費やした時間の母標準偏差（ミリ秒単位）
rows bigint SQL文によって取得された、あるいは影響を受けた行の総数
shared_blks_hit bigint SQL文によってヒットした共有ブロックキャッシュの総数
shared_blks_read bigint SQL文によって読み込まれた共有ブロックの総数
shared_blks_dirtied bigint SQL文によってダーティ状態となった共有ブロックの総数
shared_blks_written bigint SQL文によって書き込まれた共有ブロックの総数
local_blks_hit bigint SQL文によってヒットしたローカルブロックキャッシュの総数
local_blks_read bigint SQL文によって読み込まれたローカルブロックの総数
local_blks_dirtied bigint SQL文によってダーティ状態となったローカルブロックの総数
local_blks_written bigint SQL文によって書き込まれたローカルブロックの総数
temp_blks_read bigint SQL文によって読み込まれた一時ブロックの総数
temp_blks_written bigint SQL文によって書き込まれた一時ブロックの総数
blk_read_time double precision SQL文がブロック読み取りに費やした総時間（ミリ秒単位）（track_io_timingが有効な場合。無効であればゼロ）
blk_write_time double precision SQL文がブロック書き出しに費やした総時間（ミリ秒単位）（track_io_timingが有効な場合。無効であればゼロ）
wal_records bigint SQL文により生成されたWALレコードの総数
wal_fpi bigint SQL文により生成されたWALフルページイメージの総数
wal_bytes numeric SQL文により生成されたWALバイトの総量

セキュリティ上の理由から、スーパーユーザとpg_read_all_statsロールのメンバだけが、他のユーザによって実行されたSQLテキストや問い合わせのqueryidを見ることができます。ただし、ユーザのデータベースにビューがインストールされている場合、統計情報については他のユーザから見ることができます。

計画作成が可能な問い合わせ（つまりSELECT、INSERT、UPDATE、DELETE）は、内部のハッシュ計算に従った、同一の問い合わせ構造を持つ限り、1つのpg_stat_statements項目に組み合わせられます。典型的には、2つの問い合わせは、問い合わせの中に現れるリテラル定数の値以外、意味的に等価である場合、この目的では同一とみなされます。しかし、ユーティリティコマンド（つまりこの他のコマンドすべて）は問い合わせ文字列のテキストを基に厳密に比較されます。

他の問い合わせと合致させるために定数値が無視された場合、pg_stat_statementsの表示の中で定数は$1のようなパラメータ記号に置換されます。問い合わせの残りのテキストは、pg_stat_statements項目に関連付いた特定のqueryidハッシュ値を持つ、1つ目の問い合わせのテキストです。

一部の状況では、見た目上異なるテキストを持つ問い合わせが1つのpg_stat_statements項目にまとめられることがあります。通常これは意味的に等しい問い合わせでのみ発生しますが、関連がない問い合わせが1つの項目にまとめられるハッシュ競合の可能性がわずかながら存在します。（しかしこれは別のユーザまたは別のデータベースに属する問い合わせでは発生することはあり得ません。）

queryidハッシュ値は問い合わせの解析後の表現に対して計算されますので、search_pathの設定が異なる等の要因の結果として異なる意味を持つ場合、同じテキストを持つ問い合わせが別の項目として現れるという、反対もまたあり得ます。

pg_stat_statementsの消費者は、問い合わせテキストよりもより安定で信頼できる各項目への識別子として(おそらくdbidやuseridと組み合わせて)queryidを使いたいかもしれません。しかし、queryidハッシュ値の安定性には限定された保証しかないのを理解することは重要です。識別子は解析後の木から得られますので、その値は、とりわけ、この表現に現れる内部オブジェクト識別子の関数です。これは少々直観に反する結果です。例えば、pg_stat_statementsは見たところは同一の問い合わせを、それらが2つの問い合わせの実行の間に削除され再作成されたテーブルを参照しているのであれば、別個のものとみなします。ハッシュ処理はプラットフォームのマシンアーキテクチャやその他の面の違いにも敏感です。その上、PostgreSQLのメジャーバージョンをまたがってqueryidが安定であるとみなすのは安全ではありません。

経験上、queryid値は、基礎となるサーバのバージョンとカタログメタデータの詳細が全く同じである限り、安定していて比較可能とみなすことができます。物理WAL再生に基づくレプリケーションに参加する2つのサーバでは、同じ問い合わせに対して同一のqueryid値を持つことが期待できます。しかし、論理レプリケーションの仕組みは、レプリカが対応する詳細すべてで同一であることを約束しません。そのため、論理レプリカの集まりで増えるコストを識別するのにqueryidは有用ではありません。疑わしければ、直接テストすることを薦めます。

代表的な問い合わせテキストの定数を置き換えるのに使われたパラメータ記号は、元の問い合わせテキストの最も大きな$nパラメータの次の数字から、もしそれがなければ$1から始まります。ある場合には、この番号付けに影響する隠れたパラメータ記号があるかもしれないことに言及しておく価値はあります。例えば、PL/pgSQLは関数の局所変数の値を問い合わせに挿入するために隠れたパラメータ記号を使います。そのため、SELECT i + 1 INTO jのようなPL/pgSQL文はSELECT i + $2のような代表的なテキストになります。

代表的な問い合わせテキストは外部ディスクファイルに保持され、共有メモリを消費しません。そのため、非常に長い問い合わせテキストであっても保存に成功します。しかし、数多くの長い問い合わせテキストが蓄積されると、外部ファイルは手に負えないくらい大きくなるかもしれません。もしそのようなことが起きれば、回復手法として、pg_stat_statementsは問い合わせテキストを破棄することを選ぶで

しょう。その結果、各queryidに関連する統計は保存されるものの、pg_stat_statementsビュー内に存在するエントリはすべてqueryフィールドがヌルになります。もしこのようなことが起きたら、再発防止のためpg_stat_statements.maxを減らすことを検討してください。

プラン生成時と実行時の統計情報はそれぞれの終了フェーズで更新され、操作に成功した場合にのみ更新されるため、plansとcallsは常に一致するとは限りません。例えば、SQL文のプラン生成に成功しても実行フェーズの間に失敗した場合、そのプラン生成時の統計情報のみが更新されます。キャッシュされたプランが使用されプラン生成がスキップされた場合、実行時の統計情報のみが更新されます。

F.29.2. 関数

pg_stat_statements_reset(userid Oid, dbid Oid, queryid bigint) returns void

pg_stat_statements_resetは指定されたuserid、dbid、queryidに対応するpg_stat_statementsによってこれまでに収集したすべての統計情報を削除します。いずれかのパラメータを指定しないのであれば、デフォルト値0(無効)を使ってください。他のパラメータに一致する統計情報がリセットされます。どのパラメータも指定しない、または、すべての指定されたパラメータが0(無効)ならば、すべての統計情報を削除します。デフォルトでは、この関数はスーパーユーザのみ実行できます。GRANTを使ってアクセス権を他のユーザに付与できます。

pg_stat_statements(showtext boolean) returns setof record

pg_stat_statementsビューは同じくpg_stat_statementsという名前の関数の項で定義されています。クライアントがpg_stat_statements関数を直接呼び出し、showtext := falseと指定することで問い合わせテキストを省略することが可能です(すなわち、ビューのquery列に対応するOUT引数はNULLを返します)。この機能は不定長の問い合わせテキストを繰り返し取得するオーバヘッドを避けたいと考える外部のツールをサポートすること意図したものです。そのようなツールは代わりに、それがpg_stat_statements自身が行なっていることのすべてですので、各項目で最初に観察された問い合わせテキストをキャッシュし、必要とされる問い合わせテキストのみを取得できます。サーバは問い合わせテキストをファイルに格納しますので、この方法はpg_stat_statementsデータの繰り返しの検査に対する物理I/Oを減らすでしょう。

F.29.3. 設定パラメータ

pg_stat_statements.max (integer)

pg_stat_statements.maxは、このモジュールによって記録されるSQL文の最大数(すなわち、pg_stat_statementsビューにおける行の最大数)です。これを超えて異なるSQL文を検出した場合は、最も実行回数の低いSQL文の情報が捨てられます。デフォルトは5000です。このパラメータはサーバの起動時にのみ指定できます。

pg_stat_statements.track (enum)

pg_stat_statements.trackは、どのSQL文をモジュールによって計測するかを制御します。topを指定した場合は(直接クライアントによって発行された)最上層のSQL文を記録します。allは(関数の中から呼び出された文などの)入れ子になった文も記録します。noneは文に関する統計情報収集を無効にします。デフォルトはtopです。この設定はスーパーユーザだけが変更できます。

pg_stat_statements.track_utility (boolean)

pg_stat_statements.track_utilityは、このモジュールがユーティリティコマンドを記録するかどうかを指定します。ユーティリティコマンドとは、SELECT、INSERT、UPDATEおよびDELETE以外のすべてです。デフォルトはonです。この設定はスーパーユーザのみが変更できます。

pg_stat_statements.track_planning (boolean)

pg_stat_statements.track_planningは、このモジュールがプラン生成の操作と時間を記録するかどうかを指定します。このパラメータを有効にすると、特に多数の同時接続で実行されるクエリの種類が少ない場合にパフォーマンスが著しく低下する可能性があります。デフォルトはoffです。この設定はスーパーユーザのみが変更できます。

pg_stat_statements.save (boolean)

pg_stat_statements.saveは、サーバを終了させる際に文の統計情報を保存するかどうかを指定します。offの場合、統計情報は終了時に保存されず、サーバ開始時に再読み込みもされません。デフォルト値はonです。このパラメータはpostgresql.confファイル、またはサーバコマンドラインでのみ設定できます。

このモジュールは、pg_stat_statements.maxに比例する追加の共有メモリを必要とします。pg_stat_statements.trackにnoneが設定されていても、モジュールがロードされている限り常にこのメモリが消費されることに注意してください。

これらのパラメータはpostgresql.confの中で設定しなければなりません。典型的な使用方法は以下のようになります。

```
# postgresql.conf
shared_preload_libraries = 'pg_stat_statements'

pg_stat_statements.max = 10000
pg_stat_statements.track = all
```

F.29.4. サンプル出力

```
bench=# SELECT pg_stat_statements_reset();

$ pgbench -i bench
$ pgbench -c10 -t300 bench

bench=# \x
bench=# SELECT query, calls, total_exec_time, rows, 100.0 * shared_blks_hit /
               nullif(shared_blks_hit + shared_blks_read, 0) AS hit_percent
          FROM pg_stat_statements ORDER BY total_exec_time DESC LIMIT 5;
-[ RECORD 1 ]---+----------------------------------------------------------------------
query           | UPDATE pgbench_branches SET bbalance = bbalance + $1 WHERE bid = $2
calls           | 3000
total_exec_time | 25565.855387
rows            | 3000
```

```
hit_percent     | 100.0000000000000000
-[ RECORD 2 ]---+----------------------------------------------------------------
query           | UPDATE pgbench_tellers SET tbalance = tbalance + $1 WHERE tid = $2
calls           | 3000
total_exec_time | 20756.669379
rows            | 3000
hit_percent     | 100.0000000000000000
-[ RECORD 3 ]---+----------------------------------------------------------------
query           | copy pgbench_accounts from stdin
calls           | 1
total_exec_time | 291.865911
rows            | 100000
hit_percent     | 100.0000000000000000
-[ RECORD 4 ]---+----------------------------------------------------------------
query           | UPDATE pgbench_accounts SET abalance = abalance + $1 WHERE aid = $2
calls           | 3000
total_exec_time | 271.232977
rows            | 3000
hit_percent     | 98.8454011741682975
-[ RECORD 5 ]---+----------------------------------------------------------------
query           | alter table pgbench_accounts add primary key (aid)
calls           | 1
total_exec_time | 160.588563
rows            | 0
hit_percent     | 100.0000000000000000


bench=# SELECT pg_stat_statements_reset(0,0,s.queryid) FROM pg_stat_statements AS s
            WHERE s.query = 'UPDATE pgbench_branches SET bbalance = bbalance + $1 WHERE bid =
 $2';

bench=# SELECT query, calls, total_exec_time, rows, 100.0 * shared_blks_hit /
               nullif(shared_blks_hit + shared_blks_read, 0) AS hit_percent
          FROM pg_stat_statements ORDER BY total_exec_time DESC LIMIT 5;
-[ RECORD 1 ]---+----------------------------------------------------------------
query           | UPDATE pgbench_tellers SET tbalance = tbalance + $1 WHERE tid = $2
calls           | 3000
total_exec_time | 20756.669379
rows            | 3000
hit_percent     | 100.0000000000000000
-[ RECORD 2 ]---+----------------------------------------------------------------
query           | copy pgbench_accounts from stdin
calls           | 1
total_exec_time | 291.865911
rows            | 100000
hit_percent     | 100.0000000000000000
```

```
-[ RECORD 3 ]---+------------------------------------------------------------------
query           | UPDATE pgbench_accounts SET abalance = abalance + $1 WHERE aid = $2
calls           | 3000
total_exec_time | 271.232977
rows            | 3000
hit_percent     | 98.8454011741682975
-[ RECORD 4 ]---+------------------------------------------------------------------
query           | alter table pgbench_accounts add primary key (aid)
calls           | 1
total_exec_time | 160.588563
rows            | 0
hit_percent     | 100.0000000000000000
-[ RECORD 5 ]---+------------------------------------------------------------------
query           | vacuum analyze pgbench_accounts
calls           | 1
total_exec_time | 136.448116
rows            | 0
hit_percent     | 99.9201915403032721

bench=# SELECT pg_stat_statements_reset(0,0,0);

bench=# SELECT query, calls, total_exec_time, rows, 100.0 * shared_blks_hit /
               nullif(shared_blks_hit + shared_blks_read, 0) AS hit_percent
          FROM pg_stat_statements ORDER BY total_exec_time DESC LIMIT 5;
-[ RECORD 1 ]---+------------------------------------------------------------------------------
query           | SELECT pg_stat_statements_reset(0,0,0)
calls           | 1
total_exec_time | 0.189497
rows            | 1
hit_percent     |
-[ RECORD 2 ]---+------------------------------------------------------------------------------
query           | SELECT query, calls, total_exec_time, rows, $1 * shared_blks_hit /          +
                |                nullif(shared_blks_hit + shared_blks_read, $2) AS hit_percent+
                |           FROM pg_stat_statements ORDER BY total_exec_time DESC LIMIT $3
calls           | 0
total_exec_time | 0
rows            | 0
hit_percent     |
```

F.29.5. 作者

板垣 貴裕 <itagaki.takahiro@oss.ntt.co.jp>。Peter Geoghegan <peter@2ndquadrant.com>により問い合わせの正規化が追加されました。

F.30. pgstattuple

pgstattupleモジュールはタプルレベルの統計情報を入手するための各種関数を提供します。

これらの関数は詳細なページレベルの情報を返しますので、デフォルトではアクセスが制限されています。デフォルトではpg_stat_scan_tablesロールだけがEXECUTE権限を持っています。スーパーユーザは、当然、この制限を無視します。拡張がインストールされた後、ユーザはGRANTコマンドを発行して他のユーザがそれらを実行できるよう関数に対する権限を変更できます。しかしながら、その代わりにpg_stat_scan_tablesロールにそのユーザを追加する方が好ましいでしょう。

F.30.1. 関数

```
pgstattuple(regclass) returns record
```

pgstattupleはリレーションの物理的な長さ、「無効」なタプルの割合、およびその他の情報を返します。これはバキュームが必要かどうかユーザが判断する時に有用かもしれません。引数は対象とするリレーションの名前（スキーマ修飾可）もしくはOIDです。以下に例を示します。

```
test=> SELECT * FROM pgstattuple('pg_catalog.pg_proc');
-[ RECORD 1 ]------+-------
table_len          | 458752
tuple_count        | 1470
tuple_len          | 438896
tuple_percent      | 95.67
dead_tuple_count   | 11
dead_tuple_len     | 3157
dead_tuple_percent | 0.69
free_space         | 8932
free_percent       | 1.95
```

出力列を表 F.22で説明します。

表F.22 pgstattupleの出力列

列	型	説明
table_len	bigint	リレーションのバイト単位の物理長
tuple_count	bigint	有効なタプル数
tuple_len	bigint	有効なタプルの物理長（バイト単位）
tuple_percent	float8	有効タプルの割合
dead_tuple_count	bigint	無効なタプル数
dead_tuple_len	bigint	バイト単位の総無効タプル長
dead_tuple_percent	float8	無効タプルの割合
free_space	bigint	バイト単位の総空き領域

列	型	説明
free_percent	float8	空き領域の割合

> **注記**
>
> table_lenは、tuple_len、dead_tuple_len、free_spaceの合計よりも常に大きいです。違いの原因は、固定ページのオーバヘッド、ページ毎のタプルへのポインタのテーブル、タプルが正しく整列することを確実にするためのパディングです。

pgstattuple はリレーション上で読み取りロックのみを獲得します。ですので、結果はこの瞬間のスナップショットを考慮しません。つまり、同時実行の更新がその結果に影響を与えます。

pgstattupleは、HeapTupleSatisfiesDirtyが偽を返すかどうかで、タプルが「無効」かどうか判定します。

pgstattuple(text) returns record

TEXTで対象リレーションを指定する点を除き、これはpgstattuple(regclass)と同じです。この関数は今までのところ後方互換のために残されており、近い将来のリリースでは廃止予定になるでしょう。

pgstatindex(regclass) returns record

pgstatindexはB-treeインデックスに関する情報を示すレコードを返します。以下は例です。

```
test=> SELECT * FROM pgstatindex('pg_cast_oid_index');
-[ RECORD 1 ]------+------
version            | 2
tree_level         | 0
index_size         | 16384
root_block_no      | 1
internal_pages     | 0
leaf_pages         | 1
empty_pages        | 0
deleted_pages      | 0
avg_leaf_density   | 54.27
leaf_fragmentation | 0
```

出力列は以下の通りです。

列	型	説明
version	integer	B-treeバージョン番号
tree_level	integer	ルートページのツリーレベル
index_size	bigint	バイト単位のインデックスサイズ
root_block_no	bigint	ルートページの場所（存在しない場合はゼロ）

列	型	説明
internal_pages	bigint	「内部」(上位レベル)ページ数
leaf_pages	bigint	リーフページ数
empty_pages	bigint	空ページ数
deleted_pages	bigint	削除ページ数
avg_leaf_density	float8	リーフページの平均密度
leaf_fragmentation	float8	リーフページの断片化

報告されるindex_sizeは、通常、internal_pages + leaf_pages + empty_pages + deleted_pagesが占めるより1多いページに相当するでしょう。これは、index_sizeがインデックスメタページも含むためです。

pgstattuple同様、結果はページ毎に累積されます。この瞬間のインデックス全体のスナップショットが存在すると想定してはいけません。

pgstatindex(text) returns record

TEXTで対象インデックスを指定する点を除き、これはpgstatindex(regclass)と同じです。この関数は今までのところ後方互換のために残されており、近い将来のリリースでは廃止予定になるでしょう。

pgstatginindex(regclass) returns record

pgstatginindexは、GINインデックスに関する情報を示すレコードを返します。以下に例を示します。

```
test=> SELECT * FROM pgstatginindex('test_gin_index');
-[ RECORD 1 ]--+--
version        | 1
pending_pages  | 0
pending_tuples | 0
```

出力列は以下の通りです。

列	型	説明
version	integer	GINバージョン番号
pending_pages	integer	待機中リスト内のページ数
pending_tuples	bigint	待機中リスト内のタプル数

pgstathashindex(regclass) returns record

pgstathashindexは、HASHインデックスに関する情報を示すレコードを返します。以下に例を示します。

```
test=> select * from pgstathashindex('con_hash_index');
-[ RECORD 1 ]--+-----------------
version        | 4
bucket_pages   | 33081
overflow_pages | 0
```

```
bitmap_pages   | 1
unused_pages   | 32455
live_items     | 10204006
dead_items     | 0
free_percent   | 61.8005949100872
```

出力列は以下の通りです。

列	型	説明
version	integer	HASHバージョン番号
bucket_pages	bigint	バケットページの数
overflow_pages	bigint	オーバーフローページの数
bitmap_pages	bigint	ビットマップページの数
unused_pages	bigint	使われていないページの数
live_items	bigint	有効なタプルの数
dead_tuples	bigint	無効なタプルの数
free_percent	float	空き領域の割合

pg_relpages(regclass) returns bigint

pg_relpagesはリレーション内のページ数を返します。

pg_relpages(text) returns bigint

TEXTで対象リレーションを指定する点を除き、これはpg_relpages(regclass)と同じです。この関数は今までのところ後方互換のために残されており、近い将来のリリースでは廃止予定になるでしょう。

pgstattuple_approx(regclass) returns record

pgstattuple_approxはpgstattupleの代わりとなる高速なバージョンで、近似の結果を返します。引数は対象のリレーションの名前またはOIDです。以下に例を示します。

```
test=> SELECT * FROM pgstattuple_approx('pg_catalog.pg_proc'::regclass);
-[ RECORD 1 ]--------+-------
table_len            | 573440
scanned_percent      | 2
approx_tuple_count   | 2740
approx_tuple_len     | 561210
approx_tuple_percent | 97.87
dead_tuple_count     | 0
dead_tuple_len       | 0
dead_tuple_percent   | 0
approx_free_space    | 11996
approx_free_percent  | 2.09
```

出力列については表 F.23で説明します。

pgstattupleが常に全件走査を実行し、有効タプルと無効タプルの正確な数(およびそのサイズ)と空き領域を返すのに対し、pgstattuple_approxは全件走査を避けようとし、無効タプルの正確な統計情報および有効タプルと空き領域の数とサイズの近似値を返します。

可視性マップに従えば可視のタプルしかないページ(ページに対応するVMビットがセットされているなら、無効タプルが含まれていないとみなします)についてスキップすることで、これを実現します。そのようなページについて空き領域の値を空き領域マップから導き、ページ上の残りのスペースは有効タプルによって占められているとみなします。

スキップできないページについては、各タプルをスキャンし、その存在とサイズを適切なカウンターに記録し、ページ上の空き領域を加算します。最後に有効タプルの合計数をスキャンしたページとタプルの数に基づいて推定します(VACUUMがpg_class.reltuplesを推定するのと同じ方法です)。

表F.23 pgstattuple_approxの出力列

列	型	説明
table_len	bigint	リレーションの物理的なバイト長(正確)
scanned_percent	float8	スキャンしたテーブルの割合
approx_tuple_count	bigint	有効タプル数(推定)
approx_tuple_len	bigint	有効タプルの合計のバイト長(推定)
approx_tuple_percent	float8	有効タプルの割合
dead_tuple_count	bigint	無効タプル数(正確)
dead_tuple_len	bigint	無効タプルの合計のバイト長(正確)
dead_tuple_percent	float8	無効タプルの割合
approx_free_space	bigint	空き領域の合計バイト数(推定)
approx_free_percent	float8	空き領域の割合

上記の出力で、空き領域の数字はpgstattupleの出力と正確には一致しないかもしれません。これは空き領域マップは正確な数字を提供しますが、バイト単位で正確であることまでは保証されていないためです。

F.30.2. 作者

Tatsuo Ishii、Satoshi Nagayasu、Abhijit Menon-Sen

F.31. pg_trgm

pg_trgmモジュールは、類似文字列の高速検索をサポートするインデックス演算子クラスだけではなく、トライグラム一致に基づく英数字の類似度の決定に関する関数と演算子も提供します。

このモジュールは「trusted」と見なされます。つまり、現在のデータベースに対してCREATE権限を持つ非スーパーユーザがインストールできます。

F.31.1. トライグラム(またはトリグラフ)の概念

トライグラムは文字列から3つの連続する文字を取り出したグループです。共有するトライグラムの個数を数えることで、2つの文字列の類似度を測定することができます。この単純な考えが、多くの自然言語における単語の類似度を測定する際に非常に効率的であることが判明しています。

> **注記**
>
> pg_trgmは、文字列からトライグラムを抽出する時に単語以外の文字(英数字以外)を無視します。文字列内に含まれるトライグラム集合を決める際、文字列の前に2つの空白、後に1つの空白が付いているものとみなされます。例えば、「cat」という文字列のトライグラム集合は、「 c」、「 ca」、「cat」、「at 」です。「foo|bar」という文字列のトライグラム集合は、「 f」、「 fo」、「foo」、「oo 」、「 b」、「 ba」、「bar」、「ar 」です。

F.31.2. 関数と演算子

pg_trgmモジュールで提供される関数を表 F.24に、演算子を表 F.25に示します。

表F.24 pg_trgm Functions

関数 説明
similarity (text, text) → real 2つの引数がどの程度似ているかを示す数値を返します。結果はゼロ(2つの文字列がまったく類似していないことを示す)から1(2つの文字列が同一であることを示す)までの範囲です。
show_trgm (text) → text[] 与えられた文字列内のすべてのトライグラムからなる配列を返します。(実際これはデバッグ時を除いて役に立つことはほぼありません。)
word_similarity (text, text) → real 最初の引数文字列中のトライグラムの集合と、二番目の引数文字列中の順序付きトライグラム集合の中で最も類似度の高い連続した範囲の類似度を表す数字を返します。詳細は以下の説明をご覧ください。
strict_word_similarity (text, text) → real word_similarityと同様ですが、境界の範囲を単語の境界に一致させます。単語間にまたがるトライグラムは無いため、この関数は実際のところ最初の文字列と二番目の文字列の単語の任意の連続した範囲との間での最大類似度を返します。
show_limit () → real %演算子で使用される現在の類似度閾値を返します。これは、例えば2つの単語それぞれでミススペルがあったとしても類似しているものとみなす、その2つの単語間の最低の類似度を設定します。(廃止予定です。代わりにSHOW pg_trgm.similarity_thresholdを使ってください。)
set_limit (real) → real %演算子で使用される現在の類似度閾値を設定します。閾値は0から1までの値でなければなりません(デフォルトは0.3です)。渡された値と同じ値が返ります。(廃止予定です。; 代わりにSET pg_trgm.similarity_thresholdを使ってください。)

以下の例を考えます。

```
# SELECT word_similarity('word', 'two words');
 word_similarity
-----------------
             0.8
(1 row)
```

最初の文字列では、トライグラムの集合は{" w"," wo","wor","ord","rd "}です。二番目の文字列では、順序付きトライグラムの集合は{" t"," tw",two,"wo "," w"," wo","wor","ord","rds", "ds "}です。二番目の文字列中の順序付きトライグラムの集合の中で最も類似度の高い範囲は、{" w"," wo","wor","ord"}で、類似度は0.8となります。

この関数が返す値は、最初の文字列と、二番目の文字列の部分文字列の間の最大の類似度を表す値であると、概ね解釈できます。しかし、この関数は範囲の境界に対してパディングを行いません。ですから、一致しない語の境界を除くと、二番目の文字列中に存在する追加文字数は考慮されません。

一方で、strict_word_similarityは二番目の文字列の単語の範囲を選択します。上記の例でstrict_word_similarityは単語'words'の範囲を選択して、そのトライグラム集合は{" w"," wo","wor","ord","rds","ds "}になるでしょう。

```
# SELECT strict_word_similarity('word', 'two words'), similarity('word', 'words');
 strict_word_similarity | similarity
------------------------+------------
               0.571429 |   0.571429
(1 row)
```

このようにstrict_word_similarity関数は単語の並び全体の類似度を求めるのに有益で、対して、word_similarityは単語の並びの一部の類似度を求めるのにより適しています。

表F.25 pg_trgm Operators

演算子 説明
text % text → boolean 2つの引数がpg_trgm.similarity_thresholdで設定された類似度閾値以上の類似度を持つ場合trueを返します。
text <% text → boolean 最初の引数中のトライグラムの集合と、二番目の引数中の順序付きトライグラム集合の中の範囲の類似度が、pg_trgm.word_similarity_thresholdパラメータで設定した現在の語類似度の閾値よりも高い場合にtrueを返します。
text %> text → boolean <%演算子の交代演算子です。
text <<% text → boolean 二番目の引数が単語の境界と一致する順序付きトライグラム集合の連続した範囲を持っていて、かつ、最初の引数のトライグラム集合に対する類似度がpg_trgm.strict_word_similarity_thresholdパラメータで設定される現在の厳密な単語類似度の閾値より大きい場合、trueを返します。
text %>> text → boolean <<%演算子の交代演算子です。

演算子 説明
text <-> text → real 引数間の「距離」、この場合は1 - similarity()の値を返します。
text <<-> text → real 引数間の「距離」、この場合は1 - word_similarity()の値を返します。
text <->> text → real <<->演算子の交代演算子です。
text <<<-> text → real 引数間の「距離」、この場合は1 - strict_word_similarity()の値を返します。
text <->>> text → real <<<->演算子の交代演算子です。

F.31.3. GUCパラメータ

pg_trgm.similarity_threshold (real)

%演算子が使用する現在の類似度閾値を設定します。閾値は0から1の間でなければなりません。(デフォルトは0.3です)

pg_trgm.word_similarity_threshold (real)

<%と%>演算子が使用する現在の語類似度閾値を設定します。閾値は0から1の間でなければなりません。(デフォルトは0.6です)

pg_trgm.strict_word_similarity_threshold (real)

<<%と%>>演算子が使用する現在の厳密な語類似度閾値を設定します。閾値は0から1の間でなければなりません。(デフォルトは0.5です)

F.31.4. インデックスサポート

pg_trgmモジュールは、テキスト列全体に非常に高速な類似度検索を行うためのインデックスを作成することができるように、GiSTおよびGINインデックス演算子クラスを提供します。これらのインデックス種類は上記類似度演算子をサポートし、さらにLIKE、ILIKE、~および~*問い合わせにおけるトライグラムを基にしたインデックス検索をサポートします。(これらのインデックスは等価性や単純な比較演算子をサポートしません。ですので通常のB-treeインデックスも必要になるかもしれません。)

例:

```
CREATE TABLE test_trgm (t text);
CREATE INDEX trgm_idx ON test_trgm USING GIST (t gist_trgm_ops);
```

または

```
CREATE INDEX trgm_idx ON test_trgm USING GIN (t gin_trgm_ops);
```

gist_trgm_ops GiST演算子クラスはトライグラムの集合をビットマップ署名として近似します。オプションの整数パラメータsiglenは、署名の長さをバイト単位で決定します。デフォルトの署名の長さは12バイトです。署名の長さの有効な値は1から2024バイトまでです。長い署名では、インデックスはより大きくなってしまいますが、(インデックスのより小さな部分とより少ないヒープページを走査することで)検索がより正確になります。

署名の長さが32バイトのインデックスを作成する例

```
CREATE INDEX trgm_idx ON test_trgm USING GIST (t gist_trgm_ops(siglen=32));
```

この段階で、テキスト列tに類似度検索で使用可能なインデックスがあります。典型的な問い合わせを以下に示します。

```
SELECT t, similarity(t, 'word') AS sml
  FROM test_trgm
  WHERE t % 'word'
  ORDER BY sml DESC, t;
```

これは、wordに十分似たテキスト列の値をすべて、類似度の高い順番に返します。インデックスは非常に大規模なデータ群に対する高速操作を行うために使用されます。

以下は上の問い合わせを変形させたものです。

```
SELECT t, t <-> 'word' AS dist
  FROM test_trgm
  ORDER BY dist LIMIT 10;
```

これはGINインデックスではなくGiSTインデックスにより非常に効率的に実装することができます。通常、類似度が高いものの中から少ない個数のみを必要とする場合、1番目の式よりも効率的です。

また、単語の類似度あるいは厳密な単語の類似度に対してt列のインデックスを使うことができます。典型的な問い合わせは、

```
SELECT t, word_similarity('word', t) AS sml
  FROM test_trgm
  WHERE 'word' <% t
  ORDER BY sml DESC, t;
```

および

```
SELECT t, strict_word_similarity('word', t) AS sml
  FROM test_trgm
  WHERE 'word' <<% t
  ORDER BY sml DESC, t;
```

です。これは、wordのトライグラム集合に十分似ている順序付きトライグラム集合に対応する連続した領域が存在するテキスト列中のすべての値を返します。結果は、最も似ているものから、最も似ていないものへの順にソートされます。インデックスは、非常に大きなデータ集合に対しても高速な操作ができるようにするために使われます。

いくつか例を示します。

```
SELECT t, 'word' <<-> t AS dist
  FROM test_trgm
  ORDER BY dist LIMIT 10;
```

```
SELECT t, 'word' <<<-> t AS dist
  FROM test_trgm
  ORDER BY dist LIMIT 10;
```

これはGINインデックスではなく、GiSTインデックスによって極めて効率的に実装できます。

PostgreSQL 9.1から、これらのインデックス種類はLIKEおよびILIKEにおけるインデックス検索をサポートします。以下に例を示します。

```
SELECT * FROM test_trgm WHERE t LIKE '%foo%bar';
```

インデックス検索は、検索文字列からトライグラムを抽出し、それらをインデックスから検索することによって動作します。検索文字列内のトライグラムが多ければ、よりインデックス検索が効率的になります。B-treeを基にした検索とは異なり、検索文字列の左側が固定されている必要はありません。

PostgreSQL 9.3から、これらの種類のインデックスは正規表現一致(~および~*演算子)に対するインデックス検索もサポートします。以下に例を示します。

```
SELECT * FROM test_trgm WHERE t ~ '(foo|bar)';
```

インデックス検索は、正規表現からトライグラムを抽出し、それらをインデックスから検索することで動作します。より多くのトライグラムが正規表現から抽出される場合、インデックス検索はより効率的になります。B-treeを基にした検索と異なり、検索文字列は先頭一致である必要はありません。

LIKEおよび正規表現検索の両方で、トライグラムが抽出されないパターンでは完全インデックススキャンより性能が落ちることに注意してください。

GiSTまたはGINインデックスの選択は、他で説明されるGiSTとGINの相対的な性能特性に依存します。これについては、別途議論されています。

F.31.5. テキスト検索の統合

トライグラム一致は全文テキストインデックスと一緒に使用する時、非常に有用なツールです。特に、全文検索機構では直接一致しない、ミススペルがある入力単語の認識を行うために役に立ちます。

第一段階は文書内で一意な単語からなる補助テーブルを生成することです。

```
CREATE TABLE words AS SELECT word FROM
        ts_stat('SELECT to_tsvector(''simple'', bodytext) FROM documents');
```

ここでdocumentsは、検索対象のbodytextテキストフィールドを持つテーブルです。言語固有の設定を使用するのではなく、to_tsvector関数でsimple設定を使用する理由は、元の(語幹抽出されていない)単語のリストが欲しいためです。

次にword列にトライグラムインデックスを作成します。

```
CREATE INDEX words_idx ON words USING GIN (word gin_trgm_ops);
```

これで、上の例に似たSELECT問い合わせを使用して、ユーザの検索語内でミススペルのある単語を提示できるようになります。有用な追加された試験は、選択された単語の長さがミススペルのある単語の長さとほぼ同じになることを要求するものです。

> **注記**
>
> wordsテーブルは別に生成された静的なテーブルですので、文書群の更新に合理的に追従できるよう定期的に再生成する必要があります。正確に最新状態を維持する必要性は通常ありません。

F.31.6. 参考

GiST開発サイト http://www.sai.msu.su/~megera/postgres/gist/

Tsearch2開発サイト http://www.sai.msu.su/~megera/postgres/gist/tsearch/V2/

F.31.7. 作者

Oleg Bartunov <oleg@sai.msu.su>, Moscow, Moscow University, Russia

Teodor Sigaev <teodor@sigaev.ru>, Moscow, Delta-Soft Ltd.,Russia

Alexander Korotkov <a.korotkov@postgrespro.ru>, Moscow, Postgres Professional, Russia

文書作成: Christopher Kings-Lynne

本モジュールはロシアモスクワのDelta-Soft Ltd.による後援です。

F.32. pg_visibility

pg_visibilityモジュールは可視性マップ(Visibility Map, VM)およびテーブルのページレベルでの可視性情報を検査する手段を提供します。このモジュールはまた、可視性マップの整合性を検査し、強制的に再構築する機能も提供します。

ページレベルの可視性についての情報を格納するために、3つの異なるビットが使用されます。可視性マップの全可視ビットは、対応するリレーションのページの全タプルがすべての現在および将来のトランザクションに対して可視であることを示します。可視性マップの全凍結ビットは、そのページのすべてのタプルが凍結されていることを示します。これはすなわち、そのページに対してタプルの挿入、更新、削除、ロックなどが発生しない限り、将来もバキュームによる修正が必要ないことを意味します。ページヘッダのPD_ALL_VISIBLEビットは、可視性マップの全可視ビットと同じ意味ですが、別のデータ構造ではなく、データページ自体の中に格納されています。これら2つのビットは通常は同じ値になりますが、クラッシュリカバリの後は、ページの全可視ビットがセットされているのに、可視性マップの全可視ビットはクリアされているということもあります。また、pg_visibilityが可視性マップを検査した後、データページを検査する前に更新が行われたために、これらについて報告される値が一致しないということもあり得ます。データ破壊を起こすような何らかのイベントの後も、これらのビットが異なることがあり得ます。

PD_ALL_VISIBLEビットに関する情報を表示する関数は、可視性マップのみを参照する関数に比べるとずっと高価です。これは、可視性マップだけではなく、(それよりずっと大きな)リレーションのデータブロックを読む必要があるからです。リレーションのデータブロックを検査する関数は、同様に高価です。

F.32.1. 関数

pg_visibility_map(relation regclass, blkno bigint, all_visible OUT boolean, all_frozen OUT boolean) returns record

指定のリレーションの指定のブロックについて、可視性マップ内の全可視ビットと全凍結ビットを返します。

pg_visibility(relation regclass, blkno bigint, all_visible OUT boolean, all_frozen OUT boolean, pd_all_visible OUT boolean) returns record

指定のリレーションの指定のブロックについて、可視性マップ内の全可視ビットと全凍結ビット、およびそのブロックのPD_ALL_VISIBLEを返します。

pg_visibility_map(relation regclass, blkno OUT bigint, all_visible OUT boolean, all_frozen OUT boolean) returns setof record

指定のリレーションの各ブロックについて、全可視ビットと全凍結ビットを返します。

pg_visibility(relation regclass, blkno OUT bigint, all_visible OUT boolean, all_frozen OUT boolean, pd_all_visible OUT boolean) returns setof record

指定のリレーションの各ブロックについて、全可視ビットと全凍結ビット、および各ブロックのPD_ALL_VISIBLEビットを返します。

pg_visibility_map_summary(relation regclass, all_visible OUT bigint, all_frozen OUT bigint) returns record

可視性マップに従って、リレーション内の全可視ページの数と全凍結ページの数を返します。

pg_check_frozen(relation regclass, t_ctid OUT tid) returns setof tid

可視性マップ内で全凍結と印を付けられたページ内に格納されている非凍結タプルのTIDを返します。この関数が返すTIDの集合が空でないなら、可視性マップは壊れています。

pg_check_visible(relation regclass, t_ctid OUT tid) returns setof tid

可視性マップ内で全可視と印を付けられたページ内に格納されている全可視でないタプルのTIDを返します。この関数が返すTIDの集合が空でないなら、可視性マップは壊れています。

pg_truncate_visibility_map(relation regclass) returns void

指定のリレーションの可視性マップを切り詰めます。そのリレーションの可視性マップが壊れていると思われ、強制的に再構築したい場合にこの関数は有効です。この関数を実行した後に、指定のリレーション上で実行される最初のVACUUMにおいて、リレーション内の全ページがスキャンされ、可視性マップが再構築されます。(それが終わるまでは、可視性マップの中がすべてゼロになっているものとして問い合わせは動作します。)

デフォルトでは、これらの関数はスーパーユーザとpg_stat_scan_tablesのメンバのみが実行可能です。pg_truncate_visibility_map(relation regclass)は例外で、スーパーユーザのみが実行可能です。

F.32.2. 作者

Robert Haas <rhaas@postgresql.org>

F.33. postgres_fdw

postgres_fdwモジュールは、外部のPostgreSQLサーバに格納されたデータをアクセスするために使用する、postgres_fdw外部データラッパを提供します。

実質上、本モジュールの提供する機能は以前のdblinkモジュールが提供する機能と重複していますが、postgres_fdwはリモートのテーブルにアクセスするためにより透過的で標準に準拠した構文を利用できるほか、多くの場合においてより良い性能を得る事ができます。

postgres_fdwを使用したリモートアクセスを準備するには:

1. CREATE EXTENSIONを使用してpostgres_fdw拡張をインストールします。
2. CREATE SERVERを使用して、接続しようとする各リモートデータベースを定義する外部サーバオブジェクトを作成します。userおよびpasswordを除く接続パラメータを、外部サーバオブジェクトのオプションとして指定します。
3. CREATE USER MAPPINGを使用して、外部サーバへのアクセスを許可するデータベースユーザごとにユーザマッピングを作成します。ユーザマッピングのuserおよびpasswordオプションを使用してリモートユーザのためのユーザ名とパスワードを指定します。
4. CREATE FOREIGN TABLEもしくはIMPORT FOREIGN SCHEMAを使用して、アクセスしたいリモートテーブルごとに外部テーブルを作成します。外部テーブルのカラム定義は被参照側のリモートテーブルに一致していなければなりません。しかしながら、外部テーブルのオプションとして正しいリモートの名前を外部テーブルのオプションに指定すれば、テーブルおよびカラム名はリモートのものと異なった名前を付ける事ができます。

今のところ、リモートテーブルに格納されているデータにアクセスするには少なくとも外部テーブルに対するSELECT権限が必要です。また、INSERTやUPDATE、DELETEを使用してリモートテーブルを操作する事もでき

ます。(もちろん、ユーザマッピングで指定されたリモートユーザは、これらの操作を実行する権限を有している必要があります)

postgres_fdwは今のところ、ON CONFLICT DO UPDATE句のあるINSERT文をサポートしていないことに注意して下さい。しかし、一意インデックスの推定の指定を省略しているならば、ON CONFLICT DO NOTHING句はサポートされます。postgres_fdwは、パーティションテーブルで実行されたUPDATE文により引き起こされる行の移動をサポートしますが、移動した行を挿入するよう選択されたリモートパーティションが後で更新されるUPDATE対象のパーティションでもある場合は、今のところ扱わないことにも注意してください。

一般的な推奨事項として、可能であれば外部テーブルのカラムを、被参照側のリモートテーブル側のカラムと全く同一のデータ型および照合順序によって定義してください。postgres_fdwは必要に応じてデータ型の変換を行いますが、リモートサーバがローカルサーバとは少々違ったWHERE句の解釈を行うため、データ型や照合順序が一致していないと、時には予期しない結果を得る事があるかもしれません。

リモートテーブルより少ないカラム数で、あるいは異なった順序であっても外部テーブルを定義できる事に留意してください。リモートテーブル側のカラムとの対応付けは、その位置ではなく、名前によって行われます。

F.33.1. postgres_fdwの外部データラッパオプション

F.33.1.1. 接続オプション

postgres_fdw外部データラッパを使用する外部サーバは、以下に記す許されていないものや特別な取り扱いのものを除き、33.1.2に記載されているlibpqが接続文字列としてサポートするものと同一のオプションを使用する事ができます。

- user、passwordおよびsslpassword(これらは代わりにユーザマッピングのオプションの中で指定するか、サービスファイルを使用します)
- client_encoding(これはローカルサーバのエンコーディングが自動的にセットされます)
- fallback_application_name(自動的にpostgres_fdwとセットされます)
- sslkeyとsslcert - これは、接続とユーザマッピングのどちらか一方、または両方に現れます。両方に存在する場合、ユーザマッピングの設定が接続設定に優先します。

スーパーユーザのみがsslcertやsslkeyの設定のあるユーザマッピングを作成したり修正したりできます。

スーパーユーザのみが外部サーバに対してパスワードなしの認証で接続できます。したがって、非特権ユーザのユーザマッピングにはpasswordを必ず指定するようにして下さい。

ユーザマッピングオプションpassword_required 'false'を設定することで、スーパーユーザはこのユーザマッピング単位の検査を無効にできます。例えば以下の通りです。

```
ALTER USER MAPPING FOR some_non_superuser SERVER loopback_nopw
OPTIONS (ADD password_required 'false');
```

非特権ユーザが、postgresサーバが動作しているunixユーザの認証権限を悪用して、スーパーユーザ権限へ昇格するのを防ぐため、スーパーユーザのみがユーザーマッピングでこのオプションを設定できます。

これが、CVE-2007-3278やCVE-2007-6601のように、マップされたユーザがマップされたデータベースにスーパーユーザとして接続するのを許可しないことを確実にするよう注意が必要です。publicロールで

はpassword_required=falseを設定しないでください。postgresサーバが動作しているシステムユーザのunixのホームディレクトリにある、任意のクライアント証明書や.pgpass、.pg_service.confなどをマップされたユーザは潜在的には利用可能なことを心に留めておいてください。peerやident認証のような認証モードで付与された信頼関係を使うこともできます。

F.33.1.2. オブジェクト名オプション

これらのオプションによりリモートのPostgreSQLサーバに送出されるSQL文で使用される名前を制御する事ができます。外部テーブルがリモートテーブルとは異なった名前で定義されている場合、これらのオプションは必須です。

schema_name

外部テーブルに対して指定できるこのオプションは、リモートサーバ上のリモートテーブルのスキーマ名を与えます。省略された場合、外部テーブルのスキーマ名が使用されます。

table_name

外部テーブルに対して指定できるこのオプションは、リモートサーバ上のリモートテーブル名を与えます。省略された場合、外部テーブルのテーブル名が使用されます。

column_name

外部テーブルのカラムに対して指定できるこのオプションは、リモートサーバ上のカラム名を与えます。省略された場合、外部テーブルのカラム名が使用されます。

F.33.1.3. コスト推定オプション

postgres_fdwはリモートサーバに対するクエリを実行しリモートのデータを受信します。したがって、理想的には外部テーブルをスキャンする推定コストは、それをリモートサーバで実行するコストと通信オーバーヘッドの和となります。この推定を行うための最も信頼できる方法は、リモートサーバに問い合わせを行い、その結果にオーバーヘッド分を加算する事ですが、小さいクエリではコスト推定を得るための追加的な問い合わせに要するコストに見合わないかもしれません。そこで、どのようにコスト推定を行うかを制御するため、postgres_fdwは以下のようなオプションを提供します。

use_remote_estimate

外部テーブルまたは外部サーバに指定できるこのオプションは、コスト推定を得るためにpostgres_fdwがリモートのEXPLAINコマンドを発行するかどうかを制御します。外部テーブルに対する設定は、関連付けられた外部サーバに対する設定を上書きしますが、その効果は当該外部テーブルに限定されます。デフォルト値はfalseです。

fdw_startup_cost

外部テーブルまたは外部サーバに指定できるこのオプションは、当該外部サーバに関連付けられた全ての外部テーブルスキャンの推定開始コストに加算される数値です。これは、接続の確立、リモート側でのクエリのパース・最適化など、追加的なオーバーヘッドを表現します。デフォルト値は100です。

fdw_tuple_cost

外部サーバに指定できるこのオプションは、このサーバでの外部テーブルのスキャンにおいて、各タプル毎に発生する追加的なコストとして使用される数値です。これは、サーバ間のデータ転送における追加的なオーバーヘッドを表現し、リモートサーバへのネットワーク遅延の高低を反映するためにこの数値を増減することができます。デフォルト値は0.01です。

use_remote_estimateがtrueの時、postgres_fdwはリモートサーバから行数とコスト推定値を取得し、それをfdw_startup_costとfdw_tuple_costに加算します。一方、use_remote_estimateがfalseの時、postgres_fdwはローカルの行数とコスト推定値を取得しfdw_startup_costとfdw_tuple_costをコスト推定値に加算します。このローカルな推定は、リモートテーブルの統計情報のローカルコピーが利用可能でないと、正確である見込みはほとんどありません。ローカルな統計情報を更新するには外部テーブルに対するANALYZEを実行します。これはリモートテーブルに対するスキャンを実行し、あたかもローカルなテーブルであるかのように統計情報の計算と保存を行います。ローカルな統計情報を保存する事で、クエリの度にリモートテーブルの実行計画を作成するオーバヘッドを削減する事ができます。しかしながら、リモートテーブルの更新頻度が高ければローカルの統計情報はすぐに実態を反映しなくなるでしょう。

F.33.1.4. リモート実行オプション

デフォルトでは、組み込みの演算子および関数を使ったWHERE句のみがリモートサーバでの実行を考慮されます。組み込みでない関数を含む句は、行が取得された後、ローカルで検査されます。そのような関数がリモートサーバで利用でき、かつローカルで実行するのと同じ結果を生成すると信頼できるときは、そのようなWHERE句をリモートでの実行のために送出することでパフォーマンスを向上することができます。この動作は以下のオプションを使うことで制御できます。

extensions

このオプションは、PostgreSQLの拡張で、ローカルとリモートの両方に、互換のバージョンがインストールされているものの名前のリストです。IMMUTABLEで、列挙された拡張に属する関数と演算子は、リモートサーバに送出可能とみなされます。このオプションは外部サーバについてのみ指定可能で、テーブル毎の指定ではありません。

extensionsオプションを使用する場合、列挙する拡張が存在し、かつローカルとリモートのサーバで同一の動作をするようにすることはユーザの責任です。そうでない場合、リモートの問い合わせは失敗したり、期待と異なる動作をするかもしれません。

fetch_size

このオプションは、postgres_fdwが1回のフェッチの動作で何行のデータを取得するかを指定します。これは外部テーブルあるいは外部サーバに対して指定できます。テーブルに対して指定されたオプションは、サーバに対して指定されたオプションよりも優先します。デフォルトは100です。

F.33.1.5. 更新機能オプション

デフォルトではpostgres_fdwを使用する全ての外部テーブルは更新可能であると想定されます。以下のオプションにより、この挙動を上書きする事ができます。

updatable

このオプションは、postgres_fdwがINSERT、UPDATEあるいはDELETEコマンドを使用して外部テーブルを操作する事を許可するかどうかを規定します。外部テーブルで指定されたオプションは、外部サーバにおいて指定されたオプションを上書きします。デフォルト値はtrueです。

もちろん、リモートテーブルが実際には更新可能ではなかった場合、いずれにしてもエラーが発生するでしょう。このオプションを使用することで、リモートサーバへの問い合わせを行う事なくローカルでエラーを発生させることができます。また、information_schemaビューは、このオプションの値に従ってpostgres_fdw管理下の外部テーブルを更新可能(あるいは不可能)であるとレポートする事に留意してください。リモートサーバ側のチェックは一切行われません。

F.33.1.6. インポートのオプション

postgres_fdwはIMPORT FOREIGN SCHEMAを使って、外部テーブルの定義をインポートすることができます。このコマンドは、リモートのサーバ上に存在するテーブルあるいはビューとマッチする外部テーブルの定義をローカルサーバ上に作成します。インポートするリモートのテーブルにユーザ定義のデータ型の列がある場合、ローカルサーバにも同じ名前の互換性のある型がなければなりません。

インポートの動作は以下のオプションでカスタマイズできます(IMPORT FOREIGN SCHEMAコマンドで指定します)。

import_collate

このオプションは、列のCOLLATEオプションが、外部サーバからインポートする外部テーブルの定義に含まれているかどうかを制御します。デフォルトはtrueです。リモートサーバとローカルサーバで照合順序の名前の集合が異なる場合は、この設定を無効にする必要があるでしょう。リモートサーバが異なるOSで動作しているなら、そういうことがありそうです。

import_default

このオプションは、列のDEFAULT式が外部サーバからインポートされる外部テーブルの定義に含まれているかどうかを制御します。デフォルトはfalseです。このオプションを有効にする場合は、ローカルサーバとリモートサーバで異なる計算をされるデフォルトに注意して下さい。nextval()はよくある問題の一つです。インポートされるデフォルト式がローカルには存在しない関数または演算子を使っていた場合、IMPORTは失敗します。

import_not_null

このオプションは、列のNOT NULL制約が、外部サーバからインポートされる外部テーブルの定義に含まれているかどうかを制御します。デフォルトはtrueです。

NOT NULL以外の制約は決してリモートのテーブルからインポートされないことに注意して下さい。PostgreSQLは外部テーブルのCHECK制約をサポートしていますが、それを自動的にインポートする予定はありません。なぜなら、制約の式はローカルとリモートのサーバで異なる評価をされる危険があるからです。CHECK制約でそのような一貫しない動作があると、問い合わせの最適化で検知するのが難しい誤りが発生するかもしれません。そのため、CHECK制約をインポートしたい場合は、それを手作業で実行する必要があり、またその一つ一つの意味を注意深く確認するべきです。外部テーブルのCHECK制約の取扱いについて、詳しくはCREATE FOREIGN TABLEを参照して下さい。

他のテーブルのパーティションであるテーブルや外部テーブルは、自動的に除外されます。パーティション化されたテーブルは、他のテーブルのパーティションでない限り、インポートされます。パーティション化階層のルートであるパーティションテーブルを介してすべてのデータにアクセスできるため、この方法では余分なオブジェクトを作成せずにすべてのデータにアクセスできます。

F.33.2. 接続管理

postgres_fdwは、外部サーバに関連付けられた外部テーブルを参照するクエリを最初に実行する際に、外部サーバへの接続を確立します。この接続は保持され、同じセッションで以降の問い合わせのために再利用されます。しかし、外部サーバへのアクセスに対して複数のユーザ識別子（ユーザマッピング）が使用される場合には、接続はユーザマッピング毎に確立される事になります。

F.33.3. トランザクション管理

外部サーバ上のリモートテーブルを参照する際に、まだトランザクションが開始されていなければpostgres_fdwはリモートサーバ上でトランザクションを開始します。ローカルのトランザクションがコミット、あるいはアボートした時、リモートのトランザクションも同様にコミット、あるいはアボートします。セーブポイントも同様に管理され、リモート側に関連付けられたセーブポイントが作成されます。

ローカルトランザクションがSERIALIZABLE分離レベルを用いている時、リモートトランザクションもSERIALIZABLE分離レベルを使用します。それ以外の場合にはREPEATABLE READ分離レベルを使用します。これは、あるクエリが複数のテーブルスキャンをリモート側で行う際に、確実に全てのスキャンにおいて一貫したスナップショットで結果を取り出すためです。その結果、別の要求によってリモートサーバ側で競合する更新が発生したとしても、あるトランザクション内の問い合わせはリモートサーバからの一貫したデータを参照する事となります。ローカルのトランザクションがSERIALIZABLEあるいはREPEATABLE READ分離レベルを用いている場合、この動作は期待通りのものでしょう。一方、ローカルのトランザクションがREAD COMMITTED分離レベルを使用している場合には、予想外の動作かもしれません。将来のPostgreSQLリリースではこれらのルールに変更が加えられるかもしれません。

postgres_fdwは今のところ、二相コミットのためのリモートトランザクションの準備をサポートしていないことに注意して下さい。

F.33.4. リモート問い合わせの最適化

外部サーバからのデータ転送量を削減するため、postgres_fdwはリモート問い合わせを最適化しようと試みます。これは問い合わせのWHERE句をリモートサーバに送出する事、およびクエリで必要とされていないカラムを取得しない事により行われます。問い合わせの誤作動のリスクを下げるため、組み込みあるいは外部サーバのextensionsオプションに列挙されている拡張に属するデータ型、演算子、関数だけを用いたものでない限り、リモートサーバにWHERE句は送出されません。また、そのようなWHERE句で使われる演算子と関数はIMMUTABLEでなければなりません。UPDATEあるいはDELETEの問い合わせについては、リモートサーバに送出できないWHERE句がなく、問い合わせにローカルな結合がなく、対象のテーブルにローカルな行レベルのBEFOREあるいはAFTERトリガーや格納された生成列がなく、親ビューからのCHECK OPTION制約がないのであれば、postgres_fdwは問い合わせ全体をリモートサーバに送出することで、問い合わせの実行の最適化

を図ります。UPDATEでは、問い合わせの誤った実行のリスクを低減するため、対象列への代入式では組み込みのデータ型、IMMUTABLE演算子、IMMUTABLE関数のみを使わなければなりません。

同一の外部サーバ上の外部テーブルの間の結合がある場合、postgres_fdwはその結合全体を外部サーバに送出します。ただし、何らかの理由で各テーブルから個別に行を取得する方が効率的だと思われる場合、あるいは結合に含まれるテーブルの参照が異なるユーザマッピングに従う場合を除きます。JOIN句を送出するにあたり、WHERE句に関して上で説明したことと同じ注意が払われます。

リモートサーバでの実行のために実際に送出される問い合わせはEXPLAIN VERBOSEを用いて調べる事ができます。

F.33.5. リモート問い合わせ実行環境

postgres_fdwが開いたリモートセッションでは、search_pathパラメータはpg_catalogにだけ設定されますので、スキーマで修飾しなければ組み込みオブジェクトだけが可視です。postgres_fdw自身が生成した問い合わせでは、常にそのような修飾を行ないますので、これは問題になりません。しかし、リモートテーブルのトリガやルールによってリモートサーバ上で実行された関数にとっては問題の原因となり得ます。例えば、リモートテーブルが実際にはビューであれば、そのビューで使われている関数はすべて制限された検索パスで実行されるでしょう。期待される検索パス環境を確立できるよう、そのような関数では名前はすべてスキーマ修飾するか、そのような関数にSET search_pathオプション(CREATE FUNCTION参照)を付けることをお薦めします。

postgres_fdwは、同様に様々なパラメータでリモートセッション設定を確立します。

- TimeZoneはUTCに設定されます。
- DateStyleはISOに設定されます。
- IntervalStyleはpostgresに設定されます。
- extra_float_digitsはリモートサーバが9.0以上では3に設定され、それより古いバージョンでは2に設定されます。

これはsearch_pathほど問題にはならないでしょうが、もし必要になったら関数のSETオプションで処理してください。

上のパラメータのセッションレベルの設定を変更することで、この振舞いを覆すことはお薦めしません。postgres_fdwが正常に動作しない原因となるでしょう。

F.33.6. バージョン間互換性

postgres_fdwのリモートサーバにはPostgreSQL 8.3以降のバージョンを使用する事ができます。読み取り専用であれば、8.1以降のバージョンまで可能です。一方、postgres_fdwはIMMUTABLE属性を持った組み込みの演算子と関数が外部テーブルのWHERE句に含まれる場合、リモート側で実行しても安全であると仮定します。そのため、リモートサーバのリリース後に追加された関数が実行のために送出されるかもしれず、結果として「関数が見つかりません」あるいは類するエラーを発生させる事になります。この種の問題は問い合わせの書き換えによって対処する事ができます。例えば、最適化を妨げるため、外部テーブルへの参照をOFFSET 0を付けて副問い合わせに埋め込み、問題のある関数や演算子を副問い合わせの外に配置するなどの方法があります。

F.33.7. 例

これはpostgres_fdwで外部テーブルを作成する例です。まず、拡張をインストールします。

```
CREATE EXTENSION postgres_fdw;
```

次に、CREATE SERVERを使って外部サーバを作成します。この例では、ホスト192.83.123.89でポート5432を監視しているPostgreSQLサーバに接続します。接続されるデータベースはリモートサーバ上でforeign_dbという名前です。

```
CREATE SERVER foreign_server
        FOREIGN DATA WRAPPER postgres_fdw
        OPTIONS (host '192.83.123.89', port '5432', dbname 'foreign_db');
```

リモートサーバで使われるロールを特定するためにユーザマッピングも必要です。ユーザマッピングはCREATE USER MAPPINGで定義されます。

```
CREATE USER MAPPING FOR local_user
        SERVER foreign_server
        OPTIONS (user 'foreign_user', password 'password');
```

これでCREATE FOREIGN TABLEにより外部テーブルが作成できるようになりました。この例では、リモートサーバのsome_schema.some_tableという名前のテーブルにアクセスします。対応するローカルの名前はforeign_tableです。

```
CREATE FOREIGN TABLE foreign_table (
        id integer NOT NULL,
        data text
)
        SERVER foreign_server
        OPTIONS (schema_name 'some_schema', table_name 'some_table');
```

CREATE FOREIGN TABLEで宣言した列のデータ型やその他の属性は、実際のリモートテーブルと一致していることが必須です。リモートテーブルでどのような名前なのかを個々の列に対してcolumn_nameオプションで指定しない限り、列名も一致していなければなりません。多くの場合、外部テーブルの定義を手作業で作成するよりも、IMPORT FOREIGN SCHEMAを使用する方が望ましいです。

F.33.8. 作者

花田 茂 <shigeru.hanada@gmail.com>

F.34. seg

本モジュールは線分、浮動小数点区間を表現するsegデータ型を実装します。segは区間の終端内の不確定性を表すことができ、特に実験計測の表現に有用です。

このモジュールは「trusted」と見なされます。つまり、現在のデータベースに対してCREATE権限を持つ非スーパーユーザがインストールできます。

F.34.1. 原理

計測の幾何は数値の連続における点より通常より複雑です。計測は通常、多少あいまいな制限を持つ連続の部分となります。不確実性と不規則性のため、さらに、タンパク質を安定させる温度範囲など計測される値は本質的に何らかの状態を示す区間となる可能性があるため、計測は区間として現れます。

一般的な見方を使うと、こうしたデータは値の組合せではなく区間としてデータを格納する方が便利なようです。実際、ほとんどのアプリケーションでより効率的であると判明してさえいます。

一般的な見方をさらに進めると、制限の曖昧さは、伝統的な数値データ型を使用することで情報がある程度損失してしまうことを暗示しています。これを考えてみましょう。計測機器で6.50と読み取り、読み取ったデータをデータベースに格納します。それを取り出す時にどうなるでしょう。見てみましょう。

```
test=> select 6.50 :: float8 as "pH";
 pH
---
6.5
(1 row)
```

計測という世界では6.50は6.5と同じではありません。時としてこれが致命的な違いになる場合があります。実験者は信頼する桁を書き出し（公開し）ます。6.50は実際には、6.5というより大きくよりあいまいな区間に含まれるあいまいな区間です。2つに共通するものは（おそらく）その中央の値だけでしょう。私達は厳密にこうした異なるデータ項目が同じものとして現れることを好みません。

まとめ? 任意の可変精度を持つ区間の制限を記録できる特別なデータ型を持つことは素晴らしいことでしょう。各データ要素が独自の精度を記録するという意味での可変です。

以下を見てください。

```
test=> select '6.25 .. 6.50'::seg as "pH";
          pH
------------
6.25 .. 6.50
(1 row)
```

F.34.2. 構文

区間の外部表現は、1つまたは2つの浮動小数点値を範囲演算子（..または...）で結び付けた形になります。他にも、中央値と正負の偏差として指定することも可能です。省略可能な確実性指示子（<、>、~）を格納す

ることもできます。(しかし、確実性指示子はすべての組み込みの演算子で無視されます。) 表 F.26に許される表現についての概要を、表 F.27にいくつか例を示します。

表 F.26では、x、y、deltaは浮動小数点数値を表します。delta以外のxとyの前に確実性指示子を付与することができます。

表F.26 seg外部表現

x	単一値(幅0の区間)
x .. y	xからyまでの区間
x (+-) delta	x - deltaからx + deltaまでの区間
x ..	下限値xを持つ閉じていない区間
.. x	上限値xを持つ閉じていない区間

表F.27 有効なSEG入力の例

5.0	幅0のセグメントを作成します(こうすると点になります)。
~5.0	幅0のセグメントを作成し、データ内に~を記録します。~はseg型の演算では無視されますが、コメントとして保持されます。
<5.0	5.0という点を作成します。<は無視されますが、コメントとして保持されます。
>5.0	5.0という点を作成します。>は無視されますが、コメントとして保持されます。
5(+-)0.3	4.7 .. 5.3という区間を作成します。(+-)という記述は保持されないことに注意してください。
50 ..	50以上のすべて
.. 0	0以下のすべて
1.5e-2 .. 2E-2	0.015 .. 0.02という区間を作成します
1 ... 2	1...2、1 .. 2、1..2と同じです(範囲演算子前後の空白は無視されます)。

データソースで...演算子が広く使用されるため、..演算子の代わりの綴りとして許可されています。残念なことにこれにより解析上の曖昧性が生じました。0...23の上限が23なのか0.23なのかが明確ではありません。これは、segの入力において、少なくとも1つの桁を数値内の小数点の前に書くことを要求することで解決されます。

健全性検査としてsegは、5 .. 2のような、下限値が上限値より大きな区間を拒絶します。

F.34.3. 精度

内部的にseg値には32ビット浮動小数点数値の組合せが格納されます。これは7桁以上の有効桁を持つ数値が切り詰められることを意味します。

有効桁が正確に7桁、または7桁未満の数値は元の精度が保たれます。つまり、問い合わせが0.00を返す場合、後ろに続く0は書式付けのための見かけのものではないことが確実です。これは元のデータの精度を反映します。前にある0の数は精度には影響しません。0.0067は2有効桁のみを持つものと考えられます。

F.34.4. 使用方法

segモジュールにはseg値用のGiSTインデックス演算子クラスが含まれます。GiST演算子クラスでサポートされる演算子を表 F.28に示します。

表F.28 Seg GiST演算子

演算子 説明
seg << seg → boolean 1番目のsegが完全に2番目の左側に存在するか。b < cならば[a, b] << [c, d]は真です。
seg >> seg → boolean 1番目のsegが完全に2番目の右側に存在するか。a > dならば[a, b] >> [c, d]は真です。
seg &< seg → boolean 1番目のsegは2番目の右側にはみ出さないか。b <= dならば[a, b] &< [c, d]は真です。
seg &> seg → boolean 1番目のsegは2番目の左側にはみ出さないか。a >= cならば[a, b] &> [c, d]は真です。
seg = seg → boolean 2つのsegは等しいか。
seg && seg → boolean 2つのsegは重なるか。
seg @> seg → boolean 1番目のsegは2番目を包含するか。
seg <@ seg → boolean 1番目のsegは2番目に包含されるか。

(PostgreSQL 8.2以前では、包含演算子@>および<@はそれぞれ@および~という名前でした。以前の名前もまだ利用できますが、廃止予定であり、いずれなくなります。以前の名前は、コアの幾何データ型で以前従っていた規約と反対であることに注意してください。)

上記の演算子に加えて、seg型では表 9.1にある通常の比較演算子が利用可能です。これらの演算子はまず(a)と(c)を比べ、等しければ(b)と(d)を比べます。論理的にはほとんどの場合優れたソート処理と思えます。こうした型をORDER BYで使用したい場合に有用です。

F.34.5. 注釈

使用方法の例はリグレッションテストのsql/seg.sqlを参照してください。

(+-)を通常の範囲に変換する機構は、境界で有効な桁数を決定するという点で完全に正確ではありません。例えば以下のように、結果の区間に10の冪乗が含まれる場合、下限値に余計な桁を追加します。

```
postgres=> select '10(+-)1'::seg as seg;
      seg
---------
```

```
9.0 .. 11             -- 次のようになるべきです: 9 .. 11
```

R-treeインデックスの性能は入力値の初期の順序に大きく依存する可能性があります。seg列で入力テーブルをソートすることは非常に役に立つでしょう。例としてsort-segments.plスクリプトを参照してください。

F.34.6. クレジット

原作者：Gene Selkov, Jr. <selkovjr@mcs.anl.gov>, Mathematics and Computer Science Division, Argonne National Laboratory.

GiST (http://gist.cs.berkeley.edu/)の要旨（gist）を説明していただいたJoe Hellerstein博士（https://dsf.berkeley.edu/jmh/）に感謝します。また、自分の世界を作成できるようにし、静かに生活できるようにしてもらった、過去から現在までのすべてのPostgres開発者に感謝します。データベース研究を長年誠実にサポートしてくれたArgonne LabとU.S. Department of Energyにも感謝します。

F.35. sepgsql

sepgsqlは、SELinuxのセキュリティポリシーに基づいた、ラベルベースの強制アクセス制御（MAC; Mandatory Access Control）機能を提供するモジュールです。

> **警告**
>
> 現在の実装にはいくつかの重要な制限事項があり、そのため、全ての操作に対して強制アクセス制御を適用するわけではありません。詳細は F.35.7 をご覧ください。

F.35.1. 概要

このモジュールは、PostgreSQLが標準で提供しているものに加えて、SELinuxと統合されたアクセス制御のレイヤーを追加します。SELinuxの視点からは、このモジュールがPostgreSQLをユーザ空間オブジェクトマネージャとして機能することを可能にします。すなわち、DMLクエリによる個々のテーブルや関数へのアクセスは、オペレーティングシステムのセキュリティポリシーによってチェックされます。このチェックは、PostgreSQLによる通常のSQLパーミッションチェックに対して追加的に実施されます。

SELinuxにおけるアクセス制御の意思決定は、system_u:object_r:sepgsql_table_t:s0のような書式を持ったセキュリティラベルと呼ばれる文字列を用いて行われます。個々のアクセス制御の意思決定には、2種類のラベルが利用されます。すなわち、ある操作を行おうとする主体（サブジェクト）のラベルと、その操作の対象となるオブジェクトのラベルです。これらのラベルはあらゆる種類のオブジェクトに対して適用されるため、（このモジュールを用いる事で）データベースに格納されたオブジェクトに対するアクセス制御は、他の一般的なオブジェクト、例えばファイルに対するものと同一の基準に従って意思決定される事になります。このデザインは、情報資産を格納する方法とは独立に、一元管理されたセキュリティポリシーによって情報資産を保護することを意図しています。

SECURITY LABELを用いてデータベースオブジェクトにセキュリティラベルを設定することができます。

F.35.2. インストール

sepgsqlはSELinuxが有効なLinuxカーネル 2.6.28 以上で動作します。その他のプラットフォーム上では利用する事はできません。加えて、(ディストリビューションによっては、必要なルールを古いバージョンのポリシーにバックポートしている可能性がありますが)libselinux 2.1.10以上、selinux-policy 3.9.13以上が必要です。

sestatusコマンドを用いてSELinuxの状態を確認することができます。典型的な出力例は以下の通りです。

```
$ sestatus
SELinux status:                 enabled
SELinuxfs mount:                /selinux
Current mode:                   enforcing
Mode from config file:          enforcing
Policy version:                 24
Policy from config file:        targeted
```

SELinuxが無効化されている、あるいはインストールされていない場合、このモジュールのインストールの前に、SELinuxのセットアップを行わねばなりません。

このモジュールをビルドするには、PostgreSQLのconfigureコマンドに--with-selinuxオプションを追加してください。これは、ビルド時にlibselinux-develパッケージがインストールされている事を確認します。

このモジュールを利用するには、postgresql.confのshared_preload_librariesパラメータに sepgsqlを含める必要があります。これ以外の方法でロードされた場合、このモジュールは正しく機能しません。このモジュールのロード後、各データベースに対してsepgsql.sqlを実行し、セキュリティラベル管理のための関数のインストールや、初期セキュリティラベルの設定を行うべきです。

以下にsepgsql関数およびセキュリティラベルと共にデータベースクラスタを初期化する手順を示します。インストール先に応じて、適宜パス名を読み替えるようにしてください。

```
$ export PGDATA=/path/to/data/directory
$ initdb
$ vi $PGDATA/postgresql.conf

    #shared_preload_libraries = ''                # (change requires restart)

  を
    shared_preload_libraries = 'sepgsql'          # (change requires restart)
  に変更
$ for DBNAME in template0 template1 postgres; do
    postgres --single -F -c exit_on_error=true $DBNAME \
      </usr/local/pgsql/share/contrib/sepgsql.sql >/dev/null
  done
```

libselinuxとselinux-policyのバージョンによっては以下のようなメッセージが出力される事があります。

```
/etc/selinux/targeted/contexts/sepgsql_contexts:  line 33 has invalid object type db_blobs
/etc/selinux/targeted/contexts/sepgsql_contexts:  line 36 has invalid object type db_language
/etc/selinux/targeted/contexts/sepgsql_contexts:  line 37 has invalid object type db_language
/etc/selinux/targeted/contexts/sepgsql_contexts:  line 38 has invalid object type db_language
/etc/selinux/targeted/contexts/sepgsql_contexts:  line 39 has invalid object type db_language
/etc/selinux/targeted/contexts/sepgsql_contexts:  line 40 has invalid object type db_language
```

これらのメッセージは無害ですので無視してください。

インストール手順が正常に終了したら、通常通り、サーバを起動することができます。

F.35.3. リグレッションテスト

SELinuxの性質上、sepgsqlのリグレッションテストを実行するには、いくつかの追加的な設定が必要で、 そのうちの幾つかはrootで実行する必要があります。リグレッションテストは通常のmake checkやmake installcheckコマンドで実行する事はできません。必要な設定を行い、テスト用スクリプトを手動で実行する必要があります。このテストはPostgreSQLビルドツリーのcontrib/sepgsqlディレクトリで実行する必要があります。しかしビルドツリーを必要とする一方、このテストはインストールされたサーバに対して実行する必要があります。これは、make checkではなく、make installcheckによく似ています。

最初に、F.35.2に従ってsepgsqlをデータベースにセットアップします。使用するOS上のユーザは、認証無しでデータベース特権ユーザとして接続できる必要があることに留意してください。

次に、リグレッションテスト用のポリシーパッケージのビルドとインストールを行ってください。sepgsql-regtestポリシーはリグレッションテストの実行に必要な一連のルールを含む特別な目的のポリシーパッケージです。ポリシーのソースファイルであるsepgsql-regtest.teから、SELinuxの提供するMakefileを用いてmakeコマンドでビルドする事ができます。この時、インストール先システムにおいて、適切なMakefileの位置を指定する必要があります。以下の例で示されているパスは一例です。(このMakefileは通常selinux-policy-develやselinux-policyパッケージで提供されています。) ビルドが完了したら、semoduleを用いてこのポリシーパッケージをインストールする事ができます。このコマンドは、指定されたポリシーパッケージをリンクし、カーネル空間にロードする役割を果たします。インストールが正常終了したら、semodule -lにより有効なパッケージの一覧としてsepgsql-regtestが表示されるはずです。

```
$ cd .../contrib/sepgsql
$ make -f /usr/share/selinux/devel/Makefile
$ sudo semodule -u sepgsql-regtest.pp
$ sudo semodule -l | grep sepgsql
sepgsql-regtest 1.07
```

次に、sepgsql_regression_test_modeを有効化してください。安全のため、デフォルトではsepgsql-regtestに含まれる全てのルールが有効化されている訳ではありません。sepgsql_regression_test_modeパラメータはリグレッションテストを起動するために必要となる幾つかのルールを有効にします。setseboolコマンドによって有効化する事ができます。

```
$ sudo setsebool sepgsql_regression_test_mode on
```

```
$ getsebool sepgsql_regression_test_mode
sepgsql_regression_test_mode --> on
```

次に、シェルがunconfined_tドメインで動作している事を確認して下さい。

```
$ id -Z
unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
```

利用者の動作ドメインを設定する方法について、必要であれば、詳細はF.35.8をご覧ください。

最後に、リグレッションテストのスクリプトを実行します。

```
$ ./test_sepgsql
```

このスクリプトは全ての設定ステップが正常に行われていることを確認し、その後、sepgsqlモジュールに対するリグレッションテストを実行します。

テストの実行後はsepgsql_regression_test_modeパラメータを無効化する事をお勧めします。

```
$ sudo setsebool sepgsql_regression_test_mode off
```

sepgsql-regtestポリシーをアンロードする際は、以下のコマンドを実行してください。

```
$ sudo semodule -r sepgsql-regtest
```

F.35.4. GUCパラメータ

sepgsql.permissive (boolean)

このパラメータにより、オペレーティングシステムの設定に関わらず、sepgsqlをパーミッシブモードで動作させる事ができます。デフォルトの設定値はoffです。postgresql.conf内、およびサーバ起動時のコマンドラインでのみ、このパラメータを設定する事ができます。

このパラメータがonの場合、たとえSELinuxがエンフォーシングモードで動作していたとしても、sepgsql関数はパーミッシブモードで動作します。このパラメータは主としてテストの目的に有用です。

sepgsql.debug_audit (boolean)

このパラメータにより、セキュリティポリシーの設定とは無関係に監査ログを出力する事が可能になります。デフォルト値はoff（セキュリティポリシーの設定に従う）です。

SELinuxのセキュリティポリシーには、特定のアクセスを監査ログに記録するか否かを制御するルールも存在します。デフォルトでは、ポリシーに違反したアクセスを記録し、それ以外はログに記録されません。

システムのポリシーとは無関係に、このパラメータは全ての監査ログの出力を強制します。

F.35.5. 機能

F.35.5.1. 制御されるオブジェクトの種類

SELinuxのセキュリティモデルでは、全てのアクセス制御ルールは動作主体（サブジェクト; 典型的にはデータベースクライアント）と対象オブジェクト間の関係として記述し、これらはセキュリティラベルによって識別されます。ラベル付けされていないオブジェクトに対するアクセスが発生した場合、そのオブジェクトはあたかもunlabeled_tタイプが割り当てられているかのように振舞います。

現在のsepgsqlでは、スキーマ、テーブル、カラム、シーケンス、ビューおよび関数に対するラベル付けがサポートされています。sepgsqlの利用時には、これらのデータベースオブジェクトに対して、その生成時に自動的にセキュリティラベルが割り当てられます。このラベルはデフォルトセキュリティラベルと呼ばれ、作成者のラベル、親関係にあたるオブジェクトのラベル、そして場合によっては作成されたオブジェクトの名前に基づいて、システムのセキュリティポリシーが決定します。

新しいデータベースオブジェクトのラベルは、タイプ遷移と呼ばれる異なったラベルを設定するための特別なルールがセキュリティポリシーに設定されている場合を除き、親関係にあるオブジェクトのラベルを引き継ぎます。スキーマの親オブジェクトはデータベースであり、テーブル、シーケンス、ビュー、および関数はその属するスキーマが、カラムはその属するテーブルが親オブジェクトという事になります。

F.35.5.2. DMLパーミッション

テーブルに対しては、構文の種類に応じてdb_table:select、db_table:insert、db_table:updateあるいはdb_table:delete権限が全ての被参照テーブルに対してチェックされます。加えて、WHERE句やRETURNING句で参照されるカラム、又はUPDATEの際のデータ元として利用されるカラムの属するテーブルに対してdb_table:select権限もチェックされます。

参照された全てのカラムに対して列レベルの権限チェックが行われます。SELECT構文で読み出されるカラムに対してだけでなく、他のDML構文で参照されているカラムに対してもdb_column:select権限がチェックされます。また、UPDATEやINSERTによって操作の対象となっているカラムに対しても、db_column:updateやdb_column:insert権限がそれぞれチェックされます。

以下の例を見てください

```
UPDATE t1 SET x = 2, y = func1(y) WHERE z = 100;
```

ここでは、更新されるt1.xに対してdb_column:update権限が、更新と同時に参照されるt1.yに対してはdb_column:{select update}権限が、そして参照されるだけのt1.zにはdb_column:select権限がチェックされます。また、テーブルレベルではdb_table:{select update}権限がチェックされます。

SELECT構文を用いてシーケンスを参照する場合、db_sequence:get_valueがチェックされます。しかし、現在のところlastval()など関連する関数の実行時にはパーミッションチェックを行わない事に留意してください。

ビューに対してはdb_view:expand権限がチェックされ、次いで、ビューから展開されたオブジェクトに対するパーミッションが個別にチェックされます。

利用者がクエリの一部として、あるいは近道インタフェースを用いた呼び出しによって関数を実行しようとするとき、db_procedure:{execute}権限がチェックされます。関数がトラステッドプロシージャである場合、関数がトラステッドプロシージャのエントリーポイントとして振る舞う事ができるかどうかを検査するためにdb_procedure:{entrypoint}権限がチェックされます。

あらゆるスキーマオブジェクトにアクセスするためには、それらを含むスキーマに対するdb_schema:search権限が必要です。あるスキーマオブジェクトがスキーマ修飾無しに参照された場合、この権限を与えられていないスキーマは探索されません(あたかも利用者がスキーマに対するUSAGE権限を有していないかのように振る舞います)。明示的なスキーマ修飾があり、このスキーマに対して利用者が要求された権限を有していない場合、エラーが発生します。

クライアントは全ての被参照テーブル・カラムに対して参照の権限を有している必要があります。それらがビューに由来し、展開されたものであっても同様です。これにより、テーブルの内容がどのような方法によって参照されるかに関係なく、一貫したアクセス制御ルールを適用する事ができます。

データベーススーパーユーザに対して、標準のデータベース権限システムはDMLを用いたシステムカタログの更新と、TOASTテーブルの参照および更新を許していますが、sepgsqlが有効なとき、これらの操作は禁止されます。

F.35.5.3. DDLパーミッション

オブジェクトの作成、変更、削除やセキュリティラベルの変更など、SELinuxは各オブジェクトに共通の操作を制御するためのパーミッションを何個か定義しています。また、特定のスキーマに対する名前の追加や削除など、いくつかのオブジェクトにはそれ固有の操作を制御するための特別なパーミッションも定義されています。

新しいデータベースオブジェクトの作成にはcreate権限が必要です。SELinuxは利用者のセキュリティラベルと新しいオブジェクトに付与される事になるセキュリティラベルの対に基づいて、この権限を許可、あるいは拒否します。いくつかの場合では、追加的な権限チェックが行われます。

- CREATE DATABASEは、複製元またはテンプレートのデータベース に対するgetattr権限を要求します。
- スキーマオブジェクトの作成は、それを含むスキーマに対してadd_name権限を要求します。
- テーブルの作成は同時に、それがあたかも独立したオブジェクトであるかのように、個々のテーブル列を作成する権限を要求します。
- LEAKPROOF属性を持った関数の作成はinstall権限を要求します。(これはまた、既存の関数にLEAKPROOF属性を設定する時にも要求されます)

DROP構文の実行時、削除されるオブジェクトに対してdrop権限がチェックされます。CASCADEにより間接的に削除されるオブジェクトに対してもチェックは行われます。特定のスキーマに含まれるオブジェクト(テーブル、ビュー、シーケンスや関数)の削除に際しては、スキーマに対するremove_name権限も併せてチェックされます。

ALTER構文の実行時、テーブルに対するインデックスやトリガなど別オブジェクトに従属するもの(これらは代わりに親オブジェクトに対する権限がチェックされる)を除き、setattrがチェックされます。いくつかの場合では、追加的な権限チェックが行われます。

- オブジェクトを新しいスキーマに移動させるには、古いスキーマに対してremove_name権限が、新しいスキーマに対してadd_name権限が必要です。
- 関数に対するLEAKPROOF属性の設定はinstall権限を要求します。
- SECURITY LABELコマンドの実行時、ラベル付けされるオブジェクトの古いラベルに対してsetattr権限とrelabelfrom権限が、入力された新しいラベルに対してrelabelto権限がチェックされます。(複数のラベルプロバイダがインストールされており、利用者がSELinuxの管理下にないラベルを設定しようとした場合、setattr権限だけがチェックされるべきです。しかし実装上の制約により、現在はこれをチェックしていません。)

F.35.5.4. トラステッドプロシージャ

トラステッドプロシージャはSECURITY DEFINER関数やSet-UIDコマンドに似ています。通常、機密データに対する高度にコントロールされたアクセス手段(例えば行を削除したり、保存された値の精度をさげたりします)を提供する目的で、SELinuxは利用者のものとは異なるセキュリティラベルで信頼済みのコードを実行するための機能を持っています。関数がトラステッドプロシージャとして振舞うかどうかは、関数のセキュリティラベルおよびオペレーティングシステムのセキュリティポリシーに従って決まります。例えば:

```
postgres=# CREATE TABLE customer (
               cid     int primary key,
               cname   text,
               credit  text
           );
CREATE TABLE
postgres=# SECURITY LABEL ON COLUMN customer.credit
               IS 'system_u:object_r:sepgsql_secret_table_t:s0';
SECURITY LABEL
postgres=# CREATE FUNCTION show_credit(int) RETURNS text
             AS 'SELECT regexp_replace(credit, ''-[0-9]+$'', ''-xxxx'', ''g'')
                         FROM customer WHERE cid = $1'
           LANGUAGE sql;
CREATE FUNCTION
postgres=# SECURITY LABEL ON FUNCTION show_credit(int)
               IS 'system_u:object_r:sepgsql_trusted_proc_exec_t:s0';
SECURITY LABEL
```

これらの操作は管理権限を持つ利用者で行ってください。

```
postgres=# SELECT * FROM customer;
ERROR:  SELinux: security policy violation
postgres=# SELECT cid, cname, show_credit(cid) FROM customer;
 cid | cname  |     show_credit
-----+--------+---------------------
   1 | taro   | 1111-2222-3333-xxxx
   2 | hanako | 5555-6666-7777-xxxx
```

```
(2 rows)
```

この場合、一般の利用者はcustomer.creditを直接参照することはできませんが、トラステッドプロシージャであるshow_creditを用いる事で、一部の桁がマスクされた顧客のクレジットカード番号をプリントする事が可能になります。

F.35.5.5. 動的ドメイン遷移

セキュリティポリシーによって許可されている場合、SELinuxの動的ドメイン遷移機能を用いて、利用者のラベルを新しいものに切り替える事ができます。利用者のドメインはsetcurrent権限および、古いドメインから新しいドメインに遷移するためのdyntransition権限を有している必要があります。

利用者に自身のラベル、すなわち権限を切り替える事を可能にする動的ドメイン遷移は、システムによる自動切り替え(トラステッドプロシージャの場合)に比べて慎重に取り扱う必要があります。dyntransition権限は元々のドメインよりも小さな権限セットを持つドメインに切り替える場合にのみ安全であると考えられます。例えば、

```
regression=# select sepgsql_getcon();
                    sepgsql_getcon
-----------------------------------------------------
 unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
(1 row)

regression=# SELECT sepgsql_setcon('unconfined_u:unconfined_r:unconfined_t:s0-s0:c1.c4');
 sepgsql_setcon
----------------
 t
(1 row)

regression=# SELECT sepgsql_setcon('unconfined_u:unconfined_r:unconfined_t:s0-s0:c1.c1023');
ERROR:  SELinux: security policy violation
```

上記の例では、広いMCSレンジc1.c1023から狭いMCSレンジc1.c4への遷移は許可されているものの、その逆は禁止されています。

動的ドメイン遷移とトラステッドプロシージャの組み合わせは、典型的なコネクションプーラのライフサイクルに適合する興味深い利用法を提供します。たとえコネクションプーリングソフトウェアが大半のSQLの実行を許可されていない場合でも、トラステッドプロシージャの内側からsepgsql_setcon()関数を用いて利用者のセキュリティラベルを切り替える事ができます。(トラステッドプロシージャは利用者のセキュリティラベルを切り替えるための認証情報を要求すべきです。) この後、現在のセッションはコネクションプーラの権限ではなく、対象となる利用者の権限で動作する事になります。また、sepgsql_setcon()にNULL引数を与えて(適切な権限チェックを行う)トラステッドプロシージャから再び呼び出す事で、コネクションプーラは後でセキュリティラベルを元に戻す事ができます。ここでのポイントは、トラステッドプロシージャだけが実際に有効なセキュリティラベルを変更する権限を持っており、正しい認証情報が与えられた場合にのみそれを実行するという事です。言うまでもなく、安全な操作のためには、権限のないアクセスから認証情報を保持するテーブルや関数定義などを保護しなければなりません。

F.35.5.6. その他

ロードされたモジュールは、セキュリティポリシーの適用を容易にバイパスできるため、LOADコマンドの実行は全面的に禁止されています。

F.35.6. sepgsql関数

表 F.29に利用可能な関数を示します。

表F.29 sepgsql関数

関数 説明
sepgsql_getcon () → text 利用者のドメイン、つまり、現在の利用者ラベルを返却します。
sepgsql_setcon (text) → boolean セキュリティポリシーで許可されている場合、現在のセッションの利用者ドメインを新しいドメインへと切り替えます。NULLを引数に取る事も可能で、その場合、元々の利用者ドメインへの遷移を意味します。
sepgsql_mcstrans_in (text) → text mcstransデーモンが動作している場合、入力されたMLS/MCSレンジを修飾フォーマットから直接フォーマットに変換します。
sepgsql_mcstrans_out (text) → text mcstransデーモンが動作している場合、入力されたMLS/MCSレンジを直接フォーマットから修飾フォーマットに変換します。
sepgsql_restorecon (text) → boolean 現在のデータベース内のすべてのオブジェクトに対して初期セキュリティラベルを割り当てます。引数はNULLもしくはシステムの標準に代わる定義ファイルの名前です。

F.35.7. 制限事項

Data Definition Language (DDL) パーミッション

実装上の制約により、いくつかのDDL操作はパーミッションをチェックしません。

Data Control Language (DCL) パーミッション

実装上の制約により、DCL操作はパーミッションをチェックしません。

行レベルアクセス制御

PostgreSQLは行レベルアクセス制御をサポートしていますが、sepgsqlはサポートしていません。

隠れチャネル

たとえ利用者が参照を許可されていないオブジェクトであっても、sepgsqlはその存在を隠すことを意図していません。例えば、我々があるオブジェクトの内容を参照する事ができなくても、主キーの競合や外部キー違反、その他の方法によって不可視なオブジェクトが存在する事を推測できます。"最高機密"テーブルの存在を隠すことは不可能であり、その内容を秘匿することだけを意図しています。

F.35.8. 外部リソース

SE-PostgreSQL Introduction[5]

このwikiページでは、概要、セキュリティ・デザイン、アーキテクチャ、管理、および将来の機能について紹介しています。

SELinux User's and Administrator's Guide[6]

このドキュメントはSELinuxシステム管理に対する広範な知識を提供しています。主としてRed Hatオペレーティングシステムを対象としていますが、それに限ったものではありません。

Fedora SELinux FAQ[7]

このドキュメントはSELinuxに関するよくある質問と回答(FAQ)です。主としてFedoraを対象としていますが、それに限ったものではありません。

F.35.9. 作者

KaiGai Kohei <kaigai@ak.jp.nec.com>

F.36. spi

spiモジュールは、サーバプログラミングインタフェース(SPI)およびトリガを使用した、動作可能な例を複数提供します。これらの関数は独自の何らかの価値を持つものですが、目的に応じて変更するための例としてより有用です。関数は任意のテーブルと使用できるほど一般的なものですが、トリガを作成する場合は(後述のように)テーブル名とフィールド名を指定する必要があります。

以下で説明する関数グループのそれぞれは、別々にインストールすることができる拡張として提供されます。

F.36.1. refint — 参照整合性を実装する関数

check_primary_key()およびcheck_foreign_key()は、外部キー制約を検査するために使用されます。(当然ながら、この機能はかなり前に組み込みの外部キー機能に取って代わりました。しかし例としてはまだ有用です。)

check_primary_key()は参照テーブルを検査します。使用方法は、この関数を使用するBEFORE INSERT OR UPDATEトリガを他のテーブルを参照するテーブルに作成することです。トリガ引数は、外部キーを形成する参照テーブルの列名、被参照テーブル名、プライマリ/一意キーを形成する被参照テーブルの列名です。複数の外部キーを扱うためには、各参照に対してトリガを作成してください。

check_foreign_key()は被参照テーブルを検査します。使用方法は、この関数を使用するBEFORE DELETE OR UPDATEトリガを他のテーブルで参照されるテーブルに作成することです。トリガ引数は、この関数が検査

[5] https://wiki.postgresql.org/wiki/SEPostgreSQL

[6] https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/SELinux_Users_and_Administrators_Guide/

[7] https://fedoraproject.org/wiki/SELinux_FAQ

を実行しなければならない参照テーブル数、参照キーが見つかった場合の動作(cascade — 参照行を削除、restrict — 参照キーが存在する場合トランザクションをアボート、setnull —参照キーフィールドをNULLに設定)、プライマリ/一意キーを形成するトリガを発行したテーブルの列名、参照テーブルの名前と列名(最初の引数で指定された数のテーブル分繰り返す)です。プライマリ/一意キー列はNOT NULLと指定されていなければならず、また、一意性インデックスを持つべきであることに注意してください。

refint.exampleに例があります。

F.36.2. autoinc — フィールド自動増分用の関数

autoinc()は、整数型フィールドにシーケンスの次の値を格納するトリガです。これは、組み込みの「連番列」機能と一部重複しますが、同一ではありません。autoinc()は挿入時に別のフィールド値に置き換える試みを上書きし、さらに省略可能ですが、更新時にフィールドを増加させるために使用することもできます。

使用方法は、この関数を使用するBEFORE INSERT(または BEFORE INSERT OR UPDATE)トリガを作成することです。2つのトリガ引数、変更する整数型列の名前と値を生み出すシーケンスオブジェクトの名前を指定します。(実際、自動増分列を複数更新したい場合、これらの名前の組み合わせを任意の数指定することができます。)

autoinc.exampleに例があります。

F.36.3. insert_username — 誰がテーブルを変更したかを追跡する関数

insert_username()は現在のユーザ名をテキスト型のフィールドに格納するトリガです。これはテーブル内のある行を最後に変更したユーザを追跡する際に有用です。

使用方法は、この関数を使用するBEFORE INSERT、UPDATEまたはその両方のトリガを作成することです。1つのトリガ引数、変更するテキスト型の列の名前を指定してください。

insert_username.exampleに例があります。

F.36.4. moddatetime — 最終更新時刻を追跡する関数

moddatetime()は現在時刻をtimestamp型のフィールドに格納するトリガです。これは、テーブル内のある行の最終更新時刻を追跡する際に有用です。

使用方法は、この関数を使用するBEFORE UPDATEトリガを作成することです。1つのトリガ引数、変更する列名を指定してください。この列はtimestamp型またはtimestamp with time zone型でなければなりません。

moddatetime.exampleに例があります。

F.37. sslinfo

現在のクライアントがPostgreSQLに接続する際に提供する SSL 証明書に関する情報を、sslinfoモジュールは提供します。現在の接続が SSL を使用しない場合、モジュールは無用です(大部分の関数は NULL を返します)。

このモジュールを通じて取得できる情報の中には、組み込みシステムビューpg_stat_sslを使っても取得できるものがあります。

インストールを--with-openssloptionオプション付きで構築しない限り、この拡張は全く構築されません。

F.37.1. 提供される関数

ssl_is_used() returns boolean

サーバへの現在の接続においてSSLを使用する場合true、使用しない場合falseを返します。

ssl_version() returns text

SSL接続に使われているプロトコルの名前（例えば、TLSv1.0、TLSv1.1またはTLSv1.2）を返します。

ssl_cipher() returns text

SSL接続に使われている暗号の名前（例えば、DHE-RSA-AES256-SHA）を返します。

ssl_client_cert_present() returns boolean

現在のクライアントがサーバに対して、有効なSSLクライアント証明書を提示した場合true、そうでない場合falseを返します。（サーバがクライアントに対して、クライアント証明書を要求する方式と要求しない方式があります）。

ssl_client_serial() returns numeric

現在のクライアント証明書のシリアル番号を返します。証明書のシリアル番号と証明書の発行者との組み合わせにより、証明書が一意に識別されることが保証されます（しかし、証明書の所有者の保証ではありません。所有者は定期的にその鍵を変更し、発行者から新しい証明書を取得すべきだからです）。

したがって、自分で認証局を設立し、その認証局の証明書だけをサーバが受理する場合、シリアル番号は利用者を識別するのに最も信頼できる方法です（あまり記憶の助けにはなりませんが）。

ssl_client_dn() returns text

現在のクライアント証明書の所有者の内容を全て返します。文字データは現在のデータベースのエンコーディングに変換されます。なお、証明書名で非ASCII文字を用いる場合、データベースでもその文字を使用できると仮定します。データベースがSQL_ASCIIエンコーディングを使用する場合、証明書名で用いる非ASCII文字はUTF-8のユニコードとして表示されます。

その結果は/CN=Somebody /C=Some country /O=Some organizationのようになります。

ssl_issuer_dn() returns text

現在のクライアント証明書の発行者名を全て返します。文字データは現在のデータベースのエンコーディングに変換されます。エンコーディングの変換法はssl_client_dnと同じです。

本関数の戻り値と証明書シリアル番号の組み合わせにより、証明書を一意に識別します。

実際に本関数が有用となるのは、サーバの認証局ファイルの中に信頼できる認証局の証明書を複数保有している場合、またはこの認証局が中間認証局の証明書を発行している場合だけです。

ssl_client_dn_field(fieldname text) returns text

この関数は証明書の所有者の指定した項目の内容を返します。指定した項目が存在しない場合は NULL を返します。項目の名前は OpenSSL オブジェクトデータベースを使用して ASN1 オブジェクト識別子に変換された文字列定数です。以下の項目が受理できます。

```
commonName (alias CN)
surname (alias SN)
name
givenName (alias GN)
countryName (alias C)
localityName (alias L)
stateOrProvinceName (alias ST)
organizationName (alias O)
organizationalUnitName (alias OU)
title
description
initials
postalCode
streetAddress
generationQualifier
description
dnQualifier
x500UniqueIdentifier
pseudonym
role
emailAddress
```

commonName を除き、全ての項目は任意です。認証局の方針によって、どの項目を含み、どの項目を含まないかが全て決まります。しかし、X.500 および X.509 標準によって、項目の意味は厳格に定義されています。したがって、項目に任意の意味を持たせることはできません。

ssl_issuer_field(fieldname text) returns text

証明書の所有者に対するものではなく証明書の発行者に対するものであるという点を除き、ssl_client_dn_field と同様の関数です。

ssl_extension_info() returns setof record

クライアント証明書の拡張に関する情報を提供します。拡張に関する情報とは、拡張の名前、拡張の値、クリティカルな拡張か否かです。

F.37.2. 作者

Victor Wagner <vitus@cryptocom.ru>, Cryptocom LTD

Dmitry Voronin <carriingfate92@yandex.ru>

Cryptocom 社 OpenSSL 開発グループのメールアドレス <openssl@cryptocom.ru>

F.38. tablefunc

tablefuncモジュールにはテーブル（つまり複数行）を返す各種関数があります。これらの関数は、その独自の目的として、および、複数行を返すC関数の作成方法を示す例として、有用です。

このモジュールは「trusted」と見なされます。つまり、現在のデータベースに対してCREATE権限を持つ非スーパーユーザがインストールできます。

F.38.1. 提供される関数

tablefuncモジュールにより提供される関数を表 F.30にまとめます。

表F.30 tablefuncの関数

関数 説明
normal_rand (numvals integer, mean float8, stddev float8) → setof float8 正規分布乱数値の集合を生成します。
crosstab (sql text) → setof record 行の名前とN個の値列からなる「ピボット表」を生成します。ここでNは呼出元の問い合わせで指定される行型で決定します。
crosstabN (sql text) → setof table_crosstab_N 行の名前とN個の値列からなる「ピボット表」を生成します。crosstab2、crosstab3、crosstab4が定義されていますが、後述する手順で追加のcrosstabN関数を作成することが可能です。
crosstab (source_sql text, category_sql text) → setof record 2番目の問い合わせで指定された値列を持つ「ピボット表」を生成します。
crosstab (sql text, N integer) → setof record 廃止予定のcrosstab(text)です。値列の数は呼び出す問い合わせで常に決まりますので、現在パラメータNは無視されます。
connectby (relname text, keyid_fld text, parent_keyid_fld text [, orderby_fld text], start_with text, max_depth integer [, branch_delim text]) → setof record 階層ツリー構造表現を生成します。

F.38.1.1. normal_rand

```
normal_rand(int numvals, float8 mean, float8 stddev) returns setof float8
```

normal_randは正規乱数値の集合（ガウス分布）を生成します。

ここでnumvalsはこの関数が返す値の数です。meanは正規分布の平均値、stddevは正規分布値の標準偏差です。

例えば、以下の呼出しは、平均5、標準偏差3で1000個の値を要求します。

```
test=# SELECT * FROM normal_rand(1000, 5, 3);
```

```
     normal_rand
----------------------
     1.56556322244898
     9.10040991424657
     5.36957140345079
   -0.369151492880995
    0.283600703686639
        .
        .
        .
     4.82992125404908
     9.71308014517282
     2.49639286969028
(1000 rows)
```

F.38.1.2. crosstab(text)

```
crosstab(text sql)
crosstab(text sql, int N)
```

crosstab関数は「ピボット」表示を生成するために使用されます。ここでは、データは下方向にではなくページ横方向に渡って列挙されます。例えば、以下のようなデータがあるとします。

```
row1    val11
row1    val12
row1    val13
...
row2    val21
row2    val22
row2    val23
...
```

これを次のように表示したいとします。

```
row1    val11   val12   val13   ...
row2    val21   val22   val23   ...
...
```

crosstab関数は、最初のような書式を持つ生データを生成するSQL問い合わせとなるテキストパラメータを取り、2番目のような書式を持つテーブルを生成します。

sqlパラメータは元となるデータ集合を生成するSQL文です。この文はrow_name列を1つ、category列を1つ、value列を1つ返さなければなりません。Nは廃れたパラメータであり、指定されたとしても無視されます。(これまでは、これは出力値列の数と一致する必要がありました。しかし、現在これは呼び出し元の問い合わせにより決まります。)

例：指定したSQLは以下のような集合を生成しても構いません。

```
row_name    cat    value
----------+-------+-------
  row1      cat1    val1
  row1      cat2    val2
  row1      cat3    val3
  row1      cat4    val4
  row2      cat1    val5
  row2      cat2    val6
  row2      cat3    val7
  row2      cat4    val8
```

crosstab関数はsetof recordを返すものとして宣言されています。このため、出力列の実際の名前と型を呼び出し元のSELECT文のFROM内で宣言しなければなりません。以下に例を示します。

```
SELECT * FROM crosstab('...') AS ct(row_name text, category_1 text, category_2 text);
```

この例は以下のような集合を生成します。

```
<== value  columns  ==>
 row_name   category_1   category_2
----------+------------+------------
  row1        val1         val2
  row2        val5         val6
```

FROM句は出力を1つのrow_name列（SQL問い合わせの最初の結果列と同一データ型）と続くN個のvalue列（SQL問い合わせの3番目の結果列とすべて同じデータ型）を持つものとして定義しなければなりません。必要なだけの個数の値列を出力するように設定することができます。出力列の名前は使用者に任されています。

crosstab関数は、同じrow_name値を持つ入力行の各連続的なグループに対して、1つの出力行を生成します。左から右へこれらの行のvalueフィールドで出力value列を埋めていきます。もしグループ内の行が存在する出力value列より少なければ、余った出力列はNULLになります。もし行が多ければ、余った入力行は無視されます。

実際のところ、入力行の順序が適切になるように、つまり、同じrow_nameを持つ値がまとまり、行内で正しく順序付けられるように、SQL問い合わせは常にORDER BY 1,2を指定しなければなりません。crosstab自体が問い合わせ結果の2番目の列に注意を払わないことに注意してください。これは順序付けのため、3番目の列の値がページに渡って現れる順序を制御するためだけに存在します。

以下に複雑な例を示します。

```
CREATE TABLE ct(id SERIAL, rowid TEXT, attribute TEXT, value TEXT);
INSERT INTO ct(rowid, attribute, value) VALUES('test1','att1','val1');
INSERT INTO ct(rowid, attribute, value) VALUES('test1','att2','val2');
INSERT INTO ct(rowid, attribute, value) VALUES('test1','att3','val3');
```

```
INSERT INTO ct(rowid, attribute, value) VALUES('test1','att4','val4');
INSERT INTO ct(rowid, attribute, value) VALUES('test2','att1','val5');
INSERT INTO ct(rowid, attribute, value) VALUES('test2','att2','val6');
INSERT INTO ct(rowid, attribute, value) VALUES('test2','att3','val7');
INSERT INTO ct(rowid, attribute, value) VALUES('test2','att4','val8');

SELECT *
FROM crosstab(
  'select rowid, attribute, value
   from ct
   where attribute = ''att2'' or attribute = ''att3''
   order by 1,2')
AS ct(row_name text, category_1 text, category_2 text, category_3 text);

 row_name | category_1 | category_2 | category_3
----------+------------+------------+------------
 test1    | val2       | val3       |
 test2    | val6       | val7       |
(2 rows)
```

必要な出力行型をその定義に反映した独自のcrosstab関数を構築することで、常に出力列を定義するためのFROM句を書く必要性をなくすことができます。これは次節で説明します。他にも必要なFROM句をビュー定義に埋め込むことでも実現可能です。

> **注記**
>
> psqlの\crosstabviewコマンドも参照してください。crosstab()と類似の機能を提供します。

F.38.1.3. crosstabN(text)

```
crosstabN(text sql)
```

crosstabN関数は、呼び出し元のSELECT問い合わせで列名と型を書き出す必要性をなくすことができるように、一般的なcrosstab関数に対する独自のラッパを構築する方法の例です。tablefuncモジュールには、次のように出力行型が定義されたcrosstab2、crosstab3、crosstab4が含まれています。

```
CREATE TYPE tablefunc_crosstab_N AS (
    row_name TEXT,
    category_1 TEXT,
    category_2 TEXT,
        .
        .
        .
    category_N TEXT
```

```
);
```

このように、入力問い合わせがtext型のrow_name列とvalue列を生成し、かつ、2、3、または4個の出力値列を持つ場合、これらの関数を直接使用することができます。この他の点はすべて、上述の一般的なcrosstab関数で説明した通りの動作をします。

例えば、上で挙げた例は下のように動作します。

```
SELECT *
FROM crosstab3(
  'select rowid, attribute, value
   from ct
   where attribute = ''att2'' or attribute = ''att3''
   order by 1,2');
```

これらの関数はほぼ説明を目的として提供されたものです。背後のcrosstab()関数に基いた独自の戻り型と関数を作成することができます。独自のcrosstab関数を構築する方法は2つあります。

- contrib/tablefunc/tablefunc--1.0.sqlの例と同様にして、必要な出力列を記述する複合型を作成します。そして、text型のパラメータを1つ取り、setof your_type_nameを返す一意な名前の関数を、同じ背後のcrosstab C関数をリンクさせて定義します。例えば、元データが行名としてtext型を、値としてfloat8を生成し、5つの値列を希望する場合、以下のようになります。

  ```
  CREATE TYPE my_crosstab_float8_5_cols AS (
      my_row_name text,
      my_category_1 float8,
      my_category_2 float8,
      my_category_3 float8,
      my_category_4 float8,
      my_category_5 float8
  );

  CREATE OR REPLACE FUNCTION crosstab_float8_5_cols(text)
      RETURNS setof my_crosstab_float8_5_cols
      AS '$libdir/tablefunc','crosstab' LANGUAGE C STABLE STRICT;
  ```

- 暗黙的に戻り値の型を定義する場合はOUTパラメータを使用してください。同じ例を以下のように書くこともできます。

  ```
  CREATE OR REPLACE FUNCTION crosstab_float8_5_cols(
      IN text,
      OUT my_row_name text,
      OUT my_category_1 float8,
      OUT my_category_2 float8,
      OUT my_category_3 float8,
      OUT my_category_4 float8,
      OUT my_category_5 float8)
  ```

```
  RETURNS setof record
  AS '$libdir/tablefunc','crosstab' LANGUAGE C STABLE STRICT;
```

F.38.1.4. crosstab(text, text)

```
crosstab(text source_sql, text category_sql)
```

単一パラメータのcrosstab構文の大きな制限は、各値を最初の利用可能な列に挿入して、すべての値をグループのように扱う点です。値列を特定のデータカテゴリに対応させ、グループの一部はカテゴリの一部のデータを持たない可能性がある場合は、うまく動作しません。2パラメータを取るcrosstab構文は、出力列に対応するカテゴリのリストを明示的に提供することで、こうした状況を扱います。

source_sqlは元となるデータ集合を生成するSQL文です。このSQL文はrow_name列を1つcategory列を1つ、value列を1つ返さなければなりません。また1つ以上の「追加」の列を持つこともできます。row_name列が先頭でなければなりません。categoryとvalue列は、この順番で最後の2列でなければなりません。row_nameとcategoryとの間の列はすべて「追加」の列とみなされます。「追加」の列は同じrow_name値を持つ行すべてで同一であるということが前提です。

例えば、source_sqlは以下のような集合を生成しなければなりません。

```
SELECT row_name, extra_col, cat, value FROM foo ORDER BY 1;

 row_name    extra_col   cat    value
----------+------------+-----+---------
  row1         extra1    cat1    val1
  row1         extra1    cat2    val2
  row1         extra1    cat4    val4
  row2         extra2    cat1    val5
  row2         extra2    cat2    val6
  row2         extra2    cat3    val7
  row2         extra2    cat4    val8
```

category_sqlはカテゴリの集合を生成するSQL文でなければなりません。このSQL文は1つの列のみを返さなければなりません。また、少なくとも1つの結果行を生成しなければならず、さもないと、エラーになります。さらに重複するカテゴリを生成してはなりません。さもないとエラーとなります。category_sqlは以下のようなものになります。

```
SELECT DISTINCT cat FROM foo ORDER BY 1;
    cat
  -------
    cat1
    cat2
    cat3
    cat4
```

crosstab関数はsetof recordを返すものとして宣言されていますので、出力列の実際の名前と型を、以下の例のように、呼出元のSELECTのFROM句で定義しなければなりません。

```
SELECT * FROM crosstab('...', '...')
    AS ct(row_name text, extra text, cat1 text, cat2 text, cat3 text, cat4 text);
```

これは以下のような集合を生成します。

```
<==  value  columns   ==>
row_name   extra   cat1   cat2   cat3   cat4
---------+-------+------+------+------+------
  row1     extra1  val1   val2          val4
  row2     extra2  val5   val6   val7   val8
```

FROM句は、出力列の適切な個数、およびその適切なデータ型を定義しなければなりません。source_sql問い合わせ結果にN個の列がある場合、最初のN-2は最初のN-2出力列と一致しなければなりません。残りの出力列はsource_sql問い合わせ結果の最後の列の型を持たなければならず、かつ、category_sql問い合わせ結果内の行と同じ個数でなければなりません。

crosstab関数は、同一row_name値を持つ入力行の連続したグループ毎に1つの出力行を生成します。row_name出力列と任意の「追加」列はグループの最初の行からコピーされます。value出力列は、category値と一致する行のvalueで埋められます。行のcategoryがcategory_sql問い合わせの出力とまったく一致しなかった場合、そのvalueは無視されます。グループの入力行内にまったくカテゴリに一致する出力列が存在しない場合、NULLで埋められます。

実際は、同じrow_nameを持つ値をまとめられるように、source_sql問い合わせでは常にORDER BY 1を指定すべきです。しかし、グループ内のカテゴリの順序は重要ではありません。また、category_sql問い合わせの出力順序が指定された出力列の順序と一致することを確実にすることが重要です。

以下に複雑な例を2つ示します。

```
create table sales(year int, month int, qty int);
insert into sales values(2007, 1, 1000);
insert into sales values(2007, 2, 1500);
insert into sales values(2007, 7, 500);
insert into sales values(2007, 11, 1500);
insert into sales values(2007, 12, 2000);
insert into sales values(2008, 1, 1000);

select * from crosstab(
  'select year, month, qty from sales order by 1',
  'select m from generate_series(1,12) m'
) as (
  year int,
  "Jan" int,
  "Feb" int,
  "Mar" int,
  "Apr" int,
  "May" int,
  "Jun" int,
```

```
  "Jul" int,
  "Aug" int,
  "Sep" int,
  "Oct" int,
  "Nov" int,
  "Dec" int
);
 year | Jan  | Feb  | Mar | Apr | May | Jun | Jul | Aug | Sep | Oct | Nov  | Dec
------+------+------+-----+-----+-----+-----+-----+-----+-----+-----+------+------
 2007 | 1000 | 1500 |     |     |     |     | 500 |     |     |     | 1500 | 2000
 2008 | 1000 |      |     |     |     |     |     |     |     |     |      |
(2 rows)
```

```
CREATE TABLE cth(rowid text, rowdt timestamp, attribute text, val text);
INSERT INTO cth VALUES('test1','01 March 2003','temperature','42');
INSERT INTO cth VALUES('test1','01 March 2003','test_result','PASS');
INSERT INTO cth VALUES('test1','01 March 2003','volts','2.6987');
INSERT INTO cth VALUES('test2','02 March 2003','temperature','53');
INSERT INTO cth VALUES('test2','02 March 2003','test_result','FAIL');
INSERT INTO cth VALUES('test2','02 March 2003','test_startdate','01 March 2003');
INSERT INTO cth VALUES('test2','02 March 2003','volts','3.1234');

SELECT * FROM crosstab
(
  'SELECT rowid, rowdt, attribute, val FROM cth ORDER BY 1',
  'SELECT DISTINCT attribute FROM cth ORDER BY 1'
)
AS
(
       rowid text,
       rowdt timestamp,
       temperature int4,
       test_result text,
       test_startdate timestamp,
       volts float8
);
 rowid |          rowdt           | temperature | test_result |      test_startdate      | volts
-------+--------------------------+-------------+-------------+--------------------------+
+--------
 test1 | Sat Mar 01 00:00:00 2003 |          42 | PASS        |                          | 2.6987
 test2 | Sun Mar 02 00:00:00 2003 |          53 | FAIL        | Sat Mar 01 00:00:00 2003 | 3.1234
(2 rows)
```

各問い合わせで結果列の名前と型を記述する必要性をなくすために、事前定義した関数を作成することができます。前節の例を参照してください。このcrosstab構文用の背後のC関数はcrosstab_hashという名前です。

F.38.1.5. connectby

```
connectby(text relname, text keyid_fld, text parent_keyid_fld
          [, text orderby_fld ], text start_with, int max_depth
          [, text branch_delim ])
```

connectby関数はテーブル内に格納された階層データ表示を生成します。テーブルは行を一意に識別するキーフィールドと各行の親(もしあれば)を参照する親キーフィールドを持たなければなりません。connectbyは任意の行から辿った部分ツリーを表示することができます。

表 F.31ではパラメータを解説します。

表F.31 connectbyパラメータ

パラメータ	説明
relname	元となるリレーション名
keyid_fld	キーフィールドの名前
parent_keyid_fld	親のキーフィールドの名前
orderby_fld	兄弟の順序付け用のフィールド名(省略可能)
start_with	開始行のキー値
max_depth	辿る深さに対する制限。無制限の場合はゼロ
branch_delim	キーと分岐出力で区切る文字列(省略可能)

キーおよび親キーフィールドは任意のデータ型を取ることができますが、これらは同じデータ型でなければなりません。キーフィールドのデータ型に関係なく、start_withはテキスト文字列として入力されなければならないことに注意してください。

connectby関数はsetof recordを返すものとして宣言されていますので、以下の例のように、出力列の実際の名前と型を呼出し元のSELECT文のFROM句で定義しなければなりません。

```
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid', 'pos', 'row2', 0, '~')
    AS t(keyid text, parent_keyid text, level int, branch text, pos int);
```

先頭から2つの出力列は、現在の行のキーおよび親行のキーとして使用されます。これらはテーブルのキーフィールドのデータ型と一致する必要があります。3番目の出力列はツリーの深さであり、integer型である必要があります。branch_delimパラメータが与えられた場合、次の出力列は分岐表示であり、text型である必要があります。最後に、orderby_fldパラメータが与えられた場合、最後の出力列は連番であり、integer型である必要があります。

「分岐」出力列は現在の行まで達するために取られるキーの経路を示します。キーは指定されたbranch_delim文字列で区切られます。分岐表示が不要ならば、branch_delimパラメータと出力列リスト内の分岐列を省略してください。

同じ親を持つ兄弟の順序が重要な場合、どのフィールドで兄弟の順序付けを行うかを指定するorderby_fldパラメータを含めてください。このフィールドは任意のソート可能なデータ型を取ることができ

ます。orderby_fldが指定された場合のみ、出力列リストには、最終整数型連番列を含めなければなりません。

テーブルおよびフィールド名を表すパラメータはそのままconnectbyが内部的に生成するSQL問い合わせにコピーされます。したがって、大文字小文字が混在した名前または特殊文字を含む名前の場合は二重引用符で括ってください。またテーブル名をスキーマで修飾する必要があるかもしれません。

大規模なテーブルでは、親キーフィールド上にインデックスがないと性能が劣化します。

branch_delim文字列がキー値内にまったく出現しないことが重要です。さもないと、connectbyは無限再帰エラーを間違って報告するかもしれません。branch_delimが提供されていない場合、再帰を検知するためにデフォルト値~が使用されます。

以下に例を示します。

```
CREATE TABLE connectby_tree(keyid text, parent_keyid text, pos int);

INSERT INTO connectby_tree VALUES('row1',NULL, 0);
INSERT INTO connectby_tree VALUES('row2','row1', 0);
INSERT INTO connectby_tree VALUES('row3','row1', 0);
INSERT INTO connectby_tree VALUES('row4','row2', 1);
INSERT INTO connectby_tree VALUES('row5','row2', 0);
INSERT INTO connectby_tree VALUES('row6','row4', 0);
INSERT INTO connectby_tree VALUES('row7','row3', 0);
INSERT INTO connectby_tree VALUES('row8','row6', 0);
INSERT INTO connectby_tree VALUES('row9','row5', 0);


-- 分岐あり、
      orderby_fldなし(結果の順序は保証されない)
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid', 'row2', 0, '~')
 AS t(keyid text, parent_keyid text, level int, branch text);
 keyid | parent_keyid | level |       branch
-------+--------------+-------+--------------------
 row2  |              |     0 | row2
 row4  | row2         |     1 | row2~row4
 row6  | row4         |     2 | row2~row4~row6
 row8  | row6         |     3 | row2~row4~row6~row8
 row5  | row2         |     1 | row2~row5
 row9  | row5         |     2 | row2~row5~row9
(6 rows)


-- 分岐なし、
      orderby_fldなし(結果の順序は保証されない)
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid', 'row2', 0)
 AS t(keyid text, parent_keyid text, level int);
 keyid | parent_keyid | level
```

```
-------+--------------+-------
 row2  |              |     0
 row4  | row2         |     1
 row6  | row4         |     2
 row8  | row6         |     3
 row5  | row2         |     1
 row9  | row5         |     2
(6 rows)


-- 分岐あり、
      orderby_fldあり(row5がrow4の前に来ていることに注目)
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid', 'pos', 'row2', 0, '~')
 AS t(keyid text, parent_keyid text, level int, branch text, pos int);
 keyid | parent_keyid | level |       branch        | pos
-------+--------------+-------+---------------------+-----
 row2  |              |     0 | row2                |   1
 row5  | row2         |     1 | row2~row5           |   2
 row9  | row5         |     2 | row2~row5~row9      |   3
 row4  | row2         |     1 | row2~row4           |   4
 row6  | row4         |     2 | row2~row4~row6      |   5
 row8  | row6         |     3 | row2~row4~row6~row8 |   6
(6 rows)


-- 分岐なし、orderby_fldあり(row5がrow4の前に来ていることに注目)
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid', 'pos', 'row2', 0)
 AS t(keyid text, parent_keyid text, level int, pos int);
 keyid | parent_keyid | level | pos
-------+--------------+-------+-----
 row2  |              |     0 |   1
 row5  | row2         |     1 |   2
 row9  | row5         |     2 |   3
 row4  | row2         |     1 |   4
 row6  | row4         |     2 |   5
 row8  | row6         |     3 |   6
(6 rows)
```

F.38.2. 作者

Joe Conway

F.39. tcn

tcnモジュールは関連づけされたテーブル上の変更を監視者に通知するトリガ関数を提供します。これはFOR EACH ROWのAFTERトリガとして使用しなければなりません。

このモジュールは「trusted」と見なされます。つまり、現在のデータベースに対してCREATE権限を持つ非スーパーユーザがインストールできます。

CREATE TRIGGER文の中で与えることができるパラメータは1つしかありませんが、省略することができます。与えられた場合、それは通知のチャネル名として使用されます。省略された場合はチャネル名としてtcnが使用されます。

通知のペイロードにはテーブル名、どのような種類の操作が行われたかを示す文字、主キー列における列名と値の組み合わせが含まれます。部位はそれぞれカンマで区切られています。正規表現を使用して簡単に解析するために、テーブル名と列名は常に二重引用符で括られ、またデータ値は常に単一引用符で括られています。内部に含まれる引用符は二重化されます。

この拡張を使用する簡単な例を以下に示します。

```
test=# create table tcndata
test-#   (
test(#     a int not null,
test(#     b date not null,
test(#     c text,
test(#     primary key (a, b)
test(#   );
CREATE TABLE
test=# create trigger tcndata_tcn_trigger
test-#   after insert or update or delete on tcndata
test-#   for each row execute function triggered_change_notification();
CREATE TRIGGER
test=# listen tcn;
LISTEN
test=# insert into tcndata values (1, date '2012-12-22', 'one'),
test-#                             (1, date '2012-12-23', 'another'),
test-#                             (2, date '2012-12-23', 'two');
INSERT 0 3
Asynchronous notification "tcn" with payload ""tcndata",I,"a"='1',"b"='2012-12-22'" received from
 server process with PID 22770.
Asynchronous notification "tcn" with payload ""tcndata",I,"a"='1',"b"='2012-12-23'" received from
 server process with PID 22770.
Asynchronous notification "tcn" with payload ""tcndata",I,"a"='2',"b"='2012-12-23'" received from
 server process with PID 22770.
test=# update tcndata set c = 'uno' where a = 1;
UPDATE 2
Asynchronous notification "tcn" with payload ""tcndata",U,"a"='1',"b"='2012-12-22'" received from
 server process with PID 22770.
Asynchronous notification "tcn" with payload ""tcndata",U,"a"='1',"b"='2012-12-23'" received from
 server process with PID 22770.
```

```
test=# delete from tcndata where a = 1 and b = date '2012-12-22';
DELETE 1
Asynchronous notification "tcn" with payload ""tcndata",D,"a"='1',"b"='2012-12-22'" received from
 server process with PID 22770.
```

F.40. test_decoding

test_decodingは論理デコード出力プラグインの例です。これは特に有用なことはまったく行いませんが、独自出力プラグイン開発の開始点として使えます。

test_decodingは論理デコード機構を通してWALを受け取り、実行された操作のテキスト表現にデコードします。

このプラグインがSQL論理デコードインタフェースで使われると、そこからの典型的な出力は以下のようになるでしょう。

```
postgres=# SELECT * FROM pg_logical_slot_get_changes('test_slot', NULL, NULL, 'include-xids',
 '0');
   lsn     | xid |                       data
-----------+-----+-------------------------------------------------
 0/16D30F8 | 691 | BEGIN
 0/16D32A0 | 691 | table public.data: INSERT: id[int4]:2 data[text]:'arg'
 0/16D32A0 | 691 | table public.data: INSERT: id[int4]:3 data[text]:'demo'
 0/16D32A0 | 691 | COMMIT
 0/16D32D8 | 692 | BEGIN
 0/16D3398 | 692 | table public.data: DELETE: id[int4]:2
 0/16D3398 | 692 | table public.data: DELETE: id[int4]:3
 0/16D3398 | 692 | COMMIT
(8 rows)
```

F.41. tsm_system_rows

tsm_system_rowsモジュールはSYSTEM_ROWSというテーブルサンプリングメソッドを提供します。これはSELECTコマンドのTABLESAMPLE句で利用できます。

このテーブルサンプリングメソッドは読み込む最大行数を指定する整数の引数を1つ取ります。結果のサンプルにはいつでもそれと正確に同じだけの行数が含まれます。ただしテーブルにそれだけの行数がないときは、テーブル全体が選択されることになります。

組み込みのSYSTEMサンプリングメソッドと同様、SYSTEM_ROWSはブロックレベルのサンプリングを行うため、サンプルは完全にはランダムではなく、特にごく少数の行が要求されたときはクラスタリングの影響を受けます。

SYSTEM_ROWSはREPEATABLE句をサポートしません。

このモジュールは「trusted」と見なされます。つまり、現在のデータベースに対してCREATE権限を持つ非スーパーユーザがインストールできます。

F.41.1. 例

以下にSYSTEM_ROWSを使ってテーブルのサンプルをSELECTする例を示します。まず、拡張をインストールします。

```
CREATE EXTENSION tsm_system_rows;
```

これで、例えば以下のようにSELECTコマンドを使うことができます。

```
SELECT * FROM my_table TABLESAMPLE SYSTEM_ROWS(100);
```

このコマンドはテーブルmy_tableからサンプルの100行を返します。（ただし、テーブルに可視の行が100行ないときは、すべての行が返されます。）

F.42. tsm_system_time

tsm_system_timeモジュールはSYSTEM_TIMEというテーブルサンプリングメソッドを提供します。これはSELECTコマンドのTABLESAMPLE句で利用できます。

このテーブルサンプリングメソッドはテーブルを読み込みのに消費する最大ミリ秒を指定する浮動小数点の引数を1つ取ります。これにより、サンプルのサイズを予測するのが難しくなる代わりに、問い合わせに要する時間に関する直接的な制御が得られます。結果のサンプルには、指定した時間内に読み込めただけの数の行が含まれます。ただし、テーブル全体を先に読み終わった時は除きます。

組み込みのSYSTEMサンプリングメソッドと同様、SYSTEM_TIMEはブロックレベルのサンプリングを行うため、サンプルは完全にはランダムではなく、特にごく少数の行がSELECTされたときはクラスタリングの影響を受けます。

SYSTEM_TIMEはREPEATABLE句をサポートしません。

このモジュールは「trusted」と見なされます。つまり、現在のデータベースに対してCREATE権限を持つ非スーパーユーザがインストールできます。

F.42.1. 例

以下にSYSTEM_TIMEを使ってテーブルのサンプルをSELECTする例を示します。まず、拡張をインストールします。

```
CREATE EXTENSION tsm_system_time;
```

これで、例えば以下のようにSELECTコマンドを使うことができます。

```
SELECT * FROM my_table TABLESAMPLE SYSTEM_TIME(1000);
```

このコマンドは1秒（1000ミリ秒）の間にmy_tableから読み込めるだけのサンプルを返します。もちろん、テーブル全体が1秒いないに読み込めるときは、すべての行が返されます。

F.43. unaccent

unaccentは語彙素からアクセント(発音区分記号)を取り除く全文検索用の辞書です。これはフィルタ処理を行う辞書、つまり、標準の動作と異なり、その出力が常に次の辞書（もしあれば）に渡されるものです。これにより全文検索においてアクセントを無視した処理を行うことができます。

現在のunaccentの実装ではthesaurus辞書向けの正規化用辞書として使用することはできません。

このモジュールは「trusted」と見なされます。つまり、現在のデータベースに対してCREATE権限を持つ非スーパーユーザがインストールできます。

F.43.1. 設定

unaccent辞書は以下のオプションを受け付けます。

- RULESは翻訳規則の一覧を含むファイルのベースネームです。このファイルは$SHAREDIR/tsearch_data/内に格納しなければなりません。（ここで$SHAREDIRはPostgreSQLインストレーションの共有データディレクトリを意味します。） この名前は.rulesで終わらなければなりません。（.rulesはRULESパラメータには含まれません。）

rulesファイルの書式は以下の通りです。

- 各行は、アクセント付き文字とその後にアクセントを取り除いた文字から構成される、1つの変換規則です。一つ目が二つ目に変換されます。以下に例を示します。

```
À       A
Á       A
Â       A
Ã       A
Ä       A
Å       A
Æ       AE
```

  2つの文字は空白で分けられていなければならず、行の先頭や末尾の空白は無視されます。

- あるいは、一行に一文字だけ指定された場合、その文字は削除されます。これは、アクセントが分かれた文字で表現される言語では便利です。

- 実のところ、各「文字」は空白を含まなければいかなる文字列でも良いので、unaccent辞書は発音区別符号の除去に加えて、部分文字列の置換などに使うこともできます。

- 他のPostgreSQLテキスト検索設定ファイルと同じように、rulesファイルはUTF-8エンコーディングで保存しなければなりません。データはロード時に自動的に現在のデータベースのエンコーディングに変換されます。rulesファイルが現在のエンコーディングで適用できない規則も含むことができるように、変換できない文字を含む行は単に無視されます。

unaccent.rulesは、ほとんどの欧州圏の言語で直接使用できる、より複雑な例です。これはunaccentモジュールをインストールした時に$SHAREDIR/tsearch_data/にインストールされます。このrulesファイルは、アクセント記号のある文字をアクセント記号のない同じ文字に変換し、また、合字を同等な普通の文字の並びに(例えば、ÆをAEに)展開します。

F.43.2. 使用方法

unaccent拡張をインストールすることで、unaccent全文検索テンプレートとそれに基づくデフォルトのパラメータを持つunaccent辞書が生成されます。unaccent辞書はRULES='unaccent'というデフォルトパラメータ設定を持ちます。これは標準のunaccent.rulesファイルを即座に使用可能にします。次の例のようにパラメータを変更することができます。

```
mydb=# ALTER TEXT SEARCH DICTIONARY unaccent (RULES='my_rules');
```

また、このテンプレートに基づいた辞書を新規に作成することができます。

以下を行うことで、辞書の動作を確認することができます。

```
mydb=# select ts_lexize('unaccent','Hôtel');
 ts_lexize
-----------
 {Hotel}
(1 row)
```

全文検索設定にunaccent辞書を組み込む方法を示す例を以下に示します。

```
mydb=# CREATE TEXT SEARCH CONFIGURATION fr ( COPY = french );
mydb=# ALTER TEXT SEARCH CONFIGURATION fr
        ALTER MAPPING FOR hword, hword_part, word
        WITH unaccent, french_stem;
mydb=# select to_tsvector('fr','Hôtels de la Mer');
    to_tsvector
-------------------
 'hotel':1 'mer':4
(1 row)

mydb=# select to_tsvector('fr','Hôtel de la Mer') @@ to_tsquery('fr','Hotels');
 ?column?
----------
 t
(1 row)

mydb=# select ts_headline('fr','Hôtel de la Mer',to_tsquery('fr','Hotels'));
      ts_headline
-----------------------
 <b>Hôtel</b> de la Mer
```

```
(1 row)
```

F.43.3. 関数

unaccent関数は与えられた文字列からアクセント（発音区別符号）を取り除きます。基本的にこれはunaccent型の辞書のラッパです。しかし通常の全文検索以外の文脈で使用することができます。

```
unaccent([dictionary regdictionary, ] string text) returns text
```

引数dictionaryが省略された場合、unaccentという名前でunaccent()関数自体と同じスキーマにある全文検索用の辞書が使われます。

下記は使用例です。

```
SELECT unaccent('unaccent', 'Hôtel');
SELECT unaccent('Hôtel');
```

F.44. uuid-ossp

uuid-osspモジュールは複数の標準的なアルゴリズムの1つを使用して汎用一意識別子（UUID）を生成する関数を提供します。また、特殊なUUID定数を生成する関数も提供します。このモジュールは、コアのPostgreSQLで利用可能なものを超える特別な要件にのみ必要となります。UUIDを生成する組み込みの方法は9.14を参照してください。

このモジュールは「trusted」と見なされます。つまり、現在のデータベースに対してCREATE権限を持つ非スーパーユーザがインストールできます。

F.44.1. uuid-ossp関数

UUIDを生成するために利用できる関数を表 F.32に示します。関連する標準ITU-T Rec. X.667、ISO/IEC 9834-8:2005、RFC 4122はUUIDの生成に関して、バージョン番号1、3、4、5で識別される4つのアルゴリズムを規定します。（バージョン2アルゴリズムは存在しません。）これらのアルゴリズムのそれぞれは、異なるアプリケーション群に適切でしょう。

表F.32 UUID生成用関数

関数 説明
uuid_generate_v1 () → uuid バージョン1 UUIDを生成します。これはコンピュータのMACアドレスとタイムスタンプが含まれます。この種のUUIDは識別子を生成したコンピュータを識別できる情報や生成した時刻をあばくことができますので、ある種のセキュリティに注意すべきアプリケーションでは適しません。
uuid_generate_v1mc () → uuid

関数 説明
コンピュータの実MACアドレスではなくランダムなマルチキャストMACアドレスを使用して、バージョン1 UUIDを作成します。
uuid_generate_v3 (namespace uuid, name text) → uuid 入力で指定されたnameを使用して、与えられた名前空間でバージョン3 UUIDを生成します。名前空間は、表 F.33に示されるuuid_ns_*()関数で生成される特殊な定数の1つでなければなりません。(理論上これは何らかのUUIDになります。) nameは選択された名前空間内の識別子です。 例えば以下の様になります。 `SELECT uuid_generate_v3(uuid_ns_url(), 'http://www.postgresql.org');` nameパラメータはMD5でハッシュ化されます。このため、生成されたUUIDから平文が分かることはありません。この方法によるUUIDの生成は不規則性はなく、また、環境に依存する要素もないため、再度生成されます。
uuid_generate_v4 () → uuid バージョン4 UUIDを生成します。これは完全にランダムな数から生成されます。
uuid_generate_v5 (namespace uuid, name text) → uuid バージョン5 UUIDを生成します。バージョン3 UUIDと似ていますが、ハッシュ方式としてSHA-1を使用することが異なります。SHA-1がMD5より安全であることから、バージョン5はバージョン3に比べて好まれるはずです。

表F.33 UUID定数を返す関数

関数 説明
uuid_nil () → uuid 「「nil」」UUID定数を返します。これは実際のUUIDになることはありません。
uuid_ns_dns () → uuid DNS名前空間をUUIDに選定した定数を返します。
uuid_ns_url () → uuid URL名前空間をUUIDに選定した定数を返します。
uuid_ns_oid () → uuid ISOオブジェクト識別子(OID)をUUIDに選定した定数を返します。(これはASN.1のOIDに関するもので、PostgreSQLで使われるOIDとは関係ありません。)
uuid_ns_x500 () → uuid X.500区分名(DN)をUIDに選定した定数を返します。

F.44.2. uuid-osspの構築

歴史的にこのモジュールは、モジュールの名前の由来となったOSSP UUIDライブラリに依存していました。OSSP UUIDライブラリはまだhttp://www.ossp.org/pkg/lib/uuid/にありますが、あまりよく維持されておらず、より新しいプラットフォームへ移植することがますます困難になってきています。uuid-osspは今やいくつかのプラットフォームではOSSPライブラリなしで構築できます。FreeBSD、NetBSD、その他のBSDから派生したプラットフォームでは、適切なUUID生成関数がコアlibcライブラリに含まれています。Linux、OS X、その他のプラットフォームでは、適切な関数がlibuuidライブラリで提供されており、(現在のLinuxではutil-linux-ngの一部と考えられていますが)そのライブラリはe2fsprogsプロジェクトに由来します。

configureを実行する時に、BSD関数を使うのなら--with-uuid=bsdを、e2fsprogsのlibuuidを使うのなら--with-uuid=e2fsを、OSSP UUIDライブラリを使うのなら--with-uuid=osspを指定してください。あるマシンではこのライブラリのうち二つ以上が利用可能かもしれませんので、configureは自動的に一つを選びません。

F.44.3. 作者

Peter Eisentraut <peter_e@gmx.net>

F.45. xml2

xml2モジュールはXPath問い合わせとXSLT機能を提供します。

F.45.1. 廃止予定の可能性についてのお知らせ

PostgreSQL 8.3から、SQL/XML標準に基づくXML関連の機能はコアサーバ内に存在します。その機能は、XML構文検査、XPath問い合わせなど本モジュールが行なうことと同等のこととそれ以上のことを範囲としますが、APIには互換性はありません。新しい標準APIのため、本モジュールは今後のバージョンのPostgreSQLで削除される予定ですので、アプリケーションの変換が推奨されています。本モジュールの機能に新しいAPIに適用できないものがあることが分かった場合、その不足に取り組むことができるように<pgsql-hackers@lists.postgresql.org>にその問題を表明してください。

F.45.2. 関数の説明

表 F.34に本モジュールで提供する関数を示します。これらの関数は簡単なXML解析とXPath問い合わせを提供します。

表F.34 xml2関数

関数 説明
xml_valid (document text) → boolean 与えられた文書を解析し、文書が整形式のXMLであれば真を返します。(注意:これは標準のPostgreSQL関数xml_is_well_formed()の別名です。XMLでは整形と検証が異なる意味を持つため、xml_valid()と言う名前は技術的には正しくありません。)
xpath_string (document text, query text) → text 与えられた文書に対するXPath問い合わせを評価し、結果をtextにキャストします。
xpath_number (document text, query text) → real 与えられた文書に対するXPath問い合わせを評価し、結果をrealにキャストします。
xpath_bool (document text, query text) → boolean 与えられた文書に対するXPath問い合わせを評価し、結果をbooleanにキャストします。
xpath_nodeset (document text, query text, toptag text, itemtag text) → text

関数 説明

文書に対する問い合わせを評価し、XMLタグ内に結果を包みます。結果が複数の値であれば、出力は以下のようになります。
`<toptag>`
`<itemtag>Value 1 which could be an XML fragment</itemtag>`
`<itemtag>Value 2....</itemtag>`
`</toptag>`
toptagまたはitemtagが空文字だった場合、対応するタグは省略されます。

xpath_nodeset (document text, query text, itemtag text) → text xpath_nodeset(document, query, toptag, itemtag)と同様ですが、結果はtoptagを省きます。
xpath_nodeset (document text, query text) → text xpath_nodeset(document, query, toptag, itemtag)と同様ですが、結果は両方のタグを省きます。
xpath_list (document text, query text, separator text) → text 文書に対する問い合わせを評価し、複数の値を指定した区切り文字で区切って返します。例えば、separatorが,ならばValue 1,Value 2,Value 3となります。
xpath_list (document text, query text) → text これは、,を区切り文字として使用する、上の関数のラッパです。

F.45.3. xpath_table

```
xpath_table(text key, text document, text relation, text xpaths, text criteria) returns setof
 record
```

xpath_tableは各文書集合に対するXPath問い合わせ集合を評価し、結果をテーブルとして返すテーブル関数です。元文書テーブルの主キーフィールドが結果の第一列として返されますので、結果セットを容易に結合で使用することができます。パラメータについては表 F.35で説明します。

表F.35 xpath_tableのパラメータ

パラメータ	説明
key	「key」フィールドの名前です。これは、出力テーブルの第一列として使用される単なるフィールドです。つまり、これは各出力行の出現元を識別するレコードです。（後述の複数値に関する注記を参照してください。）
document	XML文書を含むフィールドの名前です。
relation	文書を含むテーブルまたはビューの名前です。
xpaths	|で区切られた、1つ以上のXPath式です。
criteria	WHERE句の内容です。これは省略することができません。リレーション内の全行を処理したい場合はtrueまたは1=1を使用してください。

（XPath文字列を除く）これらのパラメータは普通のSQL SELECT 文に単純に置換されます。このため、多少の柔軟性があります。

```
SELECT <key>, <document> FROM <relation> WHERE <criteria>
```

文は上の通りですので、これらのパラメータにはそれぞれの場所で有効なものであれば何でもよいわけです。このSELECTの結果は正確に2つの列を返さなければなりません（キーまたは文書に対して複数のフィールドを列挙させようとしない限りです）。この簡略された手法では、SQLインジェクション攻撃を防ぐためにユーザから与えられた値をすべて検証しなければならないことに注意してください。

この関数は、出力列を指定するためのAS句を付けたFROM式内で使用されなければなりません。以下に例を示します。

```
SELECT * FROM
xpath_table('article_id',
            'article_xml',
            'articles',
            '/article/author|/article/pages|/article/title',
            'date_entered > ''2003-01-01'' ')
AS t(article_id integer, author text, page_count integer, title text);
```

このAS句は、出力テーブルの列名とその型を定義します。先頭が「key」フィールド、残りがXPath問い合わせに対応します。結果列より多くのXPath問い合わせが存在する場合、余った問い合わせは無視されます。XPath問い合わせより多くの結果列が存在する場合は余った列はNULLになります。

この例でpage_count結果列が整数として定義されていることに注意してください。関数は内部的に文字列表現で扱います。このため、出力内で整数で扱いたいと言っている時、XPath結果の文字列表現を取り出し、整数（またはAS句で要求した任意の型）に変換するためにPostgreSQLの入力関数を使用します。例えば結果が空など、変換できない場合はエラーになります。ですので、データに何らかの問題があると考えられる場合、列型としてtextに限定する方がよいかもしれません。

SELECT文の呼び出しでは、単なるSELECT *でなければならない必要性はありません。出力列を名前で参照することも他のテーブルと結合することも可能です。この関数は希望の何らかの操作（例えば集約、結合、ソートなど）を行うことができる仮想テーブルを生成します。このため以下をより複雑な例として示すことができます。

```
SELECT t.title, p.fullname, p.email
FROM xpath_table('article_id', 'article_xml', 'articles',
                 '/article/title|/article/author/@id',
                 'xpath_string(article_xml,''/article/@date'') > ''2003-03-20'' ')
       AS t(article_id integer, title text, author_id integer),
     tblPeopleInfo AS p
WHERE t.author_id = p.person_id;
```

当然ながら、簡便にするためにこれをすべてビューとして包み隠すことができます。

F.45.3.1. 複数値の結果

xpath_table関数は各XPath問い合わせの結果が複数の値を持つ可能性があることを前提としています。このため、この関数が返す行数は入力文書の数と同じにならない可能性があります。返される最初の行には各

問い合わせの最初の結果が、2番目の行には各問い合わせの2番目の結果が含まれます。問い合わせの1つが他よりも少ない値を持つ場合は代わりにNULL値が返されます。

指定したXPath問い合わせが単一の結果（おそらく一意な文書識別子）のみを返すことがユーザが分かっている場合があります。もしこれを複数の結果を返すXPathと一緒に使用されると、単一値の結果は結果の最初の行にのみ現れます。この解決方法はより単純なXPath問い合わせに対する結合部分としてキーフィールドを使用することです。以下に例を示します。

```
CREATE TABLE test (
    id int PRIMARY KEY,
    xml text
);

INSERT INTO test VALUES (1, '<doc num="C1">
<line num="L1"><a>1</a><b>2</b><c>3</c></line>
<line num="L2"><a>11</a><b>22</b><c>33</c></line>
</doc>');

INSERT INTO test VALUES (2, '<doc num="C2">
<line num="L1"><a>111</a><b>222</b><c>333</c></line>
<line num="L2"><a>111</a><b>222</b><c>333</c></line>
</doc>');

SELECT * FROM
  xpath_table('id','xml','test',
              '/doc/@num|/doc/line/@num|/doc/line/a|/doc/line/b|/doc/line/c',
              'true')
  AS t(id int, doc_num varchar(10), line_num varchar(10), val1 int, val2 int, val3 int)
WHERE id = 1 ORDER BY doc_num, line_num

 id | doc_num | line_num | val1 | val2 | val3
----+---------+----------+------+------+------
  1 | C1      | L1       |    1 |    2 |    3
  1 |         | L2       |   11 |   22 |   33
```

各行にdoc_numを付けるためには、2つのxpath_tableを呼び出し、その結果を結合することです。

```
SELECT t.*,i.doc_num FROM
  xpath_table('id', 'xml', 'test',
              '/doc/line/@num|/doc/line/a|/doc/line/b|/doc/line/c',
              'true')
    AS t(id int, line_num varchar(10), val1 int, val2 int, val3 int),
  xpath_table('id', 'xml', 'test', '/doc/@num', 'true')
    AS i(id int, doc_num varchar(10))
WHERE i.id=t.id AND i.id=1
ORDER BY doc_num, line_num;
```

```
 id | line_num | val1 | val2 | val3 | doc_num
----+----------+------+------+------+---------
  1 | L1       |    1 |    2 |    3 | C1
  1 | L2       |   11 |   22 |   33 | C1
(2 rows)
```

F.45.4. XSLT関数

libxsltがインストールされている場合、以下の関数を使用することができます。

F.45.4.1. xslt_process

```
xslt_process(text document, text stylesheet, text paramlist) returns text
```

この関数はXSLスタイルシートを文書に適用し、変換した結果を返します。paramlistは、'a=1,b=2'という形で指定された、変換で使用されるパラメータ代入式のリストです。パラメータ解析はあまり熟考されたものではないことに注意してください。パラメータ値にカンマを入れることができません。

また、変換用のパラメータを渡さない、2つのパラメータを取るバージョンのxslt_processも存在します。

F.45.5. 作者

John Gray <jgray@azuli.co.uk>

本モジュールの開発はTorchbox Ltd. (www.torchbox.com)が後援しました。PostgreSQLと同じBSDライセンスです。

付録G 追加で提供されるプログラム

この付録と前の付録には、PostgreSQL配布物のcontribディレクトリにあるモジュールに関する情報があります。contribセクションの概要や、特にcontribにあるサーバエクステンションやプラグインに関する情報は、付録Fを参照してください。

この付録ではcontribにあるユーティリティプログラムを説明します。ソースからでもパッケージシステムからでも、いったんインストールされるとそれらはPostgreSQLがインストールされた場所のbinディレクトリに入り、他のプログラムと同様に使用することができます。

G.1. クライアントアプリケーション

このセクションでは、contribにあるPostgreSQLのクライアントアプリケーションを説明します。それらは、データベースサーバがどこで稼働しているかに依存せず、どこからでも実行することが出来ます。PostgreSQLのコア配布物に含まれるクライアントアプリケーションに関する情報は、PostgreSQLクライアントアプリケーションを参照してください。

oid2name

oid2name — OIDとPostgreSQLデータディレクトリ内のファイルノードを解決する

概要

oid2name [option...]

説明

oid2nameは、管理者がPostgreSQLで使用されるファイル構造を確認することを補助するユーティリティプログラムです。使用できるようになるためには、第68章で説明されるデータベースファイル構造についての知識が必要です。

> **注記**
>
> 「oid2name」という名前は歴史的なものであり、これを使用する場合のほとんどでは、本当はテーブルのファイルノード番号（これはデータベースディレクトリ内で可視なファイル名）が関係しますので、実際誤解されやすいものです。テーブルのOIDとテーブルファイルノードの違いを確実に理解してください。

oid2nameは対象データベースに接続し、OID、ファイルノード、テーブル名情報を抽出します。また、データベースOIDまたはテーブル空間OIDを示すようにさせることもできます。

オプション

oid2nameは以下のコマンドライン引数を受け付けます。

-f filenode
--filenode=filenode

filenodeというファイルノードを持つテーブルの情報を表示します。

-i
--indexes

一覧にインデックスおよびシーケンスを含めます。

-o oid
--oid=oid

oidというOIDを持つテーブルの情報を表示します。

-q
--quiet

ヘッダを省略します。（スクリプト処理に適しています）

-s
--tablespaces

テーブル空間OIDを表示します。

-S
--system-objects

システムオブジェクト（information_schema、pg_toast、pg_catalogスキーマ内に存在するもの）を含めます。

-t tablename_pattern
--table=tablename_pattern

tablename_patternに一致するテーブル（複数可）の情報を表示します。

-V
--version

oid2nameのバージョンを表示し、終了します。

-x
--extended

表示対象の各オブジェクトに関してさらに情報を表示します。テーブル空間名、スキーマ名、OID。

-?
--help

oid2nameのコマンドライン引数の説明を表示し、終了します。

またoid2nameは以下の接続用のパラメータに関するコマンドライン引数を受け付けます。

-d database
--dbname=database

接続データベース。

-h host
--host=host

データベースサーバのホスト。

-H host

データベースサーバのホスト。このパラメータの使用はPostgreSQL 12以降で廃止予定です。

-p port
--port=port

データベースサーバのポート。

-U username
--username=username

接続ユーザ名。

特定のテーブルを表示するために、-o、-f、-tを使用して表示するテーブルを選択してください。-oはOIDを、-fはファイルノードを、-tはテーブル名(実際はLIKEパターンです。ですのでfoo%などが使用できます)を引数として取ります。これらのオプションを必要なだけ使用することができます。一覧には、オプションのいずれかで一致したオブジェクトがすべて含まれます。しかしこれらのオプションでは、-dで指定したデータベース内に存在するオブジェクトしか表示しないことに注意してください。

-o、-f、-tのいずれも指定せずに-dを指定した場合、-dで指定したデータベース上のすべてのテーブルを列挙します。このモードでは、-Sおよび-iスイッチが何を列挙するかを制御します。

-dも指定しなかった場合、データベースOIDの一覧を示します。他にも-sを指定してテーブル空間の一覧を得ることもできます。

環境

PGHOST
PGPORT
PGUSER

デフォルトの接続パラメータ。

このユーティリティは、他の多くPostgreSQLユーティリティと同様に、libpqがサポートする環境変数(33.14参照)も使います。

環境変数PG_COLORは診断メッセージで色を使うかどうかを指定します。可能な値はalways、auto、neverです。

注釈

oid2nameは破損のないシステムカタログで実行中のデータベースサーバが必要です。したがって、破滅的にデータベースが破損したような状況からの復旧には限定的にしか役に立ちません。

例

```
$ # とにかく、
      このデータベースサーバの中には何があるのだろう
$ oid2name
All databases:
    Oid  Database Name  Tablespace
----------------------------------
  17228       alvherre  pg_default
  17255     regression  pg_default
  17227      template0  pg_default
      1      template1  pg_default

$ oid2name -s
All tablespaces:
     Oid  Tablespace Name
```

```
-------------------------
    1663       pg_default
    1664        pg_global
  155151         fastdisk
  155152          bigdisk


$ # さて、
      データベースalvherreの中を見てみよう
$ cd $PGDATA/base/17228


$ # デフォルトテーブル空間のデータベースオブジェクトを大きさの順に上位10個取得
$ ls -lS * | head -10
-rw-------  1 alvherre alvherre 136536064 sep 14 09:51 155173
-rw-------  1 alvherre alvherre  17965056 sep 14 09:51 1155291
-rw-------  1 alvherre alvherre   1204224 sep 14 09:51 16717
-rw-------  1 alvherre alvherre    581632 sep  6 17:51 1255
-rw-------  1 alvherre alvherre    237568 sep 14 09:50 16674
-rw-------  1 alvherre alvherre    212992 sep 14 09:51 1249
-rw-------  1 alvherre alvherre    204800 sep 14 09:51 16684
-rw-------  1 alvherre alvherre    196608 sep 14 09:50 16700
-rw-------  1 alvherre alvherre    163840 sep 14 09:50 16699
-rw-------  1 alvherre alvherre    122880 sep  6 17:51 16751


$ # ファイル155173は何だろう
$ oid2name -d alvherre -f 155173
From database "alvherre":
  Filenode  Table Name
----------------------
    155173    accounts


$ # 2つ以上のオブジェクトについて問い合わせることもできる
$ oid2name -d alvherre -f 155173 -f 1155291
From database "alvherre":
  Filenode     Table Name
-------------------------
    155173       accounts
   1155291  accounts_pkey


$ # オプションを複数指定することもできて、
      -xではより詳細を得ることができる
$ oid2name -d alvherre -t accounts -f 1155291 -x
```

```
From database "alvherre":
  Filenode     Table Name      Oid  Schema  Tablespace
----------------------------------------------------
    155173       accounts   155173  public  pg_default
   1155291  accounts_pkey  1155291  public  pg_default


$ # 各データベースオブジェクトのディスク容量を表示
$ du [0-9]* |
> while read SIZE FILENODE
> do
>   echo "$SIZE       `oid2name -q -d alvherre -i -f $FILENODE`"
> done
16             1155287  branches_pkey
16             1155289  tellers_pkey
17561             1155291  accounts_pkey
...


$ # 同上、
      ただし大きさの順
$ du [0-9]* | sort -rn | while read SIZE FN
> do
>   echo "$SIZE   `oid2name -q -d alvherre -f $FN`"
> done
133466              155173    accounts
17561             1155291  accounts_pkey
1177               16717  pg_proc_proname_args_nsp_index
...


$ # テーブル空間に何があるのか見たければ、
      pg_tblspcディレクトリを使う
$ cd $PGDATA/pg_tblspc
$ oid2name -s
All tablespaces:
     Oid  Tablespace Name
------------------------
    1663       pg_default
    1664        pg_global
  155151         fastdisk
  155152          bigdisk


$ # テーブル空間"fastdisk"にはどのデータベースのオブジェクトがあるのだろうか
$ ls -d 155151/*
```

```
155151/17228/  155151/PG_VERSION


$ # おや、
      データベース17228がまた出てきた
$ oid2name
All databases:
    Oid  Database Name  Tablespace
----------------------------------
  17228       alvherre  pg_default
  17255     regression  pg_default
  17227      template0  pg_default
      1      template1  pg_default


$ # このデータベースがどのオブジェクトをこのテーブル空間に持っているのか見てみよう
$ cd 155151/17228
$ ls -l
total 0
-rw-------  1 postgres postgres 0 sep 13 23:20 155156


$ # 分かった、これはかなり小さなテーブルだ、、、でも何のテーブルだろう
$ oid2name -d alvherre -f 155156
From database "alvherre":
  Filenode  Table Name
----------------------
    155156         foo
```

作者

B. Palmer <bpalmer@crimelabs.net>

vacuumlo

vacuumlo — PostgreSQLデータベースから孤児となったラージオブジェクトを削除する

概要

vacuumlo [option...] dbname...

説明

vacuumloはPostgreSQLデータベースから「孤児になった」ラージオブジェクトをすべて削除する、単純なユーティリティです。データベース内でoidまたはloデータ型列内にまったく現れないOIDを持つすべてのラージオブジェクト(LO)を「孤児になった」LOとみなします。

これを使用する場合にはまた、loモジュール内のlo_manageトリガに興味を持つかもしれません。lo_manageは初期段階で孤児になったLOの生成を防止しようと試みます。

コマンドラインで指名された全てのデータベースに対して処理が行われます。

オプション

vacuumloは以下のコマンドライン引数を受け付けます。

-l limit
--limit=limit

1トランザクションに付き、limit個（デフォルトは1000）より多くのラージオブジェクトを削除しません。サーバは削除されるLO毎に一つのロックを取得するため、多数のLOの削除を1トランザクションで行う場合、max_locks_per_transactionを超える恐れがあります。もし1トランザクションで全ての削除を行いたい場合は、このlimit値を0に指定してください。

-n
--dry-run

ラージオブジェクトの削除を行わず、単に何が行われるはずかを示します。

-v
--verbose

多くの進行メッセージを出力します。

-V
--version

vacuumloのバージョンを表示し終了します。

-?
--help

vacuumloのコマンドライン引数に関するヘルプを表示し終了します。

vacuumloは接続パラメータとして以下のコマンドライン引数も受け付けます。

-h host
--host=host

データベースサーバのホスト名です。

-p port
--port=port

データベースサーバのポート番号です。

-U username
--username=username

接続ユーザ名です。

-w
--no-password

パスワード入力のプロンプトを出しません。もし、サーバがパスワード認証を必要としており、.pgpassファイルを用いる様な、プロンプト入力とは別の手段を通してパスワードを利用できない場合は、接続に失敗します。このオプションは、バッチ処理やスクリプト処理の様なパスワードを入力するユーザがいないケースで役に立つかもしれません。

-W
--password

vacuumloは強制的にデータベースに接続する前にパスワード入力を促します。

サーバがパスワード認証を要求する場合vacuumloは自動的にパスワード入力を促しますので、これが重要になることはありません。しかし、vacuumloは、サーバにパスワードが必要かどうかを判断するための接続試行を無駄に行います。こうした余計な接続試行を防ぐために-Wの入力が有意となる場合もあります。

環境

PGHOST
PGPORT
PGUSER

デフォルトの接続パラメータ。

このユーティリティは、他の多くPostgreSQLユーティリティと同様に、libpqがサポートする環境変数(33.14参照)も使います。

環境変数PG_COLORは診断メッセージで色を使うかどうかを指定します。可能な値はalways、auto、neverです。

注意

vacuumloは下記の手法で動作します。まずvacuumloは選択されたデータベース内のラージオブジェクトのOIDをすべて含む一時テーブルを構築します。そしてデータベース内でoid型またはlo型を型として持つ全列をスキャンし、一時テーブルから一致する項目を削除します。(注意:これらの名前の型のみが対象となります。特に、これらの型を伴ったドメインなどはスキャン対象にはなりませんので注意が必要です。) 一時テーブルに残った項目を孤児LOと識別します。これらが削除されます。

作者

Peter Mount <peter@retep.org.uk>

G.2. サーバアプリケーション

このセクションでは、contribにあるPostgreSQLのサーバ関連のアプリケーションを説明します。それらは、一般的にはデータベースサーバが稼働するホスト上で実行されます。PostgreSQLのコア配布物に含まれるサーバアプリケーションに関する情報は、PostgreSQLサーバアプリケーションを参照してください。

pg_standby

pg_standby — PostgreSQLウォームスタンバイサーバの作成をサポートする

概要

pg_standby [option...] archivelocation nextwalfile walfilepath [restartwalfile]

説明

pg_standbyは「ウォームスタンバイ」データベースサーバの作成をサポートします。これは、特定の変更が必要となるカスタマイズ可能なテンプレートを持ち、実運用環境で利用可能なプログラムとして設計されています。

pg_standbyは、標準のアーカイブリカバリからウォームスタンバイに切り替えるために必要な待機コマンドrestore_commandとして設計されています。他の設定も必要ですが、それらはすべてメインのサーバマニュアルで説明されています（26.2を参照してください）。

pg_standbyを使用して待機サーバを構築するには、postgresql.conf設定ファイルに以下を追加します。

```
restore_command = 'pg_standby archiveDir %f %p %r'
```

ここでarchiveDirは、リストアすべきWALセグメントファイルが存在するディレクトリです。

通常、%rマクロを使用してrestartwalfileが指定された場合、このファイルより論理的に前のすべてのWALファイルはarchivelocationから削除されます。これによってクラッシュからの再起動ができることを担保しつつ、保持する必要があるファイルの数を最小化します。archivelocationが、この特定の待機サーバで一時使用用の領域である場合、このパラメータの使用は適切です。しかし、archivelocationが長期間のWAL保管を目的とした領域である場合には、不適切となります。

pg_standbyは、archivelocationがサーバを所有するユーザから読み取り可能なディレクトリであることを前提とします。また、restartwalfile（または-k）が指定される場合、archivelocationディレクトリは書き込み可能である必要があります。

マスタサーバが失敗した時の「ウォームスタンバイ」データベースサーバへフェイルオーバーする方法には、以下の2つがあります。

スマートフェイルオーバー

スマートフェイルオーバーは、アーカイブとして利用可能なすべてのWALファイルを適用した後に、待機サーバが準備完了となります。待機サーバが遅れてもデータロスとなることは全くありませんが、適用されていないWALが大量にある場合、待機サーバが利用可能になるまでには長時間かかるかもしれません。スマートフェイルオーバーのトリガとなるためには、単語smartを含むトリガファイルを作成するか、単に空のファイルを作成してください。

ファストフェイルオーバー

ファストフェイルオーバーでは、待機サーバはすぐに準備完了となります。アーカイブ内の未適用のWALファイルは無視され、それらのファイルに記録されていたすべてのトランザクションは失われます。ファ

ストフェイルオーバーのトリガとなるためには、トリガファイルを作成し、単語fastを書き込んでください。また、指定した時間内に新しいWALファイルが出現しない場合に、自動的にファストフェイルオーバーを実行するようにpg_standbyを設定することもできます。

オプション

pg_standbyは、以下のコマンドライン引数を受け付けます。

`-c`

アーカイブからWALファイルをリストアするために cpまたはcopyコマンドを使用します。これが唯一サポートされている動作ですので、このオプションには意味はありません。

`-d`

stderrに大量のデバッグログを出力します。

`-k`

archivelocationからファイルを削除することによって、現在のWALファイルよりも古いWALファイルが、ここで指定した数以上アーカイブ内に保持されないようにします。ゼロ(デフォルト)はarchivelocationからファイルをまったく削除しないことを意味します。restartwalfileが指定された場合、このパラメータは警告なく無視されます。アーカイブ内の正確な切り捨て点を決定する際には、そちらの指定方法の方がより正確だからです。PostgreSQL 8.3の時点では、restartwalfileパラメータによる指定の方が安全、かつ効率的であるため、このパラメータの使用は廃止予定です。あまりにも小さな値を設定すると、待機サーバの再起動に必要とするファイルも削除されてしまう可能性があり、一方であまりに大きな値を設定するとアーカイブ領域を無駄に消費します。

`-r maxretries`

コピーが失敗した場合のリトライ回数の最大値を設定します(デフォルトは3です)。失敗する度に、失敗回数に比例して待ち時間が増加するようにsleeptime * num_retries秒間待機します。そのため、デフォルトでは待機サーバに失敗を返す前に、5秒、10秒、15秒待機することになります。これはリカバリの完了と解釈され、その結果としてスタンバイが完全に起動するでしょう。

`-s sleeptime`

リストアすべきWALがアーカイブ内で見つかるかどうか、その確認をする間隔を秒単位で設定します(最大60秒、デフォルト5秒)。デフォルト設定は必ずしも推奨するものではありません。26.2を参考に検討してください。

`-t triggerfile`

存在すればフェイルオーバー発生のきっかけとなるトリガファイルを指定します。同一システムに複数のサーバが存在する場合、たとえば/tmp/pgsql.trigger.5432などのように構造を持ったファイル名を使用して、どのサーバのトリガか混乱しないようにすることを推奨します。

`-V`
`--version`

pg_standbyのバージョンを表示して終了します。

-w maxwaittime

ファストフェイルオーバー実行後に、次のWALファイルを待機する最大秒数を設定します。ゼロ(デフォルト)に設定することは永久に待機することを意味します。デフォルトの設定は必ずしも推奨されません。26.2を参考にして検討してください。

-?
--help

pg_standbyのコマンドライン引数に関するヘルプを表示して終了します。

注釈

pg_standbyは、PostgreSQL 8.2以降で動作するよう設計されています。

PostgreSQL 8.3は、保持しておく必要がある最後のWALファイルをpg_standbyに渡すための%rマクロを提供しています。PostgreSQL 8.2では、アーカイブファイルの削除が必要な場合-kオプションを使用しなければなりません。このオプションは8.3でもまだ利用可能ですが、非推奨です。

PostgreSQL 8.4は、recovery_end_commandオプションを提供しています。このオプションを指定しないと、残ったトリガファイルが問題を引き起こす可能性があります。

pg_standbyはC言語で書かれており、必要に応じて修正すべき部分が明確に示されているので、修正の容易なソースコードとなっています。

例

LinuxまたはUnixシステムでは以下のように使用できます。

```
archive_command = 'cp %p .../archive/%f'

restore_command = 'pg_standby -d -s 2 -t /tmp/pgsql.trigger.5442 .../archive %f %p %r
 2>>standby.log'

recovery_end_command = 'rm -f /tmp/pgsql.trigger.5442'
```

ここでは、アーカイブディレクトリは物理的には待機サーバ上にあります。そのため、archive_commandはNFS経由でアーカイブディレクトリにアクセスします。しかし、このファイルは(lnの使用を有効にした)待機サーバではローカルです。そのため、以下のようになります。

- standby.logにデバッグ用の出力を書き出します。
- 次のWALファイルが利用可能になったかどうかを確認するまで2秒間待機します。
- /tmp/pgsql.trigger.5442というトリガファイルが出現すると待機状態を解除し、トリガファイルの内容に従ってフェイルオーバーを実行します。
- 復旧が終了した時点で、トリガファイルを削除します。

- 必要なくなったファイルをアーカイブディレクトリから削除します。

Windowsでは以下のように使用できます。

```
archive_command = 'copy %p ...\\archive\\%f'

restore_command = 'pg_standby -d -s 5 -t C:\pgsql.trigger.5442 ...\archive %f %p %r
 2>>standby.log'

recovery_end_command = 'del C:\pgsql.trigger.5442'
```

archive_commandではバックスラッシュを二重にする必要がありますが、
 restore_commandやrecovery_end_commandでは必要ないことに注意してください。これは以下のような内容になります。

- アーカイブからWALファイルをリストアするためにcopyコマンドを使用します。
- standby.logにデバッグ用の出力を書き出します。
- 次のWALファイルが利用可能になったかどうかを確認するまで5秒間待機します。
- C:\pgsql.trigger.5442というトリガファイルが出現すると待機を中止し、トリガファイルの内容に従ってフェイルオーバーを実行します。
- 復旧が終了した時点で、トリガファイルを削除します。
- 必要なくなったファイルをアーカイブディレクトリから削除します。

Windowsのcopyコマンドは、ファイルが完全にコピーされる前に、最終的なファイルサイズを設定します。これは通常pg_standbyを誤動作させます。したがって、pg_standbyは適切なファイルサイズを見てから、いったんsleeptime秒待ちます。GNUWin32のcpは、ファイルコピーが完了した後にだけ、ファイルサイズを設定します。

Windowsの例では両方のサーバでcopyを使用していますので、どちらか一方、または両サーバがネットワーク経由でアーカイブディレクトリにアクセスすることになります。

作者

Simon Riggs <simon@2ndquadrant.com>

関連項目

pg_archivecleanup

付録H 外部プロジェクト

PostgreSQLは複雑なソフトウェアプロジェクトであり、そのプロジェクト管理は困難です。PostgreSQLに対する拡張をコアプロジェクトと分離して開発する方がより効率的であることがわかりました。

H.1. クライアントインタフェース

PostgreSQL基本配布物内には、以下の2つのクライアントインタフェースのみが存在します。

- libpqは基本C言語インタフェースであり、他の多くのクライアントインタフェース構築に必要なため、存在します。
- ECPGはサーバサイドのSQL文法に依存し、PostgreSQL自体の変更に敏感であるため、存在します。

この他の言語についてのインタフェースは外部プロジェクトのもので、別に提供されています。表 H.1にこれらのプロジェクトの一部を示します。パッケージの中にはPostgreSQLと同じライセンスで提供されていないものがあることに注意してください。ライセンスなど各言語インタフェースの詳細についてはそのWebサイトや文書を参照してください。

表H.1 外部管理のクライアントインタフェース

名称	言語	コメント	Webサイト
DBD::Pg	Perl	Perl DBIドライバ	https://metacpan.org/release/DBD-Pg
JDBC	Java	タイプ4 JDBCドライバ	https://jdbc.postgresql.org/
libpqxx	C++	C++インタフェース	https://pqxx.org/
node-postgres	JavaScript	Node.jsドライバ	https://node-postgres.com/
Npgsql	.NET	.NET データプロバイダ	https://www.npgsql.org/
pgtcl	Tcl		https://github.com/flightaware/Pgtcl
pgtclng	Tcl		https://sourceforge.net/projects/pgtclng/
pq	Go	Goのdatabase/sql向けのPure Goドライバ	https://github.com/lib/pq
psqlODBC	ODBC	ODBCドライバ	https://odbc.postgresql.org/
psycopg	Python	DB API 2.0互換	https://www.psycopg.org/

H.2. 管理ツール

PostgreSQLで使用できる管理ツールが複数あります。最も人気のあるのはpgAdmin[1]ですが、複数の商用版もあります。

[1] https://www.pgadmin.org/

H.3. 手続き言語

PostgreSQLの基本配布には複数の手続き言語が存在します。PL/pgSQL、PL/Tcl、PL/PerlおよびPL/Pythonです。

さらに、PostgreSQLのコア配布以外で開発、保守される手続き言語も多く存在します。表 H.2に一部のパッケージを示します。これらのプロジェクトの中には、PostgreSQLと同じライセンスで提供されないものがあることに注意してください。ライセンス情報など各手続き言語の詳細については、そのWebサイトや文書を参照してください。

表H.2 外部管理の手続き言語

名称	言語	Webサイト
PL/Java	Java	https://tada.github.io/pljava/
PL/Lua	Lua	https://github.com/pllua/pllua-ng
PL/R	R	https://github.com/postgres-plr/plr
PL/sh	Unixシェル	https://github.com/petere/plsh
PL/v8	JavaScript	https://github.com/plv8/plv8

H.4. 拡張

PostgreSQLは容易に拡張可能なように設計されています。このため、データベースに読み込まれる拡張は、データベースに組み込まれた機能と同様に働きます。ソースコードに同梱されているcontribディレクトリには複数の拡張が含まれています。付録Fで説明します。他にもPostGIS[2]などが独立して開発されています。PostgreSQLのレプリケーションソリューションですら外部で開発することができます。例えば、人気の高いマスタ/スタンバイレプリケーションツールである Slony-I[3]はコアプロジェクトと独立して開発されています。

[2] https://postgis.net/

[3] https://www.slony.info

付録I ソースコードリポジトリ

PostgreSQLのソースコードはGitバージョン管理システムを使って保存管理されています。マスタのリポジトリの公的なミラーが使用可能です。このミラーは、マスタのリポジトリに変更があればすぐに更新されます。

Wikihttps://wiki.postgresql.org/wiki/Working_with_GitにGitを使用する際の手順が掲載されています。

ソースのリポジトリからPostgreSQLをビルドするにはbison、flexそしてPerlの適度に新しいバージョンが必要であることに注意してください。配布されているtarballからビルドするためには、これらのツールは必要ではありません。なぜならこれらのツールを使ってビルドされるファイルはtarballに含まれているためです。他のツールの必要条件は、16.2に示されているものと同様です。

I.1. Gitを使ってソースを入手する

Gitを使用すると、コードのリポジトリのコピーがローカルマシンに作成されます。よってすべての履歴とブランチにオフラインでアクセスすることができます。これは開発もしくはパッチをテストするには最速で最も柔軟性のある方法です。

Git

1. Gitのインストール版が必要となります。インストール版はhttps://git-scm.comから入手可能です。多くのシステムはデフォルトでGitの比較的新しい版がインストールされているか、パッケージ配布システムにおいて利用可能です。

2. Gitリポジトリを使用するには、公式ミラーサイトのクローンを作成してください。

   ```
   git clone https://git.postgresql.org/git/postgresql.git
   ```

   これは、ローカルマシンにすべてのリポジトリをコピーします。よってインターネット接続が遅い場合には特に時間がかかるでしょう。ファイルは、カレントディレクトリのpostgresqlのサブディレクトリに配置されるでしょう。

   Gitミラーサイトには、Gitプロトコルでも到達できます。URLのプレフィックスをgitに変更して以下のようにしてください。

   ```
   git clone git://git.postgresql.org/git/postgresql.git
   ```

3. システムの最新の更新を入手する場合は、cdを(ローカルマシンの)リポジトリに対して実行し、次のコマンドを実行してください。

   ```
   git fetch
   ```

Gitはソースコードを取得する以外に、もっと多くのことが実行できます。より詳細な情報は、Gitのmanページを参照するか、またはhttps://git-scm.comのウェブサイトを参照してください。

付録Jドキュメント作成

PostgreSQLの文書には4つの主要なフォーマットがあります。

- 平文：インストール前の情報
- HTML：オンラインでの閲覧と参照
- PDF：印刷用
- マニュアルページ：すぐ知りたい時

さらに、PostgreSQLソースツリー全体にわたり、様々な実装に関する問題を記述した平文のREADMEファイルが数多くあります。

HTML文書とマニュアルページは標準の配布物の一部でデフォルトでインストールされます。PDF形式の文書は別途ダウンロードすることで入手できます。

J.1. DocBook

文書のソースは、XMLで定義されたマークアップ言語である*DocBook*で作成されています。ここからは、DocBookとXML両方の用語が使用されますが、技術的に相互の互換性はありません。

DocBookを使用して作成することで、最終的な見栄えがどうなるかに気を遣わずに技術文書の構造と内容を指定できます。ドキュメントスタイルは、いくつかの最終的な形のいずれかにコンテンツをレンダリングする方法を定義します。DocBookはOASISグループ[1]により保守されています。公式DocBookサイト[2]では役に立つ入門用と参照用の文書、そしてO'Reilly社の本の完全版をオンラインで読むことができます。NewbieDoc Docbook Guide[3]は初心者にとりとても役に立ちます。FreeBSDドキュメントプロジェクト[4]でも同様にDocBookを使用していて注目すべき数多くのスタイルに関する指針を含め、役立つ情報があります。

J.2. ツールセット

文書を処理する過程で次のようなツールが使われます。そのうちのいくつかは付記されているように省略しても構いません。

DocBook DTD[5]

> DocBookそのものの定義です。現在はバージョン4.5を使用しており、これより古いまたは新しいバージョンは使用できません。DocBook DTDのSGML対応版ではなく、XML対応版が必要です。

[1] https://www.oasis-open.org

[2] https://www.oasis-open.org/docbook/

[3] http://newbiedoc.sourceforge.net/metadoc/docbook-guide.html

[4] https://www.freebsd.org/docproj/docproj.html

[5] https://www.oasis-open.org/docbook/

DocBook XSL Stylesheets[6]

DocBookのソースをHTMLのような他のフォーマットに変換する処理手順が含まれています。

現在最低必要なバージョンは1.77.0ですが、最良の結果を得るために入手可能な最新の版を使うことをお勧めします。

xmllintのためのLibxml2[7]

このライブラリとそこに含まれるxmllintツールはXMLを処理するのに使われます。PostgreSQLのコードを構築する時にも使われますので、多くの開発者はすでにLibxml2をインストールしているでしょう。しかしながら、xmllintは別のサブパッケージからインストールする必要があるかもしれないことに注意してください。

xsltprocのためのLibxslt[8]

xsltprocはXSLTプロセッサ、すなわち、XSLTスタイルシートを使ってXMLを他のフォーマットに変換するプログラムです。

FOP[9]

これは変換、とりわけXMLからPDFへの変換のためのプログラムです。

文書を作成するために必要な様々なツールのインストール方法についての経験をまとめました。以下に記載します。これらのツールは別にパッケージ化されて配布されていることも考えられます。もしそのような配布物を見つけた場合はdocのメーリングリストに報告してください。そのような情報をここに付け加えたいと思います。

必要なファイルはインターネットからダウンロードされ、ローカルにキャッシュされるので、DocBook XMLとDocBook XSLTスタイルシートをローカルにインストールせずに済ませることもできます。オペレーティングシステムのパッケージで特にスタイルシートの古いバージョンしか提供されていない場合や、パッケージが全く利用できない場合には、実際、これが好ましい解決法でしょう。詳細はxmllintとxsltprocの--nonetオプションを参照してください。

J.2.1. Fedora、RHEL、およびその派生版でのインストール

要求されるパッケージをインストールするには以下のようにしてください。

```
yum install docbook-dtds docbook-style-xsl fop libxslt
```

J.2.2. FreeBSDでのインストール

pkgで必要なパッケージをインストールするには以下を使います。

[6] https://github.com/docbook/wiki/wiki/DocBookXslStylesheets

[7] http://xmlsoft.org/

[8] http://xmlsoft.org/XSLT/

[9] https://xmlgraphics.apache.org/fop/

```
pkg install docbook-xml docbook-xsl fop libxslt
```

提供されているMakefileはFreeBSDのmake用ではありませんので、docディレクトリから文書を作成するにはgmakeを使うことが必要でしょう。

J.2.3. Debianパッケージ

Debian GNU/Linux用の文書作成パッケージの一式が揃っています。インストールには以下を使います。

```
apt-get install docbook-xml docbook-xsl fop libxml2-utils xsltproc
```

J.2.4. macOS

macOSでは、他に追加でインストールしなくても、HTMLとmanドキュメントを構築できます。PDFを構築する、あるいはDocBookのローカルなコピーが必要な場合は、お気に入りのパッケージマネジャーを使って入手できます。

MacPortsを使っているのであれば、以下のようにすれば設定されます。

```
sudo port install docbook-xml-4.5 docbook-xsl fop
```

Homebrewを使っているのであれば、こちらを使ってください。

```
brew install docbook docbook-xsl fop
```

J.2.5. configureによる検出

PostgreSQL本体のプログラムを構築した時のように、文書を構築する際にconfigureスクリプトを実行する必要があります。実行が終わる近辺の次のような出力を確認してください。

```
checking for xmllint... xmllint
checking for DocBook XML V4.5... yes
checking for dbtoepub... dbtoepub
checking for xsltproc... xsltproc
checking for fop... fop
```

xmllintが見つからない時は次のテストのいくつかは省略されます。

J.3. 文書の構築

全ての設定が終了したら、doc/src/sgmlディレクトリに移動して以下のコマンドの1つを実行してください（GNU makeを使うのを忘れずに）。

J.3.1. HTML

HTML形式の文書を作成するには次のようにします。

```
doc/src/sgml$ make html
```

これはデフォルトでの対象です。出力はhtmlサブディレクトリに作成されます。

デフォルトの簡単なスタイルではなくpostgresql.org[10]で使われているスタイルシートのHTMLの文書を作成するには、次のようにします。

```
doc/src/sgml$ make STYLE=website html
```

STYLE=websiteオプションが使われると、生成されるHTMLファイルはpostgresql.org[11]で提供されるスタイルシートへの参照を含み、見るにはネットワークアクセスが必要です。

J.3.2. マニュアルページ

DocBook refentryページをマニュアルページに対応した*roff形式に変換するには、DocBook XSLスタイルシートを使用します。マニュアルページを作成するには次のようにします。

```
doc/src/sgml$ make man
```

J.3.3. PDF

FOPを使ってPDF文書を作成したい時は、対象の用紙のサイズに合わせて以下のコマンドの中から1つを選んでください。

- A4サイズ:

  ```
  doc/src/sgml$ make postgres-A4.pdf
  ```

- U.S.レターサイズ:

  ```
  doc/src/sgml$ make postgres-US.pdf
  ```

PostgreSQL文書はかなり大きいので、FOPはかなりの量のメモリを必要とするでしょう。そのため、システムの中には、メモリに関連するエラーメッセージを出して作成に失敗するものもあるでしょう。これは、設定ファイル~/.foprcのJavaヒープ設定を設定することで回避できます。例えば、

```
# FOP binary distribution
FOP_OPTS='-Xmx1500m'
```

[10] https://www.postgresql.org/docs/current/

[11] https://www.postgresql.org/docs/current/

```
# Debian
JAVA_ARGS='-Xmx1500m'
# Red Hat
ADDITIONAL_FLAGS='-Xmx1500m'
```

必要なメモリの最小量があり、ある程度はメモリが多いほどより速くなるようです。非常に少ないメモリ(1GB以下)のシステムでは、作成はスワップのために非常に遅くなるか、全く動かないでしょう。

他のXSL-FOプロセッサも手動で使えますが、自動化された作成プロセスではFOPだけがサポートされています。

J.3.4. 平文ファイル

より良い表示ツールがない状況で必要とされる場合に備えて、インストールの手順は平文でも配布されています。INSTALLファイルは第16章と連係していて、異なった内容を説明するためにちょっとした変更が加えられています。このファイルを生成するには、doc/src/sgmlディレクトリに移動し**make INSTALL**と入力します。テキスト出力の構築には、追加の構築ツールとしてバージョン1.13以降のPandocが必要です。

以前はリリースノートとリグレッションテストの手順も平文で配布されていましたが、打ち切りになりました。

J.3.5. 構文検証

文書の構築にはとても時間がかかります。でも文書ファイルの正しい構文だけを検証する方法があります。以下のように入力します。ほんの数秒しかかかりません。

```
doc/src/sgml$ make check
```

J.4. 文書の起草

XMLの編集モードを持つエディタを使えばドキュメントソースの編集が非常に便利になります。更にそれがXMLスキーマ言語を理解すると更に便利になり、DocBook構文を考慮することができるようになります。

歴史的な理由により、今はXMLファイルであるドキュメントのソースファイルは.sgml拡張子を持つファイル名称となっていることに注意してください。ですから、正しいモードになるようにエディタ設定を調整する必要があるかも知れません。

J.4.1. Emacs

Emacsと一緒に提供されるnXML Modeは、EmacsでXML文書を編集するためのモードとしてもっとも広く使われています。このモードではEmacsでタグを挿入してマークアップの一貫性をチェックでき、出荷時の状態でDocBookをサポートします。詳細なドキュメントは、nXML manual[12]を確認してください。

src/tools/editors/emacs.samplesにはこのモードで使う推奨設定が含まれています。

[12] https://www.gnu.org/software/emacs/manual/html_mono/nxml-mode.html

J.5. スタイルガイド

J.5.1. リファレンスページ

リファレンスページは標準のレイアウトに従う必要があります。このことにより、ユーザが必要な情報を素早く見つけられるようになり、同時にあるコマンドに関連する全ての特徴を文書化するよう書き手を励ましてもいます。一貫性は、PostgreSQLの各リファレンスページ間だけでなく、オペレーティングシステムや他のパッケージソフトのリファレンスページとの関係でも求められるものです。そのために、以下のガイドラインが作成されました。このガイドラインのほとんどの部分は、様々なオペレーティングシステムで定められている同様のガイドラインと一貫性を持つものです。

実行可能なコマンドを説明するリファレンスページには、以下の節がこの順序で含まれる必要があります。該当しない節は除外しても構いません。これらと同レベルの節は特殊な状況でのみ追加すべきです。そのような情報は多くの場合、「使用方法」節に入れることができます。

名前

この節は自動的に生成されます。ここには、コマンドの名前および機能の簡単な概要が入ります。

概要

この節にはコマンドの構文図が入ります。この概要には、通常、各コマンドラインオプションを記載しません（それらは、以下の節に記載されます）。その代わり、入出力ファイルの宛先などのコマンドラインの主要なコンポーネントを記載します。

説明

コマンドによって何が行われるかを説明する文章です。

オプション

各コマンドラインオプションについて説明するリストです。オプションが数多くある場合、副節を使用することができます。

終了ステータス

成功の場合はゼロを使用し、失敗の場合には非ゼロを使用するプログラムでは、この節を記載する必要はありません。複数の非ゼロの終了コードに異なる意味があれば、ここに記載します。

使用方法

プログラムの副言語またはランタイムインタフェースを全て記載します。プログラムが対話型でない場合、通常はこの節を除外することができます。それ以外の場合、この節は実行時の特徴のすべてを記載するために使用されます。必要に応じて副節を作成してください。

環境

プログラムが使用する可能性のある全ての環境変数を列挙します。なるべく完全なリストを作成してください。SHELLのような些細に見える変数でも、ユーザに必要なことがあります。

ファイル

プログラムが暗黙的にアクセスする可能性のある全てのファイルを列挙します。つまり、コマンドラインで指定された入出力ファイルではなく、設定ファイルなどを列挙するということです。

診断

プログラムが作成する可能性のある全ての異常な出力についての説明を記載します。全てのエラーメッセージを列挙することは避けてください。作業が大変な割に、実際にはほとんど役に立たないからです。ただし、エラーメッセージにユーザが解析できる標準のフォーマットがあれば、ここに記載してください。

注釈

特定の不具合、実装の問題点、セキュリティの考慮事項、互換性の問題など、他の節に該当しない全てのものを記載します。

例

例を記載します。

更新履歴

プログラムの更新履歴に主要な変更点があった場合、ここに列挙します。通常この節は省くことができます。

作者

作者（contrib節のみ使用）

関連項目

次の順序で列挙されるクロスリファレンスです。他のPostgreSQLコマンドのリファレンスページ、PostgreSQLのSQLコマンドのリファレンスページ、PostgreSQLマニュアルの引用、他のリファレンスページ（オペレーティングシステム、他のパッケージソフトなど）、その他の文書。同一グループに属するものは、アルファベット順に列挙します。

SQLコマンドを説明するリファレンスページには、次の節を含める必要があります。名前、概要、説明、パラメータ、使用方法、診断、注釈、例、互換性、更新履歴、関連項目。パラメータ節はオプション節と似ていますが、リストに加えることのできるコマンドの句についてより自由度が高いものです。互換性節では、このコマンドがどの程度まで標準SQLに準拠しているか、または、他のどのデータベースシステムに対して互換性があるかを説明します。SQLコマンドの関連項目節では、プログラムへのクロスリファレンスよりも前にSQLコマンドを記載する必要があります。

付録K PostgreSQLの制限

表 K.1にPostgreSQLの様々なハード制限を示します。しかしながら、絶対的なハード制限に達する前に、パフォーマンスの制限や利用可能なディスク容量などの実際の制限が適用されるかもしれません。

表K.1 PostgreSQLの制限

項目	上限	コメント
データベースの大きさ	無制限	
データベースの数	4,294,950,911	
データベース当たりのリレーション	1,431,650,303	
リレーションの大きさ	32 TB	BLCKSZがデフォルトの8192バイトの場合。
テーブル当たりの行	4,294,967,295ページに収まるタプルの数により制限されます。	
テーブル当たりの列	1600	1ページに収まるタプルの大きさによりさらに制限されます。以下の注意書きを参照してください。
フィールドの大きさ	1 GB	
識別子の長さ	63バイト	PostgreSQLを再コンパイルすることで増やせます。
テーブル当たりのインデックス	無制限	データベース当たりの最大リレーションで制限されます。
インデックス当たりの列	32	PostgreSQLを再コンパイルすることで増やせます。
パーティションキー	32	PostgreSQLを再コンパイルすることで増やせます。

格納されるタプルが8192バイトの1つのヒープページに収まらないといけませんので、テーブル当たりの列の最大数はさらに少なくなります。例えば、タプルヘッダを除いて、1600のintの列は6400バイトを消費しますのでヒープページ1つに収まりますが、1600のbigintの列は12800バイトを消費しますのでヒープページ1つの中には収まりません。text、varchar、charのような可変長の型のフィールドは、その必要があるほど値が長くなれば、行に収まらないその値をテーブルのTOASTテーブルに格納します。18バイトのポインタだけがテーブルのヒープのタプル内に残ります。より短い長さの可変長フィールドでは、4バイトまたは1バイトのフィールドヘッダが使われ、値はヒープタプルの中に格納されます。

テーブルから削除された列も列の上限の一因となります。さらに、新しく作られたタプルに対する削除された列の値は、内部ではタプルのNULLビットマップにNULLと印を付けられますが、NULLビットマップも容量を占めます。

付録L 頭字語

以下はPostgreSQLドキュメントと、PostgreSQLの論議で一般に使用される頭字語の一覧です。

ANSI

American National Standards Institute（米国規格協会）[1]

API

Application Programming Interface（エイ・ピー・アイ）[2]

ASCII

American Standard Code for Information Interchange（情報交換用米国標準コード）[3]

BKI

Backend Interface（バックエンドインタフェース）

CA

Certificate Authority（認証局）[4]

CIDR

Classless Inter-Domain Routing（クラスレスドメイン間ルーティング）[5]

CPAN

Comprehensive Perl Archive Network（シーパン）[6]

CRL

Certificate Revocation List（証明書失効リスト）[7]

CSV

Comma Separated Values（コンマ区切り値）[8]

CTE

Common Table Expression（共通テーブル式）

[1] https://en.wikipedia.org/wiki/American_National_Standards_Institute

[2] https://en.wikipedia.org/wiki/API

[3] https://en.wikipedia.org/wiki/Ascii

[4] https://en.wikipedia.org/wiki/Certificate_authority

[5] https://en.wikipedia.org/wiki/Classless_Inter-Domain_Routing

[6] https://www.cpan.org/

[7] https://en.wikipedia.org/wiki/Certificate_revocation_list

[8] https://en.wikipedia.org/wiki/Comma-separated_values

CVE

Common Vulnerabilities and Exposures（共通脆弱性識別子）[9]

DBA

Database Administrator（データベース管理者）[10]

DBI

Database Interface (Perl)（[Perlの]データベースインタフェース）[11]

DBMS

Database Management System（データベース管理システム）[12]

DDL

Data Definition Language（データ定義言語）[13]、CREATE TABLE、ALTER USERなどのSQLコマンド

DML

Data Manipulation Language（データ操作言語）[14]、INSERT、UPDATE、DELETEなどのSQLコマンド

DST

Daylight Saving Time（夏時間）[15]

ECPG

Embedded C for PostgreSQL（PostgreSQL用の組み込みC）

ESQL

Embedded SQL（組み込みSQL）[16]

FAQ

Frequently Asked Questions（よくある質問）[17]

FSM

Free Space Map（空き領域マップ）

[9] https://cve.mitre.org/

[10] https://en.wikipedia.org/wiki/Database_administrator

[11] https://dbi.perl.org/

[12] https://en.wikipedia.org/wiki/Dbms

[13] https://en.wikipedia.org/wiki/Data_Definition_Language

[14] https://en.wikipedia.org/wiki/Data_Manipulation_Language

[15] https://en.wikipedia.org/wiki/Daylight_saving_time

[16] https://en.wikipedia.org/wiki/Embedded_SQL

[17] https://en.wikipedia.org/wiki/FAQ

GEQO

Genetic Query Optimizer（遺伝的問い合わせオプティマイザ）

GIN

Generalized Inverted Index（汎用転置インデックス）

GiST

Generalized Search Tree（汎用検索ツリー）

Git

Git[18]

GMT

Greenwich Mean Time（グリニッジ標準時）[19]

GSSAPI

Generic Security Services Application Programming Interface（汎用セキュリティサービスアプリケーションプログラミングインタフェース）[20]

GUC

Grand Unified Configuration、サーバ構成処理を行うPostgreSQLのサブシステム

HBA

Host-Based Authentication（ホストベース認証）

HOT

Heap-Only Tuples（ヒープ専用タプル）[21]

IEC

International Electrotechnical Commission（国際電気標準会議）[22]

IEEE

Institute of Electrical and Electronics Engineers（米国電気電子学会）[23]

IPC

Inter-Process Communication（プロセス間通信）[24]

[18] https://en.wikipedia.org/wiki/Git_(software)

[19] https://en.wikipedia.org/wiki/GMT

[20] https://en.wikipedia.org/wiki/Generic_Security_Services_Application_Program_Interface

[21] https://git.postgresql.org/gitweb/?p=postgresql.git;a=blob;f=src/backend/access/heap/README.HOT;hb=HEAD

[22] https://en.wikipedia.org/wiki/International_Electrotechnical_Commission

[23] https://standards.ieee.org/

[24] https://en.wikipedia.org/wiki/Inter-process_communication

ISO

International Organization for Standardization（国際標準化機構）[25]

ISSN

International Standard Serial Number（国際標準逐次刊行物番号）[26]

JDBC

Java Database Connectivity[27]

JIT

Just-in-Time compilation[28]

JSON

JavaScript Object Notation[29]

LDAP

Lightweight Directory Access Protocol[30]

LSN

Log Sequence Number（ログシーケンス番号）、pg_lsnおよびWALの内部参照。

MSVC

Microsoft Visual C[31]

MVCC

Multi-Version Concurrency Control（多版型同時実行制御）

NLS

National Language Support[32]

ODBC

Open Database Connectivity[33]

[25] https://www.iso.org/home.html

[26] https://en.wikipedia.org/wiki/Issn

[27] https://en.wikipedia.org/wiki/Java_Database_Connectivity

[28] https://en.wikipedia.org/wiki/Just-in-time_compilation

[29] https://www.json.org

[30] https://en.wikipedia.org/wiki/Lightweight_Directory_Access_Protocol

[31] https://en.wikipedia.org/wiki/Visual_C++

[32] https://en.wikipedia.org/wiki/Internationalization_and_localization

[33] https://en.wikipedia.org/wiki/Open_Database_Connectivity

OID

Object Identifier（オブジェクト識別子）

OLAP

Online Analytical Processing[34]

OLTP

Online Transaction Processing（オンライントランザクション処理）[35]

ORDBMS

Object-Relational Database Management System（オブジェクトリレーショナルデータベース管理システム）[36]

PAM

Pluggable Authentication Modules（着脱可能認証モジュール）[37]

PGSQL

PostgreSQL

PGXS

PostgreSQL Extension System（PostgreSQL拡張システム）

PID

Process Identifier（プロセス識別子）[38]

PITR

Point-In-Time Recovery（ポイントインタイムリカバリ）（Continuous Archiving - 継続的アーカイブ）

PL

Procedural Languages（手続き言語）（サーバ側）

POSIX

Portable Operating System Interface[39]

RDBMS

Relational Database Management System（リレーショナルデータベース管理システム）[40]

[34] https://en.wikipedia.org/wiki/Olap

[35] https://en.wikipedia.org/wiki/OLTP

[36] https://en.wikipedia.org/wiki/ORDBMS

[37] https://en.wikipedia.org/wiki/Pluggable_Authentication_Modules

[38] https://en.wikipedia.org/wiki/Process_identifier

[39] https://en.wikipedia.org/wiki/POSIX

[40] https://en.wikipedia.org/wiki/Relational_database_management_system

RFC

Request For Comments[41]

SGML

Standard Generalized Markup Language（標準汎用マークアップ言語）[42]

SPI

Server Programming Interface（サーバプログラミングインタフェース）

SP-GiST

Space-Partitioned Generalized Search Tree（空間分割汎用検索ツリー）

SQL

Structured Query Language（構造化問い合わせ言語）[43]

SRF

Set-Returning Function（集合を返す関数）

SSH

Secure Shell[44]

SSL

Secure Sockets Layer[45]

SSPI

Security Support Provider Interface[46]

SYSV

Unix System V[47]

TCP/IP

Transmission Control Protocol (TCP) / Internet Protocol (IP)[48]

[41] https://en.wikipedia.org/wiki/Request_for_Comments

[42] https://en.wikipedia.org/wiki/SGML

[43] https://en.wikipedia.org/wiki/SQL

[44] https://en.wikipedia.org/wiki/Secure_Shell

[45] https://en.wikipedia.org/wiki/Secure_Sockets_Layer

[46] https://msdn.microsoft.com/en-us/library/aa380493%28VS.85%29.aspx

[47] https://en.wikipedia.org/wiki/System_V

[48] https://en.wikipedia.org/wiki/Transmission_Control_Protocol

TID

Tuple Identifier（タプル識別子）

TOAST

The Oversized-Attribute Storage Technique（過大属性格納技法）

TPC

Transaction Processing Performance Council（トランザクション処理性能評議会）[49]

URL

Uniform Resource Locator（統一資源位置指定子）[50]

UTC

Coordinated Universal Time（協定世界時）[51]

UTF

Unicode Transformation Format（ユニコード変換書式）[52]

UTF8

Eight-Bit Unicode Transformation Format（8ビットユニコード変換書式）[53]

UUID

Universally Unique Identifier（汎用一意識別子）

WAL

Write-Ahead Log（ログ先行書き込み）

XID

Transaction Identifier（トランザクション識別子）

XML

Extensible Markup Language（拡張可能マークアップ言語）[54]

[49] http://www.tpc.org/

[50] https://en.wikipedia.org/wiki/URL

[51] https://en.wikipedia.org/wiki/Coordinated_Universal_Time

[52] https://www.unicode.org/

[53] https://en.wikipedia.org/wiki/Utf8

[54] https://en.wikipedia.org/wiki/XML

付録M 用語集

これはPostgreSQLと一般的なリレーショナルデータベースシステムにおける用語とその意味のリストです。

ACID	Atomicity（原子性）、Consistency（一貫性）、Isolation（独立性）、Durability（永続性）。このデータベーストランザクションの性質の集合は、並列操作及び電源障害などによるエラーの際にも正当性を保証することを意図しています。
Aggregate function【集約関数】（ルーチン）	たとえば数え上げ、平均、加算によって複数の入力値をまとめて（集約して）、単一の値を出力する関数。 詳細については9.21を参照してください。 Window function【ウィンドウ関数】（ルーチン）参照
Analytic function【分析関数】	Window function【ウィンドウ関数】（ルーチン）参照
Analyze【アナライズ】（操作）	テーブルと他のリレーションから統計情報データを収集し、問い合わせプランナが問い合わせをどのように実行するかの決定を支援するプロセス。 （この用語を、EXPLAINコマンドのANALYZEオプションと混同しないでください。） 詳細についてはANALYZEを参照してください。
Atomic【原子的】	*datum*との関連においては、値がより小さな構成要素に分解できないこと。 データベーストランザクションとの関連においては、原子性を参照のこと。
Atomicity【原子性】	すべての操作が不可分なものとして完了するか、あるいは何もなかったことになるかのどちらかで終わるトランザクションの性質。加えて、トランザクションの実行中にシステム障害が起きても、リカバリ後には中途半端な結果が見えるようなことはないこと。これはACID特性の一部です。
Attribute【属性】	タプル内にある特定の名前とデータ型を持つ要素。
Autovacuum【自動バキューム】（プロセス）	バキューム及びアナライズ操作を定期的に実行する一連のバックグラウンドプロセス。 詳細については24.1.6を参照してください。
Backend【バックエンド】（プロセス）	クライアントセッションのために活動し、その要求を処理するインスタンスのプロセス。 （この用語を、類似の用語であるバックグラウンドワーカやバックグラウンドライタと混同しないでください。）
Background worker【バックグラウンドワーカ】（プロセス）	システムあるいはユーザが提供するコードを実行するインスタンス内のプロセス。論理レプリケーション、パラレルクエリといったPostgreSQLの

	機能の基盤を供給します。更に、拡張によってカスタムバックエンドワーカプロセスを追加することができます。 詳細については第47章を参照してください。
Background writer【バックグラウンドライタ】(プロセス)	共有メモリからファイルシステムに、変更されたデータページを書き出すプロセス。他のプロセスをブロックしてしまうより大きなI/Oピークを引き起こしてしまうことを避けて、高価なI/O活動を時系列で平均化するために、定期的に目覚めて短い期間だけ活動します。 詳細については19.4.5を参照してください。
Bloat【ブロート】	未使用(自由)空間、あるいは古くなった行バージョンのように、現在の行バージョンを含まないデータページ中の空間。
Cast【キャスト】	*datum*を現在のデータ型から別のデータ型に変換すること。 詳細についてはCREATE CASTを参照してください。
Catalog	標準SQLではこの用語を使用してPostgreSQLで*database*と呼ばれる用語を示します。 (この用語を、*system catalog*と混同しないでください。) 詳細については22.1を参照してください。
Check constraint【チェック制約】	一つ以上の属性の取り得る値が制限されるリレーションに定義される制約の一形式。チェック制約は同じ行内のすべての属性を参照できますが、同じあるいは別のリレーションの別の行は参照できません。 詳細については5.4を参照してください。
Checkpoint【チェックポイント】	一連の*WAL*の中で、そのチェックポイント以前に更新された共有メモリのすべての情報がヒープとインデックスのデータファイルに反映されたことが保証されている点。その点を記録するために、チェックポイントレコードがWALに書き出されます。 また、チェックポイントは、上で定義されているチェックポイントに到達するために必要なすべてのアクションを実行に移すことでもあります。このプロセスはあらかじめ決められた条件、たとえば一定の時間が経過した、またはある量のレコードが書き出されたなどの条件が整うことで開始されます。あるいは、CHECKPOINTコマンドでユーザが起動することもできます。 詳細については29.4を参照してください。
Checkpointer【チェックポインター】(プロセス)	チェックポイントを実行するための特別なプロセス。
Class【クラス】(旧用語)	Relation【リレーション】参照
Client【クライアント】(プロセス)	データベースと情報交換するために、遠隔の可能性があり、インスタンスに接続することによってセッションを確立するすべてのプロセス。

Column【列】

テーブルまたはビューに含まれる属性。

Commit【コミット】

他のトランザクションに対してトランザクションを可視化し、その永続性を保証し、データベース中のトランザクションの終了を実行すること。

詳細についてはCOMMITを参照してください。

Concurrency【並行性】

データベースの中で複数の独立した操作が同時に行われる概念。PostgreSQLにおいては、並行性は複数バージョン並行性制御*(multiversion concurrency control*機構によって制御されます。

Connection【接続】

通常ネットワーク越しにクライアントプロセスとバックエンドプロセスの間で確立された通信回線。セッションをサポートします。この用語は時にセッションの同義語として使われることがあります。

詳細については19.3を参照してください。

Consistency【一貫性】

データベース中のデータが常に一貫性制約*(integrity constraints)*に従う性質。トランザクションは、コミット前には一時的に制約の一部に違反する可能性もありますが、コミット時点までにそうした違反が解決されなければ、トランザクションは自動的にロールバックされます。これはACID特性の一部です。

Constraint【制約】

テーブル中のデータの値、あるいはドメインの属性の制限。

詳細については5.4を参照してください。

Data area【データ領域】

Data directory【データディレクトリ】参照

Database【データベース】

ローカルな*SQL*オブジェクトの名前付き集合。

詳細については22.1を参照してください。

Database cluster【データベースクラスタ】

データベース、グローバルなSQLオブジェクト、そしてそれらの静的あるいは動的なメタデータの集合。時には*クラスタ(cluster)*として参照されます。

PostgreSQLでは、インスタンスを指すために*cluster*という用語も使用されることがあります。(この用語を、SQLコマンドのCLUSTERと混同しないでください。)

Database server【データベースサーバ】

Instance【インスタンス】参照

Data directory【データディレクトリ】

データベースクラスタに関連付けられるすべてのデータファイルとサブディレクトリ(テーブルスペースと、オプションで*WAL*を除く)を含むサーバのファイルシステム上のベースディレクトリ。環境変数PGDATAは、通常データディレクトリを参照するために使用されます。

クラスタのストレージ空間は、データディレクトリに加えてすべての追加のテーブルスペースを含みます。

	詳細については68.1を参照してください。
Data page【データページ】	リレーションデータを格納するための基本的なデータ構造。すべてのページは同じサイズです。データページはそれぞれ特定のファイルに置かれ、典型的にはディスク上に格納され、共有バッファに読み込むことができ、その上で変更を加えることが可能で、その結果変更済み*(dirty)*となります。それらはディスクに書き出されたときにきれい(clean)になります。最初はメモリ上にのみ存在したページも書き出されるまでは変更済みです。
Datum	SQLデータ型の1つの値の内部表現。
Delete	指定したテーブルまたはリレーションから行を削除するSQLコマンド。 詳細についてはDELETEを参照してください。
Durability【永続性】	トランザクションが一旦コミットされると、その変更がシステム故障あるいはクラッシュ後も維持されることの保証。これはACID特性の一部です。
Epoch【エポック】	Transaction ID【トランザクションID】参照
Extension【拡張】	他の機能を入れるためにインスタンスにインストールするソフトウェアの追加パッケージ。 詳細については37.17を参照してください。
File segment【ファイルセグメント】	あるリレーションのデータを格納するための物理ファイル。ファイルセグメントのサイズは設定値（通常1ギガバイト）に制限されます。したがってリレーションサイズがそれを超えると、複数のセグメントに分割されます。 詳細については68.1を参照してください。 （この用語を、類似の用語である*WAL segment*と混同しないでください。）
Foreign data wrapper【外部データラッパ】	あたかもローカルテーブルであるかのように見せるための、ローカルデータベースには含まれないデータの表現手法。外部データラッパを使うと、外部サーバと外部テーブルを定義することができます。 詳細についてはCREATE FOREIGN DATA WRAPPERを参照してください。
Foreign key【外部キー】	テーブルの一つあるいは複数の列に定義される制約の一形式。その制約により、別な（稀に同じ）テーブルの0あるいは1個の行を識別するための列の値が存在することが必要となる。
Foreign server【外部サーバ】	同じ外部データラッパを使い、他の設定値を共通に持つ外部テーブルの名前付きの集合。 詳細についてはCREATE SERVERを参照してください。
Foreign table【外部テーブル】（リレーション）	通常のテーブルと同じように行と列を持つかのように見えるが、外部テーブルの定義に従った構造で結果集合を返す外部データラッパを通じてデータ要求を転送するリレーション。

詳細についてはCREATE FOREIGN TABLEを参照してください。

Fork【フォーク】

リレーションが格納される個々のセグメントファイルの集合。主フォークには、実際のデータが格納されます。また、メタデータのための2つの二次フォークが存在します。フリースペースマップと可視性マップです。*unlogged*リレーションには初期化フォーク*(init fork)*もあります。

Free space map【フリースペースマップ】(フォーク)

テーブルのメインフォークの個々のデータページ関するメタデータを保持する格納構造。個々のページに対応するフリースペースマップのエントリには今後追加されるタプルが使用できるフリースペースの量が格納され、与えられた大きさの新しいタプルで使用できるフリースペースの量を効率的に探索できる構造になっています。

詳細については68.3を参照してください。

Function【関数】(ルーチン)

ゼロ以上の引数を受け取り、ゼロ以上の出力値を返すルーチンの一形式で、一つのトランザクション内で実行されるように制限されています。関数はたとえばSELECTクエリの一部として起動されます。ある関数は集合を返すことができます。そうした関数は集合を返す関数と呼ばれます。

関数はまた、トリガを起動するのにも用いられます。

詳細についてはCREATE FUNCTIONを参照してください。

Grant

ユーザやロールがデータベース内の特定のオブジェクトにアクセスすることを許可するために使われるSQLコマンド。

詳細についてはGRANTを参照してください。

Heap【ヒープ】

リレーションの行の属性(つまりデータ)の値を含む。ヒープは、リレーションのメインフォークの一つ以上のファイルセグメントとして実現されています。

Host【ホスト】

ネットワークを通じて他のコンピュータと通信するコンピュータ。時にはサーバの同義語として用いられます。また、クライアントプロセスを実行するコンピュータを指す用語としても用いられます。

Index【インデックス】(リレーション)

テーブルあるいはマテリアライズドビューから派生したデータを含むリレーション。その内部構造は、元のデータの高速な取り出しとアクセスをサポートします。

詳細についてはCREATE INDEXを参照してください。

Insert

テーブルに新しいデータを追加するために使用されるSQLコマンド。

詳細についてはINSERTを参照してください。

Instance【インスタンス】

共通する共有メモリを使って通信する一群のバックエンドとその他のプロセス。一つの*postmaster*プロセスがインスタンスを管理します。一つのインスタンスは、すべてのデータベースを含む、正確に一つのデータベー

	スクラスタを管理します。TCPポートが重ならない限り、同じサーバ内に多くのインスタンスを走らせることができます。 インスタンスは、ファイルと共有メモリからの読み出しおよび書き込みアクセス、ACID特性の保障、クライアントプロセスへの接続、権限の検証、クラッシュからの回復、レプリケーションその他のDBMSのすべての重要な機能を管理します。
Isolation【独立性】	コミット前にはトランザクションの効果が同時実行するトランザクションから見えない性質。これはACID特性の一部です。 詳細については13.2を参照してください。
Join【結合】	操作の一つであり、複数のリレーションのデータを組み合わせる問い合わせで用いられるSQLキーワード。
Key【キー】	リレーションの一つ以上の属性に含まれる値を使ってテーブルあるいは他のリレーションの行を識別するための手段。
Lock【ロック】	プロセスがあるリソースへの同時アクセスを制限、あるいは阻止できるようにするための機構。
Log file【ログファイル】	ログファイルには事象に関する人間可読なテキスト行が含まれます。例として、ログイン失敗、長期に渡って実行中の問い合わせなどがあります。 詳細については24.3を参照してください。
Logged【ログされる】	*WAL*に変更が送信されると、テーブルはログされると見なされます。デフォルトでは、すべての通常のテーブルはログされます。生成時、あるいはALTER TABLEコマンドで*unlogged*とテーブルを指定することができます。
Logger【ロガー】(プロセス)	有効な場合、そのプロセスは現在のログファイルにデータベースのイベントに関する情報を書き込みます。ある時刻あるいは容量による条件に到達すると、新しいログファイルが作られます。*syslogger*とも呼ばれます。 詳細については19.8を参照してください。
Log record【ログレコード】	*WAL*レコードの旧用語。
Master【マスタ】(サーバ)	Primary【プライマリ】(サーバ)参照
Materialized【マテリアライズド】	ある情報をその場で計算するのではなく、後で使うために前もって計算し、格納する属性。 この用語は、ビュー問い合わせから派生するデータが、そのデータのソースとは別に格納されることを意味する目的で、マテリアライズドビューで用いられます。 またこの用語は、与えられたステップを実行した結果のデータをメモリに格納し(ディスクに吐き出す可能性もあります)、別のステップで複数回読み出されるようにすることを意味する複数ステップの問い合わせを指すためにも使われます。

Materialized view【マテリアライズドビュー】(リレーション)	SELECT文により定義されたリレーション(ビューと同様)ですが、テーブルと同じ方法でデータが格納されます。INSERT、UPDATE、DELETEの操作では変更できません。 詳細についてはCREATE MATERIALIZED VIEWを参照してください。
Multi-version concurrency control【多版型同時実行制御】(MVCC)	複数のトランザクションが、あるプロセスが他のプロセスを停止させることなく同じ行を読み書き可能にするように設計された機構。PostgreSQLでは、タプルが変更されたときに、その複製(版)を作ることによりMVCCが実装されています。古い版を見ることができるプロセスが終了した後、これらの古い版は削除する必要があります。
Null	関係データベース理論の中心的な原理の一つである、存在しないという概念。明示的な値が存在しないことを表現します。
Optimizer【オプティマイザ】	Query planner【問い合わせプランナ】参照
Parallel query【パラレルクエリ】	複数CPUを持つサーバ上の並行プロセスの利点を活かすために、問い合わせの実行の各部分を扱うことが可能である機能。
Partition【パーティション】	大きな集合の中の互いに素な(重ならない)部分集合の一つ。 パーティション化テーブルとの関連では、親*(parent)*であるパーティション化テーブルの一部のデータを持つテーブルを意味します。パーティション自身もテーブルなので、直接問い合わせ対象になります。また、パーティションもパーティション化テーブルになることができるので、階層を作ることができます。 問い合わせの中のウィンドウ関数との関連では、パーティションは問い合わせの結果集合中のどの隣接する行であるかをその関数によって識別するユーザ定義の基準です。
Partitioned table【パーティション化テーブル】(リレーション)	意味論的にはテーブルと同じですが、格納場所が複数のパーティションに渡って分散しているリレーション。
Postmaster(プロセス)	インスタンスの最初期のプロセス。他の補助プロセスを起動して管理し、要求に応じてバックエンドプロセスを生成します。 詳細については18.3を参照してください。
Primary key【主キー】	主キーのどの属性も*null*値を持たないことが保証されているテーブルあるいは他のリレーション上に定義される一意性制約の特殊なケース。その名前から連想されるように、一つのテーブルにはひとつだけ主キーが存在することができます。しかし、NULLにならない属性を持つ複数の一意性制約を持つことも可能です。
Primary【プライマリ】(サーバ)	2つ以上のデータベースがレプリケーションを通じて連携するときに、情報の信頼できるソースであると見なされるサーバはプライマリと呼ばれます。マスタという用語でも知られています。

Procedure【プロシージャ】（ルーチン）

ルーチンの一種。違いは値を返さないことと、COMMITやROLLBACKといったトランザクション文を発行することが許されていることです。CALLコマンドを通じて呼び出されます。

詳細についてはCREATE PROCEDUREを参照してください。

Query【問い合わせ】

通常結果を返す、あるいはデータベース上のデータを変更するためにクライアントからバックエンドに送信される要求。

Query planner【問い合わせプランナ】

問い合わせを実行する最も効率の良い方法を決定する（計画する）ために使われるPostgreSQLの一部分。問い合わせオプティマイザ、オプティマイザ、あるいは単にプランナとしても知られています。

Record

Tuple【タプル】参照

Recycling

WAL file【WALファイル】参照

Referential integrity【参照整合性】

外部キーによってあるリレーションのデータを制限し、他のリレーションに対応するデータが必ず存在しなければならないようにする手段。

Relation【リレーション】

名前と特定の順序で定義された属性のリストを持つ、データベース内の全てのオブジェクトの総称。テーブル、シーケンス、ビュー、外部テーブル、マテリアライズドビュー、複合型、インデックスはすべてリレーションです。

より一般的にはリレーションはタプルの集合です。例えば問い合わせの結果もリレーションです。

PostgreSQLでは、クラスはリレーションの旧用語の同義語です。

Replica【レプリカ】（サーバ）

プライマリデータベースと対になり、プライマリデータベースのデータのある部分、あるいはすべてのコピーを維持するデータベース。これを行う大きな理由は、データへのアクセスを効率化し、プライマリが動作しなくなったときにデータの可用性を維持するためです。

Replication【レプリケーション】

あるサーバのデータの複製をレプリカに作る過程。ここでは、あるサーバのファイルの変更をそのまま複製する物理レプリケーションと、あらかじめ定義したデータの部分集合に対する変更を高レベルの表現を使って転送する論理レプリケーションの2つの形式が可能です。

Result set【結果集合】

SQLコマンドの完了時にバックエンドプロセスからクライアントに送信されるリレーション。SQLコマンドは通常SELECTですが、RETURNING節が指定されればINSERT、UPDATE、DELETEも可能です。

結果集合がリレーションであるということは、問い合わせが他の問い合わせの定義に使用でき、副問合せとなるということです。

Revoke

ロールの名前付きリストに対してデータベースオブジェクトの名前付き集合に対するアクセスを防ぐコマンド。

詳細についてはREVOKEを参照してください。

Role【ロール】

インスタンスに対するアクセス権限の集まり。ロールはそれ自身が他のロールへ与えることのできる権限です。これは利便性、あるいは複数のユーザが同じ権限を必要とする際に漏れがないようにするためにしばしば行われます。

詳細についてはCREATE ROLEを参照してください。

Rollback【ロールバック】

トランザクションの開始以来実行されたすべての操作を取り消すためのコマンド。

詳細についてはROLLBACKを参照してください。

Routine【ルーチン】

データベースシステムに格納され、実行するために起動可能な予め定義された操作の集合。ルーチンは多様なプログラミング言語で記述できます。ルーチンは、関数(集合を返す関数とトリガ関数)を含みます)、集約関数、プロシージャのいずれかです。

多くのルーチンはPostgreSQL自身にすでに含まれていますが、ユーザ定義のルーチンを追加することもできます。

Row【行】

Tuple【タプル】参照

Savepoint【セーブポイント】

トランザクション中の一連のステップ中の特別な印。セーブポイント時点以降のデータ変更は、この時点まで遡って取り消すことができます。

詳細についてはSAVEPOINTを参照してください。

Schema【スキーマ】

スキーマは、同じデータベースに存在するSQLオブジェクトのための名前空間です。各SQLオブジェクトは正確に一つのスキーマに存在する必要があります。

すべてのシステム定義のSQLオブジェクトはpg_catalogスキーマに存在します。

より一般的には、スキーマという用語は、与えられたデータベースあるいはその部分集合中のすべてのデータの記述(テーブル定義、制約、コメントなど)の意味で用いられます。

詳細については5.9を参照してください。

Segment【セグメント】

File segment【ファイルセグメント】参照

Select

データベースに対してデータを要求するためのSQLコマンド。通常SELECTコマンドはデータベースを変更しないものと期待されますが、問い合わせ中で起動される関数がデータを変更する副作用を持つことはあり得ます。

詳細についてはSELECTを参照してください。

Sequence【シーケンス】(リレーション)

値を生成するために使用されるリレーションの一種。通常、生成される値は非反復な連番です。これらは通常、代理主キーの値を生成するために使用されます。

Server【サーバ】

PostgreSQLインスタンスを実行するコンピュータ。サーバという用語は、実際のハードウェア、コンテナ、あるいは仮想マシンを意味します。

この用語は時にはインスタンスあるいはホストに関連して用いられます。

Session【セッション】

接続を通じて通信し、クライアントとバックエンドが関わり合いを持つことが可能な状態。

Shared memory【共有メモリ】

一つのインスタンスに共通のプロセスによって使用されるRAM。*database*ファイルの一部をコピーし、*WAL*レコードのために一時的な領域を提供し、追加の共通情報を格納します。共有メモリは完全なインスタンスに所属し、単一のデータベースには所属しないことに注意してください。

共有メモリの最大の部分は共有バッファとして知られ、ページに分割されてデータファイルのコピーを保持するために使用されます。ページが変更されると、ファイルシステムに書き出されるまではダーティページ(dirty page)と呼ばれます。

詳細については19.4.1を参照してください。

SQL object【SQLオブジェクト】

CREATEコマンドで作られるあらゆるオブジェクト。ほとんどのオブジェクトは一つのデータベースに限定され、ローカルオブジェクトとして一般的に知られています。

ほとんどのローカルオブジェクトは、リレーション（すべての種類）、ルーチン（すべての種類）データ型などのように、データベース中の特定のスキーマに所属します。スキーマの同じ種類の中のそうしたオブジェクト同士は、名前がユニークであることが強制されます。

スキーマに所属しないローカルオブジェクトも存在します。例としては、拡張、データ型キャスト、外部データラッパがあります。データベースの同じ種類の中のそうしたオブジェクト同士は、名前がユニークであることが強制されます。

他のオブジェクト型、たとえばロール、テーブル空間、レプリケーション起点、論理レプリケーションのサブスクライブ、データベース自体は、完全に特定のデータベースの外に存在するので、ローカルSQLオブジェクトではありません。それらはグローバルオブジェクトと呼ばれます。データベースクラスタ全体の同じ種類の中のそうしたオブジェクト同士は、名前がユニークであることが強制されます。

詳細については22.1を参照してください。

SQL standard【標準SQL】

SQL言語を義する一連の文書。

Standby【スタンバイ】（サーバ）

Replica【レプリカ】（サーバ）参照

Stats collector【統計情報収集器】（プロセス）

このプロセスは、*instance*の活動に関する統計情報を収集します。

詳細については27.2を参照してください。

System catalog【システムカタログ】

インスタンスのすべての*SQL*オブジェクトの構造を記述するテーブルの集まり。システムカタログはpg_catalogスキーマに存在します。これらのテーブルは内部表現のデータを格納しているので、典型的にはユーザが調べる目的には適しません。pg_catalogスキーマにもユーザによりわかりやすい多くのビューが提供されており、一部の情報にはより便利なアクセスを提供しています。一方*SQL*標準によって管理されているものと同じあるいはさらに追加の情報を提供するinformation_schemaスキーマ（第36章参照）に追加のテーブルとビューがあります。

詳細については5.9を参照してください。

Table【テーブル】

共通のデータ構造を持つタプルの集合（同じ数の属性が同じ順序で、位置ごとに同じ名前と型を持ちます）。テーブルは、PostgreSQLにおけるリレーションの最も一般的な形式です。

詳細についてはCREATE TABLEを参照してください。

Tablespace【テーブルスペース】

サーバファイルシステムの名前付き場所。すべての*SQL*オブジェクトは、システムカタログ内の定義を超えた格納領域が要求され、単一のテーブルスペースに属している必要があります。最初にデータベースクラスは、単一で使用可能なテーブルスペースが含まれています。それはpg_defaultと呼ばれ、全てのSQLオブジェクトでデフォルトとして使用されます。

詳細については22.6を参照してください。

Temporary table【一時テーブル】

セッションまたはトランザクションのどちらか（作成時に指定します）の存続期間中にのみ存在するテーブル。そのデータは他のセッションからは見られず、ログされることはありません。一時テーブルはしばしば複数ステップ操作の中間データを格納するために使用されます。

詳細についてはCREATE TABLEを参照してください。

TOAST

テーブル行の大きな属性を分割して副テーブルに格納する機構。*TOAST*テーブルと呼ばれます。大きな属性を持つ各リレーションには、独自のTOASTテーブルがあります。

詳細については68.2を参照してください。

Transaction【トランザクション】

単一の原子性コマンドとして動作する必要があるコマンドの組み合わせ。それらは単一の組としてすべて成功かすべて失敗し、トランザクションが完了するまで（分離レベルによってはその後でさえ）、他のセッションからはその効果が見えません。

詳細については13.2を参照してください。

Transaction ID【トランザクションID】

個々のトランザクションが最初にデータベースに変更を加える際に、ユニークな数値である順序数としてアサインされる識別子です。しばしば*xid*と略されます。ディスク上ではxidは32ビット幅しかないので、約4億の書き込みトランザクションIDしか生成できません。それよりも長くシス

	テムが実行できるようにするために、これもまた32ビット幅であるエポックが用いられます。カウンタがxidの最大値に到達すると、xidは3（これよりも小さな値は予約されています）から再開し、エポックの値は1増えます。ときにはエポックとxidの値を組み合わせて、単一の64ビット値として扱うこともあります。 詳細については8.19を参照してください。
Transactions per second (TPS)	1秒あたりに実行されたトランザクションの平均数。測定された実行中にアクティブな全てのセッションで合計されます。これはインスタンスのパフォーマンス特性の指標として使用されます。
Trigger【トリガ】	特定の操作(INSERT、UPDATE、DELETE、TRUNCATE)がリレーションに適用されるたびに実行することを定義できる関数。トリガは、トリガを起動した文と同じトランザクション内で実行されます。関数が失敗すると、起動した文も失敗します。 詳細についてはCREATE TRIGGERを参照してください。
Tuple【タプル】	属性を一定の順序で集めたもの。この順序はタプルが含まれるテーブル（または他のリレーション）によって定義されます。その場合タプルは、しばしば行と呼ばれます。また結果セットの構造によって定義される場合もあります。その場合、タプルはレコードと呼ばれることがあります。
Unique constraint【一意性制約】	リレーションに定義される制約の一形式で、一つ以上の複数の列の組み合わせで許可される値を制限して、各値または組み合わせの値がリレーションの中で一度しか現れないように制限します。— つまり、リレーション内の他の行にそれらと等しい値が含まれないようにします。 *NULL*値は互いに等しいとは見なされないため、一意性制約の違反にはならず、NULL値は複数の行に存在することが許可されます。
Unlogged【ログを取らない】	特定の性質のリレーションで、それらに対する変更が*WAL*に反映されません。これらのリレーションのレプリケーションとクラッシュリカバリは無効になります。 ログを取らないテーブルの主な用途は、プロセス間で共有する必要がある一時的な作業データを格納することです。 一時テーブルは常にログを取りません。
Update	指定されたテーブルに既にある行を修正するために使われるSQLコマンド。行を作成したり削除したりはできません。 詳細についてはUPDATEを参照してください。
User【ユーザ】	LOGIN権限を持つロール。
User mapping【ユーザマッピング】	ローカルデータベース内のログイン認証情報をリモートデータシステム内の外部データラッパによって定義された認証情報に変換すること。

詳細についてはCREATE USER MAPPINGを参照してください。

Vacuum【バキューム】

テーブルまたはマテリアライズドビューから古いものとなったタプルバージョンを削除し、またPostgreSQLの*MVCC*の実装に必要なその他の密接に関連する処理をおこなうプロセス。これはVACUUMコマンドを使用して開始できますが、自動バキュームプロセスを介して自動的に処理することもできます。

詳細については24.1を参照してください。

View【ビュー】

SELECT文によって定義されたリレーションですが、それ自体は格納されません。問い合わせがビューを参照すると、ビューの名前ではなく副問合せとして入力したかのように、ビューの定義が問い合わせに代入されます。

詳細についてはCREATE VIEWを参照してください。

Visibility map【可視性マップ】(フォーク)

テーブルのメインフォークの各データページに関するメタデータ保持する格納構造です。各ページの可視性マップのエントリに2ビットが格納されます。1番目のビット(all-visible)は、ページ内のすべてのタプルがすべてのトランザクションに対して可視であることを示します。2番目のビット(all-frozen)は、ページ内のすべてのタプルが凍結とマークされていることを示します。

WAL

Write-ahead log【ログ先行書き込み】参照

WAL archiver【WALアーカイバ】(プロセス)

バックアップの作成またはレプリカを最新の状態に保つために、*WAL*ファイルのコピーを保持するプロセス。

詳細については25.3を参照してください。

WAL file【WALファイル】

*WAL*セグメントや*WAL*セグメントファイルとしても知られています。*WAL*の格納領域を提供する連番のファイル。ファイルはすべて、事前に定義された同じサイズであり、連続した順序で書き込まれます。複数のセッションで同時に発生する変更が分散しています。システムがクラッシュした場合、ファイルは順番に読み込まれ、各変更が再生されてクラッシュする前の状態にシステムが復元されます。

各WALファイルはチェックポイントがすべての変更を対応するデータファイルをすべて書き込んだ後に解放できます。ファイルを解放するには、削除するか、名前を変えて将来使用できるようにします。これはrecycling(再利用)と呼ばれます。

詳細については29.5を参照してください。

WAL record【WALレコード】

個々のデータの変更を低レベルで記述したもの。システム障害によって変更が失われた場合に、データの変更を再実行(再生)するための十分な情報が含まれています。WALレコードは表示できないバイナリフォーマットを使用します。

詳細については29.5を参照してください。

WAL segment【WALセグメント】

WAL file【WALファイル】参照

WAL writer【WALライタ】（プロセス）

*WAL*レコードを共有メモリから*WAL*ファイルに書き出すプロセス。

詳細については19.5を参照してください。

Window function【ウィンドウ関数】（ルーチン）

問い合わせ内で使用される関数の一種で、問い合わせの結果セットのパーティションに適用されます。この関数の結果は、同じパーティションまたはフレームの行の値を元にしている。

すべての集約関数はウィンドウ関数として使用できますが、ウィンドウ関数は例えば、パーティション内の各行にランク付けすることもできます。分析関数とも呼ばれます。

詳細については3.5を参照してください。

Write-ahead log【ログ先行書き込み】

ユーザおよびシステム起因の操作によるデータベースクラスタ内の変更を追跡するジャーナル。それは、多くの個別の*WAL*レコードが連続して*WAL*ファイルに書き込まれます。

付録N 色対応

PostgreSQLパッケージのたいていのプログラムは、色付けされたコンソール出力が可能です。この付録では、色付けがどのように設定されるかを説明します。

N.1. いつ色が使われるか

色付けされた出力を使うには、環境変数PG_COLORを以下のように設定します。

1. 値がalwaysの場合、色が使われます。
2. 値がautoで標準エラーストリームがターミナルデバイスに関連付けられている場合、色が使われます。
3. それ以外の場合には色は使われません。

N.2. 色を設定する

実際に使われる色は環境変数PG_COLORS（複数形であることに注意）を使って設定されます。この値はコロン区切りのkey=valueのリストです。キー（key）はどこに色が使われるかを示します。値（value）はSGR（Select Graphic Rendition）の記述で、ターミナルにより解釈されます。

現在、以下のキーが使われています。

error

エラーメッセージでテキスト「error」を強調するのに使われます

warning

警告メッセージでテキスト「warning」を強調するのに使われます

locus

メッセージで位置情報を強調するのに使われます（例えば、プログラム名やファイル名）

デフォルト値はerror=01;31:warning=01;35:locus=01です（01;31 = 太字の赤、01;35 = 太字のマゼンタ、01 = 太字のデフォルト色）。

> **ヒント**
>
> この色指定書式は、GCC、GNU coreutils、および、GNU grepなど、他のソフトウェアパッケージでも使われています。

付録O 貢献者

以下の方々に日本語ドキュメント作成に貢献いただきました。ありがとうございました。(PostgreSQL9.6以降の貢献者を記載)

PostgreSQL13

- 翻訳者

  小泉 悟

  斉藤 登

  田中 響

  SRA OSS, Inc. 日本支社 石井 達夫

  SRA OSS, Inc. 日本支社 北山 貴広

  SRA OSS, Inc. 日本支社 高塚 遙

  NTTコムウェア 星合 拓馬

  株式会社アシスト 田中 健一朗

  株式会社スカイアーチHRソリューションズ 橋本 淳一

- その他の貢献

  富士通株式会社 黒田 隼人

PostgreSQL12

- 翻訳者

  小泉 悟

  斉藤 登

  藤井 隆夫

  星合 拓馬

  SRA OSS, Inc. 日本支社 石井 達夫

  SRA OSS, Inc. 日本支社 高塚 遙

  株式会社アシスト 田中 健一朗

- その他の貢献

  大平 直宏

  篠田 典良

SRA OSS, Inc. 日本支社 佐藤 友章

SRA OSS, Inc. 日本支社 矢吹 洋一

NTT OSSセンタ 大塚 憲司

NTT OSSセンタ 坂田 哲夫

NTTコムウェア 山田達朗

株式会社NTTデータ 藤井 雅雄

株式会社ソニックガーデン 西川 一樹

富士通株式会社 黒田 隼人

PostgreSQL11

- 翻訳者

  小泉 悟

  斉藤 登

  星合 拓馬

  SRA OSS, Inc. 日本支社 石井 達夫

  SRA OSS, Inc. 日本支社 千田 貴大

  SRA OSS, Inc. 日本支社 高塚 遙

  NTT OSSセンタ 細谷柚子

  株式会社アシスト 田中 健一朗

- その他の貢献

  azarakko

  上原 一樹

  篠田 典良

  堀田　倫英

  株式会社NTTデータ 藤井 雅雄

  中央大学 遠藤 杏奈

  日本電信電話株式会社 澤田 雅彦

PostgreSQL10

- 翻訳者

小泉 悟

斉藤 登

寺内 大輝

SRA OSS, Inc. 日本支社 石井 達夫

SRA OSS, Inc. 日本支社 高塚 遙

株式会社アシスト 田中 健一朗

株式会社シーエーシー　松田 神一

- その他の貢献

SRA OSS, Inc. 日本支社 佐藤 友章

SRA OSS, Inc. 日本支社 彭 博

PostgreSQL9.6

- 翻訳者

垣谷 学

小泉 悟

斉藤 登

SRA OSS, Inc. 日本支社 石井 達夫

SRA OSS, Inc. 日本支社 千田 貴大

SRA OSS, Inc. 日本支社 高塚 遙

株式会社アシスト 田中 健一朗

株式会社シーエーシー　松田 神一

- その他の貢献

大塚 憲司

小山 哲志

篠田 典良

寺内 大輝

ぬこ@横浜

松枝 智也

SRA OSS, Inc. 日本支社 佐藤 友章

NTT OSSセンタ 澤田 雅彦

NECソリューションイノベータ(株) 近藤 太樹

NECソリューションイノベータ(株) Dang Minh Huong

株式会社NTTデータ 藤井 雅雄

参考文献

厳選されたSQLとPostgreSQLに関する参考文献と図書です。

元となったPOSTGRES開発チームの白書と技術レポートが、カリフォルニア大学バークレイ校コンピュータサイエンス学部のWebサイト[1]にあります。

SQL参考図書

[bowman01] *The Practical SQL Handbook*. Using SQL Variants. Fourth Edition. Judith Bowman, Sandra Emerson, Marcy Darnovsky. ISBN 0-201-70309-2. Addison-Wesley Professional. 2001.

[date97] *A Guide to the SQL Standard* [訳注：翻訳は『標準*SQL*ガイド』、4-7561-2047-4]. A user's guide to the standard database language SQL. Fourth Edition. C. J. Date, Hugh Darwen. ISBN 0-201-96426-0. Addison-Wesley. 1997.

[date04] *An Introduction to Database Systems*. Eighth Edition. C. J. Date. ISBN 0-321-19784-4. Addison-Wesley. 2003.

[elma04] *Fundamentals of Database Systems*. Fourth Edition. Ramez Elmasri, Shamkant Navathe. ISBN 0-321-12226-7. Addison-Wesley. 2003.

[melt93] *Understanding the New SQL* [訳注：改訂版の翻訳は『SQL:1999リレーショナル言語詳解』、4-8947-1531-7]. A complete guide. Jim Melton, Alan R. Simon. ISBN 1-55860-245-3. Morgan Kaufmann. 1993.

[ull88] *Principles of Database and Knowledge-Base Systems*. Classical Database Systems. Jeffrey D. Ullman. Volume 1. Computer Science Press. 1988.

[sqltr-19075-6] *SQL Technical Report*[2]. Part 6: SQL support for JavaScript Object Notation (JSON). First Edition. 2017.

PostgreSQLに特化した文書

[sim98] *Enhancement of the ANSI SQL Implementation of PostgreSQL*. Stefan Simkovics. Department of Information Systems, Vienna University of Technology. Vienna, Austria. November 29, 1998.

[yu95] *The Postgres95. User Manual*. A. Yu, J. Chen. University of California. Berkeley, California. Sept. 5, 1995.

[1] https://dsf.berkeley.edu/papers/

[2] https://standards.iso.org/ittf/PubliclyAvailableStandards/c067367_ISO_IEC_TR_19075-6_2017.zip

[fong] *The design and implementation of the POSTGRES query optimizer*[3]. Zelaine Fong. University of California, Berkeley, Computer Science Department.

会報ならびに記事

[ports12] 「Serializable Snapshot Isolation in PostgreSQL[4]」. D. Ports, K. Grittner. VLDB Conference, August 2012.

[berenson95] 「A Critique of ANSI SQL Isolation Levels[5]」. H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, P. O'Neil. ACM-SIGMOD Conference on Management of Data, June 1995.

[olson93] *Partial indexing in POSTGRES: research project*. Nels Olson. UCB Engin T7.49.1993 O676. University of California. Berkeley, California. 1993.

[ong90] 「A Unified Framework for Version Modeling Using Production Rules in a Database System」. L. Ong, J. Goh. *ERL Technical Memorandum M90/33*. University of California. Berkeley, California. April, 1990.

[rowe87] 「The POSTGRES data model[6]」. L. Rowe, M. Stonebraker. VLDB Conference, Sept. 1987.

[seshadri95] 「Generalized Partial Indexes[7]」. P. Seshadri, A. Swami. Eleventh International Conference on Data Engineering, 6–10 March 1995. Cat. No.95CH35724. IEEE Computer Society Press. Los Alamitos, California. 1995. 420–7.

[ston86] 「The design of POSTGRES[8]」. M. Stonebraker, L. Rowe. ACM-SIGMOD Conference on Management of Data, May 1986.

[ston87a] 「The design of the POSTGRES. rules system」. M. Stonebraker, E. Hanson, C. H. Hong. IEEE Conference on Data Engineering, Feb. 1987.

[ston87b] 「The design of the POSTGRES storage system[9]」. M. Stonebraker. VLDB Conference, Sept. 1987.

[ston89] 「A commentary on the POSTGRES rules system[10]」. M. Stonebraker, M. Hearst, S. Potamianos. *SIGMOD Record 18(3)*. Sept. 1989.

[ston89b] 「The case for partial indexes[11]」. M. Stonebraker. *SIGMOD Record 18(4)*. Dec. 1989. 4–11.

[3] https://dsf.berkeley.edu/papers/UCB-MS-zfong.pdf

[4] https://arxiv.org/pdf/1208.4179

[5] https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-95-51.pdf

[6] https://dsf.berkeley.edu/papers/ERL-M87-13.pdf

[7] https://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.40.5740

[8] https://dsf.berkeley.edu/papers/ERL-M85-95.pdf

[9] https://dsf.berkeley.edu/papers/ERL-M87-06.pdf

[10] https://dsf.berkeley.edu/papers/ERL-M89-82.pdf

[11] https://dsf.berkeley.edu/papers/ERL-M89-17.pdf

[ston90a] 「The implementation of POSTGRES[12]」. M. Stonebraker, L. A. Rowe, M. Hirohama. *Transactions on Knowledge and Data Engineering 2(1)*. IEEE. March 1990.

[ston90b] 「On Rules, Procedures, Caching and Views in Database Systems[13]」. M. Stonebraker, A. Jhingran, J. Goh, S. Potamianos. ACM-SIGMOD Conference on Management of Data, June 1990.

[12] https://dsf.berkeley.edu/papers/ERL-M90-34.pdf

[13] https://dsf.berkeley.edu/papers/ERL-M90-36.pdf

索引

シンボル

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z