

2017年2月11日
PostgreSQL 9.6 対応版

PostgreSQL Internals (1)

日本ヒューレット・パッカード株式会社
篠田典良

謝辞

本資料の作成と公開にあたり、永安悟史様（アップタイム・テクノロジーズ合同会社）、渡部亮太様（株式会社コーソル）にレビューいただきました。アドバイスありがとうございます。日本ヒューレット・パッカード株式会社の社内では高橋智雄さん、北山貴広さん、竹島彰子さん（いずれもテクノロジー事業統括　トランسفォーメーション・コンサルティング本部）にレビューいただきました。ありがとうございます。

PostgreSQL 9.4 対応版の作成にあたり、永安悟史様、渡部亮太様、高橋智雄さん、北山貴広さん、竹島彰子さんにあらためてご意見をいただきました。

PostgreSQL 9.6 対応版の作成にも高橋智雄さん、北山貴広さん、竹島彰子さんにレビューを依頼し、有益なアドバイスをいただきました。ありがとうございます。

オープンソース製品を開発するすべてのエンジニアに感謝します。本文書が少しでも PostgreSQL を利用する皆様の役に立ちますように。

2014 年 7 月 16 日 PostgreSQL 9.3 版

2015 年 3 月 16 日 PostgreSQL 9.4 版

2017 年 2 月 11 日 PostgreSQL 9.6 版

篠田典良

目次

謝辞.....	2
目次.....	3
用語集.....	10
1. 本文書について.....	12
1.1 本文書の目的.....	12
1.2 本文書の対象読者	12
1.3 本文書の範囲.....	12
1.4 本文書の対応バージョン	12
1.5 本文書の更新.....	12
1.6 本文書に対する質問・意見および責任	13
1.7 表記	13
1.7.1 表記の変換.....	13
1.7.2 例の表記	14
2. プロセスとメモリー	16
2.1 プロセス構成.....	16
2.1.1 プロセスの親子関係	16
2.1.2 プロセス名.....	17
2.1.3 プロセスとシグナル	19
2.1.4 プロセスの起動と停止	28
2.1.5 Microsoft Windows 環境のプロセス構成.....	30
2.2 メモリー構成.....	31
2.2.1 共有バッファ概要.....	31
2.2.2 共有バッファの実装	32
2.2.3 Huge Pages.....	32
2.2.4 セマフォ	35
2.2.5 チェックポイント	36
2.2.6 リング・バッファ	38
2.3 インスタンス起動／停止時の動作.....	39
2.3.1 起動／停止の待機.....	39
2.3.2 パラメーターの設定	42
2.3.3 外部ライブラリ	46
2.3.4 インスタンス停止失敗時の動作.....	47
2.3.5 インスタンス起動時の読み込みライブラリ	47
2.3.6 主な入出力ファイル	48

2.3.7 Windows Service 停止時の動作	50
3.ストレージ構成の検証	51
3.1 ファイルシステムの構造	51
3.1.1 ディレクトリ構造	51
3.1.2 データベース・ディレクトリの内部	52
3.1.3 TOAST 機能	55
3.1.4 TRUNCATE 文とファイルの関係	57
3.1.5 FILLFACTOR 属性	59
3.2 テーブル空間	61
3.2.1 テーブル空間とは	61
3.2.2 オブジェクトとファイルの関係	63
3.3 ファイルシステムと動作	67
3.3.1 データベース・クラスターの保護モード	67
3.3.2 ファイルの更新	69
3.3.3 Visibility Map と Free Space Map	71
3.3.4 VACUUM 動作	73
3.3.5 オープン・ファイル	83
3.3.6 プロセスの動作 (WAL の書き込み)	85
3.3.7 プロセスの動作 (checkpointer による書き込み)	88
3.3.8 プロセスの動作 (writer による書き込み)	89
3.3.9 プロセスの動作 (archiver)	89
3.4 オンライン・バックアップ	91
3.4.1 オンライン・バックアップの動作	91
3.4.2 バックアップ・ラベル・ファイル	93
3.4.3 レプリケーションとオンライン・バックアップ	95
3.4.4 オンライン・バックアップとインスタンス停止	96
3.5 ファイルのフォーマット	97
3.5.1 postmaster.pid	97
3.5.2 postmaster.opts	98
3.5.3 PG_VERSION	98
3.5.4 pg_control	99
3.5.5 pg_filenode.map	103
3.6 ブロックのフォーマット	107
3.6.1 ブロックとページ	107
3.6.2 タプル	108
3.7 トランザクション ID の周回問題	111

3.7.1 トランザクション ID.....	111
3.7.2 FREEZE 処理に関するパラメーター	113
3.8 ロケール指定.....	114
3.8.1 ロケールとエンコーディングの指定	114
3.8.2 LIKE によるインデックスの使用.....	117
3.8.3 <>演算子によるインデックスの使用	119
3.8.4 ロケールおよびエンコードの指定	121
3.9 チェックサム.....	122
3.9.1 チェックサムの指定	122
3.9.2 チェックサムの場所	122
3.9.3 チェックサム・エラー	123
3.9.4 チェックサムの有無確認.....	124
3.10 ログファイル.....	125
3.10.1 ログファイルの出力	125
3.10.2 ログファイル名	126
3.10.3 ローテーション	128
3.10.4 ログの内容	129
3.10.5 ログのエンコード.....	130
4. 障害対応.....	133
4.1 インスタンス起動前のファイル削除	133
4.1.1 pg_control 削除.....	133
4.1.2 WAL 削除	133
4.1.3 データファイル削除時の動作（正常終了時）	135
4.1.4 データファイル削除時の動作（クラッシュ時／変更なし）	135
4.1.5 データファイル削除時の動作（クラッシュ時／変更あり）	137
4.1.6 その他のファイル	138
4.2 インスタンス稼働中のファイル削除	139
4.2.1 pg_control 削除.....	139
4.2.2 WAL 削除	139
4.3 プロセス障害.....	141
4.3.1 プロセス異常終了時の再起動	141
4.3.2 プロセス異常終了時のトランザクション	142
4.4 その他の障害.....	143
4.4.1 クラッシュ・リカバリ	143
4.4.2 オンライン・バックアップ中のインスタンス異常終了	143
4.4.3 アーカイブ処理の失敗	144

5. パフォーマンス関連	148
5.1 統計情報の自動収集	148
5.1.1 タイミング	148
5.1.2 条件	148
5.1.3 サンプル・レコード数	148
5.1.4 統計として収集される情報	151
5.1.5 統計情報の保存先	154
5.2 自動 VACUUM	155
5.2.1 タイミング	155
5.2.2 条件	155
5.2.3 autovacuum worker プロセス起動	155
5.2.4 使用メモリー容量	156
5.3 実行計画	158
5.3.1 EXPLAIN 文	158
5.3.2 コスト	159
5.3.3 実行計画	160
5.3.4 実行時間	164
5.3.5 空テーブルのコスト計算	164
5.3.6 ディスクソート	165
5.3.8 テーブル・シーケンシャル・スキャンとインデックス・スキャン	168
5.3.9 BUFFERS 指定	170
5.4 パラメーター	172
5.4.1 パフォーマンスに関連するパラメーター	172
5.4.2 effective_cache_size	172
5.4.3 effective_io_concurrency	172
5.5 システム・カタログ	174
5.5.1 システム・カタログの実体	174
6. SQL 文の仕様	175
6.1 ロック	175
6.1.1 ロックの種類	175
6.1.2 ロックの取得	176
6.2 パーティション・テーブル	177
6.2.1 パーティション・テーブルとは	177
6.2.2 パーティション・テーブルの実装	177
6.2.3 実行計画の確認	179
6.2.4 制約	183

6.2.5 パーティション間のレコード移動	183
6.2.6 パーティション・テーブルと統計情報	184
6.2.7 パーティション・テーブルと外部テーブル	184
6.3 シーケンス	188
6.3.1 シーケンスの使い方	188
6.3.2 キャッシュ	189
6.3.3 トランザクション	191
6.4 バインド変数と PREPARE 文	192
6.5 INSERT ON CONFLICT	194
6.5.1 INSERT ON CONFLICT 文の基本構文	194
6.5.2 ON CONFLICT 句とトリガーの関係	196
6.5.3 ON CONFLICT 句と実行計画	197
6.5.4 ON CONFLICT 句とパーティション・テーブル	198
6.6 TABLESAMPLE	201
6.6.1 概要	201
6.6.2 SYSTEM と BERNOULLI	201
6.6.3 実行計画	203
6.7 テーブルの属性変更	205
6.7.1 ALTER TABLE SET UNLOGGED	205
6.7.2 ALTER TABLE SET WITH OIDS	207
6.7.3 ALTER TABLE MODIFY COLUMN TYPE	208
6.8 ECPG	210
6.8.1 ホスト変数のフォーマット	210
6.8.2 領域不足時の動作	211
6.9 Parallel Query	213
6.9.1 概要	213
6.9.2 実行計画	215
6.9.3 並列処理と関数	216
6.9.4 並列度の計算	220
7. 権限とオブジェクト作成	222
7.1 オブジェクト権限	222
7.1.1 テーブル空間の所有者	222
7.1.2 データベースの所有者	222
7.2 Row Level Security	223
7.2.1 Row Level Security とは	223
7.2.2 準備	223

7.2.3 ポリシーの作成	224
7.2.4 パラメーターの設定	227
8. ユーティリティ	229
8.1 ユーティリティ使用方法	229
8.1.1 pg_basebackup コマンド	229
8.1.2 pg_archivecleanup コマンド	232
8.1.3 psql コマンド	234
8.1.4 pg_resetxlog コマンド	235
8.1.5 pg_rewind コマンド	236
8.1.6 vacuumdb コマンド	239
8.2 ユーティリティの終了ステータス	241
8.2.1 pg_ctl コマンド	241
8.2.2 psql コマンド	241
8.2.3 pg_basebackup コマンド	243
8.2.4 pg_archivecleanup コマンド	243
8.2.5 initdb コマンド	243
8.2.6 pg_isready コマンド	243
8.2.7 pg_recvexlog コマンド	244
9. システム構成	245
9.1 パラメーターのデフォルト値	245
9.1.1 initdb コマンド実行時に導出されるパラメーター	245
9.2 推奨構成	246
9.2.1 ロケール設定	246
9.2.2 推奨パラメーター	246
10. ストリーミング・レプリケーション	248
10.1 ストリーミング・レプリケーションの仕組み	248
10.1.1 ストリーミング・レプリケーションとは	248
10.1.2 ストリーミング・レプリケーションの構成	248
10.2 レプリケーション環境の構築	250
10.2.1 レプリケーション・スロット	250
10.2.2 同期と非同期	253
10.2.3 パラメーター	255
10.2.4 recovery.conf ファイル	256
10.3 フェイルオーバーとスイッチオーバー	259
10.3.1 スイッチオーバー	259
10.3.2 pg_ctl promote コマンド	259

10.3.3 トリガー・ファイルによるマスター昇格	260
10.3.4 障害発生時のログ	261
11. ソースコード構造	262
11.1 ディレクトリ構造	262
11.1.1 トップ・ディレクトリ	262
11.1.2 src ディレクトリ	263
11.2 ビルド環境	263
11.2.1 configure コマンド・パラメータ	263
11.2.2 make コマンド・パラメータ	263
12. Linux オペレーティング・システム設計	265
12.1 カーネル設定	265
12.1.1 メモリー・オーバーコミット	265
12.1.2 I/O スケジューラ	265
12.1.3 SWAP	265
12.1.4 Huge Pages	266
12.1.5 セマフォ	266
12.2 ファイルシステム設定	266
12.2.1 ext4 使用時	266
12.2.2 XFS 使用時	266
12.3 Core ファイル	267
12.3.1 Core ファイル出力設定	267
12.3.2 ABRT による Core 管理	267
12.4 ユーザー制限	269
12.5 systemd 対応	269
12.5.1 サービス登録	269
12.5.2 サービス起動と停止	270
12.6 その他	273
12.6.1 SSH	273
12.6.2 Firewall	273
12.6.3 SE-Linux	273
12.6.4 systemd	273
付録. 参考文献	274
付録 1. 書籍	274
付録 2. URL	275
変更履歴	276

用語集

表 1 略語／用語

略語/用語	説明
ACID 特性	トランザクションが保持すべき特性（Atomicity、Consistency、Isolation、Durability）を示す。
Contrib モジュール	PostgreSQL の拡張モジュールを差す。標準で使用できる Contrib モジュールの一覧はマニュアル「Appendix F. Additional Supplied Modules ¹ 」に掲載されている。
ECPG	PostgreSQL が提供する埋め込み SQL 開発のためのプリプロセッサ
EnterpriseDB	Postgres Plus を開発／販売している会社
GUC	PostgreSQL のパラメーターが保存されるメモリー領域（Global Unified Configuration）
OID (オブジェクト ID)	データベース内部で作成されるオブジェクトを識別する ID で、符号なし 32 ビット値を持つ。
PL/pgSQL	PostgreSQL のストアド・プロシージャ記述言語のひとつ Oracle Database の PL/SQL とある程度互換性がある。
Postgres Plus	PostgreSQL をベースにした商用データベース製品
PostgreSQL	オープンソースデータベース製品
psql	PostgreSQL に付属する SQL 文を実行するためのユーティリティ
TID (Tuple ID)	テーブル内のレコードを一意に示す ID。レコードの物理位置を示す。
WAL	PostgreSQL のトランザクション・ログ（Write Ahead Logging）ファイル
XID (トランザクション ID)	トランザクションを一意に識別する ID、レコードの新旧を識別する符号なし 32 ビット値
アーカイブログ	リカバリーに使用される WAL のコピー

¹ <https://www.postgresql.org/docs/9.6/static/contrib.html>

表 1 (続) 略語／用語

略語/用語	説明
システム・カタログ	PostgreSQL データベース全体のメタ情報を格納している領域
タプル	テーブル内のレコードを示す
データベース・クラスター	PostgreSQL データベース全体の管理情報が格納されているディレクトリ
リレーション	テーブルをリレーションと呼ぶ場合がある
テーブル空間	オブジェクトが格納されるファイルシステム上のディレクトリ。表領域と呼ばれる場合もある。

1. 本文書について

1.1 本文書の目的

本文書は PostgreSQL を利用するエンジニア向けに、PostgreSQL の内部構造やマニュアルに記載されていない動作に関する知識を提供することを目的としています。

1.2 本文書の対象読者

本文書は、既にある程度 PostgreSQL に関する知識を持っているエンジニア向けに記述しています。インストール、基本的な管理等は実施できることを前提としています。

1.3 本文書の範囲

本文書の記述範囲は PostgreSQL が使用するストレージの内部構造や、マニュアルには記載されていない内部動作の検証が中心です。作成者が独習用に調査した結果をまとめた資料であるため、技術レベルや網羅性にはばらつきがあります。

1.4 本文書の対応バージョン

本文書は原則として以下のバージョンを検証の対象としています。

表 2 対象バージョン

種別	バージョン	備考
データベース	PostgreSQL 9.6	9.6.2
オペレーティング・システム	Red Hat Enterprise Linux 7 Update 1 (x86-64)	3.10.0-229
	Microsoft Windows Server 2008 R2	一部

1.5 本文書の更新

本文書は要望があれば更新する予定ですが、時期や更新内容は決定していません。

1.6 本文書に対する質問・意見および責任

本文書の内容は日本ヒューレット・パッカード株式会社の公式見解ではありません。また内容の間違いにより生じた問題について作成者および所属会社は責任を負いません。本文書に対するご意見、ご感想等については日本ヒューレット・パッカード株式会社 テクノロジーコンサルティング事業統括 篠田典良 (noriyoshi.shinoda@hpe.com) までお知らせください。

1.7 表記

1.7.1 表記の変換

中括弧 {} で囲まれた部分は、何等かの文字列に変換されることを示しています。以下の表記を使用します。

表 3 表記

表記	説明	例
{999999}	任意の数字列	16495
{9}	一桁の数字	1
{ARCHIVEDFILE}	アーカイブログ・ファイル名	00000001000000000000A8
{ARCHIVEDIR}	アーカイブログ出力用ディレクトリ	/usr/local/pgsql/archive
{BACKUPLABEL}	バックアップ処理時に指定されるラベル文字列	pg_basebackup base backup
{BGWORKER}	カスタム Worker プロセス名	custom_worker
{DATE}	日付と時刻	2017-02-11_122532
{HOME}	psql コマンド実行ユーザーのホーム・ディレクトリ	/home/postgres
{INSTALL}	PostgreSQL インストール・ディレクトリ	/usr/local/pgsql
{MODULENAME}	Contrib モジュールの名前	auto_explain
{OID}	任意の OID 番号	12993
{PARAMETER}	パラメーターの名前	log_checkpoints
{PASSWORD}	表示されないパスワード	secret
{PGDATA}	データベース・クラスター用ディレクトリ	/usr/local/pgsql/data

表 3 表記（続）

表記	説明	例
{PGDATABASE}	データベース名	datadb1
{PGUSER}	接続ユーザー名	user1
{PID}	プロセス ID	3468
{PORT}	接続待ちポート番号	5432
{RELFilenode}	テーブルに対応するファイル名、pg_class カタログの relfilename 列に対応	16531
{SERVICENAME}	サービス名	postgresql-9.6.2.service
{SLOT}	レプリケーション・スロット名	slot_1
{SOURCE}	パラメーター設定元のマクロ	
{SQL}	任意の SQL 文	SELECT * FROM table1
{TABLE}	任意のテーブル名	table1
{TABLESPACEDIR}	テーブル空間用ディレクトリ	/usr/local/pgsql/ts1
{TCP/IP (PORT)}	クライアントの TCP/IP アドレスとポート番号	192.168.1.100(65327)
{VERSION}	バージョン番号	9.6
{WALFILE}	WAL ファイル名	000000010000000000000000B0
{WALOFFSET}	WAL オフセット	5225832
{YYYYMMDDN}	フォーマット番号	201608131
\${文字列}	環境変数が展開されることを示す	\${PGDATA}

1.7.2 例の表記

本文書内にはコマンドや SQL 文の実行例が含まれます。例は以下の表記で記載しています。

表 4 例の表記

表記	説明
#	Linux root ユーザーのプロンプト
\$	Linux 一般ユーザーのプロンプト
太字	ユーザーが入力する文字列
postgres=#	PostgreSQL 管理者が利用する psql プロンプト
postgres=>	PostgreSQL 一般ユーザーが利用する psql プロンプト
backend>	スタンダロン・モードのプロンプト
<<途中省略>>	画面出力の中間部分を省略していることを示す。
<<以下省略>>	画面出力の後半部分を省略していることを示す。

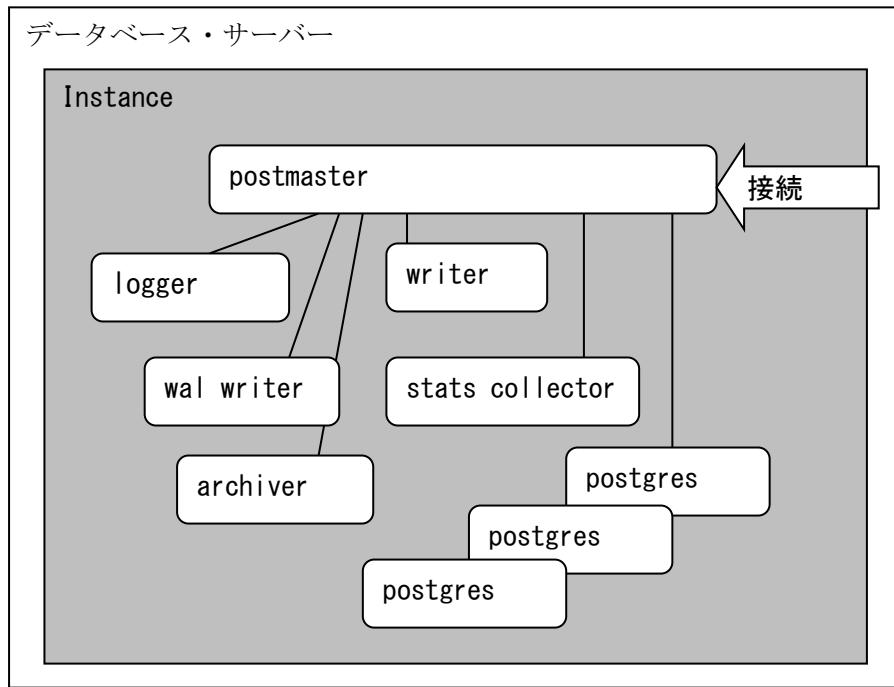
2. プロセスとメモリー

2.1 プロセス構成

2.1.1 プロセスの親子関係

PostgreSQL のプロセス構成は、postmaster²を親プロセスとした、複数のバックエンド・プロセスから構成されます。postmaster プロセスのプロセス ID は、{PGDATA}/postmaster.pid ファイルに記録されます。インスタンスが正常に停止した場合にはこのファイルは削除されます。クライアントは postmaster プロセスがリッスンするポートに対して接続を行います。

図 1 プロセスの親子関係



下記の例ではプロセス ID 2680 が postmaster プロセスになります。その他のプロセスはすべて postmaster プロセスの子プロセスであることがわかります。postmaster プロセスはクライアントからの接続を受けて認証を行い、SQL 文を実行する子プロセスとして postgres プロセスを起動します。

² すべてのプロセスの親となる postgres プロセスを歴史的な経緯で postmaster と呼んでいます。

例 1 プロセス構造の確認

```
$ ps -ef | grep postgres | grep -v grep
postgres 2680      1  0 10:25 ?    00:00:00 /usr/local/pgsql/bin/postgres -D
/usr/local/pgsql/data
postgres 2681 2680  0 10:25 ?    00:00:00 postgres: logger process
postgres 2683 2680  0 10:25 ?    00:00:00 postgres: checkpointer process
postgres 2684 2680  0 10:25 ?    00:00:00 postgres: writer process
postgres 2685 2680  0 10:25 ?    00:00:00 postgres: wal writer process
postgres 2686 2680  0 10:25 ?    00:00:00 postgres: autovacuum launcher process
postgres 2687 2680  0 10:25 ?    00:00:00 postgres: stats collector process
```

2.1.2 プロセス名

PostgreSQL インスタンスは前述の通り複数のプロセスから構成されます。ps コマンド等で参照した各プロセスの名称は以下の通りになります。パラメーター update_process_title を on に指定することでプロセス名の一部が変化します（デフォルト値 on）。

表 5 プロセス名

プロセス	プロセス名
postmaster	{INSTALL}/bin/postgres -D {PGDATA}
logger	postgres: logger process
checkpointer	postgres: checkpointer process
writer	postgres: writer process
wal writer	postgres: wal writer process
autovacuum launcher	postgres: autovacuum launcher process
autovacuum worker	postgres: autovacuum worker process {PGDATABASE}
archiver	postgres: archiver process last was {ARCHIVEDFILE}
stats collector	postgres: stats collector process
postgres (local)	postgres: {PGUSER} {PGDATABASE} [local] {SQL}
postgres (remote)	postgres: {PGUSER} {PGDATABASE} {TCP/IP (PORT)} {SQL}
wal sender	postgres: wal sender process {PGUSER} {TCP/IP (PORT)} streaming {WALFILE}
	postgres: wal sender process {PGUSER} {TCP/IP (PORT)} sending backup “{BACKUP_LABEL}”
wal receiver	postgres: wal receiver process streaming {WALFILE}
startup process	postgres: startup process recovering {WALFILE}
bgworker	postgres: bgworker: {BGWORKER}
parallel worker	postgres: bgworker: parallel worker for PID {PID}

PostgreSQL 9.5 の新機能であるパラメーターcluster_name を指定すると、プロセス名の一部に指定された文字列が出力されます。下記はパラメーターcluster_name にcluster1を指定した例です。

例 2 パラメーターcluster_name

```
$ ps -ef | grep postgres
postgres 12364      1  0 06:14 pts/0    00:00:00 /usr/localpgsql/bin/postgres -D data
postgres 12365 12364  0 06:14 ?        00:00:00 postgres: cluster1: logger process
postgres 12367 12364  0 06:14 ?        00:00:00 postgres: cluster1: checkpointer process
postgres 12368 12364  0 06:14 ?        00:00:00 postgres: cluster1: writer process
postgres 12369 12364  0 06:14 ?        00:00:00 postgres: cluster1: wal writer process
postgres 12370 12364  0 06:14 ?        00:00:00 postgres: cluster1: autovacuum launcher process
postgres 12371 12364  0 06:14 ?        00:00:00 postgres: cluster1: stats collector process
```

例からわかるように postmaster プロセスのプロセス名にはクラスターネームは出力されません。パラメーター cluster_name に指定できる文字は ASCII 文字列（0x20～0x7E）のみです。これ以外のコードはクエスチョン・マーク (?) に変換されて出力されます。

2.1.3 プロセスとシグナル

インスタンスを構成するバックエンド・プロセスに特定のシグナルを送信することでアクションを発生させることができます。ここではいくつかのシグナルを送信した場合の動作について検証しています。

□ SIGKILL シグナル

postmaster プロセスが KILL シグナルを受信した場合には子プロセスも含めて全プロセスが異常終了します。この際に postmaster.pid ファイルは削除されません。再起動時には以下のログが記録されますが、インスタンス自体は正常に起動します。

例 3 異常終了後の再起動ログ

```
LOG: database system was interrupted; last known up at 2017-02-11 11:12:03 JST
LOG: database system was not properly shut down; automatic recovery in progress
LOG: redo starts at 0/155E118
FATAL: the database system is starting up
FATAL: the database system is starting up
LOG: invalid record length at 0/5A5C050: wanted 24, got 0
LOG: redo done at 0/5A5C018
LOG: last completed transaction was at log time 2017-02-11 12:25:15.443492+09
LOG: MultiXact member wraparound protections are now enabled
LOG: autovacuum launcher started
LOG: database system is ready to accept connections
```

postgres プロセスが異常終了（KILL シグナル受信を含む）すると、該当するプロセスだけでなく、インスタンスに接続された全セッションがリセットされます。インスタンス上で実行中のトランザクションはすべてロールバックされ、シグナル受信直後の SQL 文はすべてエラーになります。postgres プロセスを安全に停止させるには、`pg_cancel_backend` 関数（SIGINT シグナルを送信）、または `pg_terminate_backend` 関数（SIGTERM シグナルを送信）を実行します。

例 4 KILL シグナル受信後のログ

```

LOG: server process (PID 3416) was terminated by signal 9: Killed
LOG: terminating any other active server processes
LOG: archiver process (PID 3404) exited with exit code 1
WARNING: terminating connection because of crash of another server process
DETAIL: The postmaster has commanded this server process to roll back the
current transaction and exit, because another server process exited abnormally
and possibly corrupted shared memory.
HINT: In a moment you should be able to reconnect to the database and repeat
your command.
LOG: all server processes terminated; reinitializing

```

各バックエンド・プロセスがシグナルを受信した場合の動作は以下の通りです。

SIG_IGN はシグナル無視、SIG_DFL は Linux プロセスのデフォルトの動作を示します。

- postgres プロセスのシグナル受信時の動作
postgres プロセスのシグナル受信時の動作は以下の通りです。

表 6 postgres プロセスの動作

シグナル	ハンドラー	動作
SIGHUP	SigHupHandler	設定ファイルの再読み込み
SIGINT	StatementCancelHandler	実行中のトランザクションの破棄 (pg_cancel_backend 関数の処理)
SIGTERM	die	トランザクション破棄とプロセス終了 (pg_terminate_backend 関数の処理)
SIGQUIT	quickdie または die	強制終了
SIGALRM	handle_sig_alarm	タイムアウト発生通知
SIGPIPE	SIG_IGN	
SIGUSR1	procsignal_sigusr1_handler	リカバリー処理
SIGUSR2	SIG_IGN	
SIGFPE	FloatExceptionHandler	ERROR ログ出力
SIGCHLD	SIG_DFL	

- autovacuum launcher プロセスのシグナル受信時の動作
autovacuum launcher プロセスのシグナル受信時の動作は以下の通りです。

表 7 autovacuum launcher プロセスの動作

シグナル	ハンドラー	動作
SIGHUP	av_sighup_handler	設定ファイルの再読み込み
SIGINT	StatementCancelHandler	実行中のトランザクションの破棄
SIGTERM	avl_sigterm_handler	正常終了
SIGQUIT	quickdie	ログ出力+強制終了
SIGALRM	handle_sig_alarm	タイムアウト発生通知
SIGPIPE	SIG_IGN	
SIGUSR1	procsignal_sigusr1_handler	リカバリー処理
SIGUSR2	avl_sigusr2_handler	autovacuum worker 終了処理
SIGFPE	FloatExceptionHandler	ERROR ログ出力
SIGCHLD	SIG_DFL	

- bgworker プロセスのシグナル受信時の動作

bgworker プロセスのシグナル受信時の動作は以下の通りです。

表 8 bgworker プロセスの動作

シグナル	ハンドラー	動作
SIGHUP	SIG_IGN	
SIGINT	StatementCancelHandler	実行中のトランザクションの破棄
	SIG_IGN	
SIGTERM	bgworker_die	FATAL エラーログの出力
SIGQUIT	bgworker_quickdie	強制終了
SIGALRM	handle_sig_alarm	タイムアウト発生通知
SIGPIPE	SIG_IGN	
SIGUSR1	procsignal_sigusr1_handler	リカバリー処理
	bgworker_sigusr1_handler	latch_sigusr1_handler 関数をコール
SIGUSR2	SIG_IGN	
SIGFPE	FloatExceptionHandler	ERROR ログ出力
	SIG_IGN	
SIGCHLD	SIG_DFL	

- writer プロセスのシグナル受信時の動作

writer プロセスのシグナル受信時の動作は以下の通りです。

表 9 writer プロセスの動作

シグナル	ハンドラー	動作
SIGHUP	BgSigHupHandler	設定ファイルの再読み込み
SIGINT	SIG_IGN	
SIGTERM	ReqShutdownHandler	正常終了
SIGQUIT	bg_quickdie	異常終了
SIGALRM	SIG_IGN	
SIGPIPE	SIG_IGN	
SIGUSR1	bgwriter_sigusr1_handler	latch_sigusr1_handler 関数をコール
SIGUSR2	SIG_IGN	
SIGCHLD	SIG_DFL	
SIGTTIN	SIG_DFL	
SIGTTOU	SIG_DFL	
SIGCONT	SIG_DFL	
SIGWINCH	SIG_DFL	

- checkpointer プロセスのシグナル受信時の動作

checkpointer プロセスのシグナル受信時の動作は以下の通りです。

表 10 checkpointer プロセスの動作

シグナル	ハンドラー	動作
SIGHUP	ChkptSigHupHandler	設定ファイルの再読み込み
SIGINT	ReqCheckpointHandler	チェックポイントの実行リクエスト
SIGTERM	SIG_IGN	
SIGQUIT	chkpt_quickdie	異常終了
SIGALRM	SIG_IGN	
SIGPIPE	SIG_IGN	
SIGUSR1	chkpt_sigusr1_handler	latch_sigusr1_handler 関数をコール
SIGUSR2	ReqShutdownHandler	WAL のクローズと正常終了
SIGCHLD	SIG_DFL	
SIGTTIN	SIG_DFL	
SIGTTOU	SIG_DFL	
SIGCONT	SIG_DFL	
SIGWINCH	SIG_DFL	

checkpointer プロセスに SIGINT シグナルを送信すると、チェックポイントが実行されます。ただしこの方法ではパラメーター log_checkpoints を on に指定してもログが出力されません。pg_stat_bgwriter カタログは更新されます。

- stats collector プロセスのシグナル受信時の動作

stats collector プロセスのシグナル受信時の動作は以下の通りです。

表 11 stats collector プロセスの動作

シグナル	ハンドラー	動作
SIGHUP	pgstat_sighup_handler	設定ファイルの再読み込み
SIGINT	SIG_IGN	
SIGTERM	SIG_IGN	
SIGQUIT	pgstat_exit	正常終了
SIGALRM	SIG_IGN	
SIGPIPE	SIG_IGN	
SIGUSR1	SIG_IGN	
SIGUSR2	SIG_IGN	
SIGCHLD	SIG_DFL	
SIGTTIN	SIG_DFL	
SIGTTOU	SIG_DFL	
SIGCONT	SIG_DFL	
SIGWINCH	SIG_DFL	

- postmaster プロセスのシグナル受信時の動作

postmaster プロセスのシグナル受信時の動作は以下の通りです。

表 12 postmaster プロセスの動作

シグナル	ハンドラー	動作
SIGHUP	SIGHUP_handler	設定ファイルの再読み込み 子プロセスに SIGHUP シグナル送信
SIGINT	pmdie	FAST シャットダウン
SIGTERM	pmdie	SMART シャットダウン
SIGQUIT	pmdie	IMMEDIATE シャットダウン
SIGALRM	SIG_IGN	
SIGPIPE	SIG_IGN	
SIGUSR1	sigusr1_handler	子プロセスからのシグナル受信処理
SIGUSR2	dummy_handler	何もしない
SIGCHLD	reaper	子プロセス終了時の処理 バックエンド・プロセスの再起動
SIGTTIN	SIG_IGN	
SIGTTOU	SIG_IGN	
SIGXFSZ	SIG_IGN	

postmaster プロセスに SIGHUP シグナルを送信すると、postgresql.conf ファイル (postgresql.auto.conf、pg_*.conf ファイルも) の再読み込みが行われます。これは pg_ctl reload コマンドの実行と同じです。以下のログが出力されます。

例 5 設定ファイルの再読み込み

```
LOG: received SIGHUP, reloading configuration files
```

- startup プロセスのシグナル受信時の動作

startup プロセスのシグナル受信時の動作は以下の通りです。

表 13 startup プロセスの動作

シグナル	ハンドラー	動作
SIGHUP	StartupProcSigHupHandler	設定ファイルの再読み込み
SIGINT	SIG_IGN	
SIGTERM	StartupProcShutdownHandler	プロセス終了
SIGQUIT	startupproc_quickdie	異常終了
SIGALRM	handle_sig_alarm	タイムアウト発生通知
SIGPIPE	SIG_IGN	
SIGUSR1	StartupProcSigUsr1Handler	latch_sigusr1_handler 関数をコール
SIGUSR2	StartupProcTriggerHandler	リカバリーを終了,マスターにプロモート
SIGCHLD	SIG_DFL	
SIGTTIN	SIG_DFL	
SIGTTOU	SIG_DFL	
SIGCONT	SIG_DFL	
SIGWINCH	SIG_DFL	

- logger プロセスのシグナル受信時の動作

logger プロセスのシグナル受信時の動作は以下の通りです。

表 14 logger プロセスの動作

シグナル	ハンドラー	動作
SIGHUP	sigHupHandler	設定ファイルの再読み込み ログ設定の再確認とディレクトリ作成
SIGINT	SIG_IGN	
SIGTERM	SIG_IGN	
SIGQUIT	SIG_IGN	
SIGALRM	SIG_IGN	
SIGPIPE	SIG_IGN	
SIGUSR1	sigUsr1Handler	ログのローテーション実行
SIGUSR2	SIG_IGN	
SIGCHLD	SIG_DFL	
SIGTTIN	SIG_DFL	
SIGTTOU	SIG_DFL	
SIGCONT	SIG_DFL	
SIGWINCH	SIG_DFL	

- wal writer プロセスのシグナル受信時の動作

wal writer プロセスのシグナル受信時の動作は以下の通りです。

表 15 wal writer プロセスの動作

シグナル	ハンドラー	動作
SIGHUP	WalSigHupHandler	設定ファイルの再読み込み
SIGINT	WalShutdownHandler	正常終了
SIGTERM	WalShutdownHandler	正常終了
SIGQUIT	wal_quickdie	異常終了
SIGALRM	SIG_IGN	
SIGPIPE	SIG_IGN	
SIGUSR1	walwriter_sigusr1_handler	latch_sigusr1_handler 関数をコール
SIGUSR2	SIG_IGN	
SIGCHLD	SIG_DFL	
SIGTTIN	SIG_DFL	
SIGTTOU	SIG_DFL	
SIGCONT	SIG_DFL	
SIGWINCH	SIG_DFL	

- archiver プロセスのシグナル受信時の動作

archiver プロセスのシグナル受信時の動作は以下の通りです。

表 16 archiver プロセスの動作

シグナル	ハンドラー	動作
SIGHUP	ArchSigHupHandler	設定ファイルの再読み込み
SIGINT	SIG_IGN	
SIGTERM	ArchSigTermHandler	正常終了
SIGQUIT	pgarch_exit	強制終了
SIGALRM	SIG_IGN	
SIGPIPE	SIG_IGN	
SIGUSR1	pgarch_waken	アーカイブ処理
SIGUSR2	pgarch_waken_stop	アーカイブ処理停止
SIGCHLD	SIG_DFL	
SIGTTIN	SIG_DFL	
SIGTTOU	SIG_DFL	
SIGCONT	SIG_DFL	
SIGWINCH	SIG_DFL	

2.1.4 プロセスの起動と停止

checkpoint、writer、stats collector プロセスは常に起動されます。その他のプロセスの起動／停止のタイミングは以下の通りです。postmaster の子プロセスは定期的に親プロセスである postmaster プロセスの存在をチェックしており、postmaster プロセスが停止していることを検知すると自プロセスを終了します。

表 17 プロセスの起動／停止

プロセス	起動／停止のタイミング
logger	パラメーターlogging_collector = on の場合に起動（デフォルト off）
autovacuum launcher	パラメーターautovacuum = on の場合に起動（デフォルト on）
autovacuum worker	autovacuum launcher プロセスがパラメーター autovacuum_naptime で指定された間隔（デフォルト 1 分）で起動、処理が終了すると停止
archiver	スタンダロンまたはレプリケーション構成のマスター・インスタンスではパラメーターarchive_mode = on または always の場合に起動（デフォルト off） レプリケーション構成のスレーブ・インスタンスでは、archive_mode = always の場合のみ起動
postgres (local)	クライアントのローカル接続時に起動、切断時に停止
postgres (remote)	クライアントのリモート接続時に起動、切断時に停止
wal sender	・ストリーミング・レプリケーション環境のマスター・インスタンスで起動。スレーブ・インスタンスが接続してくると起動、切断すると停止。 ・pg_basebackup コマンドによるバックアップ中に起動、終了すると停止
wal receiver	ストリーミング・レプリケーション環境のスレーブ・インスタンスで起動。マスター・インスタンスが停止すると、自動的に停止。マスター・インスタンスが再開されると再起動
startup process	ストリーミング・レプリケーション環境のスレーブ・インスタンスで常時起動
wal writer	レプリケーション環境のスレーブ・インスタンスでは起動しない。それ以外では常に起動
bgworker	カスタム・プロセスの仕様により動作が変化する
parallel worker	Parallel Query 実行時に起動、SQL 実行完了時に停止

□ autovacuum worker プロセス数

autovacuum worker プロセスは、プロセス名から判るようにデータベース単位で起動されます。起動される最大数はパラメーターautovacuum_max_workers（デフォルト値 3）で決まります。各 worker プロセスはテーブル単位で処理を行います。

□ postgres プロセス数

クライアントが接続すると自動的に postgres プロセスが起動します。postgres プロセスの最大数はパラメーター max_connections (デフォルト値 100) に制限されます。

SUPERUSER 権限を持たない一般ユーザーが接続できる数は「max_connections - superuser_reserved_connections (デフォルト値 3)」の計算結果になります。この制限を超過する接続要求があると以下のログが出力されます。

例 6 一般ユーザーの接続数超過

```
FATAL: remaining connection slots are reserved for non-replication superuser connections
```

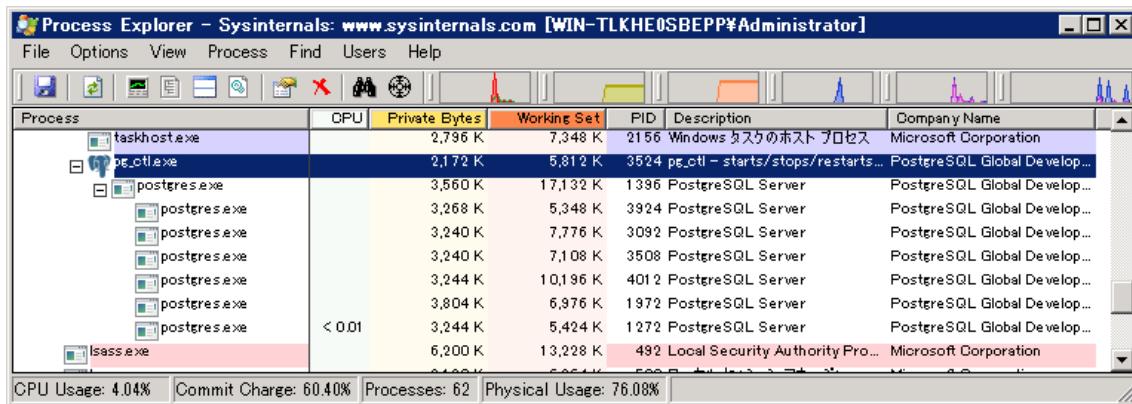
例 7 パラメーター max_connections で指定された接続数超過

```
FATAL: sorry, too many clients already
```

2.1.5 Microsoft Windows 環境のプロセス構成

Microsoft Windows 上で稼動する PostgreSQL は Windows Service として実行されます。Windows Service として登録される実行イメージは pg_ctl.exe です。pg_ctl.exe の子プロセスとして postgres.exe が実行され、postmaster として稼動します。下図は Windows 環境におけるプロセスの親子関係を示しています。

図 2 Windows 環境のプロセス構成



2.2 メモリー構成

2.2.1 共有バッファ概要

PostgreSQL はブロックのキャッシュを「共有バッファ」と呼ぶメモリー領域に保存し、複数のバックエンド・プロセス間で共有します。PostgreSQL インスタンスが使用する共有バッファは、System V Shared Memory（shmget システムコール）とメモリーマップドファイル（mmap システムコール）から構成されます。各プロセス間で協調して動作するためのロック処理には System V セマフォが利用されます。接続するクライアントが増加してもセマフォ・セットの数は変更されません。

例 8 共有メモリーの状況

```
$ ipcs -a
----- Shared Memory Segments -----
key      shmid      owner      perms      bytes      nattch      status
0x00530201 2621440  postgres    600          56          5

----- Semaphore Arrays -----
key      semid      owner      perms      nsems
0x00530201 19038210  postgres    600          17
0x00530202 19070979  postgres    600          17
0x00530203 19103748  postgres    600          17
0x00530204 19136517  postgres    600          17
0x00530205 19169286  postgres    600          17
0x00530206 19202055  postgres    600          17
0x00530207 19234824  postgres    600          17
0x00530208 19267593  postgres    600          17

----- Message Queues -----
key      msqid      owner      perms      used-bytes      messages
```

インスタンスが異常終了すると、共有バッファおよびセマフォが残ってしまうことがあります。インスタンスの再起動は正常に行われます。

2.2.2 共有バッファの実装

Linux 環境における System V Shared Memory は、shmget システムコールを使って作成します。System V Shared Memory の作成には、ホスト上で一意なキー番号とサイズを指定する必要があります。キー番号は以下の計算式を使って生成されます。キーが既に使用されている場合には、値をインクリメントさせながら空き番号を探します。この処理はソースコード (src/backend/port/sysv_shmem.c) 内の PGSharedMemoryCreate 関数内で実行しています。

計算式 1 共有メモリーのキー値

```
キー = パラメータport * 1000 + 1
```

標準では接続を待つポート番号（パラメータ port）は 5,432 であるため、共有メモリーのキーは 5,432,001 (= 0x52e2c1) となります。PostgreSQL 9.3 以降は System V Shared Memory として作成されるメモリー容量は構造体 PGShmemHeader

(include/storage/pg_shmem.h) のサイズです。テーブルやインデックス用に使用される共有バッファの大部分は、メモリマップドファイル (mmap システムコール) で作成されます。mmap で作成されるメモリー領域のサイズは 100 KB に、各種パラメーターから計算される容量を追加した値になります。Windows 環境では、CreateFileMapping システムコールによる共有メモリーを構成します (src/backend/port/win32_shmem.c)。

2.2.3 Huge Pages

大規模メモリーを搭載した Linux ではメモリー管理負荷を削減するために Huge Pages を利用することができます。Huge Pages への対応は PostgreSQL 9.4 の新機能であり、パラメーター huge_pages により決定されます。Huge Pages を使用する場合のページ・サイズは 2 MB です ($2 \times 1,024 \times 1,024$ バイト)。Huge Pages を使用する場合、確保される共有メモリーのサイズは計算値を元に 2 MB の倍数に調整され、mmap システムコールに MAP_HUGETLB マクロが指定されます。

□ パラメーター設定

PostgreSQL が使用する共有メモリーとして Huge Pages を使用するには、パラメーター huge_pages を設定します。

表 18 パラメーターhuge_pagesに指定できる値

パラメーター値	説明	備考
on	Huge Pages を使用する	
off	Huge Pages を使用しない	
try	Huge Pages の使用を試し、使えれば使う	デフォルト値

デフォルト値のtryを指定すると、mmapシステムコールにMAP_HUGETLBマクロを指定して共有メモリーを作成しようとします。処理に失敗した場合は、共有メモリーを、MAP_HUGETLBマクロを削除して再作成します。このパラメーターをonに指定すると強制的に Huge Pages を使用します。プラットフォームが Huge Pages をサポートしていない場合、pg_ctl コマンドは以下のエラー・メッセージを出力してインスタンスは起動できません。

例 9 エラー・メッセージ

```
FATAL: huge pages not supported on this platform
```

□ Huge Pages の設定方法

Linux 環境で Huge Pages を有効にするにはカーネル・パラメーターvm.nr_hugepagesに2 MB 単位のページ数の最大値を指定します。このパラメーターのデフォルト値は0です。使用中の Huge Pages の情報は、/proc/meminfo ファイルを参照します。

例 10 Linux の Huge Pages 設定

```
# sysctl -a | grep nr_hugepages
vm.nr_hugepages = 0
vm.nr_hugepages_mempolicy = 0
# sysctl -w vm.nr_hugepages = 1000
vm.nr_hugepages = 1000
# grep ^Huge /proc/meminfo
HugePages_Total:    1000
HugePages_Free:     1000
HugePages_Rsvd:      0
HugePages_Surp:      0
Hugepagesize:       2048 kB
#
```

パラメーターhuge_pages=on を指定した環境でインスタンス起動時に必要なページが確保できない場合、以下のエラーが発生してインスタンスを起動できません。

例 11 Huge Pages ページ不足エラー

```
$ pg_ctl -D data start -w
server starting
FATAL: could not map anonymous shared memory: Cannot allocate memory
HINT: This error usually means that PostgreSQL's request for a shared memory
segment exceeded available memory, swap space or huge pages. To reduce the
request size (currently 148324352 bytes), reduce PostgreSQL's shared memory
usage, perhaps by reducing shared_buffers or max_connections.
```

注意

Red Hat Enterprise Linux 6.4 では、ヘッダ・ファイルに MAP_HUGETLB マクロが欠落しているため、ソースコードからビルドすると Huge Pages 非対応のバイナリが作成されます。バイナリ作成時に、/usr/include/bits/mman.h 内に以下の行があるか確認してください。

```
# define MAP_HUGETLB      0x40000          /* Create huge page mapping. */
```

□ Huge Pages として必要なメモリー領域の計算

PostgreSQL インスタンスが使用する共有メモリーの容量はパラメーターの値から計算されます。パラメーターshared_buffers とパラメーターwal_buffers の容量に 10~50 MB 程度を追加します。この追加のメモリー量は、パラメーター max_connections 、 autovacuum_max_workers 、 max_worker_processes 等いくつかのパラメーターから計算されます。カーネル・パラメーターvm.nr_hugepages には上記の値を 2 MB 単位に切り上げて指定します。

正確な共有メモリーの必要量を知るために、パラメーター log_min_messages に DEBUG3 を指定してインスタンスを起動します。インスタンス起動ログ (pg_ctl -l で指定) に以下のメッセージが出力されます。

例 12 共有メモリー必要容量

```
DEBUG: invoking IpcMemoryCreate(size=148324352)
```

2.2.4 セマフォ

セマフォはバックエンド・プロセス間でリソース競合を防ぐロック制御のために使用されています。PostgreSQL ではインスタンス起動時に以下のパラメーターから計算された数のセマフォ集合が作成されます。

計算式 2 セマフォ集合の個数

最大バックエンド数 =

`max_connections + autovacuum_max_workers + 1 + max_worker_processes`

セマフォ集合数 = `CEIL(最大バックエンド数/17 + 1)`

各セマフォ集合には 17 個のセマフォが格納されます。Red Hat Enterprise Linux 6 の場合、セマフォ関連のカーネル・パラメーターのデフォルト値は最大セッション数が 1,000 程度のデータベースであれば十分な量が確保されています。

セマフォ関連のカーネル・パラメーターが不足している場合、以下のエラーが発生してインスタンスを起動できません。

例 13 セマフォ関連のリソース不足エラー

```
$ pg_ctl -D data start -w
waiting for server to start....
FATAL:  could not create semaphores: No space left on device
DETAIL:  Failed system call was semget(5440029, 17, 03600).
HINT:  This error does *not* mean that you have run out of disk space.  It
occurs when either the system limit for the maximum number of semaphore sets
(SEMMNI), or the system wide maximum number of semaphores (SEMMNS), would be
exceeded.  You need to raise the respective kernel parameter.  Alternatively,
reduce PostgreSQL's consumption of semaphores by reducing its max_connections
parameter.
```

The PostgreSQL documentation contains more information about configuring your system for PostgreSQL.

```
.... stopped waiting
pg_ctl: could not start server
Examine the log output.
```

セマフォ集合のキーは、共有メモリーのキーと同じロジックで作成されます (src/backend/port/sysv_sema.c)。Microsoft Windows 環境では、Windows API の CreateSemaphore を使ってセマフォ機能を作成しています (src/backend/port/win32_sema.c)。

2.2.5 チェックポイント

メモリーとストレージを同期し、永続化を保障する処理をチェックポイントと呼びます。チェックポイントを作成するために共有バッファ上で変更されたページをストレージに書き込みを行います。チェックポイントはいくつかのタイミングで発生します。

□ チェックポイントの発生契機

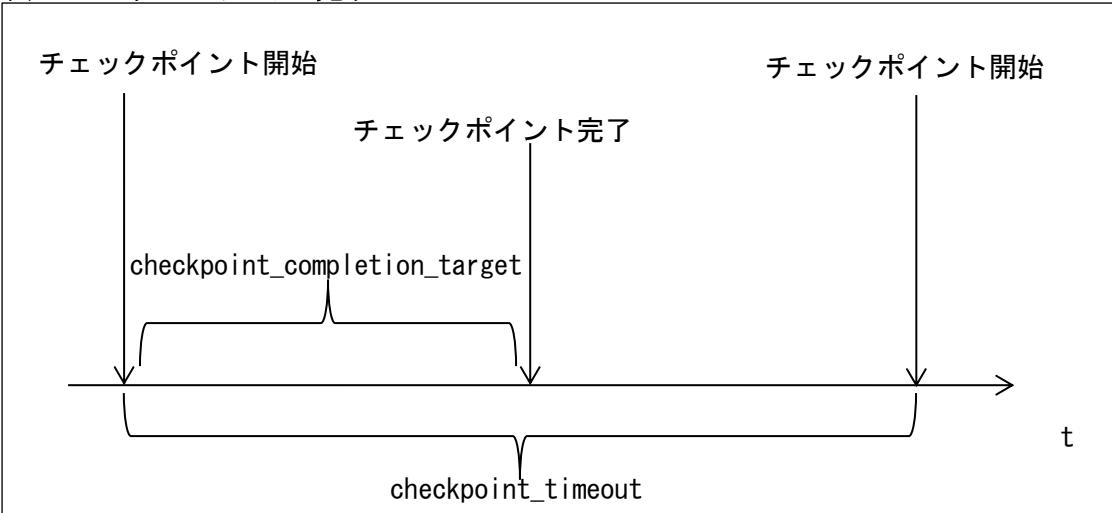
チェックポイントは以下の場合に発生します。

- CHECKPOINT 文の実行
管理者が CHECKPOINT 文を実行した場合。
- パラメーター checkpoint_timeout で設定した時間間隔
デフォルトでは 300 秒 (5 分) 間隔で実行されます。
- WAL に書き込まれたデータ量がパラメーター max_wal_size に達した場合
WAL データがパラメーターで指定された量 (デフォルト 1GB) だけ書き込まれた場合 (PostgreSQL 9.5 で変更されたパラメーター)。
- オンライン・バックアップ開始時
pg_start_backup 関数実行時
pg_basebackup コマンド実行時
- インスタンス終了時
pg_ctl stop -m immediate コマンド実行の場合を除く
- データベース構成時
CREATE DATABASE / DROP DATABASE 実行時

□ チェックポイントにおける処理の完了

チェックポイントの処理には種類が 2 つあります。一定時間間隔や WAL 書き込み量により発生する Regular Checkpoint とインスタンス停止時や CHECKPOINT 文発行時の Immediate Checkpoint です。Regular Checkpoint の処理にはダーティ・バッファを一度に書き込むのではなく、一定期間に処理を分散する機能が提供されています。パラメーター checkpoint_completion_target の設定により、次回のチェックポイント (パラメーター checkpoint_timeout で指定) 発生までに処理を完了する時間の割合を指定します。デフォルト値は 0.5 なので、次回のチェックポイント開始までの 50% の時間でチェックポイントを完了させることになります。

図 3 チェックポイントの完了



書き込みが必要なブロック数に対する書き込み完了ブロック数の割合と、チェックポイント間隔（パラメーターcheckpoint_timeout）を比較して進捗状況を確認します。書き込み量に余裕がある場合は 100 ミリ秒処理を停止して処理を再開します。この判断は IsCheckpointOnSchedule 関数 (src/backend/postmaster/checkpointer.c) で実施しています。

□ チェックポイントに関するパラメーター

チェックポイントに関するパラメーターは以下の通りです。

表 19 チェックポイントに関するパラメーター

パラメーター	説明	デフォルト値
checkpoint_timeout	チェックポイント間隔	5min
bgwriter_delay	writer プロセス書き込み間隔	200ms
bgwriter_lru_maxpages	writer プロセス書き込みページ数	100
bgwriter_lru_multiplier	writer プロセス書き込みページ数の倍数	2.0
checkpoint_completion_target	次回のチェックポイント時刻までにチェックポイントを完了させる割合。	0.5
log_checkpoints	チェックポイント情報をログに書く	off
full_page_writes	チェックポイント直後の更新時にはページ全体を WAL に書き込む	on

2.2.6 リング・バッファ

テーブルのシーケンシャル・スキャンや、COPY TO 文による一括検索が行われると、共有バッファ上のアクティブなページがメモリー上から排除される可能性があります。このためアクセスするテーブルのサイズが共有バッファの 1/4 を超えるテーブルに対するシーケンシャル・スキャンが行われる場合等には共有バッファ上的一部を循環させるリング・バッファを使用します。作成されるリング・バッファのサイズはソースコード上で固定されているため変更できません。

表 20 リング・バッファのサイズ

処理	サイズ	操作
一括読み込み	256 KB	Seq Scan CREATE MATERIALIZED VIEW
一括書き込み	16 MB	CREATE TABLE AS COPY FROM
VACUUM	256 KB	VACUUM

実際に作成されるリング・バッファのサイズは上記表のサイズと共有バッファの 1/8 を比較して小さい方が使われます (src/backend/storage/buffer/freelist.c)。リング・バッファの詳細は README (src/backend/storage/buffer/README) ファイルに記載されています。

2.3 インスタンス起動／停止時の動作

インスタンスの起動／停止時の動作についてまとめています。

2.3.1 起動／停止の待機

インスタンスの管理には pg_ctl コマンドを使用します。pg_ctl コマンドには、処理の完了を待機する -w パラメーター／待機を行わない -W パラメーターを指定することができます。マニュアルにも記載がありますが、インスタンスの起動時／再起動時は -W パラメーターがデフォルトで、インスタンスの停止時は -w パラメーターがデフォルトです (<https://www.postgresql.org/docs/9.6/static/app-pg-ctl.html>)。

表 21 pg_ctl コマンドによるインスタンス操作時の動作

動作	標準の動作	備考
start	非同期 (-W)	
restart	非同期 (-W)	停止処理は同期
stop	同期 (-w)	

待機を行う場合のタイムアウト時間は -t パラメーターで指定します。デフォルトは 60 秒です。1 秒ごとにステータスをチェックし、タイムアウトまで繰り返します。

□ インスタンス起動時の動作

pg_ctl start コマンドによるインスタンス起動は -w パラメーターを指定しない限り起動の完了を待機しません。postmaster プロセスの起動のために system 関数 (Windows 以外) の戻り値のみチェックしています。また Windows 環境では Windows API CreateRestrictedProcess を実行していますが、戻り値のチェックは行われていません。このため起動エラーが発生しても、pg_ctl コマンドの戻り値は 0 になります。

例 14 インスタンス起動失敗時の動作

```
$ pg_ctl -D data start
server starting
LOG: redirecting log output to logging collector process
HINT: Future log output will appear in directory "pg_log".
$ pg_ctl -D data start ← 同じクラスターに対して2回起動（エラーになる）
pg_ctl: another server might be running; trying to start server anyway
server starting
FATAL: lock file "postmaster.pid" already exists
Is another postmaster (PID 3950) running in data directory
"/usr/local/pgsql/data"?
$ echo $? ← pg_ctl コマンドのステータスは0
0
```

□ レプリケーション環境における待機

インスタンス停止時に-m smart パラメーター³を指定すると、クライアントの切断をタイムアウトまで待ちます。ただしレプリケーション環境でスレーブ・インスタンスによる接続はクライアントと見なされないため、スレーブの接続が行われっていてもインスタンスは停止できます。

例 15 レプリケーション時の —m smart パラメーター

```
postgres=# SELECT state FROM pg_stat_replication ;
 state
 -----
 streaming
(1 row)
postgres=# \q
$ pg_ctl stop -D data -m smart
waiting for server to shut down..... done
server stopped
```

³ PostgreSQL 9.5 で-m パラメーターのデフォルト値が smart から fast に変更されました。

ホット・スタンバイ状態になっていないスレーブ・インスタンス (hot_standby=off) を待機 (-w) 設定で起動すると、タイムアウト（デフォルトでは 60 秒）までコマンドが完了しません。これは pg_ctl コマンドが Pqping 関数を使ってインスタンスの起動を確認しているためです。

例 16 ホット・スタンバイではないスレーブ・インスタンスの起動

```
$ grep hot_standby data. stdby/postgresql.conf
hot_standby = off

$ pg_ctl -D data. stdby start -w
waiting for server to start....
LOG:  redirecting log output to logging collector process
HINT:  Future log output will appear in directory "pg_log".
..... stopped waiting
server is still starting up
$ echo $?
0
```

2.3.2 パラメーターの設定

インスタンス起動時には{PGDATA}/postgresql.conf ファイルが解析され、パラメーターが設定されます。その後、{PGDATA}/postgresql.auto.conf ファイルが解析されて設定値を上書きします。

パラメーターの一覧を取得するには pg_settings カタログを検索するか、psql ユーティリティから show all コマンドを実行します。pg_settings カタログの source 列は、パラメーターの設定元の情報が提供されます。下記列値は、ソースコード (src/backend/utils/misc/guc.c) 内の「GucSource_Names」配列で定義されている値です。実際には、enum GucSource で定義されたマクロ (PGC_S_{SOURCE}) を使用してアクセスされています。enum 値はソースコード (src/include/utils/guc.h) で定義されています。

表 22 pg_settings カタログの source 列

列値	説明	備考
default	デフォルト値	
environment variable	postmaster の環境変数から導出	
configuration file	postgresql.conf ファイルで設定	
command line	postmaster 起動パラメーター	
global	グローバル	詳細不明
database	データベース毎の設定	
user	ユーザー単位の設定	
database user	ユーザーとデータベース毎の設定	
client	クライアントからの設定	
override	強制的にデフォルト値を使用する特殊ケース	
interactive	エラー報告のための境界	
test	ユーザー毎またはデータベース毎のテスト	
session	SET 文による変更	

□ パラメーター・ファイルの動的変更

PostgreSQL 9.4 からは ALTER SYSTEM 文により、パラメーター・ファイルの設定が動的に永続化できるようになりました。ALTER SYSTEM 文は superuser 権限を持つユーザーのみ実行できます。

構文 1 ALTER SYSTEM 文

```
ALTER SYSTEM SET パラメータ名 = 値 | DEFAULT
ALTER SYSTEM RESET parameter_name
```

ALTER SYSTEM 文で変更したパラメーターの値は「{PGDATA}/postgresql.auto.conf」ファイルに書き込まれます。このファイルは手動で変更しないようにしてください。

例 17 ALTER SYSTEM 文によるパラメーター変更

```
postgres=# SHOW work_mem ;
work_mem
-----
4MB
(1 row)
postgres=# ALTER SYSTEM SET work_mem = '8MB' ;
ALTER SYSTEM
postgres=# SHOW work_mem ;
work_mem
-----
4MB
(1 row)
postgres=# $q
$ cat data/postgresql.auto.conf
# Do not edit this file manually!
# It will be overwritten by ALTER SYSTEM command.
work_mem = '8MB'
$
```

上記の例でもわかるように、ALTER SYSTEM 文はインスタンスのパラメーターは変更せず、postgresql.auto.conf ファイルのみ書き換えます。このファイルはインスタンス起動時または pg_reload_conf 関数実行時に postgresql.conf ファイルが読み込まれた後解析され、値が適用されます。

ALTER SYSTEM 文のパラメーター値として DEFAULT を指定するか、ALTER SYSTEM RESET 文を実行すると、postgresql.auto.conf ファイルからパラメーターが削除されます。

例 18 ALTER SYSTEM 文によるパラメータ・リセット

```
postgres=# ALTER SYSTEM SET work_mem = DEFAULT ;
ALTER SYSTEM
postgres=# \q
$ cat data/postgresql.auto.conf
# Do not edit this file manually!
# It will be overwritten by ALTER SYSTEM command.
$
```

□ パラメーター・ファイルと SET 文の構文

複数の値を持つパラメーターをパラメーター・ファイルに記述する場合は、値をカンマ(,)で区切り、全体をシングル・クオーテーション(')で囲みます。一方、SET 文でセッションのパラメーターを変更する場合は、シングル・クオーテーションの指定は行いません。SET 文のパラメーターをシングル・クオーテーションで囲むと単一の値として認識されます。

例 19 ファイルと SET 文の構文の違い

```
$ grep temp_tablespaces ${PGDATA}/postgresql.conf
temp_tablespaces = 'pg_default, ts1'
$ psql
postgres=# SET temp_tablespaces='ts2, ts3' ;
ERROR:  tablespace "ts2, ts3" does not exist
postgres=# SET temp_tablespaces=ts2, ts3 ;
SET
postgres=#
```

□ パラメーター記述形式とエラー

postgresql.conf ファイルには標準のパラメーター以外に Contrib モジュール等が使用する独自のパラメーターを記述できます。多くの場合パラメーター名の形式は「{MODULENAME}.{PARAMETER}」です。インスタンス起動時、この形式で記述されたパラメーターは正当性のチェックが行われません。Contrib モジュール用のパラメーターが間違って記述されてもインスタンスは正常に起動されます。また SHOW 文でも間違ったパラメーター名のまま情報を取得できます。

このため Contrib モジュール用のパラメーターを設定する場合は、設定後に動作を確認すべきです。

例 20 間違ったパラメータ名の記述

```
$ grep autoexplain postgresql.conf
autoexplain.loganalyze = true ← 正しくは auto_explain.log_analyze = true
$ pg_ctl -D data start -w
waiting for server to start...
LOG: redirecting log output to logging collector process
HINT: Future log output will appear in directory "pg_log".
done
server started ← 正常起動
$ psql
postgres=# SHOW autoexplain.loganalyze ; ← SHOW 文で参照可能
autoexplain.loganalyze
-----
true
(1 row)
```

ALTER SYSTEM 文では Contrib モジュールのパラメーターもパラメータ名のチェックが行われるため、間違った名前のパラメーターは設定できません。

□ パラメーター・ファイルの確認

パラメーター・ファイル（postgresql.conf, postgresql.auto.conf）の記述内容は、
pg_file_settings カタログから確認できます。このカタログに対する検索が行われる度
にファイル内容が解析され、ファイルに記述された情報を参照できます。

例 21 ファイル内容をカタログから確認

```
postgres=# ALTER SYSTEM SET port=5434 ;
ALTER SYSTEM
postgres=# SELECT sourcefile, name, setting FROM pg_file_settings
      WHERE name = 'port' ;
sourcefile          | name | setting
-----+-----+
/usr/local/pglsq/data/postgresql.auto.conf | port | 5434
(1 row)
```

2.3.3 外部ライブラリ

PostgreSQL には外部ライブラリを動的にロードすることで機能を拡張することができます。

□ ライブラリをロードするためのパラメーター

外部ライブラリを自動的にロードするために以下のパラメーターが定義されています。いずれのパラメーターにも、ライブラリのリスト（カンマ区切り）を指定します。

表 23 ライブラリのロードを行うパラメーター

パラメータ名	説明
shared_preload_libraries	インスタンス起動時にロード
session_preload_libraries	postgres プロセス開始時にロード、変更は superuser のみ
local_preload_libraries	postgres プロセス開始時にロード、一般ユーザー変更可能

postgres プロセス起動時は、まず session_preload_libraries に指定されたライブラリのロードが行われ、その後 local_preload_libraries に指定されたライブラリがロードされます。

□ shared_preload_libraries パラメーター設定値

パラメーター shared_preload_libraries には、Contrib モジュール等で使用する共有ライブラリ名を設定します。インスタンス起動時に共有ライブラリを見つけることができない場合、インスタンス起動はエラーになります。

以下の例では Contrib モジュール pg_stat_statements がインストールしていない環境でパラメーターを設定してインスタンスを起動した場合のエラー・メッセージです。

例 22 shared_preload_libraries パラメーターのエラー

```
$ pg_ctl -D data start -w
waiting for server to start...
FATAL:  could not access file "pg_stat_statements": No such file or directory
..... stopped waiting
pg_ctl: could not start server
Examine the log output.
```

shared_preload_libraries パラメーターに指定したライブラリは、パラメーター dynamic_library_path に指定されたパスから検索されます。

2.3.4 インスタンス停止失敗時の動作

`pg_ctl stop -m smart` コマンドは接続ユーザーの終了を待ちますが、タイムアウト（デフォルト 60 秒）を経過すると `pg_ctl` コマンドが戻り値 1 で終了します。

タイムアウトした場合でも、インスタンスはシャットダウン中のステータスのままでです。このため、新規のクライアント接続はできない状態に陥ります。既存のセッションがすべて終了すると自動的にインスタンスは終了します。タイムアウトの設定は、`pg_ctl` コマンド・パラメータ `--timeout=秒数`（または `-t 秒数`）で指定します。

例 23 インスタンス終了タイムアウト

```
$ pg_ctl -D data stop -m smart
waiting for server to shut down..... failed
pg_ctl: server does not shut down
HINT: The "-m fast" option immediately disconnects sessions rather than
waiting for session-initiated disconnection.
$

$ psql -U user1
psql: FATAL: the database system is shutting down
↑ 新規のセッションは受け付けられない
$

$ pg_ctl stop -m immediate
waiting for server to shut down.... done
server stopped
$
```

2.3.5 インスタンス起動時の読み込みライブラリ

インスタンス起動時に読み込まれる共有ライブラリを以下に示します。インスタンス起動時の動作を `strace` コマンドでトレースして確認しました。

表 24 インスタンス起動時に読み込まれるライブラリ

ライブラリ	ディレクトリ
libpq.so.5	{INSTALL}/lib
libc.so.6	/lib64
libpthread.so.6	/lib64
libtinfo.so.5	/lib64
libdl.so.2	/lib64
librt.so.1	/lib64
libm.so.6	/lib64
libnss_files.so.2	/lib64
libselinux.so.1	/lib64
libacl.so.1	/lib64
libattr.so.1	/lib64

2.3.6 主な入出力ファイル

インスタンス起動時に入出力されるファイルを示します。インスタンスの停止は正常に行われた場合を想定しています。またパラメーター等はデフォルト値を使用しています。

表 25 入出力ファイル

ファイル	パス	備考
postgresql.conf	{PGDATA}	
postgresql.auto.conf	{PGDATA}	
PG_VERSION	{PGDATA}	
postmaster.pid	{PGDATA}	
Japan	{INSTALL}/share/postgresql/timezone	
posixrules	{INSTALL}/share/postgresql/timezone	
Default	{INSTALL}/share/postgresql/timezonesets	
pg_control	{PGDATA}/global	
.s.PGSQL.5432.lock	/tmp	
.s.PGSQL.5432	/tmp	
0000	{PGDATA}/pg_notify	再作成
postmaster.opts	{PGDATA}	作成
pg_log (directory)	{PGDATA}	作成
postgresql-{DATE}.log	{PGDATA}/pg_log	
pgsql_tmp	{PGDATA}/base	
state	{PGDATA}/pg_replslot/{SLOT}	9.4 追加
pg_hba.conf	{PGDATA}	
pg_ident.conf	{PGDATA}	
pg_internal.init	{PGDATA}/global	
recovery.conf	{PGDATA}	
backup_label	{PGDATA}	
000000010...00001	{PGDATA}/pg_xlog	
0000	{PGDATA}/pg_multixact/offsets	
0000	{PGDATA}/pg_clog	
pg_filenode.map	{PGDATA}/global	
global.tmp	{PGDATA}/pg_stat_tmp	
db_{OID}.stat	{PGDATA}/pg_stat	
global.stat	{PGDATA}/pg_stat_tmp {PGDATA}/pg_stat	
db_0.tmp	{PGDATA}/pg_stat_tmp	
archive_status	{PGDATA}/pg_xlog	

2.3.7 Windows Service 停止時の動作

Microsoft Windows 環境の PostgreSQL インスタンスは Windows サービスとして動作することができます。NET STOP コマンドまたは「サービス マネージャー」を使ったインスタンス停止処理は fast モード (SIGINT シグナル) で行われます。

下記ソースは「src/bin/pg_ctl/pg_ctl.c」内の pgwin32_ServiceMain 関数の一部です。

例 24 NET STOP コマンドによるインスタンス終了

```
static void WINAPI
pgwin32_ServiceMain(DWORD argc, LPTSTR *argv)
{
    <<途中省略>>
    pgwin32_SetServiceStatus(SERVICE_STOP_PENDING);
    switch (ret)
    {
        case WAIT_OBJECT_0:           /* shutdown event */
        {
            /*
             * status.dwCheckPoint can be incremented by
             * test_postmaster_connection(), so it might not start from 0.
             */
            int maxShutdownCheckPoint = status.dwCheckPoint + 12;
            kill(postmasterPID, SIGINT);
        }
        <<途中省略>>
    }
}
```

3.ストレージ構成の検証

3.1 ファイルシステムの構造

本節ではファイルシステムに関する情報を提供しています。

3.1.1 ディレクトリ構造

ここではPostgreSQLデータベース・クラスターのディレクトリ構造を記載しています。

□ データベース・クラスター

データベース・クラスターは、PostgreSQLデータベースの永続化情報がすべて格納されます。オペレーティング・システムのディレクトリを指定して initdb コマンドにより作成されます。データベース・クラスターはインスタンス起動、停止時に使用する pg_ctl コマンドでも必ず指定され、インスタンスの運用単位になります。

例 25 データベース・クラスター内のファイル構造

```
$ ls -l ${PGDATA}
total 96
-rw----- 1 postgres postgres      4 Feb 11 12:45 PG_VERSION
drwx----- 6 postgres postgres  4096 Feb 11 13:00 base
drwx----- 2 postgres postgres  4096 Feb 11 15:52 global
drwx----- 2 postgres postgres  4096 Feb 11 12:45 pg_clog
-rw----- 1 postgres postgres  4222 Feb 11 12:45 pg_hba.conf
-rw----- 1 postgres postgres  1636 Feb 11 12:45 pg_ident.conf
drwxr-xr-x 2 postgres postgres  4096 Feb 11 15:52 pg_log
<< 途中省略 >>
drwx----- 2 postgres postgres  4096 Feb 11 15:54 pg_tblspc
drwx----- 2 postgres postgres  4096 Feb 11 12:45 pg_twophase
drwx----- 3 postgres postgres  4096 Feb 11 12:45 pg_xlog
-rw-r--r-- 1 postgres postgres   101 Feb 11 12:45 postgresql.auto.conf
-rw-r--r-- 1 postgres postgres 19598 Feb 11 12:45 postgresql.conf
-rw----- 1 postgres postgres    45 Feb 11 15:52 postmaster.opts
-rw----- 1 postgres postgres     73 Feb 11 15:52 postmaster.pid
$
```

データベース・クラスターとして指定されたディレクトリ内には多数のディレクトリとファイルが作成されます。base ディレクトリは永続化データが保存される標準のディレクトリです。base ディレクトリにはデータベースに対応するサブ・ディレクトリが作成されます。

3.1.2 データベース・ディレクトリの内部

データベースに対応するディレクトリ以下には、データベースに保存されるオブジェクトが個別のファイルとして作成されます。以下のファイルが自動的に作成されます。

表 26 データベース・ディレクトリ以下に作成されるファイル

ファイル名	説明
{999999}	セグメント・ファイル
{999999}.{9}	セグメント・ファイル (1 GB 超の場合)
{999999}_fsm	Free Space Map ファイル
{999999}_vm	Visibility Map ファイル
{999999}_init ⁴	UNLOGGED TABLE の初期化フォークを示すファイル
pg_filenode.map	一部のシステム・カタログの OID と物理ファイル名のマッピングを行う。
pg_internal.init	システム情報のキャッシュファイル。インスタンス起動時に再作成される。 {PGDATA}/global ディレクトリおよびデータベースが保存されるディレクトリ直下に作成される。
PG_VERSION	バージョン情報が記録されるテキストファイル。データベース利用時にチェックされる。

□ テーブルの構成要素とカタログ

PostgreSQL のテーブルは、実際には複数のオブジェクトの集合体です。内部的には以下の要素から構成されています。

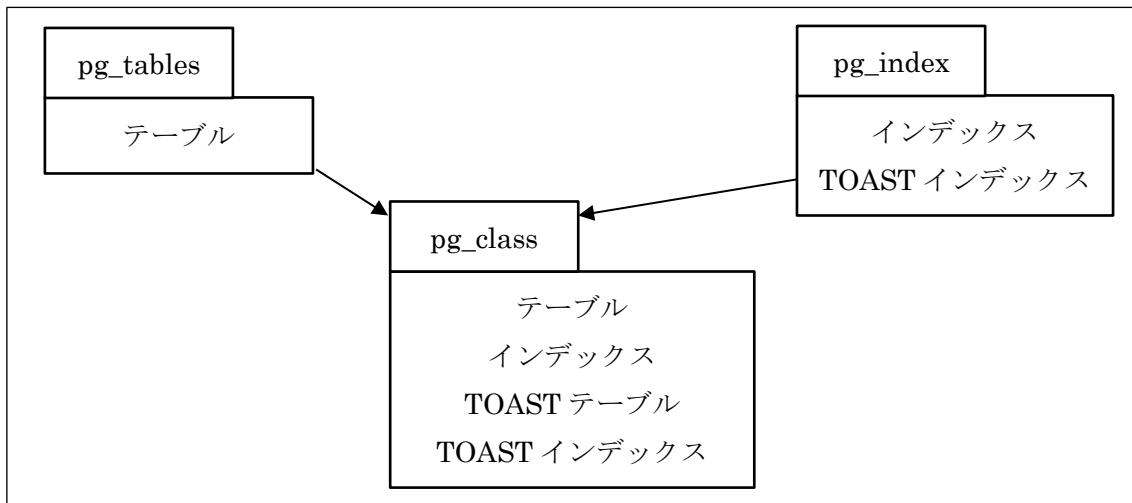
⁴ _init ファイルは UNLOGGED TABLE、UNLOGGED TABLE の TOAST テーブル、UNLOGGED TABLE の TOAST インデックス、UNLOGGED TABLE に対して作成されたインデックスに対して作成されます。

表 27 テーブルを構成する要素

要素	説明	備考
テーブル	データが保存される領域	
インデックス	検索高速化のためにテーブルに作成される索引	
TOAST テーブル	大規模データを格納する領域	後述
TOAST インデックス	TOAST テーブルの検索を高速化するインデックス	後述

上記の要素をすべて管理するカタログが pg_class です。pg_class カタログにはテーブル名 (relname)、TOAST テーブルや TOAST インデックスの OID (reltoastrelid) が格納されています。pg_tables カタログは pg_class カタログからテーブルのみを抽出するビューになっています。テーブルとインデックスの対応付けを行うカタログが pg_index です。このカタログには pg_class カタログに格納されているテーブルの OID (indexrelid) と、インデックスの OID (indrelid) 等の情報が格納されています。

図 4 テーブルとカタログ



□ テーブルとファイルの関係

テーブルやインデックスとオペレーティング・システムのファイルは pg_class カタログの relfilenode 列の値と対応しています。

オブジェクトとファイルの関係は oid2name ユーティリティを使っても確認できます。格納されるテーブル空間は pg_class カタログの reltablespace 列で確認します。この列値が 0 の場合、pg_default テーブル空間であることを示します。

例 26 ファイルの特定

```
$ oid2name -d datadb1
From database "datadb1":
  Filenode  Table Name
  -----
    16437      data1

postgres=> SELECT relname, reffilenode, reltablespace FROM pg_class
           WHERE relname IN ('data2', 'data3') ;
  relname | reffilenode | reltablespace
  -----+-----+-----+
  data2   |      34115 |          0          <- テーブル空間 pg_default
  data3   |      34119 |      32778          <- テーブル空間 tb12
```

□ セグメント・ファイル

セグメント・ファイルは、テーブルやインデックスの実データが格納されたファイルです。ファイル・サイズが 1 GB (RELSEG_SIZE×BLCKSZ) を超えると複数作成されます。元のファイルに加えて、ファイル名の末尾に「.{9}」({9}は1から始まる数字)付のファイルが作成されます。

例 27 セグメント・ファイル

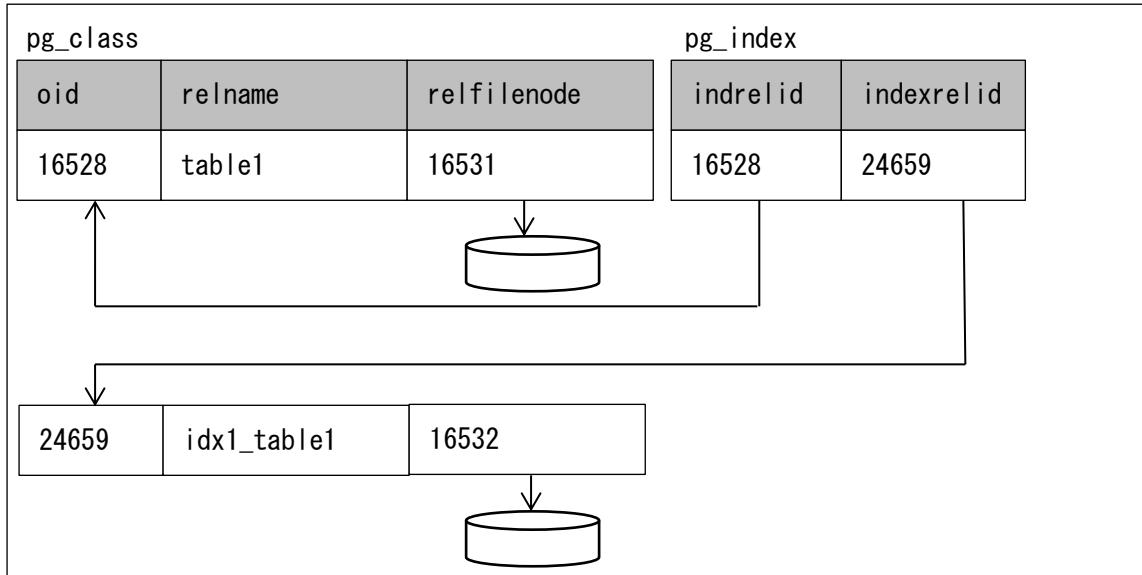
```
postgres=> SELECT oid, relname, reffilenode FROM pg_class
           WHERE relname='large1' ;
  oid | relname | reffilenode
  ----+-----+-----
  16468 | large1 |      16495
(1 row)

$ ls -l 16495*
-rw-----. 1 postgres postgres 1073741824 Feb 11 14:06 16495
-rw-----. 1 postgres postgres    96550912 Feb 11 14:06 16495.1
```

□ インデックス・ファイル

テーブルと同様にインデックスも独立したファイルとして作成されます。インデックスのファイル名も pg_class カタログの relfilenode 列に格納されています。テーブルとインデックスを結びつけるカタログが pg_index です。

図 5 pg_class カタログとインデックス



3.1.3 TOAST 機能

通常 PostgreSQL は 8 KB 単位のページにレコードを格納します。レコードがページをまたがって格納されることはありません。このため大規模なレコードはページに含めることができません。より大規模なレコードを格納するために TOAST (The Oversized-Attribute Storage Technique) と呼ばれる機能が提供されています。TOAST データは圧縮済の列データが TOAST_THRESHOLD (コンパイル時に決定) で決められたサイズを超える場合に作成されます。また TOAST_TARGET 以下に縮小されるまで TOAST テーブルにデータを格納します。

□ TOAST テーブル

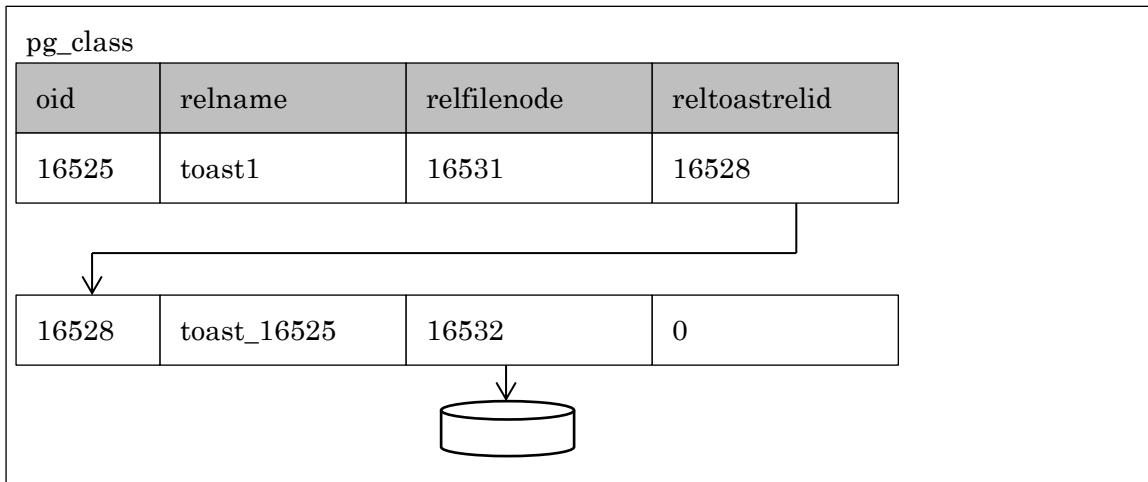
TOAST データは pg_class カタログの relfilenode 列で指定されるファイルとは別テーブル (別ファイル) に格納されます。pg_class カタログの reltoastreloid 列には、TOAST テーブルの oid が保存されます。pg_class カタログから TOAST テーブルのファイル名 (relfilenode) を検索することで、ファイルを特定することができます。TOAST テーブルの検索を高速化するために TOAST テーブルには TOAST インデックスも作成されます。TOAST テーブルは pg_tables カタログには表示されません。

表 28 pg_class カタログの relname 列

relname	説明
テーブル名	CREATE TABLE で作成したテーブル名
toast_{OID}	テーブルに対応する TOAST テーブル (OID は元テーブルの oid)
toast_{OID}_index	TOAST テーブルに対する TOAST インデックス

下記の図はテーブルと TOAST テーブルの関係を表しています。テーブル toast1 を作成すると、TOAST テーブル toast_16525 が自動的に作成され、ファイル 16532 に保存されます。TOAST インデックスは pg_index カタログから indrelid 列が 16528 のレコードを検索します。

図 6 pg_class カタログと TOAST テーブル



□ TOAST データの保存

TOAST データには保存フォーマットを指定することができます。保存フォーマットは通常自動的に決定されますが、列単位で指定することができます。

表 29 TOAST データ保存フォーマット

フォーマット	説明
PLAIN	TOAST を使用しません。
EXTENDED	圧縮と TOAST テーブルを利用します。多くの TOAST を利用できるデータ型のデフォルト値です。
EXTERNAL	圧縮は行いませんが、TOAST テーブルを利用します。
MAIN	圧縮は行いますが、TOAST テーブルは原則として使用しません。

psql コマンド内から、「\d+ テーブル名」を実行すると、TOAST 対応列の保存フォーマ

ットを確認できます。次の例では、toast1 テーブルの c1 列 (varchar 型) と c2 列 (text 型) が TOAST 対応であることがわかります。

例 28 TOAST 列の確認

```
postgres=> \d+ toast1
                                         Table "public.toast1"
  Column |          Type          | Modifiers | Storage | Stats target | Description
-----+----------------+-----+-----+-----+-----+
  c1    | numeric          |           | main   |             |
  c2    | character varying(10) |           | extended |             |
  c3    | text              |           | extended |             |
Has OIDs: no
```

デフォルトの保存フォーマットを変更するためには ALTER TABLE 文で SET STORAGE 句を使って指定します。

例 29 TOAST 保存フォーマットの変更

```
postgres=> ALTER TABLE toast1 ALTER c2 SET STORAGE PLAIN ;
ALTER TABLE
postgres=> \d+ toast1
                                         Table "public.toast1"
  Column |          Type          | Modifiers | Storage | Stats target | Description
-----+-----+-----+-----+-----+-----+
  c1    | numeric          |           | main   |             |
  c2    | character varying(10) |           | plain   |             |
  c3    | text              |           | extended |             |
Has OIDs: no
```

3.1.4 TRUNCATE 文とファイルの関係

TRUNCATE 文が実行されたトランザクションがコミットされると、テーブルと対応するファイルは、チェックポイントを待たずにサイズ 0 に切り捨てられます。また、TRUNCATE 文の実行が完了すると、次回 INSERT されるためのファイルが新規に作成され、pg_class カタログの relfilename 列は新しいファイル名に更新されます。TRUNCATE が実行されるまで使用された旧ファイルはチェックポイントのタイミングで削除されます。

例 30 TRUNCATE とファイルの対応

```
postgres=> SELECT relfilenode FROM pg_class WHERE relname='tr1' ;
relfilenode
-----
25782
(1 row)

$ ls -l 2578* ← ファイルの確認
-rw----- 1 postgres postgres 884736 Feb 11 11:23 25782
-rw----- 1 postgres postgres 24576 Feb 11 11:23 25782_fsm

postgres=> TRUNCATE TABLE tr1 ;           ← TRUNCATE 文の実行
TRUNCATE TABLE
postgres=> SELECT relfilenode FROM pg_class WHERE relname='tr1' ;
relfilenode
-----
25783      ← ファイルが新しくなった
(1 row)

$ ls -l 2578*
-rw----- 1 postgres postgres 0 Feb 11 11:25 25782      ← 旧ファイル
-rw----- 1 postgres postgres 0 Feb 11 11:25 25783      ← 新ファイル
$

postgres=# CHECKPOINT ;           ← チェックポイントの実行
CHECKPOINT
postgres=# \q
$ ls -l 2578*
-rw----- 1 postgres postgres 0 Feb 11 11:25 25783      ← 新ファイルのみ
$
```

3.1.5 FILLFACTOR 属性

INSERT 文を実行すると、ページ内にレコードが追加されます。使用中のページにレコードが格納できなくなると次の空きページを探します。ページ内にレコードを格納できる割合を示す属性が FILLFACTOR です。FILLFACTOR のデフォルト値は 100(%)です。このため、標準ではページ内に隙間なくレコードが格納されることになります。インデックスに対しても指定することができます。

FILLFACTOR を 100%以下にする利点は UPDATE 文による更新時に空き領域を使用できるため、ページ単位のアクセスでパフォーマンスが向上する点です。一方で、テーブルが使用するページ数が拡大することになるため、テーブル全体を読み込む場合には I/O が増加することになります。更新が頻繁に行われるテーブルまたはインデックスは FILLFACTOR の値をデフォルト値から下げる 것을 推奨します。

□ CREATE TABLE 文実行時の確認

テーブル作成時に FILLFACTOR を設定するには CREATE TABLE 文の WITH 句に記述します。確認するには pg_class カタログの relops 列を参照します。

例 31 FILLFACTOR の設定

```
postgres=> CREATE TABLE fill1(key1 NUMERIC, val1 TEXT) WITH (FILLFACTOR = 85) ;
CREATE TABLE
postgres=> SELECT relname, relops FROM pg_class WHERE relname='fill1' ;
   relname |    relops
-----+-----
 fill1   | {fillfactor=85}
(1 row)
```

□ ALTER TABLE 文実行時の動作

既存のテーブルに対して FILLFACTOR 属性を変更するには ALTER TABLE 文に SET 句で指定します。FILLFACTOR 属性値を変更しても既存テーブルのレコードは変更されません。

例 32 FILLFACTOR の設定による既存データへの影響

```
postgres=> INSERT INTO fill1 VALUES (generate_series(1, 1000), 'data') ;
INSERT 0 1000
postgres=> SELECT MAX(ip) FROM heap_page_items(get_raw_page('fill1', 0)) ;
max
-----
157
(1 row)
postgres=> ALTER TABLE fill1 SET (FILLFACTOR = 30) ;
ALTER TABLE
postgres=> SELECT MAX(ip) FROM heap_page_items(get_raw_page('fill1', 0)) ;
max
-----
157
(1 row)
```

テーブル fill1 にデータを格納します。heap_page_items⁵関数を使ってページの状態を確認すると、最初のページには 157 レコード格納されていることがわかります。ALTER TABLE 文を実行して、FILLFACTOR 属性を変更し、再度ページの情報を確認していますが、同じレコード数が格納されていることがわかります。

⁵ 拡張モジュール pageinspect で定義されています。実行には superuser 権限が必要です。

3.2 テーブル空間

3.2.1 テーブル空間とは

PostgreSQL ではデータベース、テーブル、インデックス、マテリアライズド・ビュー等の永続化オブジェクトはテーブル空間⁶に格納されます。データベース・クラスターを作成すると、標準で 2 つのテーブル空間が作成されます。pg_default テーブル空間は一般ユーザーが使用します。pg_global テーブル空間には全データベースで共有するシステム・カタログが格納されています。データベース作成時にテーブル空間 (TABLESPACE 句) を指定しない場合、pg_default テーブル空間が使用されます。

□ パラメーターdefault_tablespace

パラメーターdefault_tablespace はテーブル、インデックス、マテリアライズド・ビュー等のオブジェクト作成時に TABLESPACE 句を省略した場合に使用されるテーブル空間名を指定します。このパラメーターの設定は CREATE DATABASE 文によるデータベースの保存先には影響しません。

このパラメーターのデフォルト値は” (空文字列) で、オブジェクトの保存にはデータベースが保存されたテーブル空間が使用されます。このパラメーターはインスタンス全体だけではなく、セッション単位でも変更することができます。

表 30 テーブル空間の指定をしなかった場合のオブジェクト保存先

オブジェクト	パラメーターdefault_tablespace の指定	
	パラメーター指定あり	パラメーター指定なし (空文字列)
データベース	pg_default	pg_default
テーブル	指定されたテーブル空間	データベースと同じテーブル空間
インデックス	指定されたテーブル空間	データベースと同じテーブル空間
マテリアライズド・ビュー	指定されたテーブル空間	データベースと同じテーブル空間
シーケンス ⁷	指定されたテーブル空間	データベースと同じテーブル空間

⁶ Oracle Database と同様、「表領域」と呼ばれる場合もあります。

⁷ シーケンスの作成構文には TABLESPACE 句がありませんが、このパラメーターの影響を受けます。

SET 文を使ってセッション単位でパラメーターを指定した場合、指定した名前のテーブル空間が存在するかがチェックされますが、実際にオブジェクト作成権限があるかどうかはチェックされません。

postgresql.conf ファイルを使ってインスタンス単位でパラメーターを指定した場合、指定されたテーブル空間の存在はチェックされません。またその場合、TABLESPACE 句を省略すると接続先データベースのテーブル空間が使用されます。

例 33 存在しないテーブル空間を指定された場合の動作

```
postgres=> SHOW default_tablespace ;
default_tablespace
-----
ts_bad          ← postgresql.conf に存在しないテーブル空間名を指定
(1 row)

postgres=> CREATE TABLE data1 (c1 NUMERIC, c2 VARCHAR(10)) ;
CREATE TABLE
postgres=> \d data1
      Table "public.data1"
 Column |        Type         | Modifiers
-----+---------------------+-
 c1    | numeric            |
 c2    | character varying(10) | ← デフォルトのテーブル空間名が使用される
      ↓ SET 文によるパラメーター default_tablespace 変更時はチェックが行われる。
      ↑ SET default_tablespace = ts_bad2 ;
SET default_tablespace = ts_bad2 ;
ERROR: invalid value for parameter "default_tablespace": "ts_bad2"
DETAIL: Tablespace "ts_bad2" does not exist.
```

□ パラメーター temp_tablespaces

一時オブジェクトを作成するテーブル空間名のリストを指定します。複数の名前が指定されている場合、使用されるテーブル空間は無作為に選択されます。

3.2.2 オブジェクトとファイルの関係

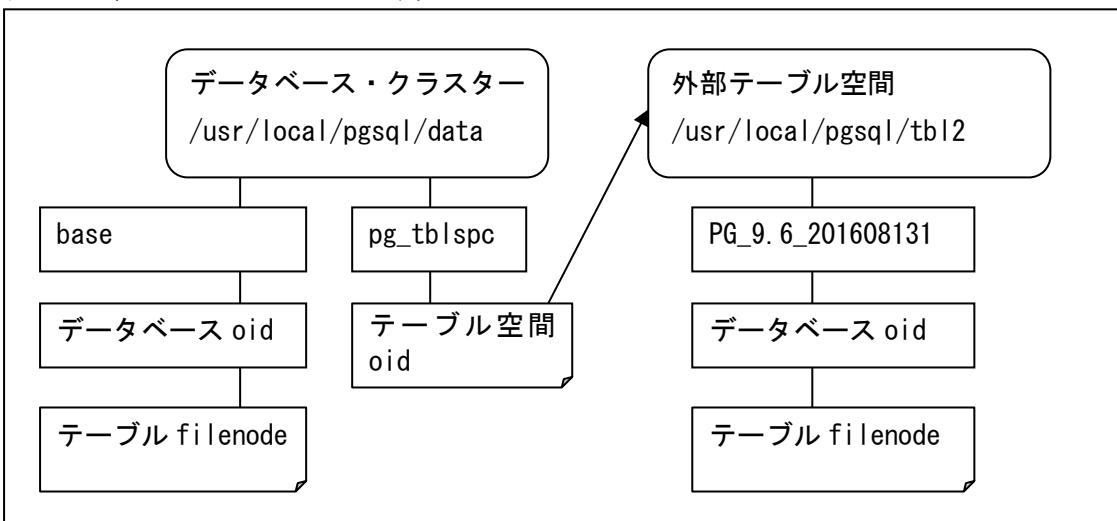
PostgreSQL ではデータベースやテーブル等のオブジェクトはオペレーティング・システムのディレクトリやファイルと対応しています。

□ テーブル空間 (TABLESPACE) の実装

`pg_default` テーブル空間はデータベース・クラスター内の `base` ディレクトリと対応します。外部のテーブル空間を作成すると、`{PGDATA}/pg_tblspc` ディレクトリにシンボリック・リンクが作成されます。

シンボリック・リンクのファイル名は、`pg_tablespace` カタログの `oid` 列に対応する名前です。

図 7 ディレクトリとテーブル空間



例 34 テーブル空間とディレクトリの対応

```
postgres=# CREATE TABLESPACE tb12 LOCATION '/usr/local/pgsql/tb12' ;
CREATE TABLESPACE
postgres=# SELECT oid, spcname FROM pg_tablespace ;
   oid  |  spcname
-----+-----
 1663  | pg_default
 1664  | pg_global
32788 | tb12
(3 rows)

$ ls -l /usr/local/pgsql/data/pg_tbspace
total 0
lrwxrwxrwx 1 postgres postgres 26 Feb 11 11:15 32788 -> /usr/local/pgsql/tb12
$
```

テーブル空間を作成すると、ディレクトリ内には「PG_{VERSION}_{YYYYMMDDN}」という名前のサブ・ディレクトリが作成されます。YYYYMMDD 部分はテーブル空間作成日付ではなく、フォーマット用の日付だと思われます。

例 35 テーブル空間の内部

```
$ ls -l /usr/local/pgsql/tb12
total 4
drwx----- 2 postgres postgres 6 Feb 11 13:23 PG_9.6_201608131
$
```

□ データベースの特定

データベースにはデータベース・クラスター全体で一意の ID (oid) が付与されます。この oid は pg_database カタログの oid 疑似列として確認することができます (または pg_stat_database カタログの datid 列)。テーブル空間内にデータベースの oid と同じ名前のディレクトリが作成されます。oid の確認は、ユーティリティ oid2name でも確認できます。

例 36 データベースの対応

```
postgres=> SELECT oid, datname FROM pg_database ;
      oid | datname
      ----+-----
    13322 | postgres
24577 | demodb
        1 | template1
    13321 | template0
(4 rows)

$ oid2name
All databases:
   Oid Database Name   Tablespace
   -----
24577      demodb  pg_default
    13322      postgres pg_default
    13321      template0 pg_default
        1      template1 pg_default
        2

$ ls -l base
total 48
drwx----- 2 postgres postgres 8192 Feb 11 10:33 1
drwx----- 2 postgres postgres 8192 Feb 11 10:33 13321
drwx----- 2 postgres postgres 8192 Feb 11 12:25 13322
drwx----- 2 postgres postgres 8192 Feb 11 12:25 24577
$
```

□ オブジェクト名からファイルを特定

`pg_class` カタログを検索する以外に、`pg_relation_filepath` 関数を使ってテーブル名からファイル名を特定することができます。この関数にテーブル名／マテリアライズド・ビュー名／インデックス名を指定すると、下記のように、データベース・クラスターからの相対パスを返します。`pg_default` 以外のテーブル空間を使用している場合は、`pg_tblspc` ディレクトリ以下に格納されているように表示されますが、実際にはシンボリック・リンク先のファイルになります。

例 37 オブジェクトとファイルの対応

```
postgres=> CREATE TABLE data1(c1 NUMERIC, c2 CHAR(10)) ;
CREATE TABLE
postgres=> SELECT pg_relation_filepath('public.data1') ;
pg_relation_filepath
-----
base/16394/16447
(1 row)

postgres=> CREATE TABLE data2 (c1 NUMERIC, c2 CHAR(10)) TABLESPACE ts1 ;
CREATE TABLE
postgres=> SELECT pg_relation_filepath('public.data2') ;
pg_relation_filepath
-----
pg_tblspc/32985/PG_9.6_201608131/16385/32986
(1 row)
```

ファイル名のみを取得する場合は、`pg_relation_filenode` 関数を使用します。

3.3 ファイルシステムと動作

3.3.1 データベース・クラスターの保護モード

データベース・クラスターに指定されているディレクトリは、起動ユーザーのみがアクセスできるモード（0700）になっている必要があります。グループや外部ユーザーにアクセス権が設定されていると、インスタンスを起動できません。

例 38 アクセス・モードとインスタンス起動

```
$ chmod g+r data
$ pg_ctl -D data start -w
server starting
FATAL:  data directory "/usr/local/pgsql/data" has group or world access
DETAIL:  Permissions should be u=rwx (0700).
$ echo $?
1
```

空きディレクトリに対して initdb コマンドが実行され、データベース・クラスターが作成されると、ディレクトリの保護モードは自動的に変更されます。またテーブル空間が作成されたディレクトリの保護モードも同様に変更されます。

例 39 保護モードの変更

```
$ mkdir data1
$ ls -ld data1
drwxrwxr-x 2 postgres postgres Feb 11 12:59 10:27 data1
$ initdb data1
The files belonging to this database system will be owned by user "postgres".
<<途中省略>>
      pg_ctl -D data1 -l logfile start
$ ls -ld data1
drwx----- 14 postgres postgres 4096 Feb 11 12:59 data1
$
$ mkdir ts1
$ ls -ld ts1
drwxr-xr-x. 2 postgres postgres 4096 Feb 11 12:59 ts1
$ psql
postgres=# CREATE TABLESPACE ts1 LOCATION '/usr/local/pgsql/ts1' ;
CREATE TABLESPACE
postgres=# \q
$ ls -ld ts1
drwx-----. 3 postgres postgres 4096 Feb 11 12:59 ts1
```

インスタンス起動時の保護モードのチェックはデータベース・クラスター外に作成されたテーブル空間では行われません。このため保護モードを変更してもデータベースに対する操作はエラーになりません。

例 40 テーブル空間の保護モード変更

```
postgres=# CREATE TABLESPACE ts1 LOCATION '/usr/local/pgsql/ts1' ;
CREATE TABLESPACE
postgres=# !q
$ ls -ld ts1
drwx----- 3 postgres postgres 4096 Feb 11 12:59 ts1
$ chmod a+r ts1
$ ls -ld ts1
drwxr--r-- 3 postgres postgres 4096 Feb 11 12:59 ts1
$ pg_ctl -D data restart -m fast -w
waiting for server to shut down.... done
server stopped
server starting
```

3.3.2 ファイルの更新

PostgreSQL ではテーブルやインデックスは個別のファイルとして作成されます。ファイルに対する I/O 状況を確認しました。

□ テーブル作成直後

テーブルを作成すると、対応するファイルが作成されます。テーブルとファイルのマッピングは oid2name コマンドや pg_class カタログの relfilenode 列で確認できます。

例 41 テーブルの作成とファイル

```
postgres=> CREATE TABLE data1(c1 VARCHAR(10), c2 VARCHAR(10)) ;
CREATE TABLE
postgres=> SELECT relfilenode FROM pg_class WHERE relname='data1' ;
relfilenode
-----
16446
(1 row)

$ cd data/base/16424/
$ ls -l 16446
-rw----- 1 postgres postgres 0 Feb 11 16:48 16446
```

テーブル作成の時点ではサイズ 0 の空ファイルであることがわかります。このテーブルにレコードを格納します。TRUNCATE 文で切り詰められた場合も同様です。

例 42 チェックポイント前のファイル

```
postgres=> INSERT INTO data1 VALUES('ABC', '123') ;  
INSERT 01  
postgres=> $q  
$ ls -l 16446  
-rw----- 1 postgres postgres 0 Feb 11 16:53 16446
```

チェックポイントが発生していないので、サイズは拡大されません。強制チェックポイントを実行するとファイルに書き込みが行われます（writer プロセスによる書き込みが行われる場合があります）。

例 43 チェックポイント後のファイル

```
postgres=# CHECKPOINT ;  
CHECKPOINT  
postgres=# $q  
  
$ ls -l 16446  
-rw----- 1 postgres postgres 8192 Feb 11 16:54 16446
```

ブロックサイズである 8 KB 単位で書き込まれていることがわかります。

データの格納と更新

PostgreSQL は追記型の RDBMS であるため、レコードの更新を行うと、旧レコードは変更されず、変更後のレコードがページ内に追加されます。

例 44 UPDATE の実行とファイルの状況

```
postgres=> UPDATE data1 SET c1='DEF', c2='456' ;
UPDATE 1
postgres=# CHECKPOINT ;
CHECKPOINT

$ od -a 16446
0000000 nul nul nul nul P bs stx dc2 soh nul nul nul sp nul @ us
0000020 nul sp eot sp ff dc4 etx nul ` us @ nul @ us @ nul
0000040 nul nul
*
0017700 ff dc4 etx nul nul
0017720 stx nul stx nul stx ( can nul ht D E F ht 4 5 6
0017740 vt dc4 etx nul ff dc4 etx nul nul nul nul nul nul nul nul
0017760 stx nul stx @ stx soh can nul ht A B C ht 1 2 3
0020000
```

ヘッダが書き込まれ、ブロックの末尾から最初のレコードが記録されていることがわかります。また更新後のレコードが追加されていることもわかります。この検証により、レコードの更新はブロック内で実行されていることがわかります。

□ 複数回の更新

レコードを複数回更新すると、ブロックはいっぱいになります。同一ブロック内に再利用可能な領域がある場合、ブロック内の不要領域が削除され、可能な限り同一ページが使用されます（Heap Only Tuples / HOT 機能）。

3.3.3 Visibility Map と Free Space Map

テーブルやインデックスは、单一（サイズにより複数）のファイルとして管理されます。データが格納されるファイル以外にオブジェクト単位に作成されるファイルが Visibility Map と Free Space Map です。Visibility Map と Free Space Map ファイルはテーブル、UNLOGGED テーブルおよびマテリアライズド・ビューにのみ作成されます。その他の永続オブジェクト（TOAST テーブル、TOAST インデックス、インデックス、シーケンス等）には作成されません。

□ Visibility Map

Visibility Map はガベージが存在するページ Freeze 処理が行われたページを記録するファイルです。テーブルのファイルに含まれる各ページを 2 ビットで管理します。ファイル名は「{RELFilenode}_vm」です。初期サイズは 8 KB です。テーブル作成後、最初のチェックポイントまたは VACUUM 時に作成されます。このファイルを参照することで、VACUUM 実行時にガベージが存在しないページをスキップし、VACUUM 処理の I/O 負荷を削減することができます。実際には不要なタプルが存在しないページ (Visible) が 32 個以上連続している場合に限り処理をスキップします。この値はソースコード (src/backend/commands/vacuumlazy.c) に SKIP_PAGES_THRESHOLD マクロで固定されています。

Visibility Map の内容を参照するためには Contrib モジュールの pg_visibility を利用することができます (PostgreSQL 9.6 から)。

例 45 Visibility Map の参照

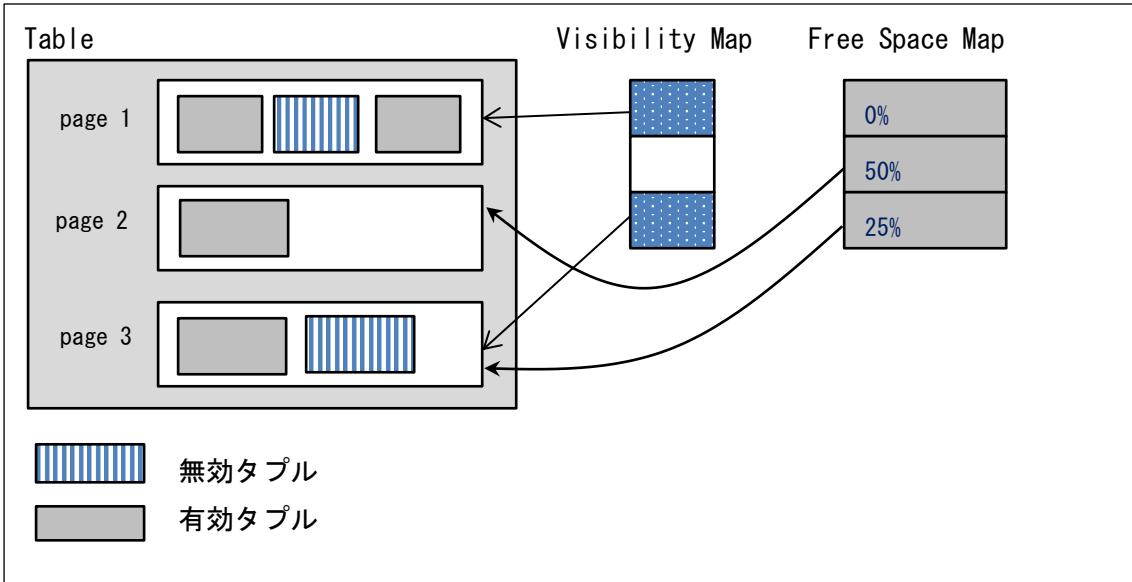
```
postgres=# CREATE EXTENSION pg_visibility ;
CREATE EXTENSION
postgres=# SELECT pg_visibility_map('data1') ;
 pg_visibility_map
-----
(0, t, f)
(1, t, f)
(2, t, f)
(3, t, f)
<<以下省略>>
```

□ Free Space Map

Free Space Map はテーブル・ファイル内の各ページがどの程度空き領域があるかを管理するファイルです。テーブルのファイルに含まれる各ページを 1 バイトで管理します。ファイル名は「{RELFilenode}_fsm」です。このファイルを参照することで、レコードの格納先を高速に発見することができるようになります。初期サイズは 24 KB です。テーブル作成後、最初の VACUUM 実行時に作成されます。また VACUUM 実行ごとに更新されます。

VACUUM は Visibility Map を参照しながら処理を行い、Free Space Map を更新します。

図 8 Visibility Map と Free Space Map



例 46 Visibility Map と Free Space Map

```
postgres=> SELECT relname, relfilenode FROM pg_class
           WHERE relname='data1' ;
relname | relfilenode
-----+-----
data1   |      16409
(1 row)

$ cd data/base/16385
$ ls 16409*
-rw---- 1 postgres postgres 8192 Feb 11 16:46 16409 ← Table
-rw---- 1 postgres postgres 24576 Feb 11 16:46 16409_fsm ← Free Space Map
-rw---- 1 postgres postgres 8192 Feb 11 16:46 16409_vm ← Visibility Map
$
```

3.3.4 VACUUM 動作

ここでは VACUUM 处理により、ファイルの内容がどのように変化するかを確認します。動作検証は自動 VACUUM を停止（パラメータ autovacuum=off）して行いました。

□ VACUUM CONCURRENT

VACUUM CONCURRENT 处理は、更新前情報を再利用可能な状態にマーキングします。

処理の前後で、ブロックの情報がどのように変化するか確認します。

例 47 データ準備（12 件挿入）

```
postgres=> CREATE TABLE data1 (c1 CHAR(500) NOT NULL, c2 CHAR(500) NOT NULL) ;
CREATE TABLE
postgres=> INSERT INTO data1 VALUES ('AAA', '111') ;
postgres=> INSERT INTO data1 VALUES ('BBB', '222') ;
postgres=> INSERT INTO data1 VALUES ('CCC', '333') ;
postgres=> INSERT INTO data1 VALUES ('DDD', '444') ;
postgres=> INSERT INTO data1 VALUES ('EEE', '555') ;
postgres=> INSERT INTO data1 VALUES ('FFF', '666') ;
postgres=> INSERT INTO data1 VALUES ('GGG', '777') ;
postgres=> INSERT INTO data1 VALUES ('HHH', '888') ;
postgres=> INSERT INTO data1 VALUES ('III', '999') ;
postgres=> INSERT INTO data1 VALUES ('JJJ', '000') ;
postgres=> INSERT INTO data1 VALUES ('AAA', 'aaa') ;
postgres=> INSERT INTO data1 VALUES ('AAA', 'bbb') ;
INSERT 0 1
postgres=# CHECKPOINT ;
CHECKPOINT
```

1 レコード 1 KB のテーブルを作成し、12 レコードを格納します。これにより 2 ブロックのテーブルが作成されます。

例 48 データ準備（ファイルの特定）

```
$ oid2name -d datadb1
From database "datadb1":
Filenode Table Name
-----
16470      data1

$ cd /usr/local/pgsql/data/base/16424
$ ls -l 16470
-rw----- 1 postgres postgres 16384 Feb 11 10:56 16470
```

例 49 ブロック初期状態（第1ブロック）

0000000 nul nul nul nul nul dle	U	stx	nak	soh	nul	nul	nul	4	nul	H	etx											
* 略																						
0001740	`	bel	nul	nul	G	G	G	sp	sp	sp	sp	sp	sp	sp	sp	sp	sp	sp	sp	sp	sp	sp
* 略																						
0002720	sp	`	bel	nul	nul	7	7	7	sp													
* 略																						
0003740	ack	nul	stx	nul	stx	bs	can	nul	`	bel	nul	nul	F	F	F	sp						
* 略																						
0004740	`	bel	nul	nul	6	6	6	sp	sp	sp	sp	sp	sp	sp	sp	sp	sp	sp	sp	sp	sp	
* 略																						
0005760	`	bel	nul	nul	E	E	E	sp	sp	sp	sp	sp	sp	sp	sp	sp	sp	sp	sp	sp	sp	
* 略																						
0006740	sp	`	bel	nul	nul	5	5	5	sp													
* 略																						
0007760	eot	nul	stx	nul	stx	bs	can	nul	`	bel	nul	nul	D	D	D	sp						
* 略																						
0010760	`	bel	nul	nul	4	4	4	sp	sp	sp	sp	sp	sp	sp	sp	sp	sp	sp	sp	sp	sp	
* 略																						
0012000	`	bel	nul	nul	C	C	C	sp	sp	sp	sp	sp	sp	sp	sp	sp	sp	sp	sp	sp	sp	
* 略																						
0012760	sp	`	bel	nul	nul	3	3	3	sp													
* 略																						
0014000	stx	nul	stx	nul	stx	bs	can	nul	`	bel	nul	nul	B	B	B	sp						
* 略																						
0015000	`	bel	nul	nul	2	2	2	sp	sp	sp	sp	sp	sp	sp	sp	sp	sp	sp	sp	sp	sp	
* 略																						
0016020	`	bel	nul	nul	A	A	A	sp	sp	sp	sp	sp	sp	sp	sp	sp	sp	sp	sp	sp	sp	
* 略																						
0017000	sp	`	bel	nul	nul	1	1	1	sp													
*																						

例 50 ブロック初期状態（第 2 ブロック）

```
0020000 nul nul nul nul nul k stx nak soh nul nul nul , nul X vt
0020020 nul sp eot sp nul nul nul nul x esc dle bs p etb dle bs
0020040 h dc3 dle bs ` si dle bs X vt dle bs nul nul nul nul
0020060 nul nul
*
0025720 nul nul nul nul nul nul nul nul 6 dc4 etx nul nul nul nul nul
* 略
0025760 ` bel nul nul L L L sp sp sp sp sp sp sp sp sp
* 略
0026740 sp sp sp sp sp sp sp ` bel nul nul b b b sp
* 略
0027760 eot nul stx nul stx bs can nul ` bel nul nul K K K sp
* 略
0030760 ` bel nul nul a a a sp sp sp sp sp sp sp sp
* 略
0032000 ` bel nul nul J J J sp sp sp sp sp sp sp sp
* 略
0032760 sp sp sp sp sp sp sp ` bel nul nul 0 0 0 sp
* 略
0034000 stx nul stx nul stx bs can nul ` bel nul nul I I I sp
* 略
0035000 ` bel nul nul 9 9 9 sp sp sp sp sp sp sp sp
* 略
0036020 ` bel nul nul H H H sp sp sp sp sp sp sp sp
* 略
0037000 sp sp sp sp sp sp sp ` bel nul nul 8 8 8 sp
0037020 sp sp
*
0040000
```

各ブロックの中間レコードを削除し、VACUUM 处理を実行します。

例 51 レコード削除と VACUUM 実行

```
postgres=> DELETE FROM data1 WHERE c1 IN ('CCC', 'JJJ') ;
DELETE 2
postgres=# CHECKPOINT ;
CHECKPOINT
postgres=> VACUUM data1 ;
VACUUM
```

この操作によりブロック内容がどのように変化したかを確認します。以下の2例は、VACUUM 後のブロック状態を示しています。ブロック内で、有効なレコードがブロック下部へ移動され、ヘッダとブロック下部の間に空き領域が作成されています。この動作により、連続した空き領域を作成していることがわかります。ただし、一部のレコード（下記例では 001740 C1='GGG', C2='777' のレコードと、003740 C1='LLL', C2='bbb' のレコード）は重複して格納されています。

注意

ブロック内のレコード整理自体は HOT (Heap On Tuples) の機能でも実施されています。ページ内に空き容量が不足していると確認された場合には、ブロック内で VACUUM 相当の動作を行います。以下のページに動作が記載されています。

http://lets.postgresql.jp/documents/tutorial/hot_2/hot2_2

例 52 VACUUM 处理後（第 1 ブロック）

0000000 nul nul nul nul sp 7 stx nak soh nul soh nul 4 nul P bel	*略
0001740 ` bel nul nul G G G sp	*略
0002720 sp sp sp sp sp sp sp ` bel nul nul 7 7 7 sp	*略
0003740 bel nul stx nul stx ht can nul ` bel nul nul G G G sp	*略
0004740 ` bel nul nul 7 7 7 sp sp sp sp sp sp sp sp	*略
0005760 ` bel nul nul F F F sp sp sp sp sp sp sp sp	*略
0006740 sp sp sp sp sp sp sp ` bel nul nul 6 6 6 sp	*略
0007760 enq nul stx nul stx ht can nul ` bel nul nul E E E sp	*略
0010760 ` bel nul nul 5 5 5 sp sp sp sp sp sp sp	*略
0012000 ` bel nul nul D D D sp sp sp sp sp sp sp	*略
0012760 sp sp sp sp sp sp sp ` bel nul nul 4 4 4 sp	*略
0014000 stx nul stx nul stx ht can nul ` bel nul nul B B B sp	*略
0015000 ` bel nul nul 2 2 2 sp sp sp sp sp sp sp	*略
0016020 ` bel nul nul A A A sp sp sp sp sp sp	*略
0017000 sp sp sp sp sp sp sp ` bel nul nul 1 1 1 sp	*

例 53 VACUUM 处理後（第 2 ブロック）

```
0020000 nul nul nul nul dle H stx nak soh nul soh nul , nul ` si
*略
0025760 ` bel nul nul L L L sp sp sp sp sp sp sp sp sp sp
*略
0026740 sp sp sp sp sp sp sp ` bel nul nul b b b sp
*略
0027760 enq nul stx nul stx ht can nul ` bel nul nul L L L sp
*略
0030760 ` bel nul nul b b b sp sp sp sp sp sp sp sp sp
*略
0032000 ` bel nul nul K K K sp sp sp sp sp sp sp sp sp
*略
0032760 sp sp sp sp sp sp sp ` bel nul nul a a a sp
*略
0034000 stx nul stx nul stx ht can nul ` bel nul nul I I I sp
*略
0035000 ` bel nul nul 9 9 9 sp sp sp sp sp sp sp sp
*略
0036020 ` bel nul nul H H H sp sp sp sp sp sp sp sp
*略
0037000 sp sp sp sp sp sp sp ` bel nul nul 8 8 8 sp
*
0040000
```

□ VACUUM 後の空間利用

VACUUM 处理で空き領域を作成できたため、データを格納します。

例 54 新規データ格納

```
postgres=> INSERT INTO data1 VALUES (' MMM', 'ccc') ;
INSERT 0 1
postgres=# CHECKPOINT ;
CHECKPOINT
```

例 55 INSERT 後（第 1 ブロック）

```
0000000 nul nul nul nul ` t stx nak soh nul soh nul 4 nul H etx
* 略
0001740 ` bel nul nul M M M sp sp sp sp sp sp sp sp sp
* 略
0002720 sp sp sp sp sp sp sp ` bel nul nul c c c sp
* 略
0003740 bel nul stx nul stx ht can nul ` bel nul nul G G G sp
* 略
0004740 ` bel nul nul 7 7 7 sp sp sp sp sp sp sp sp
* 略
```

上記のように、これまで重複して格納されていた 0001740 部分が上書きされていることがわかります。この動作はテーブルの FILLFACTOR 属性によって異なる場合があります。

□ VACUUM FULL

VACUUM FULL は、更新済レコードの再利用化だけでなく、ファイルの縮小も実施します。実際のファイルがどのように変化するかを確認します。

VACUUM FULL を実行すると、ファイル名とファイルの i-node が変更されていることから、新規のファイルが作成されることがわかります。このことから VACUUM FULL は既存のファイルを読み、レコードの整理をしながら新規ファイルを作成することでファイルの縮小を行っていることがわかります。

例 56 VACUUM 处理 (VACUUM FULL 実行)

```
$ ls -li 16470 ← VACUUM FULL 前のファイル
558969 -rw----- 1 postgres postgres 16384 Feb 11 11:39 16470

$ oid2name -d datadb1
From database "datadb1":
  Filenode  Table Name
-----
16476      data1 ← VACUUM FULL で Filenode が変更された。

$ ls -li 16476 ← ファイル名と i-node が変更された
558974 -rw----- 1 postgres postgres 16384 Feb 11 11:47 16476
```

複数のテーブルから構成されるデータベースに対して VACUUM FULL 文を実行する場合、同時に実行される VACUUM 処理は 1 テーブルだけです (CONCURRENT VACUUM も同様)。また、複数セグメントから構成されるテーブルに対して VACUUM FULL を実行する場合、全ファイルの VACUUM FULL 処理が完了するまでテーブルを構成するすべてのファイルは維持されます。このため大規模なテーブル（多数のセグメントから構成される）では、一時的にテーブルのストレージ容量が最大で 2 倍になる可能性があります。

□ 一括更新と部分更新

1,000 レコードのテーブルに対して全レコード一括で UPDATE 文を実行する場合と、各レコードに対して 1,000 回 UPDATE 文を実行する場合では、VACUUM 対象となる不要レコードの数が異なります。一括更新の場合は全レコードのコピーが作成されるのに対し、単一レコードの更新では VACUUM を実行する前に、ブロック内の不要レコードの再利用が行われるためです。

例 57 更新方法によるブロック数の差異

```
postgres=> CREATE TABLE data1(c1 NUMERIC, c2 VARCHAR(100), c3 VARCHAR(100)) ;
CREATE TABLE
-- insert 1000 records
postgres=> SELECT relpages, reltuples FROM pg_class WHERE relname='data1' ;
   relpages |   reltuples
-----+-----
     8 |      1000
(1 row)
postgres=> UPDATE data1 SET c1=c1+1 ; ← 一括更新
UPDATE 1000
postgres=> SELECT relpages, reltuples FROM pg_class WHERE relname='data1' ;
   relpages |   reltuples
-----+-----
    15 |      1000 ← ブロック数が増えている
(1 row)
postgres=> TRUNCATE TABLE data1 ;
-- insert 1000 records
postgres=> UPDATE data1 SET C2='TEST' WHERE c1=100 ; -- 1,000 回実行
UPDATE 1
postgres=> SELECT relpages, reltuples FROM pg_class WHERE relname='data1' ;
   relpages |   reltuples
-----+-----
     8 |      1000 ← ブロック数は増えていない
(1 row)
```

3.3.5 オープン・ファイル

PostgreSQL がインスタンス内でオープンするファイルについて調査しました。Linux の lsof コマンドを使用し、プロセスとオープンしているファイルの関係を調べています。

□ インスタンス起動直後

インスタンス起動後は、logger process プロセスがログファイルをオープンし、autovacuum launcher プロセスが pg_database カタログに対応するファイルをオープンします。

表 31 オープン・ファイル

プロセス	オブジェクト／ファイル
postmaster	/tmp/.s.PGSQL.{PORT}
logger process ⁸	{PGDATA}/pg_log/postgresql-{DATE}.log
autovacuum launcher	pg_database

□ ユーザー接続直後

クライアントが接続すると、バックエンド・プロセス (postgres) が、pg_am カタログをオープンします。またユーザーの接続が契機かは不明ですが、checkpointer プロセスがカレントの WAL ファイルをオープンします。

⁸ パラメーター logging_collector = on 設定時

表 32 追加オープン・ファイル

プロセス	オブジェクト／ファイル
postgres	pg_authid
postgres	pg_class
postgres	pg_attribute
postgres	pg_index
postgres	pg_am
postgres	pg_opclass
postgres	pg_amproc
postgres	pg_opclass_oid_index
postgres	pg_amproc_fam_proc_index
postgres	pg_class_oid_index
postgres	pg_attribute_relid_attnum_index
postgres	pg_index_indexrelid_index
postgres	pg_database_oid_index
postgres	pg_db_role_setting_databaseid_rol_index

□ 更新トランザクションの実行

更新トランザクションが発生すると、バックエンド・プロセスは更新対象のオブジェクトだけでなく、カレントの WAL をオープンします。

表 33 追加オープン・ファイル

プロセス	オブジェクト／ファイル
postgres	pg_xlog/{WALFILE}
postgres	更新オブジェクト

□ ユーザー切断直後

ユーザーがクローズすると、バックエンド・プロセスが停止するため、オープンしているファイルは元に戻ります。

表 34 オープン・ファイル

プロセス	オブジェクト／ファイル
autovacuum launcher	pg_database
logger process	{PGDATA}/pg_log/postgresql-{DATE}.log

上記の実験から、PostgreSQL は多くのファイルについて不要になった時点でクローズしていることがわかります。ログファイルや WAL ファイルも必要な時点でオープンし、不要になるとクローズしていることがわかります。

3.3.6 プロセスの動作 (WAL の書き込み)

トランザクションが確定すると WAL ファイルに書き込みが行われます。WAL の書き込みは wal writer プロセスまたは postgres プロセスが行います。一般的なドキュメントでは WAL の書き込みは wal writer プロセスのみが実行しているように書かれていますが、実際には postgres プロセスも書き込みを行います。WAL を書き込むプロセスの選択方法等について十分な検証を行っていません。

□ パラメーター synchronous_commit = on の場合

下記の例は、パラメータ `synchronous_commit = on` (デフォルト値) のインスタンスに対してテーブルを作成後、`INSERT` 文を発行した場合の `postgres` プロセスのシステムコールを出力しています。`postgres` プロセスが WAL の書き込みを行っています。

例 58 postgres プロセスが発行するシステムコール

表 35 実行されたシステムコール

行番号	処理
1	リモート・ホストから INSERT 文を受信
2	テーブル用ファイルのアクセス (fsm ファイルを確認)
3	テーブル用ファイルのアクセス (データファイルをオープン)
4	テーブル用ファイルの検索
5	テーブル用ファイルの検索
6	プロセス ID 7487 (writer process) ヘシグナル送信
7	テーブル用ファイルの初期化 (ページの初期化)
8	WAL のオープン
9	WAL の検索
10	WAL の書き込み
11	WAL の同期
12	UDP ネットワークへ実行結果送信
13	TCP セッションへ実行結果送信

□ パラメーター `synchronous_commit = off` の場合

パラメーター `synchronous_commit` を `off` に設定したところ、WAL の書き込みは `wal writer` プロセスが行うようになりました。`postgres` プロセスは WAL ファイルにアクセスしていません。データファイルの読み込みを行った後、`wal writer` プロセス（プロセス ID 7635）に対して `SIGUSR1` シグナルを送信しています。

例 59 postgres プロセスが発行する主なシステムコール

```
recvfrom(10, "Q¥0¥0¥0. insert into data1 values (1"..., 8192, 0, NULL, NULL) = 47
open("base/16499/16519_fsm", O_RDWR)      = 34
lseek(34, 0, SEEK_END)                  = 40960
lseek(34, 0, SEEK_SET)                  = 0
read(34, "¥0¥0¥001!¥312¥0¥0¥0¥0¥30¥0¥0 ¥0 ¥4 ¥0¥0¥0¥0¥350¥350¥0¥350"..., 8192) =
8192
open("base/16499/16519", O_RDWR)        = 35
lseek(35, 44269568, SEEK_SET)        = 44269568
read(35, "¥1¥0¥0¥0¥260774¥2P¥f¥0 ¥4 ¥0¥0¥0¥0¥330¥237D¥0¥260¥237D¥0"..., 8192) = 8192
kill(7635, SIGUSR1)                  = 0
sendto(9, "¥2¥0¥0¥0¥320¥3¥0¥0s@¥0¥0¥t¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0"..., 976, 0, NULL, 0) =
976
sendto(9, "¥2¥0¥0¥0¥320¥3¥0¥0s@¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0"..., 976, 0, NULL, 0) =
976
sendto(9, "¥2¥0¥0¥0000¥2¥0¥0s@¥0¥0¥5¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0"..., 560, 0, NULL, 0) =
560
sendto(10, "C¥0¥0¥0¥17INSERT 0 1¥0Z¥0¥0¥0¥5I", 22, 0, NULL, 0) = 22
```

例 60 wal writer プロセスが発行する主なシステムコール

```
--- SIGUSR1 (User defined signal 1) @ 0 (0) ---
write(4, "¥0", 1)                      = 1
rt_sigreturn(0x4)                      = -1 EINTR (Interrupted system call)
read(3, "¥0", 16)                      = 1
open("pg_xlog/000000100000010000017", O_RDWR) = 5
write(5, "]¥320¥6¥0¥1¥0¥27¥1¥0¥0¥0¥0¥0¥0¥0¥0¥0L<¥302x¥322cuS"..., 8192) = 8192
fdatasync(5)
```

wal writer プロセスは SIGUSR1 シグナルを受けて、パイプの処理を行い、WAL ファイルに書き込んでいます。

□ 大規模更新トランザクション

更新量が WAL バッファに格納できない場合、トランザクションの確定前に WAL ファイルが更新されます。その際は postgres プロセスと wal writer プロセスが相互に通信しながら両プロセスが WAL ファイルを更新します。

3.3.7 プロセスの動作（checkpointerによる書き込み）

`checkpointer` プロセスは文字通りチェックポイントを実行するプロセスです。チェックポイントが発生すると、データ・バッファをデータファイルに書き込み、共有バッファとストレージの内容を一致させます。以下の例は `CHECKPOINT` 文を実行した場合に `checkpointer` プロセスが実行しているシステムコールをトレースしたものです。

例 61 checkpointer による書き込み処理（一部）

```
open("pg_clog/0000", O_RDWR|O_CREAT, 0600) = 5
lseek(5, 0, SEEK_SET) = 0
write(5, "@ooooooooooooooooooooo"..., 8192) = 8192
fsync(5) = 0
close(5) = 0
open("base/16499/24980_vm", O_RDWR) = 5
write(5, "¥0¥0¥0¥0¥210¥253h¥262¥0¥0¥0¥0¥0¥0¥0¥0¥0¥200¥365?¥0"..., 8192) = 8192
open("base/16499/24980", O_RDWR) = 7 ← データファイル
lseek(7, 442368, SEEK_SET) = 442368
write(7, "¥0¥0¥0¥3r¥262¥0¥0¥1¥0¥f¥4¥200¥16¥0 ¥4 -¥10¥0:¥0¥1¥0"..., 8192) = 8192
--- lseek / write を繰り返し ---
open("base/16499/12725", O_RDWR) = 10
write(10, "¥0¥0¥0¥0¥30¥1¥0¥220¥0¥270¥17¥0 ¥4 ¥0¥0¥0¥00¥231|¥3"..., 8192) = 8192
fsync(7) = 0
fsync(8) = 0
fsync(5) = 0
fsync(10) = 0
lseek(14, 7528448, SEEK_SET) = 7528448
write(14, "}¥320¥5¥0¥1¥0¥0¥0¥0¥340r¥262¥0¥0¥0¥0&¥0¥0¥0¥0¥0¥0¥0"..., 8192) = 8192
fdatsync(14) = 0
open("global/pg_control", O_RDWR) = 11
write(11, 02x¥322cu$¥251¥3¥0¥0¥2672¥1¥f¥6¥0¥0¥311¥237S¥0¥0¥0¥0"..., 240) = 240
fsync(11) = 0
close(11) = 0
```

`pg_clog` ディレクトリ内にチェックポイント情報を書き込み、次に `vm` ファイルを更新しています。その後 1 ブロックずつデータファイルを更新し、最後に `pg_control` ファイルにチェックポイント完了の情報を書いています。

3.3.8 プロセスの動作 (writer による書き込み)

checkpointer プロセスが比較的長い周期で書き込みを行っているのに対し、writer プロセスは変更されたページ（ダーティ・バッファ）を短い周期で少量ずつ書き込んでいます。writer プロセスの書き込みにより、チェックポイントにおける I/O のピークを予防することができます。writer プロセスの書き込み間隔はパラメーター bgwriter_delay (デフォルト値 200ms) で決められています。待ち時間は Linux / UNIX プラットフォームでは select (2) システムコール、Windows 環境では Windows API WaitForMultipleObjects で待機します (WaitLatch 関数 backend/port/win32_latch.c または backend/port/pg_latch.c)。

書き込みブロック数の最大値は、パラメーター bgwriter_lru_maxpages で決まります。デフォルト値は 100 です。実際の書き込みブロック数は最近要求されたブロック数に、パラメーター bgwriter_lru_multiplier の値を掛けて計算されます。その場合でもパラメーター bgwriter_lru_maxpages を超えることはありません。パラメーター bgwriter_lru_maxpages と最近必要とされた平均ページ数等による予測される必要ページ数が、再利用可能なページ数よりも大きい場合にのみ実際の書き込みが行われます。

表 36 writer プロセスに関するパラメーター

パラメーター	説明	デフォルト値
bgwriter_delay	writer プロセスの書き込み間隔	200ms
bgwriter_lru_maxpages	書き込み最大ページ数、0 にすると書き込みは行われない	100
bgwriter_lru_multiplier	平均要求ページに掛ける値	2.0

3.3.9 プロセスの動作 (archiver)

archiver プロセスは書き込みが完了した WAL ファイルのアーカイブ化を行うプロセスです。以下のタイミングでアーカイブ処理をおこないます。

- 60 秒間隔で定期的に実施
- SIGUSR1 シグナルを受信したことを契機に実施

実際には以下の処理をおこないます。

- {PGDATA}/pg_xlog/archive_status ディレクトリを検索
- {WALFILE}.ready ファイルを発見
- パラメーター archive_command で指定されたコマンドを、system 関数を使って実行
- 実行ステータスを確認
- {PGDATA}/pg_xlog/archive_status/{WALFILE}.ready ファイルを {PGDATA}/pg_xlog/archive_status/{WALFILE}.done に変更

例 62 archiver プロセスが発行する主なシステムコール

パラメーターarchive_commandの実行パラメーターを取得するためには、パラメーター log_min_messagesにDEBUG3を指定します。以下のログが出力されます。

例 63 archive command 実行ログ

DEBUG: executing archive command "/bin/cp pg_xlog/0000000100000000000000071 /arch/0000000100000000000000071"

アーカイブ対象の WAL が決定した時点でログを出力するためには、パラメーター `log_min_messages` に `DEBUG1` を指定します。以下のログが出力されます。

例 64 アーカイブ対象 WAL のログ

3.4 オンライン・バックアップ

3.4.1 オンライン・バックアップの動作

オンライン・バックアップはインスタンス起動状態でバックアップを取得する方法です。インスタンスに対して `pg_start_backup` 関数を発行し、データベース・クラスター以下の全ファイルをバックアップします。バックアップが完了したら `pg_stop_backup` 関数を実行します。これらの操作は `pg_basebackup` コマンドで自動的に行うことができます。オンライン・バックアップはデータベース・クラスター（および外部テーブル空間）全体のバックアップを取得する必要があります。データベース単位のバックアップは `pg_dump` コマンド等の論理バックアップを使用する必要があります。

□ `pg_start_backup` 関数

オンライン・バックアップの開始を宣言します。この関数が実行されると、バックアップ開始時の WAL オフセット値が表示されます。またラベル・ファイル「`{PGDATA}/backup_label`」が作成されます。ラベル・ファイルには、バックアップの開始時間や WAL の情報が記載されます。

例 65 オンライン・バックアップの開始

```
postgres=# SELECT pg_start_backup(now()::text) ;
pg_start_backup
-----
0/59000028
(1 row)
postgres=#
```

例 66 `pg_start_backup` 関数実行時の `backup_label` ファイル

```
$ cat data/backup_label
START WAL LOCATION: 0/6000028 (file 000000010000000000000006)
CHECKPOINT LOCATION: 0/6000060
BACKUP METHOD: pg_start_backup
BACKUP FROM: master
START TIME: 2017-02-11 12:47:42 JST
LABEL: 2017-02-11 12:47:42.2466+09
```

□ pg_stop_backup 関数

オンライン・バックアップの完了を宣言します。この関数が実行されると、バックアップ終了時の WAL オフセット値が表示されます。また pg_start_backup 関数実行時に作成されたラベル・ファイル「{PGDATA}/backup_label」は削除され、アーカイブ・ログディレクトリに新規のラベル・ファイルが作成されます。

例 67 オンライン・バックアップの終了

```
postgres=# SELECT pg_stop_backup();  
NOTICE: pg_stop_backup complete, all required WAL segments have been archived  
pg_stop_backup  
-----  
0/5B0000B8  
(1 row)
```

パラメーター archive_mode=off の状態でオンライン・バックアップを実行すると、pg_stop_backup 関数実行時に以下の警告が表示されます。

例 68 オンライン・バックアップの終了 (archive_mode=off)

```
postgres=# SELECT pg_stop_backup();  
NOTICE: WAL archiving is not enabled; you must ensure that all required WAL  
segments are copied through other means to complete the backup  
pg_stop_backup  
-----  
0/590000B8  
(1 row)
```

□ Non-Exclusive モード

PostgreSQL 9.6 では Non-Exclusive モードによるオンライン・バックアップが可能になりました。pg_start_backup 関数の exclusive パラメーターに false を指定することで実行できます。このパラメーターのデフォルト値は true で、旧バージョンとの互換性が維持されています。pg_start_backup 関数をと pg_stop_backup 関数は、同じモードの指定を行う必要があります。Non-Exclusive モードで pg_stop_backup 関数を実行すると、Exclusive モードとは出力結果が異なります。

例 69 Non-Exclusive モードによるオンライン・バックアップ

```
postgres=# SELECT pg_start_backup(now()::text, false, false) ;
pg_start_backup
-----
0/8000028
(1 row)

postgres=# SELECT pg_stop_backup() ;
ERROR: non-exclusive backup in progress
HINT: Did you mean to use pg_stop_backup('f')?
postgres=#
postgres=# SELECT pg_stop_backup(false) ;
NOTICE: pg_stop_backup complete, all required WAL segments have been archived
pg_stop_backup
-----
(0/8000130, "START WAL LOCATION: 0/8000028 (file 000000010000000000000000)+"
CHECKPOINT LOCATION: 0/8000060 +
BACKUP METHOD: streamed +
BACKUP FROM: master +
START TIME: 2017-02-11 12:50:16 JST +
LABEL: 2017-02-11 12:50:15.900273+09 +
", "16384 /home/postgres/ts1 +
")
(1 row)
```

3.4.2 バックアップ・ラベル・ファイル

バックアップ・ラベル・ファイルは、オンライン・バックアップの情報が記載されるテキストファイルです。pg_start_backup 関数を実行すると、「{PGDATA}/backup_label」ファイルとして作成されます。pg_stop_backup 関数を実行すると、backup_label ファイルは内容を再読み込みした後削除され、アーカイブログと同じディレクトリに以下のフォーマットの名前で作成されます。

構文 2 バックアップ・ラベル・ファイル名のフォーマット

```
{WALFILE}. {WALOFFSET}. backup
```

インスタンス停止時に `backup_label` ファイルが残っている場合、同じディレクトリに `backup_label.old` ファイルに名前が変更されます。この処理は、`pg_ctl stop -m immediate` コマンドを実行した場合でも実施されます。

インスタンス起動時に `backup_label` ファイルが残っている場合も、ファイル名が `backup_label.old` にファイル名が変更されてからインスタンスが起動されます。ログには何も出力されません。

ラベル・ファイルはテキスト形式のファイルで、オンライン・バックアップに関する情報が記載されています。

表 37 バックアップ・ラベル・ファイルの内容

行	出力内容	説明	備考
1	START WAL LOCATION:	バックアップ開始時の WAL	
2	STOP WAL LOCATION:	バックアップ終了時の WAL	
3	CHECKPOINT LOCATION:	チェックポイント情報	
4	BACKUP METHOD:	バックアップ方法	
5	BACKUP FROM:	バックアップ元	
6	START TIME:	開始時刻	
7	LABEL:	バックアップ・ラベル	pg_start_backup 関数で指定
8	STOP TIME:	終了時刻	pg_stop_backup 関数により追記

最終行の STOP TIME は `pg_stop_backup` 関数を実行した場合に限り出力されます。

例 70 ラベル・ファイル

```
START WAL LOCATION: 0/8000028 (file 0000000100000000000000000008)
STOP WAL LOCATION: 0/8000130 (file 0000000100000000000000000008)
CHECKPOINT LOCATION: 0/8000060
BACKUP METHOD: streamed
BACKUP FROM: master
START TIME: 2017-02-11 12:50:16 JST
LABEL: 2017-02-11 12:50:15.900273+09
STOP TIME: 2017-02-11 12:50:36 JST
```

3.4.3 レプリケーションとオンライン・バックアップ

オンライン・バックアップ関数を実行できるのはマスター・インスタンスのみです。スレーブ・インスタンスで実行すると「ERROR: recovery is in progress」エラーが発生します。

例 71 スレーブ・インスタンスでオンライン・バックアップ開始

```
postgres=# SELECT pg_start_backup(now()::text) ;
ERROR: recovery is in progress
HINT: WAL control functions cannot be executed during recovery.
postgres=#
```

マスター・インスタンスでオンライン・バックアップ実行中にスレーブ・インスタンスで `pg_is_in_backup` 関数を実行すると f (オンライン・バックアップ中ではない) が返ります。

例 72 スレーブ・インスタンスでオンライン・バックアップのチェック

```
postgres=# SELECT pg_is_in_backup() ;
pg_is_in_backup
-----
f
(1 row)
```

3.4.4 オンライン・バックアップとインスタンス停止

オンライン・バックアップ中はsmartモードを指定したインスタンス停止は失敗します。停止に失敗した場合でもインスタンスはシャットダウン中のステータスになるため一般ユーザーは接続できません。オンライン・バックアップ中にインスタンスを強制的に停止するにはfastモードを指定します。

例 73 オンライン・バックアップ中のインスタンス停止

```
postgres=# SELECT pg_start_backup(now()::text) ;
pg_start_backup
-----
0/A5000028
(1 row)

postgres=# \q
$
$ pg_ctl -D data stop -m smart
WARNING: online backup mode is active
Shutdown will not complete until pg_stop_backup() is called.

waiting for server to shut down..... failed
pg_ctl: server does not shut down
HINT: The "-m fast" option immediately disconnects sessions rather than
waiting for session-initiated disconnection.

$ echo $?
1
$ pg_ctl -D data stop -m fast
waiting for server to shut down....
done
server stopped
$
```

3.5 ファイルのフォーマット

3.5.1 postmaster.pid

postmaster.pid ファイルはデータベース・クラスター内に作成されるテキストファイルです。インスタンス起動時に作成され、インスタンス正常終了時には削除されます。マニュアルには「ファイルの存在によりインスタンスの稼働を確認する」と書かれていますが、実際にはファイルの 1 行目のプロセス ID を確認しています。postmaster.pid に書き込まれる情報の定義はソースコード (src/include/miscaadmin.h) 内に「LOCK_FILE_LINE_*」マクロとして行番号が定義されています。

表 38 postmaster.pid ファイルの内容

行番号	出力内容	備考
1	postmaster プロセス ID	10 進数
2	データベース・クラスターのパス	
3	インスタンス開始時刻	
4	Ipv4 接続待ちポート番号	
5	ローカル接続用ソケット作成ディレクトリ	
6	Ipv4 接続待ちアドレス	
7	System V 共有メモリーのキー、ID 情報	10 進数

ファイルの 1 行目に書かれた数字を ID を持つプロセスが存在しない場合、pg_ctl status コマンドや pg_ctl stop コマンドは動作しません。

例 74 postmaster.pid ファイルの内容

```
$ cat data/postmaster.pid
15141
/home/postgres/data
1475205932
5432
/tmp
*
5432001    196608$
```

□ パラメーターexternal_pid_file

パラメーターexternal_pid_fileにファイル名（またはディレクトリ名を含むパス）を指定すると、postmaster.pidファイルに加えてpostmasterプロセスのIDを書き込むファイルが作成されます。ただし、external_pid_fileで指定されたファイルに出力される情報はpostmasterのプロセスIDだけです。またpg_ctlコマンドはexternal_pid_fileを参照しません。postmaster.pidファイルが保護モード-rw-----で作成されるのに対し、external_pid_fileで指定されたファイルは、保護モード-rw-r--r--で作成されるため、PostgreSQL管理者以外のユーザーが参照することができます。

インスタンス起動時にパラメーターexternal_pid_fileに指定されたファイルが作成できない場合でも起動処理は正常に行われます。

3.5.2 postmaster.opts

postmaster.optsファイルはpostmasterプロセス起動時のパラメーターを保持するファイルで、インスタンス起動時に作成されます。インスタンス停止時にも削除されません。インスタンス起動時にこのファイルに書き込み権限が無い場合、インスタンス起動はエラーになります。pg_ctl startコマンドで指定されたパラメーターがそのまま出力されます。

例 75 postmaster.opts ファイルの内容

```
$ pg_ctl start -D data
$ cat data/postmaster.opts
/usr/local/pgsql/bin/postgres "-D" "data"
$ pg_ctl stop
$ pg_ctl start
$ cat data/postmaster.opts
/usr/local/pgsql/bin/postgres
$
```

3.5.3 PG_VERSION

PG_VERSIONファイルは、データベース・クラスターおよびテーブル空間用ディレクトリ内のデータベースoidディレクトリ以下に自動的に作成されるテキストファイルです。基本的にはメジャーバージョンが記載されています。単純なテキストファイルですが、データベース・クラスター内のファイルが失われるとインスタンスが起動できず、データベースoidディレクトリ内のファイルが失われると該当データベースに接続できなくなります。

例 76 PG_VERSION ファイルの内容

```
$ cd /usr/local/pgsql/data
$ cat PG_VERSION
9.6
$
```

3.5.4 pg_control

pg_control ファイルは{PGDATA}/global ディレクトリに保存される小さなバイナリ・ファイルです。 サイズは 8 KB です (src/include/catalog/pg_control.h 内に PG_CONTROL_SIZE で定義)。 実際に書き込まれるデータは構造体 ControlFileData で定義されています (src/include/catalog/pg_control.h)。

□ pg_control ファイルの内容

pg_control ファイルの主な内容は pg_controldata コマンドまたは専用の関数で確認することができます。

例 77 pg_controldata コマンドの実行

```
$ pg_controldata data
pg_control version number:          960
Catalog version number:             201608131
Database system identifier:        6335932445819631823
Database cluster state:            in production
pg_control last modified:          Fri Feb 11 12:55:16 2017
Latest checkpoint location:        0/9000098
Prior checkpoint location:         0/8000060
Latest checkpoint's REDO location: 0/9000060
Latest checkpoint's REDO WAL file: 0000000100000000000000000000
Latest checkpoint's TimeLineID:    1
<<途中省略>>
Latest checkpoint's newestCommitTsXid:0
Time of latest checkpoint:         Fri Feb 11 12:55:16 2017
Fake LSN counter for unlogged rels: 0/1
Minimum recovery ending location:  0/0
Min recovery ending loc's timeline: 0
Backup start location:            0/0
Backup end location:              0/0
End-of-backup record required:    no
<<途中省略>>
Blocks per segment of large relation: 131072
WAL block size:                   8192
Bytes per WAL segment:            16777216
Maximum length of identifiers:    64
Maximum columns in an index:     32
Maximum size of a TOAST chunk:   1996
Size of a large-object chunk:    2048
Date/time type storage:          64-bit integers
Float4 argument passing:          by value
Float8 argument passing:          by value
Data page checksum version:      0
```

バージョン情報やデータベースの ID のような固定情報、最終更新時刻、インスタンスの状態、チェックポイントの情報、コンパイル時の情報が出力されます。pg_controldata

コマンドの出力結果から判るとおり、pg_control はチェックポイント時およびインスタンスのステータス変更時に情報が更新されます。

pg_control ファイルの情報は以下の関数を使うことでも取得できます。これらの関数は PostgreSQL 9.6 で追加されました。

表 39 pg_control 情報取得関数

関数名	説明
pg_control_checkpoint	チェックポイントの実行状況
pg_control_init	コンパイル時に決定された各種制限値の情報
pg_control_recovery	バックアップ／リカバリー情報
pg_control_system	バージョン情報、システム ID 情報

例 78 pg_control_system 関数の実行

```
postgres=> \x
Expanded display is on.
postgres=> SELECT * FROM pg_control_system() ;
-[ RECORD 1 ]-----+
pg_control_version      | 960
catalog_version_no       | 201608131
system_identifier         | 6335932445819631823
pg_control_last_modified | 2017-02-11 13:00:16+09
```

□ Database cluster state

pg_controldata コマンド実行結果の Database cluster state には現在の pg_control ファイルが認識しているデータベース・クラスターの状態が出力されます。

表 40 Database cluster state 出力

値	出力	説明	備考
0	starting up	インスタンスは起動中	
1	shut down	インスタンスは正常終了	
2	shut down in recovery	リカバリー中の停止	
3	shutting down	終了中	
4	in crash recovery	クラッシュ・リカバリ中	
5	in archive recovery	レプリケーション実行中	
6	in production	正常起動状態	
-	unrecognized status code	ステータス不明	pg_control 破壊？

□ Database system identifier

Database system identifier 項目には、各データベース・クラスターを一意に識別する ID 番号が output されます（符号なし 64 ビット整数）。この番号はデータベース・クラスター作成時に決定され、変更されることはありません。ストリーミング・レプリケーションは、この ID が同一であるデータベース・クラスター間で行われます。また、WAL ファイルの最初のブロック内にもこの番号が記録されます（XlogLongPageHeaderData 構造体）。これにより異なるデータベースの WAL ファイルを誤ってリカバリに使用することができます。

Database system identifier は以下のコードにより一意な番号を生成しています（BootStrapXLOG 関数内）。

例 79 BootStrapXLOG 関数（src/backend/access/transam/xlog.c）

```
uint64 sysidentifier;

gettimeofday(&tv, NULL);
sysidentifier = ((uint64) tv.tv_sec) << 32;
sysidentifier |= ((uint64) tv.tv_usec) << 12;
sysidentifier |= getpid() & 0xFFF;
```

例 80 WAL ファイルの先頭 (src/include/access/xlog_internal.h)

```
/*
 * When the XLP_LONG_HEADER flag is set, we store additional fields in the
 * page header. (This is ordinarily done just in the first page of an
 * XLOG file.) The additional fields serve to identify the file accurately.
 */
typedef struct XLogLongPageHeaderData
{
    XLogPageHeaderData std;          /* standard header fields */
    uint64      xlp_sysid;        /* system identifier from pg_control */
    uint32      xlp_seg_size;     /* just as a cross-check */
    uint32      xlp_xlog_blkksz;  /* just as a cross-check */
} XLogLongPageHeaderData;
```

3.5.5 pg_filenode.map

pg_filenode.map ファイルは{PGDATA}/global ディレクトリおよび、データベース用ディレクトリに保存される小さなバイナリ・ファイルです。ファイルのサイズは 512 バイトです。このファイルには、一部のシステム・カタログの OID と、実際のファイル名を対応させる情報が格納されています。格納フォーマットは RelMapFile 構造体で定義されています。

例 81 格納フォーマット (src/backend/utils/cache/relmapper.c)

```
typedef struct RelMapping
{
    Oid             mapoid;           /* OID of a catalog */
    Oid             mapfilenode;      /* its filenode number */
} RelMapping;

typedef struct RelMapFile
{
    int32          magic;           /* always RELMAPPER_FILEMAGIC */
    int32          num_mappings;    /* number of valid RelMapping entries */
    RelMapping     mappings[MAX_MAPPINGS];
    pg_crc32c     crc;             /* CRC of all above */
    int32          pad;             /* to make the struct size be 512 exactly */
} RelMapFile;
```

最初の4バイト (magic) は固定値 0x592711 (5842711) です。次の4バイトにマッピングの個数が格納されます。下記の例は特定のデータベース・ディレクトリ以下のファイルの内容です。

例 82 pg_filenode.map ファイルの内容

```
$ od -t u4 pg_filenode.map
0000000 5842711 15 1259 1259 ← 15 個のエントリー
0000020 1249 1249 1255 1255 ← OID とファイル名セット
0000040 1247 1247 2836 2836
0000060 2837 2837 2658 2658
0000100 2659 2659 2662 2662
0000120 2663 2663 3455 3455
0000140 2690 2690 2691 2691
0000160 2703 2703 2704 2704
0000200 0 0 0 0
*
0000760 0 0 983931237 0 ← CRC とパディング
0001000
```

`$(PGDATA)/global` ディレクトリにある `pg_filenode.map` には以下のテーブルが含まれます。

- `pg_pltemplate`
- `pg_pltemplate_name_index`
- `pg_tablespace`
- `pg_shdepend`
- `pg_shdepend_depender_index`
- `pg_shdepend_reference_index`
- `pg_authid`
- `pg_auth_members`
- `pg_database`
- `pg_shdescription` (TOAST テーブル、TOAST インデックスを含む)
- `pg_shdescription_o_c_index`
- `pg_database_datname_index`
- `pg_database_oid_index`
- `pg_authid rolname_index`
- `pg_authid oid_index`
- `pg_auth_members_role_member_index`
- `pg_auth_members_member_role_index`
- `pg_tablespace_oid_index`
- `pg_tablespace_spcname_index`
- `pg_db_role_setting` (TOAST テーブル、TOAST インデックスを含む)
- `pg_db_role_setting_databaseid_rol_index`
- `pg_shseclabel` (TOAST テーブル、TOAST インデックスを含む)
- `pg_shseclabel_object_index`
- `pg_replication_origin`
- `pg_replication_origin_roiident_index`
- `pg_replication_origin_roname_index`

データベース・ディレクトリにある `pg_filenode.map` には以下のテーブルが含まれます。

- `pg_class`
- `pg_attribute`
- `pg_proc` (TOAST テーブル、TOAST インデックスを含む)
- `pg_type`
- `pg_attribute_relid_attnam_index`

- pg_attribute_relid_attnum_index
- pg_class_oid_index
- pg_class_relname_nsp_index
- pg_class_tblspc_relfilenode_index
- pg_proc_oid_index
- pg_proc_proname_args_nsp_index
- pg_type_oid_index
- pg_type_typname_nsp_index

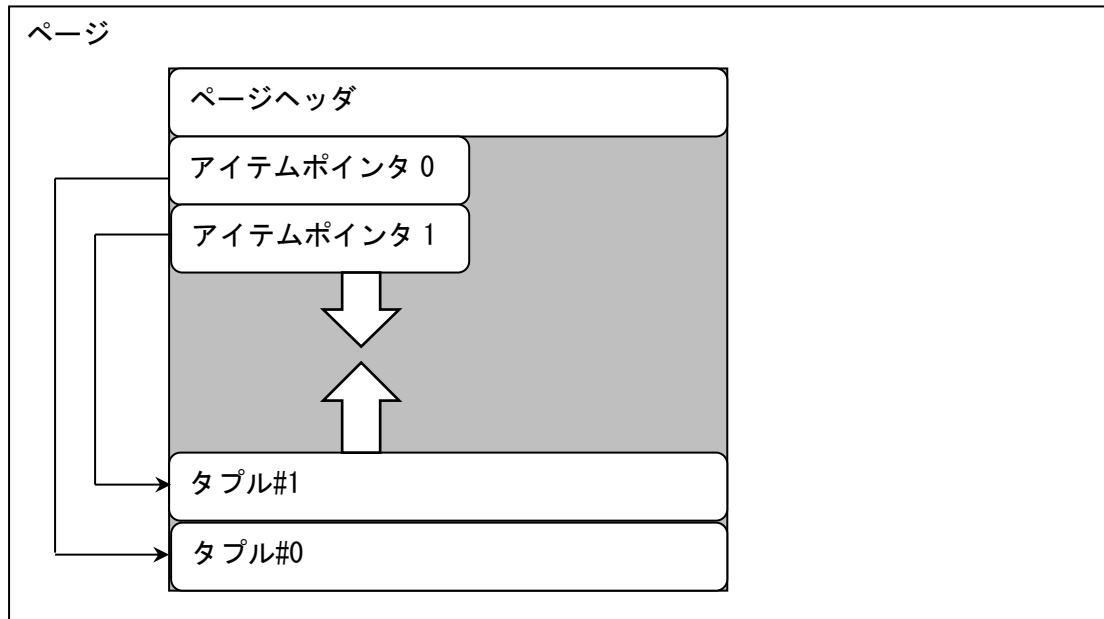
3.6 ブロックのフォーマット

3.6.1 ブロックとページ

PostgreSQL における I/O はブロック単位で行われます。ブロックのサイズは標準で 8 KB です。この値を変更するためには「src/include/pg_config.h」内の「#define BLOCKSZ 8192」を変更して再コンパイルが必要です。ブロック内にはページが格納されていますが、現在の PostgreSQL では 1 ブロックに 1 ページになるため、ブロックとページを同一のものと扱うことができます。ブロック（ページ）は 1 ファイル内に 131,072 個格納することができます。データファイルがこのサイズ ($8 \text{ KB} \times 131,072 = 1 \text{ GB}$) を超えると新しいファイルが作成されます。この値は pg_config.h ファイルの RELSEG_SIZE マクロで決定されています。

各ページにはページヘッダとページ内の各タプルへのポインターが格納されます。アイテムポインタは、ページ内のタプルの位置を示します。タプルはページの終端からページの先頭に向けて格納されます。一方でアイテムポインタはページヘッダの直後からページの終端に向けて追加されます。

図 9 ページの構成



ページヘッダは、「src/include/storage/bufpage.h」内の構造体 PageHeaderData で定義されています。

例 83 PageHeaderData

```
typedef struct PageHeaderData
{
    /* XXX LSN is member of *any* block, not only page-organized ones */
    PageXLogRecPtr pd_lsn;           /* LSN: next byte after last byte of xlog
                                         * record for last change to this page */

    uint16      pd_checksum;         /* checksum */
    uint16      pd_flags;           /* flag bits, see below */
    LocationIndex pd_lower;         /* offset to start of free space */
    LocationIndex pd_upper;         /* offset to end of free space */
    LocationIndex pd_special;       /* offset to start of special space */
    uint16      pd_pagesize_version;
    TransactionId pd_prune_xid;    /* oldest prunable XID, or zero if none */
    ItemIdData   pd_linp[FLEXIBLE_ARRAY_MEMBER]; /* line pointer array */
} PageHeaderData;
```

表 41 PageHeaderData 内部

変数	説明	備考
pd_lsn	Log Sequence Number	
pd_checksum	チェックサム	9.3 で変更
pd_flags	フラグ	
pd_lower	ページ内の空き領域の開始位置	
pd_upper	ページ内の空き領域の終了位置	
pd_special	ページ内のスペシャルスペースの終了域	
pd_pagesize_version	ページ・サイズとバージョン情報	
pd_prune_xid	ページ上でもっとも古い、切り詰められていない XMAX 値。存在しなければゼロ。	
pd_linp[]	アイテムポインタ	

3.6.2 タプル

タプル（レコード）の構造は、「タプルヘッダ」「NULL ビットマップ」「データ」から構成されています。タプルヘッダは、「src/include/access/htup_details.h」で定義されています。タプルヘッダ先頭の t_heap は、トランザクション関連の情報が記録されます。NULL ビットマップは t_bits フィールドに格納されます。

例 84 HeapTupleHeaderData 構造体

```
struct HeapTupleHeaderData
{
    union
    {
        HeapTupleFields t_heap;
        DatumTupleFields t_datum;
    } t_choice;
    ItemPointerData t_ctid; /* current TID of this or newer tuple (or a
                           * speculative insertion token) */

    /* Fields below here must match MinimalTupleData! */
    uint16          t_infomask2; /* number of attributes + various flags */
    uint16          t_infomask;  /* various flag bits, see below */
    uint8           t_hoff;      /* sizeof header incl. bitmap, padding */
    /* ^ - 23 bytes - ^ */
    bits8          t_bits[FLEXIBLE_ARRAY_MEMBER]; /* bitmap of NULLs */
    /* MORE DATA FOLLOWS AT END OF STRUCT */
};
```

例 85 HeapTupleFields (t_heap)

```
typedef struct HeapTupleFields
{
    TransactionId t_xmin; /* inserting xact ID */
    TransactionId t_xmax; /* deleting or locking xact ID */

    union
    {
        CommandId  t_cid; /* inserting or deleting command ID, or both */
        TransactionId t_xvac; /* old-style VACUUM FULL xact ID */
    } t_field3;
} HeapTupleFields;
```

表 42 HeapTupleFields 内部

変数	説明	備考
t_xmin	INSERT された XID	
t xmax	DELETE された XID	通常 0 / ROLLBACK 時も更新
t_cid	DELETE されたコマンド ID	
t_xvac	VACUUM で移動されたバージョン	

タプルの t_xmin, t_max フィールドは、仮想列 XMIN、XMAX に対応します。

3.7 トランザクション ID の周回問題

3.7.1 トランザクション ID

PostgreSQL ではトランザクションを開始すると一意なトランザクション ID が発番されます (txid_current 関数で取得)。トランザクション ID の大きさは符号なし 32 ビットです (src/include/c.h で定義)。テーブルのタプルが更新されると、各タップルヘッダには更新したトランザクション ID が格納されます。この機能により、検索時の参照整合性を維持することができます。

□ トランザクション ID の確認

テーブル上のタップルに対応するトランザクション ID は XMAX、XMIN 疑似列を指定することで確認できます。XMIN はタップルが追加された (INSERT または UPDATE による) トランザクション ID を示します。XMAX は削除されたタップルのトランザクション ID です。このため有効なタップルの XMAX は 0 になります。下記の例では C1=300 の列で XMAX が非 0 の値ですが、これは更新トランザクションがロールバックされたことを示します。

例 86 タップルのトランザクション ID

```
postgres=> SELECT XMAX, XMIN, c1 FROM data1 ;  
      xmin |      xmax | c1  
-----+-----+  
 507751 |        0 | 100  
 507752 |        0 | 200  
 507754 |        0 | 400  
 507755 |        0 | 500  
 507756 | 507757 | 300  
(5 rows)
```

□ トランザクション ID の周回

トランザクション ID は符号なし 32 ビットの大きさを持っており、単調に増加します。しかし大量のトランザクションを利用するシステムでは 32 ビットを使い果たす可能性があります。このためトランザクション ID が巡回しても問題が発生しない仕組みが用意されています。それは定期的に古いトランザクション ID を特殊な値 (FrozenXID=2) に置換することです。

この置換処理は FREEZE 処理と呼ばれ、通常は VACUUM 処理内で行われます。テーブ

ルに対してパラメーター autovacuum_freeze_max_age — vacuum_freeze_min_age で計算される数のトランザクションが発生すると、自動 VACUUM が無効でも FREEZE 処理が発生します。自動 VACUUM は処理対象ブロックを Visibility Map を使って発見します。Freeze 処理が完了すると、Visibility Map を更新します。パラメーター vacuum_freeze_table_age で指定されたトランザクション数が実行されると、Visibility Map から Freeze されていないブロックに対して検索を行います。

各テーブルのフリーズ対象となるトランザクション ID は、pg_class カタログの relfrozenxid 列で定義されています。この列はテーブル内の XMIN 仮想列の最小値です (FrozenXID を除く)。また pg_class カタログの relfrozenxid の最小値が pg_database カタログの datfrozenxid 列で確認できます。

□ 自動 VACUUM 処理が失敗した場合

何等かの原因で FREEZE 処理が行われなかつた場合、トランザクション ID の周回までに 1,000 万を切ると以下のメッセージが出力されます。

例 87 トランザクション周回の警告

```
WARNING: database "postgres" must be vacuumed within 9999999 transactions
HINT: To avoid a database shutdown, execute a database-wide VACUUM in
      "postgres" .
```

更に 100 万トランザクションを切ると、以下のログが出力され、システムは停止します。

例 88 トランザクション周回のエラー

```
ERROR: database is not accepting commands to avoid wraparound data loss in
       database "postgres"
HINT: Stop the postmaster and use a standalone backend to VACUUM in
      "postgres" .
```

このような状態に陥ったデータベースはスタンダード・モードで起動し、VACUUM 処理を行います。下記の例では postgres データベースに対して VACUUM 処理を実行しています。

例 89 スタンドアロン・モードによる起動と VACUUM

```
$ postgres --single -D data postgres
```

```
PostgreSQL stand-alone backend 9.6.2
backend> VACUUM
backend>
```

3.7.2 FREEZE 処理に関するパラメーター

トランザクション ID の FREEZE 動作に関するパラメーターは以下の通りです。

□ autovacuum_freeze_max_age

トランザクション ID の周回を防ぐために pg_class.relfrozenxid が到達できる最大の年代 (age) を指定します。自動 VACUUM が無効の場合でも、VACUUM ワーカー・プロセスが起動します。デフォルト値は 2 億 (200,000,000) です。最小値は 1 億 (100,000,000)、最大値は 20 億 (2,000,000,000) です。

□ vacuum_freeze_min_age

VACUUM がテーブルスキヤン時にトランザクション ID を FrozenXID に置き換えるカットオフ年代を指定します。デフォルト値は 5,000 万 (50,000,000) です。値は 0 から autovacuum_freeze_max_age の 50%までの値を指定できます。

□ vacuum_freeze_table_age

テーブルの pg_class.relfrozenxid がこの値で指定した時期に到達すると、VACUUM はより積極的にテーブルの走査を行い、FREEZE 処理を行います。デフォルト値は 1.5 億 (150,000,000) です。値は 0 から 10 億 (1,000,000,000) または autovacuum_freeze_max_age の 95%までの値を指定できます。

PostgreSQL 9.3.3 以降には以下のパラメーターが追加されました。マルチトランザクション ID (MultiXactId) とフリーズに関するパラメーターですが詳細は未確認です。

- autovacuum_multixact_freeze_max_age
- vacuum_multixact_freeze_min_age
- vacuum_multixact_freeze_table_age

3.8 ロケール指定

3.8.1 ロケールとエンコーディングの指定

デフォルトのロケール設定は initdb コマンドの—locale パラメーターで指定します。--locale パラメーターにエンコーディングも指定すると、デフォルトのエンコーディングになります。

例 90 ロケールとしてロケール名とエンコーディングを指定

```
$ export LANG=C
$ initdb --locale=ja_JP.utf8 data
The files belonging to this database system will be owned by user "postgres".
This user must also own the server process.

The database cluster will be initialized with locale "ja_JP.utf8".
The default database encoding has accordingly been set to "UTF8".
initdb: could not find suitable text search configuration for locale
"ja_JP.utf8"
The default text search configuration will be set to "simple".
<<以下省略>>
```

ロケール指定にロケール名のみ指定した場合は、ロケールのデフォルト・エンコードが指定されます。ロケールとして ja_JP を指定するとエンコードとして日本語 EUC(EUC_JP) が指定されます。

例 91 ロケールとしてロケール名のみ指定

```
$ export LANG=en_US.utf8
$ initdb --locale=ja_JP data
The files belonging to this database system will be owned by user "postgres".
This user must also own the server process.

The database cluster will be initialized with locale "ja_JP".
The default database encoding has accordingly been set to "EUC_JP".
initdb: could not find suitable text search configuration for locale "ja_JP"
The default text search configuration will be set to "simple".
<<以下省略>>
```

ロケールを使用せず、エンコーディングのみ指定した場合、pg_database カタログの encoding 列は指定されますが、ロケール関連列 (datcollate 等) には「C」が出力されます。

例 92 ロケールを使用せず、エンコーディングのみ指定

```
$ initdb --no-locale --encoding=utf8 data
The files belonging to this database system will be owned by user "postgres".
This user must also own the server process.

The database cluster will be initialized with locale "C".
The default text search configuration will be set to "english".
<<以下省略>>

postgres=> SELECT datname, pg_encoding_to_char(encoding), datcollate FROM
          pg_database ;
      datname | pg_encoding_to_char | datcollate
-----+-----+-----
template1 | UTF8           | C
template0 | UTF8           | C
postgres   | UTF8           | C
(3 rows)
```

以下の表に、環境変数 LANG に ja_JP.utf8 を指定した環境における initdb コマンドの指定方法とデータベース・クラスターのロケールの関係をまとめています。

表 43 initdb コマンドのロケール指定

initdb コマンド・パラメータ	lc_collate	lc_ctype	encoding
指定なし	ja_JP.utf8	ja_JP.utf8	UTF8
--locale=ja_JP	ja_JP	ja_JP	EUC_JP
--locale=ja_JP --encoding=utf8	initdb コマンド実行エラー		
--locale=ja_JP.utf8	ja_JP.utf8	ja_JP.utf8	UTF8
--locale=ja_JP.EUC_JP --encoding=utf8	initdb コマンド実行エラー		
--no-locale	C	C	SQL_ASCII
--no-locale --encoding=utf8	C	C	UTF8

□ ja_JP ロケール設定による機能

日本語ロケール (ja_JP) を設定したデータベースでは以下の機能が有効になります。

- 全角アルファベットに対する upper 関数 lower 関数

upper 関数、lower 関数が全角アルファベットでも利用できるようになります。半角／全角が維持された状態で変換されます。ロケールなしのデータベースでは変換は行われません。

例 93 upper / lower 関数

```
postgres=> SELECT upper('a'), lower('A') ; ← 半角に対する upper/lower
upper | lower
-----+
A     | a
(1 row)

postgres=> SELECT upper('あ'), lower('ア') ; ← 全角に対する upper/lower
upper | lower
-----+
ア     | あ
(1 row)
```

- money 型に対する検索で円マークの付与

psql コマンドから money 型の列を検索すると、データ先頭に通貨記号が付与されます。ただし検証の結果、ロケール付のデータベースでも、データベース・クラスターがロケール付で作成されていない場合はドル記号 (\$) になります。

例 94 money 型

```
postgres=> SELECT 100::money ;
money
-----
¥100
(1 row)
```

- ORDER BY 句による順序

ロケール機能が有効になっているデータベースで文字列型に対する ORDER BY 句を指定した場合、ロケールに応じた順序で出力されます。以下の表はエンコード UTF-8 を使用した場合の大小関係です。

表 44 ロケール指定による文字列の大小関係

大小関係	ja_JP ロケール	ロケールなし
↑ 小 ↓ 大	長さ 0 文字列	長さ 0 文字列
	半角数字	半角数字
	半角英字	半角英字
	半角カナ	ひらがな
	全角数字	全角カタカナ
	全角英字	全角数字
	ひらがな	全角英数
	全角カタカナ	半角カナ
	NULL	NULL

- 半角／全角の区別

日本語ロケールが有効なデータベースでも、全角と半角データ別データとして扱われます。またアルファベットの大文字／小文字も区別されます。

表 45 ロケール指定による大小関係

比較対象 1 (例)	比較対象 2 (例)	等しいか
半角アルファベット (A)	全角アルファベット (A)	異なる
半角大文字 (A)	半角小文字 (a)	異なる
全角カタカナ (ア)	半角カタカナ (ア)	異なる
全角カタカナ (ア)	全角ひらがな (あ)	異なる

3.8.2 LIKE によるインデックスの使用

ロケールが有効なデータベースでは、LIKE 句による前方一致でも該当列のインデックスが使用されないという仕様があります。

例 95 locale 使用時の LIKE 検索

```
postgres=> CREATE TABLE locale1(c1 varchar(10), c2 varchar(10)) ;
CREATE TABLE
postgres=> CREATE INDEX idx1_locale1 ON locale1(c1) ;
CREATE INDEX
postgres=> INSERT INTO locale1 VALUES ('ABC', 'DEF') ;
INSERT 0 1
...
postgres=> ANALYZE locale1 ;
ANALYZE
postgres=> EXPLAIN SELECT C1 FROM locale1 WHERE C1 LIKE 'A%' ;
               QUERY PLAN
-----
Seq Scan on locale1  (cost=0.00..1790.01 rows=10 width=5)
  Filter: ((c1)::text ~~ 'A%'::text)
Planning time: 1.742 ms
(3 rows)
```

実行計画が Seq Scan となり、全件検索が行われていることが確認できます。このような事態を避けるためには、CREATE INDEX 文実行時にオプションを指定し、バイナリによるインデックス検索を行うように指定します。

構文 3 CREATE INDEX 文のオプション指定

```
CREATE INDEX index_name ON table_name (column_name オプション)
```

列のデータ型に応じて以下のオプションを指定することができます。

表 46 バイナリ比較オプション

データ型	オプション
varchar	varchar_pattern_ops
char	bpchar_pattern_ops
text	text_pattern_ops
name	name_pattern_ops

例 96 オプション指定時の LIKE 検索

```
postgres=> CREATE INDEX idx2_locale1 ON locale1(c2 varchar_pattern_ops) ;
CREATE INDEX
postgres=> \d locale1
      Table "public.locale1"
 Column |          Type          | Modifiers
-----+---------------------+-----
 c1    | character varying(10) |
 c2    | character varying(10) |
Indexes:
"idx1_locale1" btree (c1)
"idx2_locale1" btree (c2 varchar_pattern_ops)
postgres=> EXPLAIN SELECT c2 FROM locale1 WHERE c2 LIKE 'A%' ;
               QUERY PLAN
-----
 Index Only Scan using idx2_locale1 on locale1  (cost=0.42..8.44 rows=10
 width=5)
   Index Cond: ((c2 ~>^ 'A' ::text) AND (c2 ~<^ 'B' ::text))
   Filter: ((c2)::text ~~ 'A%' ::text)
 Planning time: 0.541 ms
(4 rows)
```

3.8.3 <>演算子によるインデックスの使用

LIKE 演算子ではオプション指定が必要でしたが、大小を比較する演算子「<」「>」を指定してインデックスを使用したい場合には、前節のオプションを指定することができません。

例 97 オプション指定時の大小比較検索

```
postgres=> \d locale1
              Table "public.locale1"
  Column |          Type          | Modifiers
-----+-----+
    c1   | character varying(10) |
    c2   | character varying(10) |

Indexes:
  "idx1_locale1" btree (c1)
  "idx2_locale1" btree (c2 varchar_pattern_ops)

postgres=> EXPLAIN SELECT c1 FROM locale1 WHERE c1 < '10' ;
               QUERY PLAN
-----
Index Only Scan using idx1_locale1 on locale1  (cost=0.42..334.75 rows=306
width=5)
  Index Cond: (c1 < '10' ::text)
  Planning time: 0.210 ms
(3 rows)

postgres=> EXPLAIN SELECT c2 FROM locale1 WHERE c2 < '10' ;
               QUERY PLAN
-----
Seq Scan on locale1  (cost=0.00..1790.01 rows=10 width=5)
  Filter: ((c2)::text < '10' ::text)
  Planning time: 0.140 ms
(3 rows)
```

3.8.4 ロケールおよびエンコードの指定

ロケールおよびエンコードの指定はデータベース・クラスター作成時だけでなく、データベース作成時にも指定することができます。異なるロケール／エンコードを指定する場合には、テンプレートとして template0 を指定し、ENCODING 句 LC_COLLATE 句および LC_CTYPE 句を指定する必要があります。

例 98 ロケールおよびエンコードの指定

```
postgres=# CREATE DATABASE eucdb1 WITH TEMPLATE=template0 ENCODING='EUC_JP'
          LC_COLLATE='C' LC_CTYPE ='C' ;
CREATE DATABASE
postgres=# \!
      List of databases
  Name   |  Owner   | Encoding | Collate  |  Ctype   |  Access privileges
-----+-----+-----+-----+-----+-----+
eucdb1  | postgres | EUC_JP  | C        | C        |
postgres | postgres | UTF8    | ja_JP.utf8 | ja_JP.utf8 |
datadb1  | user1   | UTF8    | ja_JP.utf8 | ja_JP.utf8 |
template0 | postgres | UTF8    | ja_JP.utf8 | ja_JP.utf8 | =c/postgres      +
          |          |          |          |          | postgres=CTc/postgres
template1 | postgres | UTF8    | ja_JP.utf8 | ja_JP.utf8 | =c/postgres      +
          |          |          |          |          | postgres=CTc/postgres
(5 rows)
```

上記の例は LC_COLLATE と LC_CTYPE に'C'を指定していますが、異なるロケールでデータベースを作成する場合には両方のパラメーターにロケール名とエンコーディング名を両方指定します。またエンコーディングに日本語 EUC を指定する場合、「eucjp」、「EUC-JP」を指定することもできます。

3.9 チェックサム

PostgreSQL 9.3 からブロック・チェックサムの機能が加わりました。ブロック毎に、更新時にチェックサムが付与され、読み込み時にチェックが行われます。

3.9.1 チェックサムの指定

チェックサム機能は標準では無効化されていますが、`initdb` コマンドに`-k`⁹オプションを指定することでチェックサムを有効化したデータベース・クラスターを作成することができます。

例 99 チェックサムの有効化

```
$ initdb -k datak
The files belonging to this database system will be owned by user "postgres".
This user must also own the server process.

The database cluster will be initialized with locale "en_US.UTF-8".
The default database encoding has accordingly been set to "UTF8".
The default text search configuration will be set to "english".

Data page checksums are enabled.           ← チェックサムの有効化

<< 以下省略 >>
```

3.9.2 チェックサムの場所

チェックサムはページヘッダ内 の `pd_lsn` フィールドの後に 16 ビット領域として保存されます。この場所は PostgreSQL 9.2 まではタイムライン ID (`pd_tli`) が格納されていた部分です。チェックサムを追加してもヘッダ・サイズは変化していないため、旧バージョンと比較しても I/O 量には変化がありません。チェックサムの計算やチェックのための CPU リソースは増加すると予想されます。ページヘッダの構造体 (`PageHeaderData`) は、ヘッダ・ファイル `src/include/storage/bufpage.h` で定義されています。

チェックサムの付与は、ページの書き込み時に実施されます。実際のチェックサム作成は、ヘッダ・ファイル `include/storage/checksum_impl.h` の `pg_checksum_page` 関数で作成されます。

⁹ または`--data-checksums` オプション

例 100 ページヘッダ内のチェックサム

```
typedef struct PageHeaderData
{
    /* XXX LSN is member of *any* block, not only page-organized ones */
    PageXLogRecPtr pd_lsn;           /* LSN: next byte after last byte of xlog
                                         * record for last change to this page */

    uint16      pd_checksum;         /* checksum */
    uint16      pd_flags;           /* flag bits, see below */
    LocationIndex pd_lower;         /* offset to start of free space */
    LocationIndex pd_upper;         /* offset to end of free space */
    LocationIndex pd_special;       /* offset to start of special space */
    uint16      pd_pagesize_version;
    TransactionId pd_prune_xid;    /* oldest prunable XID, or zero if none */
    ItemIdData   pd_linp[1];        /* beginning of line pointer array */
} PageHeaderData;
```

3.9.3 チェックサム・エラー

□ チェックサムの確認

チェックサムの確認はブロックが共有バッファに読み込みが完了した時点で行われます。チェックサムのエラーが検知された場合は、ソースコード `src/backend/storage/buffer/bufmgr.c` の `ReadBuffer_common` 関数でログ出力されます。実際のチェックは、ソースコード `src/backend/storage/page/bufpage.c` の `PageIsVerified` 関数で実行しています。チェックサムの不正が検知されると以下のエラーが発生します。

例 101 チェックサム不正エラー

```
WARNING: page verification failed, calculated checksum 2773 but expected 29162
ERROR: invalid page in block 0 of relation base/12896/16385
```

チェックサム・エラーが発生したテーブルに対しては、その後は DML の実行はすべて同じエラーが発生します。

□ チェックサムの無視

パラメーター `ignore_checksum_failure` を `on` に設定すると、チェックサムのエラーを無視します（デフォルト値 `off`）。

3.9.4 チェックサムの有無確認

データベース・クラスターのチェックサム指定を確認するためには、`pg_controldata` ユーティリティの実行結果、パラメーター`data_checksums` または `pg_control_init` 関数を確認します。パラメーター`data_checksums` は PostgreSQL 9.3.4 から利用できます。

例 102 チェックサム機能の確認

```
$ pg_controldata ${PGDATA} | grep checksum
Data page checksum version:           1           ← チェックサム有効
$ 
$ psql
postgres=> SHOW data_checksums ;
data_checksums
-----
on                                         ← チェックサム有効
(1 row)
postgres=> SELECT data_page_checksum_version FROM pg_control_init() ;
data_page_checksum_version
-----
1                                         ← チェックサム有効
(1 row)
```

3.10 ログファイル

PostgreSQL が output するログファイルについて説明しています。

3.10.1 ログファイルの出力

PostgreSQL が output するログファイルにはインスタンス起動中のアクティビティを出力するログと、`pg_ctl` コマンド用のログがあります。

□ 稼働ログ

インスタンス稼働中のログの出力先はパラメーター `log_destination` (デフォルト値 `stderr`) で決定されます。このパラメーターの値を `syslog` に設定すると SYSLOG に転送されます (Windows 環境では `eventlog` に指定)。

パラメーター `log_destination` を `stderr` または `csv` に設定し、パラメーター `logging_collector` が `on` (デフォルト `off`) に指定すると、ログをファイルに出力することができます。

表 47 ログファイルの出力先 (パラメーター `log_destination`)

パラメーター値	説明	備考
<code>stderr</code>	標準エラー出力	
<code>csvlog</code>	CSV ファイル	<code>logging_collector=on</code> 必須
<code>syslog</code>	SYSLOG 転送	
<code>eventlog</code>	Windows イベント転送	Windows 環境のみ指定可能

□ 起動／停止ログ

`pg_ctl` コマンドの実行結果をログファイルに出力する場合は `-l` オプションでファイル名を指定します。指定が無い場合は標準出力に出力されます。`-l` オプションに指定されたログファイルが作成できない場合、インスタンスの起動 (`start`) はエラーになり、起動できません。インスタンスの停止 (`stop`) は正常に行われます。

`-l` オプションに既存のファイル名を指定した場合は追記されます。

□ ログファイルの出力方法

ログファイルは `fopen` 関数でオープンが行われ、`fwrite` 関数で書き込みが行われています。書き込み毎に `fflush` 関数は実行されていないため、OS のクラッシュ発生時やストレージ・レプリケーションを利用する場合には、ログが一部書き込まれていない可能性があります。

3.10.2 ログファイル名

ログファイル名を決定する要素について説明しています。

□ パラメーター指定

ログファイルの出力先とファイル名は以下のパラメーターで決定されます。

表 48 ログファイルの決定

パラメーター	説明	デフォルト値
log_directory	ログ出力ディレクトリ	pg_log
log_filename	ログファイル名	postgresql-%Y-%m-%d_%H%M%S.log
log_file_mode	ログファイルのアクセス権	0600

パラメーターlog_directory は絶対パスまたはデータベース・クラスターからの相対パスを記述できます。指定したディレクトリが存在しない場合にはインスタンス起動時に自動的に作成されます。ディレクトリが作成できない場合はインスタンス起動がエラーになります。下記はパラメーターlog_directory に「/var/log」を指定した場合のインスタンス起動エラーです。

例 103 ディレクトリ作成エラー

```
$ pg_ctl -D data start -w
pg_ctl -D data start -w
waiting for server to start...FATAL:  could not open log file
"/var/log/postgresql-2017-02-11_131642.log": Permission denied
stopped waiting
pg_ctl: could not start server
Examine the log output.
```

パラメーターlog_filename に%A を含めるとクラスターのロケールに依存せず英語の曜日名が出力されます。

□ CSV ファイル

パラメーターlog_destination に csvlog を指定すると、ログファイルの出力形式をカンマ(,) 区切りの CSV することができます。

例 104 CSV ファイル

```
2017-02-11 13:18:55.935 JST,,,15302,,57ede7af.3bc6,2,,2017-02-11 13:18:55
JST,,0,LOG,00000,"database system is ready to accept connections",,,,,,,,""
2017-02-11 13:18:55.937 JST,,,15308,,57ede7af.3bcc,1,,2017-02-11 13:18:55
JST,,0,LOG,00000,"autovacuum launcher started",,,,,,,,""
```

パラメーター `log_filename` に指定された拡張子が `.log` の場合、拡張子が `.csv` に変更されたログファイルが作成されます。また `.log` の拡張子を持つファイルも同時に作成されますが、内容は一部だけです。

□ SYSLOG 出力

パラメーター `log_destination` に `syslog` を指定するとローカルホストの `syslog` にデータが転送されます。ただし数行のレコードが記録されたログファイルも作成されます。

例 105 SYSLOG に転送された情報

```
Feb 11 13:21:04 rel71-2 postgres[15328]: [4-1] LOG: MultiXact member wraparound
protections are now enabled
Feb 11 13:21:04 rel71-2 postgres[15325]: [3-1] LOG: database system is ready
to accept connections
Feb 11 13:21:04 rel71-2 postgres[15332]: [3-1] LOG: autovacuum launcher started
```

表 49 SYSLOG 出力に関するパラメーター

パラメーター	説明	デフォルト値
<code>log_destination</code>	<code>syslog</code> に設定することで SYSLOG 転送を有効化	<code>stderr</code>
<code>syslog_facility</code>	SYSLOG のファシリティ	<code>LOCAL0</code>
<code>syslog_ident</code>	ログに出力されるアプリケーション名	<code>postgres</code>
<code>syslog_sequence_numbers</code>	SYSLOG にシーケンス番号を付与	<code>on</code>
<code>syslog_split_messages</code>	長いメッセージ (900 バイト以上) を分割	<code>on</code>

SYSLOG に出力されたログにはパラメーター `syslog_ident` で指定された名前の後にログを出力したプロセスのプロセス ID が出力されます。

3.10.3 ローテーション

`pg_rotate_logfile` 関数を実行するか、`logger` プロセスに SIGUSR1 シグナルを送信する
とログのローテーションが行われます。ただし、パラメーター`log_filename` に時刻情報が
含まれない場合等、新規のファイルが作成できない場合ローテーションは行われません。

□ `pg_rotate_logfile` 関数の仕様

本関数は `superuser` 権限を持つユーザーのみ実行できます。一般ユーザーが実行すると
以下のメッセージが出力されます。

例 106 エラー・メッセージ

```
ERROR: must be superuser to rotate log files
```

本関数は `logging_collector` パラメーターが `on` に設定されている (=logger プロセスが起
動している) 場合にのみ `true` を返します。`logging_collector` パラメーターが `off` の場合、
関数の実行は成功しますが、以下のメッセージが出力されて `false` を返します。

例 107 エラー・メッセージ

```
WARNING: rotation not possible because log collection not active
```

□ 出力中のログファイル削除

現在使用されているログファイルを削除してしまった場合、新しいログファイルは自動
的に再作成されません。強制的にローテーションを行うことで新規のログファイルを作成
させることができます。

3.10.4 ログの内容

標準設定のログには先頭にログのカテゴリーを示す文字列が output され、その後にログ内容が記録されます。カテゴリーに示される文字列は以下の通りです。

表 50 エラーログのカテゴリー

文字列	内容
DEBUG:	開発者用メッセージ
INFO:	ユーザーによって明示された詳細情報 (VACUUM VERBOSE 等)
NOTICE:	長い識別子の切り捨て等、ユーザーに対する補助情報
WARNING:	トランザクション外の COMMIT 実行等、ユーザーへの警告
ERROR:	コマンドの実行エラー等
LOG:	チェックポイントの活動等、管理者用メッセージ
FATAL:	セッションの終了を伴うエラー等
PANIC:	インスタンスの停止や全セッションの終了を伴うエラー等
???	不明のメッセージ。基本的には出力されない。

標準設定のログには、ログ出力時刻、ユーザー名、データベース名等の情報がまったく出力されません。このため監査に使用するには不十分です。ログにこれらの情報を指定するためにはパラメーター log_line_prefix を指定します。以下の文字を指定できます。

表 51 パラメーター `log_line_prefix` に指定できる文字

文字列	内容
%a	アプリケーション名 (set <code>application_name</code> で設定)
%u	接続ユーザー名
%d	接続データベース名
%r	リモート・ホスト名とポート番号 (ローカル接続時は[local])
%h	リモート・ホスト名
%p	プロセス ID
%t	ミリ秒を除いたタイムスタンプ
%m	ミリ秒込みのタイムスタンプ
%i	コマンド名 (INSERT, SELECT 等)
%e	SQLSTATE エラーコード
%c	セッション ID
%l	セッション・ライン番号
%s	セッション開始時刻
%v	仮想トランザクション ID
%x	トランザクション ID
%q	非セッションプロセスではこのエスケープ以降の出力を停止
%%	% 文字
%n	10 進数表現されたミリ秒込みのタイムスタンプ

3.10.5 ログのエンコード

エラー・メッセージを標準以外の文字コードで出力するには、PostgreSQL インストール前の `configure` コマンドに `--enable-nls=yes` を指定する必要があります。デフォルトでは指定されていませんが、コミュニティ版の rpm ファイルはこのパラメーターを指定されたバイナリが含まれます。現在利用中の PostgreSQL の設定は `pg_config` コマンドで確認できます。

例 108 `pg_config` を使った `configure` 設定の表示

```
$ pg_config | grep CONFIGURE
CONFIGURE = '--enable-nls=yes'
$
```

ログファイルに出力される文字列の文字コードは、以下の 3 つの部分で異なります。以下の例は `--enable-nls=yes` を指定した PostgreSQL で取得しています。

□ インスタンス起動／停止時のログ

`pg_ctl` コマンドのログは、コマンド実行時の OS ロケール指定（環境変数 `LANG` 等）で出力されます。

例 109 インスタンス起動時のログ（環境変数 `LANG` に依存）

```
サーバーの起動完了を待っています....  
LOG: redirecting log output to logging collector process  
HINT: Future log output will appear in directory "pg_log".  
完了
```

□ エラー・メッセージ

ロケール機能が有効なデータベースではエラー・メッセージは、パラメーター `lc_message` で指定されたエンコードで出力されます。

例 110 エラー・メッセージ（パラメーター `lc_message` に依存）

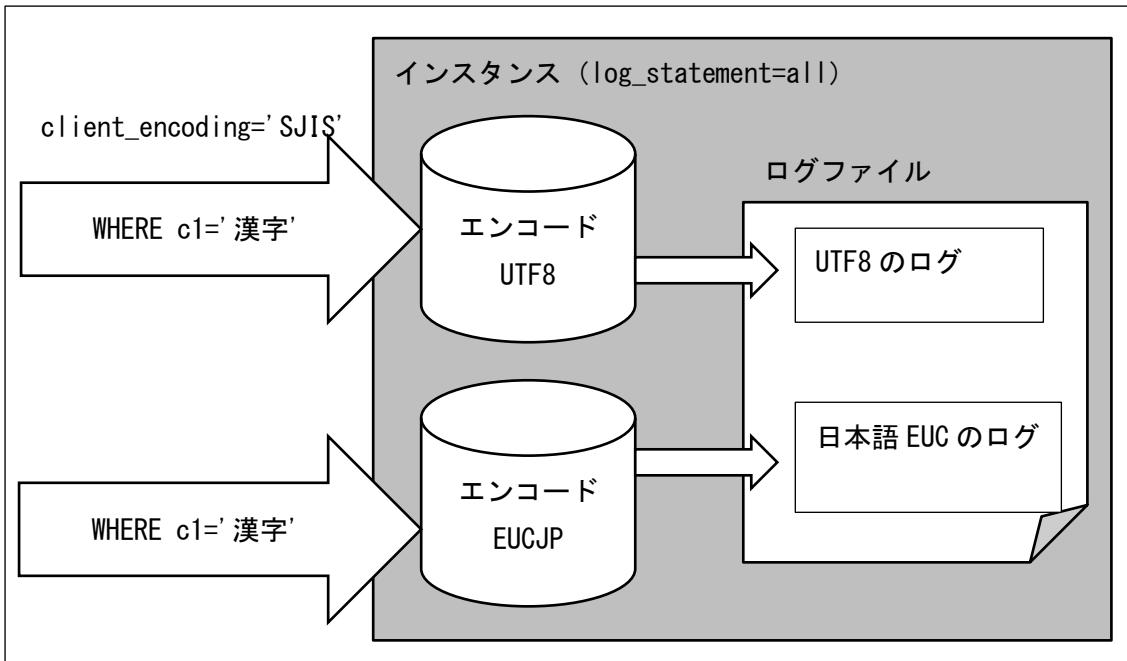
```
ERROR: リレーション"data1"は存在しません(文字位置 15)  
ステートメント: select * from data1;  
ERROR: リレーション"data1"は存在しません  
行 1: select * from data1;
```

□ SQL 文

パラメーター `log_statement` の指定により出力される SQL 文は、クライアント側のエンコードに依存せず、接続先データベースのエンコードで決定されます。このため、異なるエンコードを持つ複数のデータベースを作成した環境では、ログに出力される SQL 文の文字コードがまちまちになります。

次の図ではエンコード UTF8 のデータベースと、日本語 EUC のデータベースに対して SQL 文を発行しています。パラメーター `client_encoding` は SJIS なので、SQL 文は Shift_JIS で送信されます。パラメーター `log_statement` を `all` に指定しているため、実行された SQL 文がログに記録されます。記録される SQL 文はそれぞれ UTF8 または日本語 EUC に変換されてログファイルの内部で混在することになります。

図 10 ログファイルの文字コード



4. 障害対応

4.1 インスタンス起動前のファイル削除

4.1.1 pg_control 削除

pg_control ファイルが存在しない場合にはインスタンスを起動できません。

例 111 インスタンス起動時のエラーログ

```
postgres: could not find the database system
Expected to find it in the directory "/usr/local/pgsql/data",
but could not open file "/usr/local/pgsql/data/global/pg_control": No such file
or directory
```

pg_control ファイルをリカバリーするためには、バックアップから pg_control ファイルをリストアし、pg_resetxlog コマンドに-x オプションを指定して WAL ファイルを再作成します。インスタンスが異常終了していた場合は-f オプションも同時に指定します。

pg_control ファイルをリストアだけ行った場合はインスタンスの起動はできません。

例 112 リストアのみ実行した場合のインスタンス起動エラー

```
LOG: invalid primary checkpoint record
LOG: invalid secondary checkpoint record
PANIC: could not locate a valid checkpoint record
LOG: startup process (PID 12947) was terminated by signal 6: Aborted
LOG: aborting startup due to startup process failure
```

-x オプションに指定するトランザクション ID は、pg_resetxlog コマンドのマニュアルを参照してください。

4.1.2 WAL 削除

WAL ファイルが削除された場合の動作について検証しました。

□ 全 WAL ファイル削除時

インスタンスが正常に終了した場合、異常終了した場合にかかわらず WAL ファイルがす

べて削除された状態ではインスタンスは起動できません。pg_resetxlog コマンドを実行して WAL ファイルを再作成します。インスタンスが異常終了した直後の場合には、pg_resetxlog コマンドに-f オプションを指定して、強制的に WAL ファイルを作成します。

例 113 インスタンスの起動失敗ログ

```
LOG: could not open file "pg_xlog/0000000100000000000000000002" (log file 0,
segment 2): No such file or directory
LOG: invalid primary checkpoint record
LOG: could not open file "pg_xlog/0000000100000000000000000002" (log file 0,
segment 2): No such file or directory
LOG: invalid secondary checkpoint record
PANIC: could not locate a valid checkpoint record
LOG: startup process (PID 27972) was terminated by signal 6: Aborted
LOG: aborting startup due to startup process failure
```

□ インスタンス異常終了後、最新 WAL ファイル削除時

インスタンスが異常終了し、最新の WAL ファイルが削除された場合、ログファイルにはエラーが出力されますが、インスタンスは正常に起動します。クラッシュ・リカバリは途中までしか実行されませんが、ログには何も出力されません。

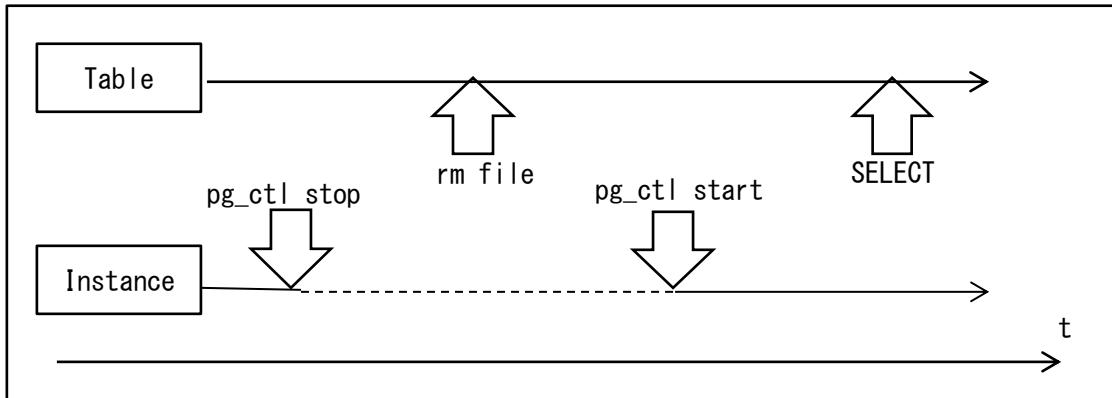
例 114 最新の WAL ファイルが削除された場合の起動ログ

```
LOG: database system was interrupted; last known up at 2017-02-11 12:07:16 JST
LOG: database system was not properly shut down; automatic recovery in progress
LOG: redo starts at 0/2000B98
FATAL: the database system is starting up
FATAL: the database system is starting up
LOG: redo done at 0/4FFFF78
LOG: last completed transaction was at log time 2017-02-11 12:08:22.592264+09
FATAL: the database system is starting up
LOG: MultiXact member wraparound protections are now enabled
LOG: database system is ready to accept connections
LOG: autovacuum launcher started
```

4.1.3 データファイル削除時の動作（正常終了時）

インスタンス正常終了後に、テーブルを構成するデータファイルを削除しました。インスタンス再起動後の動作を検証しました。

図 11 データファイル削除時の動作



□ 実行結果

インスタンスは正常に起動しました。削除されたテーブルに SELECT 文でアクセスするとエラー (ERROR カテゴリ) が発生しました。インスタンスやセッションには影響はありません。

□ ログ

インスタンス起動時にログは出力されません。SELECT 文実行時には以下のログが出力されました。

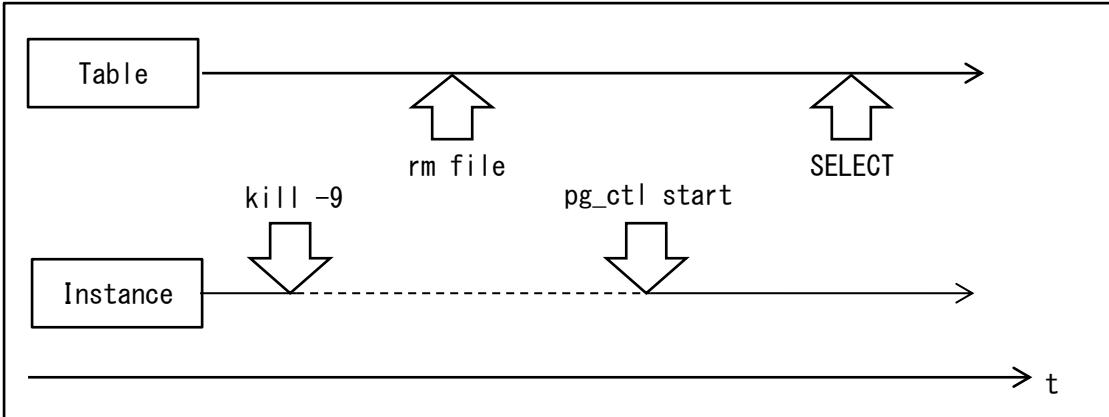
例 115 検索時のログ

```
LOG: database system is ready to accept connections
ERROR: could not open file "base/16385/16392": No such file or directory
STATEMENT: SELECT COUNT(*) FROM backup1 ;
```

4.1.4 データファイル削除時の動作（クラッシュ時／変更なし）

インスタンスが稼働中にクラッシュした場合の動作について検証しました。前回のチェックポイントから変更が無いテーブルのデータファイルが削除された場合の動作になります。これは運用しているデータベース・サーバーが OS パニックにより異常終了し、fsck コマンドによりファイルが削除されたことを想定しています。

図 12 データファイル消去の動作



□ 実行結果

インスタンスは正常に起動しました。削除されたテーブルに SELECT 文でアクセスするとエラー (ERROR カテゴリ) が発生しました。インスタンスやセッションには影響はありません。

□ ログファイル

インスタンス起動時にクラッシュ・リカバリが行われたログが出力されます。SELECT 文実行時には以下のログが出力されました。

例 116 起動時のログ

```
LOG: database system was interrupted; last known up at 2017-02-11 14:59:05 JST
LOG: database system was not properly shut down; automatic recovery in progress
LOG: redo starts at 3/A000098
LOG: invalid record length at 3/CA7AC28
LOG: redo done at 3/CA79988
FATAL: the database system is starting up
LOG: MultiXact member wraparound protections are now enabled
LOG: database system is ready to accept connections
```

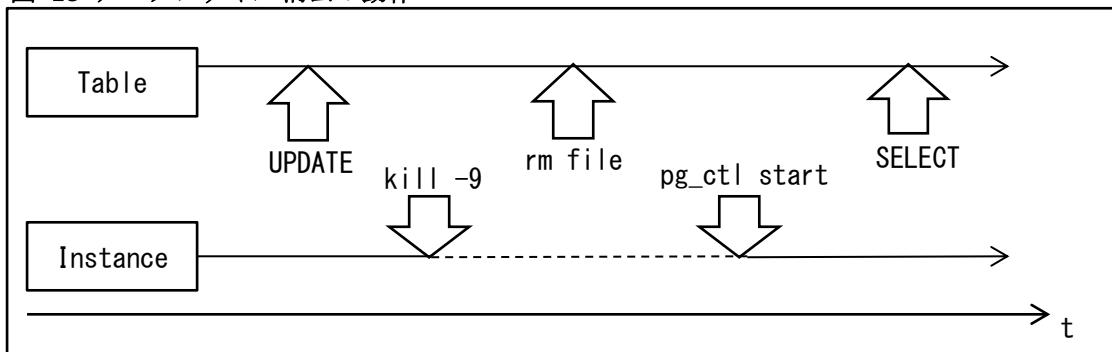
例 117 検索時のログ

```
ERROR: could not open file "base/16385/16601": No such file or directory
STATEMENT: SELECT * FROM data1 ;
```

4.1.5 データファイル削除時の動作（クラッシュ時／変更あり）

インスタンス稼働中にクラッシュした場合の動作について検証しました。前回のチェックポイント以降に更新され、WAL にトランザクション情報が記録されたテーブルのデータファイルが消滅した場合の動作になります。これは運用しているデータベース・サーバーが OS パニックにより異常終了し、fsck コマンドによりファイルが削除されたことを想定しています。

図 13 データファイル消去の動作



□ 実行結果

インスタンスは正常に起動しました。削除されたテーブルに SELECT 文でアクセスするとエラーは発生しませんでした。ただし、更新されたブロックの情報以外は損失しています。インスタンスやセッションには影響はありません。

□ ログファイル

インスタンス起動時にクラッシュ・リカバリが行われたログが出力されます。しかしデータが欠損したことに対するログは出力されませんでした。

例 118 起動時のログ

```
LOG: database system was interrupted; last known up at 2017-02-11 15:03:32
JST
LOG: database system was not properly shut down; automatic recovery in
progress
LOG: redo starts at 3/CA7AC98
LOG: invalid record length at 3/CD51DB8
LOG: redo done at 3/CD51D90
LOG: last completed transaction was at log time 2017-02-11 11:35:11.928603+09
LOG: MultiXact member wraparound protections are now enabled
LOG: database system is ready to accept connections
```

4.1.6 その他のファイル

その他のファイルを削除した場合の動作を検証しました。

□ Visibility Map (VM) /Free Space Map (FSM) ファイル削除時の動作

VM ファイル、FSM ファイルは削除されてもエラーは発生せず、対象テーブルに対する SQL 文も成功します。これらのファイルは次回の VACUUM 時には再作成されます。

□ pg_filenode.map ファイル削除時の動作

pg_filenode.map ファイルが削除されると、システム・カタログと実際のファイルがマッピングできなくなるためデータベースが利用できなくなります。

例 119 pg_filenode.map 削除時のログ

```
FATAL: could not open relation mapping file "base/16385/pg_filenode.map": No
such file or directory
```

□ PG_VERSION ファイル削除時の動作

PG_VERSION ファイルが削除されるとディレクトリが PostgreSQL 用であることが認識できなくなります。

例 120 PG_VERSION 削除時のログ

```
FATAL: "base/16385" is not a valid data directory
DETAIL: File "base/16385/PG_VERSION" is missing.
```

4.2 インスタンス稼働中のファイル削除

インスタンスが稼働中にファイルが欠損した場合の動作を検証しました。

4.2.1 pg_control 削除

インスタンス稼働中に pg_control にアクセスできない場合は PANIC が発生してインスタンスが停止します。検知はチェックポイント発生時に行われます。

例 121 チェックポイント発生時の PANIC ログ

```
PANIC: could not open control file "global/pg_control": Permission denied
LOG: checkpointer process (PID 3806) was terminated by signal 6: Aborted
LOG: terminating any other active server processes
WARNING: terminating connection because of crash of another server process
DETAIL: The postmaster has commanded this server process to roll back the
current transaction and exit, because another server process exited abnormally
and possibly corrupted shared memory.
HINT: In a moment you should be able to reconnect to the database and repeat
your command.
```

4.2.2 WAL 削除

□ インスタンス稼働中に WAL の削除（再作成可能）

WAL ファイルが削除されたことを検知すると、自動的に再作成されます。インスタンス起動状態で WAL ファイルを削除して検証しました。

□ インスタンス稼働中に WAL の削除（再作成不可能）

WAL ファイルが削除されたことを検知し、再作成できないことがわかると PANIC が発生してインスタンスが停止します。WAL ファイルを削除し、pg_xlog ディレクトリを書き込み禁止状態に設定して検証しました。

例 122 インスタンスの起動状態から WAL アクセス不可時のログ

```
PANIC:  could not open file "pg_xlog/000000010000000300000026": Permission denied
LOG:  WAL writer process (PID 2518) was terminated by signal 6: Aborted
LOG:  terminating any other active server processes
WARNING: terminating connection because of crash of another server process
DETAIL: The postmaster has commanded this server process to roll back the current transaction and exit, because another server process exited abnormally and possibly corrupted shared memory.
HINT: In a moment you should be able to reconnect to the database and repeat your command.
LOG: all server processes terminated; reinitializing
LOG: database system was interrupted; last known up at 2017-02-11 10:43:58 JST
LOG: creating missing WAL directory "pg_xlog/archive_status"
FATAL: could not create missing directory "pg_xlog/archive_status": Permission denied
LOG: startup process (PID 2624) exited with exit code 1
LOG: aborting startup due to startup process failure
```

4.3 プロセス障害

4.3.1 プロセス異常終了時の再起動

postmaster 以外のバックエンド・プロセスが異常終了した場合には、postmaster プロセスが検知して再起動を行います。いくつかのプロセスが同時に再起動する場合があります。クライアントからの SQL 文を処理する postgres プロセスは異常終了と同時にクライアントとのセッションが切断されるため、再起動は行われません。

表 52 プロセス異常終了時の動作

プロセス	動作
postmaster ¹⁰	インスタンス全体が異常終了、共有メモリーは削除されません。
logger	postmaster により再起動されます。
writer	postmaster により wal writer、autovacuum launcher も同時に再起動されます。全 postgres プロセスが停止します。
wal writer	postmaster により writer、autovacuum launcher も同時に再起動されます。全 postgres プロセスが停止します。
autovacuum launcher	postmaster により writer、wal writer も同時に再起動されます。全 postgres プロセスが停止します。
stats collector	postmaster により再起動されます。
archiver	postmaster により再起動されます。
checkpoint	postmaster により再起動されます。全 postgres プロセスが停止します。
postgres	再起動は行われません。
wal sender	postmaster により再起動されます。スレーブ・インスタンスの wal receiver プロセスも再起動されます。全 postgres プロセスが停止します。
wal receiver	postmaster により再起動されます。マスター・インスタンスの wal sender プロセスも再起動されます。スレーブ・インスタンスの全 postgres プロセスが停止します。
startup process	スレーブ・インスタンス全体が終了します。

¹⁰ postmaster に対して KILL シグナルを送信して強制終了した場合、共有メモリーおよびセマフォが削除されません。

上記動作はパラメーター `restart_after_crash` の値がデフォルト値である `on` に設定されている場合の動作です。このパラメーターが `off` に設定されると、バックエンド・プロセスが停止した場合インスタンス全体が異常終了します。

4.3.2 プロセス異常終了時のトランザクション

`stats collector` プロセス、`logger` プロセスおよび `archiver` プロセス以外のプロセスが異常終了すると、クライアントからの接続を行っていた `postgres` プロセスがすべて停止します。このため実行中のトランザクションはすべて破棄されます。レプリケーション環境では、スレーブ・インスタンスのプロセス障害は、マスター・インスタンスの動作に影響を与えません。

4.4 その他の障害

4.4.1 クラッシュ・リカバリ

インスタンスが異常終了した場合、チェックポイントが完了していないため実行した更新トランザクションの情報は WAL のみに書き込まれています。インスタンス起動時には、データファイルと WAL の不整合を調整するクラッシュ・リカバリ処理が自動的に行われます。

クラッシュ・リカバリは pg_control ファイルの読み込みからはじまります。インスタンスのステータスが DB_SHUTDOWNED (1) の場合、インスタンスは正常に終了しているためクラッシュ・リカバリは行われません。それ以外のステータスの場合、インスタンスは異常終了したことになるためクラッシュ・リカバリが必要になります。

以下に処理を簡単に示します。

1. チェックポイントの位置を確認。チェックポイントまでの WAL はデータファイルに書きこまれたことが保障されているため、リカバリーの必要がありません。
2. WAL から前回のチェックポイント後に発生したトランザクション情報を読み込み
3. チェックポイント後の最初の更新ではブロック全体が WAL に書き込まれているため、ブロックをリカバリー (パラメーター full_page_writes)
4. WAL 情報から更新トランザクションの情報を適用
5. 最新の WAL まで更新トランザクションを再実行

4.4.2 オンライン・バックアップ中のインスタンス異常終了

PostgreSQL のオンライン・バックアップは以下の手順で実施します。

1. インスタンスをアーカイブログ・モードで起動
2. pg_start_backup 関数の実行
3. データベース・クラスターのファイルをコピー (OS コマンドによる)
4. pg_stop_backup 関数の実行

上記手順のうち、3 の実行中にインスタンスが異常終了した場合の動作を検証しました。停止は postmaster に対して KILL シグナルを送信することで行いました。

例 123 オンライン・バックアップ中のインスタンス異常終了

```
postgres=# SELECT pg_start_backup('label') ;
pg_start_backup
-----
0/5000020
(1 row)

$ ps -ef | grep postgres
postgres 6016      1  0 12:46 pts/1    00:00:00 /usr/local/pgsql/bin/postgres
$ kill -KILL 6016
$ pg_ctl -D data start -w
pg_ctl: another server might be running; trying to start server anyway
server starting

postgres=# SELECT pg_stop_backup() ;
ERROR: a backup is not in progress
```

postmaster.pid ファイルが削除されていないため、警告が出力されていますが、再起動は正常に行われました。また pg_stop_backup 関数を実行したところバックアップ中ではないというエラーが返っていることからバックアップ・モードはインスタンスの再起動によってクリアされることがわかります。

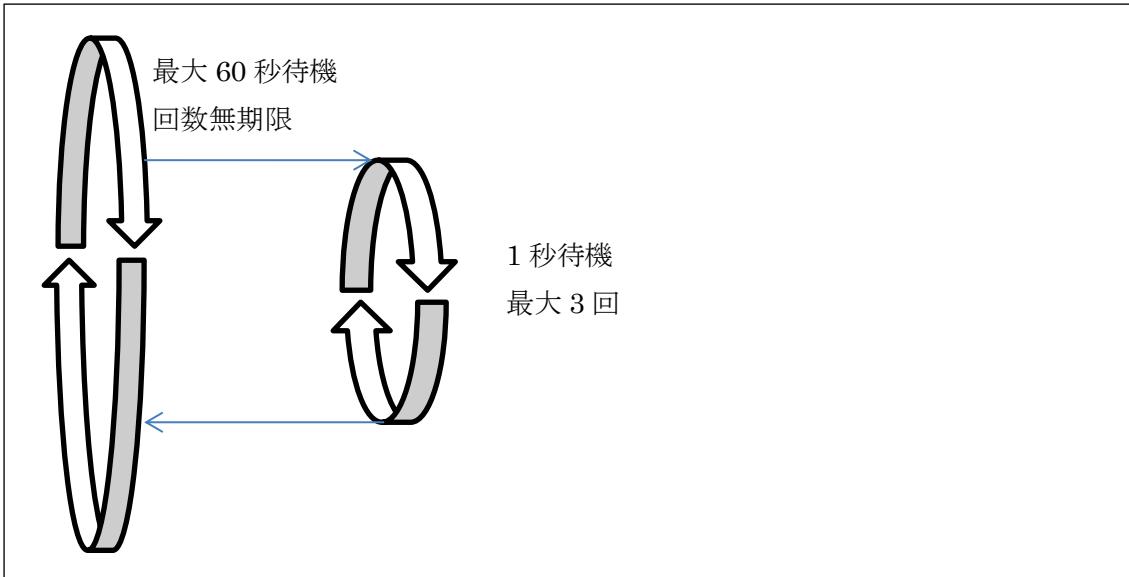
4.4.3 アーカイブ処理の失敗

WAL ファイルが完全に書き込まれると、パラメーター archive_command に指定されたコマンドが system 関数（Linux / Windows 共）を使って実行されます（src/backend/postmaster/pgarch.c 内の pgarch_archiveXlog 関数）。

□ アーカイブ処理の再実行

system 関数が 0 以外の値を返すと、アーカイブ処理が失敗したとみなされ、3回リトライを実行します。3回のリトライをすべて失敗すると最大で 60 秒待機します。

図 14 アーカイブ処理の再実行



アーカイブ処理の失敗回数等は、`pg_stat_archiver` カタログで確認することができます (PostgreSQL 9.4 から)。

□ アーカイブ処理の失敗ログ

アーカイブ処理が失敗すると、ログが出力されます。下記の例は、パラメーター `archive_command` に「`cp %p /arch/%f`」を指定し、アーカイブログ出力ディレクトリに書き込み権限が無い場合のエラーです。`cp` コマンドがステータス 1 で終了しています。

3 回 LOG レベルのエラーが続き、最後に WARNING レベルのエラーが出力されています。

例 124 アーカイブ処理失敗のログ

```

cp: accessing `/arch/0000000100000000000000070': Permission denied
LOG: archive command failed with exit code 1
DETAIL: The failed archive command was: cp pg_xlog/0000000100000000000000070
/arch/0000000100000000000000070
cp: accessing `/arch/0000000100000000000000070': Permission denied
LOG: archive command failed with exit code 1
DETAIL: The failed archive command was: cp pg_xlog/0000000100000000000000070
/arch/0000000100000000000000070
cp: accessing `/arch/0000000100000000000000070': Permission denied
LOG: archive command failed with exit code 1
DETAIL: The failed archive command was: cp pg_xlog/0000000100000000000000070
/arch/0000000100000000000000070
WARNING: archiving transaction log file "0000000100000000000000070" failed too
many times, will try again later

```

アーカイブ処理失敗時に出力されるエラー・メッセージは以下の通りです。

表 53 アーカイブ処理失敗ログ

レベル	メッセージ	説明
WARNING	archive_mode enabled, yet archive_command is not set	アーカイブログ・モードだが、パラメーターarchive_command が未設定
WARNING	archiving transaction log file ¥”%s¥” failed too many times, will try again later	3 回のリトライがすべて失敗したため、一時的に待機する
FATAL LOG	archive command failed with exit code %d	アーカイブ・コマンドが失敗ステータスで終了
FATAL	archive command was terminated by exception	アーカイブ・コマンドが例外を受けて失敗 (Windows)
FATAL	archive command was terminated by signal %d	アーカイブ・コマンドが例外を受けて失敗 (Windows 以外)
FATAL LOG	archive command exited with unrecognized status %d	アーカイブ・コマンドが不明なエラーで終了

エラー・レベルの選択 (FATAL または LOG) は、WEXITSTATUS マクロが 128 を超える場合または WIFSIGNALED マクロが真になる場合は FATAL、それ以外は LOG になります。

□ アーカイブ失敗時の WAL

アーカイブ処理が失敗すると、処理が失敗した WAL ファイルは再利用されず新規の WAL ファイルが追加されます。このためアーカイブ処理が失敗し続けると、pg_xlog ディレクトリ以下に大量の WAL ファイルが残ることになります。

5. パフォーマンス関連

5.1 統計情報の自動収集

5.1.1 タイミング

統計情報の収集は、自動 VACUUM と同時に実行されます。autovacuum launcher プロセスから、パラメーター autovacuum_naptime (デフォルト 1min) で指定された間隔で実際の処理を行う autovacuum worker プロセスが起動されます。

5.1.2 条件

統計情報の収集は、以下の計算式で取得された値と、前回統計情報を取得してから更新されたレコード数を比較して決定されます。更新されたレコード数とは UPDATE / DELETE / INSERT 文で影響を受けたレコードの総和です。条件のチェックは、ソースコード (src/backend/postmaster/autovacuum.c) 内の relation_needs_vacanalyze 関数で行っています。

計算式 3 統計情報取得の閾値

```
閾値 = autovacuum_analyze_threshold +
       autovacuum_analyze_scale_factor * reltuples
```

上記計算式の意味は以下の通りです。

- autovacuum_analyze_threshold
パラメーター autovacuum_analyze_threshold の値 (デフォルト値 50) またはテーブルの autovacuum_analyze_threshold 属性の値です。
- autovacuum_analyze_scale_factor
パラメーター autovacuum_analyze_scale_factor の値 (デフォルト値 0.1 = 10%) または、テーブルの autovacuum_analyze_scale_factor 属性の値です。
- reltuples
前回統計情報を取得した時点のテーブルの有効レコード数です。

5.1.3 サンプル・レコード数

統計情報の収集はサンプリングによって行われます。サンプリングのタプル数は、テーブルのレコード数やブロック数等に依存しません。ANALYZE 文を実行する（または自動 VACUUM 内で実行される）対象テーブルの設定により、以下の計算式で求められます。

計算式 4 サンプリング・タプル数

サンプル・タプル数 = MAX(各列 STATISTICS) * 300

各列の STATISTICS 値のデフォルト値は、パラメーター default_statistics_target が使用されます。テーブルのアクティブなレコード数がこの値以下の場合はテーブルの全アクティブ・レコードが対象になります。この計算式はソースコード (src/backend/commands/analyze.c) の std_tyanalyze 関数で定義されています。以下は計算式を決定した経緯に関するコメントです。

例 125 ソースコード内のコメント

The following choice of minrows is based on the paper "Random sampling for histogram construction: how much is enough?" by Surajit Chaudhuri, Rajeev Motwani and Vivek Narasayya, in Proceedings of ACM SIGMOD International Conference on Management of Data, 1998, Pages 436-447. Their Corollary 1 to Theorem 5 says that for table size n, histogram size k, maximum relative error in bin size f, and error probability gamma, the minimum random sample size is

$$r = 4 * k * \ln(2*n/gamma) / f^2$$

Taking f = 0.5, gamma = 0.01, n = 10^6 rows, we obtain

$$r = 305.82 * k$$

Note that because of the log function, the dependence on n is quite weak; even at n = 10^{12} , a $300*k$ sample gives ≤ 0.66 bin size error with probability 0.99. So there's no real need to scale for n, which is a good thing because we don't necessarily know it at this point.

表 54 計算式のデータ

項目	説明
r	取得が必要なサンプル・レコード数
k	ヒストグラムのサイズ (パラメーター default_statistics_target または列 STATISTICS)
n	レコード数 (定数 1,000,000 固定)
gamma	エラー確率 (定数 0.01 固定)
f	最大相対誤差 (定数 0.5 固定)

ANALYZE 文で取得されたサンプル・レコード数を出力するには、ANALYZE VERBOSE 文を実行するか、パラメーター log_min_messages を DEBUG2 に設定します。以下の例は「ANALYZE VARBOSE data1」文を実行した場合のログです。約 100,000 レコード格納されたテーブルに対して、30,000 レコードがサンプリングされたことがわかります。

例 126 統計情報収集のサーバー・ログ

```
DEBUG: analyzing "public.data1"  
DEBUG: "data1": scanned 542 of 542 pages, containing 100100 live rows and 0  
dead rows; 30000 rows in sample, 100000 estimated total rows
```

例 127 ANALYZE VERBOSE 文の出力

```
postgres=> ANALYZE VERBOSE stat1 ;  
INFO:  analyzing "public.stat1"  
INFO:  "stat1": scanned 542 of 542 pages, containing 100100 live rows and 0  
dead rows; 30000 rows in sample, 100100 estimated total rows  
ANALYZE  
postgres=>
```

ANALYZE VERBOSE 文を実行すると、サンプリング・レコード数だけでなく、ページ数、アクティブ・タプル数、デッド・タプル数、予想されたテーブル全体のタプル数の情報も出力されます。

□ 列の STATISTICS 値の変更

列の STATISTICS 設定は、ALTER TABLE ALTER COLUMN 文で実行します。
STATISTICS 設定の上限値は 10,000 です。

例 128 列の STATISTICS 設定と確認

```
postgres=> ALTER TABLE data1 ALTER COLUMN col1 SET STATISTICS 200 ;
```

```
ALTER TABLE
```

```
postgres=> \d+ data1
```

Column	Type	Modifiers	Storage	Stats target	Desc
c1	numeric		main	<u>200</u>	
c2	character varying(100)		extended		

Has OIDs: no

5.1.4 統計として収集される情報

ANALYZE 文の実行や自動 VACUUM により収集される統計情報は、以下の pg_statistic カタログと pg_class カタログに格納されます。pg_statistic カタログの形式は利用しにくいため、通常 pg_stats カタログから情報を検索します。このカタログには ANALYZE 文（自動 VACUUM 時含む）が実行されていないテーブルのレコードは格納されません。

表 55 pg_stats カタログ

列	説明
schemaname	スキーマ名
tablename	テーブル名
attname	列名
inherited	true の場合、継承子テーブルの情報を含む
null_frac	NULL 行の割合
avg_width	列の平均バイト長さ
n_distinct	一意なレコード数の推定値（正の場合と負の場合がある）
most_common_vals	最も値が多い列値
most_common_freqs	最も値が多い列値の割合
histogram_bounds	ヒストグラムの出現割合
correlation	物理配置と論理配置の相関
most_common_elems	最も値が多い列値（非スカラ一値）
most_common_elem_freqs	最も値が多い列値の割合（非スカラ一値）
elem_count_histogram	ヒストグラムの出現割合（非スカラ一値）

表 56 pg_class カタログの統計情報

列	説明
reltuples	テーブルに含まれるレコード数
relopages	テーブルのページ数

□ n_distinct 統計

pg_stats カタログの n_distinct 列はテーブル内の一意な値の数を示します。この値は負の値になる場合があります。

- サンプリングした結果、列値が一意であると判断されると n_distinct 値は-1.0 が指定されます。
- 一意な値の数が全レコード数の 10%以上であると判断された場合、以下の計算値が指定されます。

$$-1 * (\text{一意な値の推定値} / \text{トータル・レコード数})$$

- 上記以外の場合は、推定値として以下コメント内の計算式が使用されます。

例 129 n_distinct 計算式を示すコメント (src/backend/commands/analyze.c)

Estimate the number of distinct values using the estimator proposed by Haas and Stokes in IBM Research Report RJ 10025:

$$n*d / (n - f1 + f1*n/N)$$

where f1 is the number of distinct values that occurred exactly once in our sample of n rows (from a total of N), and d is the total number of distinct values in the sample.

This is their Duj1 estimator; the other estimators they recommend are considerably more complex, and are numerically very unstable when n is much smaller than N.

Overwidth values are assumed to have been distinct.

統計情報の n_distinct の値は ALTER TABLE table_name ALTER column_name SET 文で上書きできますが、次回の ANALYZE 文が実行されるまで pg_stats カタログ上の値は変更されません。

列の属性は n_distinct と n_distinct_inherited を指定することができます。

n_distinct_inherited は継承されたテーブルの列情報ですが、継承テーブルを使用してい

なくても変更することができます。 ALTER TABLE 文で変更された列属性は pg_attributes カタログの attoptions 列で確認することができます。

□ most_common_vals 統計

most_common_vals 統計 (MCV) は最も出現回数が多い列値の配列です。配列の要素数の最大値は default_statistics_target パラメーター値（デフォルト 100）または、列の STATISTICS 値が指定されていればその値です。このため default_statistics_target パラメーターを拡大するとヒストグラムのバケット数、サンプリング・レコード数、MCV の要素数を拡大することになります。

5.1.5 統計情報の保存先

オブジェクト単位の統計情報以外に、PostgreSQL では様々な統計情報が自動的に収集されます。それらは pg_stat_*カタログや pg_statio_*カタログ¹¹で確認することができます。

表 57 統計情報カタログ

カタログ名	内容
pg_stat_activity	クライアントの接続情報
pg_stat_{all sys user}_indexes	インデックスに対する統計情報
pg_stat_{all sys user}_tables	テーブルに対する統計情報
pg_stat_archiver	アーカイブログに関する統計情報
pg_stat_bgwriter	writer プロセスに関する統計情報
pg_stat_database	データベース単位の統計情報
pg_stat_database_conflicts	スタンバイ・データベースとの競合情報
pg_statio_{all sys user}_sequences	シーケンスに関する I/O 統計情報
pg_statio_{all sys user}_tables	テーブルに関する I/O 統計情報
pg_statio_{all sys user}_indexes	インデックスに関する I/O 統計情報
pg_statistic	オブジェクトに関する統計情報
pg_stat_replication	レプリケーション情報
pg_stats	pg_statistic の整形カタログ
pg_stat_ssl	クライアントの SSL 使用状況
pg_stat_user_functions	ファンクションに対する統計情報
pg_stat_xact_{all sys user}_tables	テーブルに対する更新統計情報
pg_stat_xact_user_functions	ファンクションに対する更新統計情報
pg_stat_wal_receiver	レプリケーション環境のスレーブ情報
pg_stat_progress_vacuum	VACUUM 実行状況

統計情報の主な実体はインスタンス全体の統計情報が格納される global.stat ファイルと、データベース単位に作成される db_{OID}.stat ファイルです。これらのファイルはインスタンス起動時に {PGDATA}/pg_stat ディレクトリから、パラメーター stats_temp_directory (デフォルト pg_stat_tmp) で指定されたディレクトリに移動されます。インスタンス停止時には pg_stat ディレクトリに戻されます。

統計情報の読み込みがネックになっている場合はパラメーター stats_temp_directory を高速なストレージ上のディレクトリに指定することで高速化することができます。

¹¹ マニュアルは <https://www.postgresql.org/docs/9.6/static/monitoring-stats.html>

5.2 自動 VACUUM

5.2.1 タイミング

自動 VACUUM 処理は autovacuum launcher プロセスから起動される autovacuum worker プロセスが行います。autovacuum worker プロセスの起動間隔は、パラメーター autovacuum_naptime (デフォルト 1min) で決まります。

5.2.2 条件

自動 VACUUM は、以下の計算式で取得された値と、更新されて不要になったレコード数を比較して決定されます。不要になったレコード数は基本的には UPDATE / DELETE 文により更新されたレコード数に一致しますが、同一レコードを複数回 UPDATE した場合には更新レコード数にカウントされない場合があります。これは HOT による再利用のためと想定されますが、ソースコードの確認までは実施していません。条件のチェックは、ソースコード (src/backend/postmaster/autovacuum.c) 内の relation_needs_vacanalyze 関数で行っています。

計算式 5 自動 VACUUM 閾値

```
閾値 = autovacuum_vacuum_threshold +
       autovacuum_vacuum_scale_factor * reltuples
```

上記計算式の意味は以下の通りです。

- autovacuum_vacuum_threshold
パラメーター autovacuum_vacuum_threshold の値 (デフォルト値 50) またはテーブルの autovacuum_vacuum_threshold 属性の値です。
- autovacuum_vacuum_scale_factor
パラメーター autovacuum_analyze_scale_factor の値 (デフォルト値 0.2 = 20%) または、テーブルの autovacuum_analyze_scale_factor 属性の値です。
- reltuples
前回統計情報を取得した時点のテーブルの有効レコード数です。pg_class カタログの reltuples 列で確認できます。

5.2.3 autovacuum worker プロセス起動

実際の VACUUM 処理は定期的に起動される autovacuum worker プロセスが行います。autovacuum worker プロセスは postmaster プロセスの子プロセスになります。

VACUUM が必要になるテーブルは「autovacuum_naptime パラメーター / 変更があつたデータベース数」の間隔でチェックされ、前述の閾値を超える不要レコードが存在するテーブルが見つかると autovacuum worker プロセスが起動されます。autovacuum worker プロセスは対象データベース内の VACUUM 処理を完了すると停止します。

VACUUM 対象のテーブルが、複数のデータベースに存在する場合、それぞれのデータベースに対して autovacuum worker プロセスが起動します。

次回のチェック間隔でまだ VACUUM 対象のテーブルが存在する場合、新たに autovacuum worker プロセスを起動して VACUUM 処理を行います。autovacuum worker プロセスの最大値は autovacuum_max_workers (デフォルト 3) で制限されます。最大値に到達してもログは出力されません。自動 VACUUM の実行状況は、pg_stat_progress_vacuum カタログで確認できます。ただし、VACUUM 処理実行中のみレコードが検索できます。

例 130 pg_stat_progress_vacuum カタログの検索

```
postgres=# SELECT * FROM pg_stat_progress_vacuum ;  
-[ RECORD 1 ]-----+-----  
pid | 3184  
datid | 16385  
datname | demodb  
relid | 16398  
phase | scanning heap  
heap_blks_total | 10052  
heap_blks_scanned | 2670  
heap_blks_vacuumed | 2669  
index_vacuum_count | 0  
max_dead_tuples | 291  
num_dead_tuples | 185
```

5.2.4 使用メモリー容量

自動 VACUUM の処理に使用するメモリー領域のサイズ計算を制御するパラメーターが autovacuum_work_mem です。常にこのパラメーターで指定された値が使用される訳ではなく、パラメーター値を基にした計算値が使用されます。

このパラメーターのデフォルト値は -1 です。デフォルト値の場合、パラメーター maintenance_work_mem の値を使用します。



マニュアルには記載がありませんが、パラメーターの下限値は 1024 (=1MB)です。-1以外の値で、1MB 未満の値を指定すると、パラメーター値は 1MB に設定されます。

5.3 実行計画

5.3.1 EXPLAIN 文

PostgreSQL の実行計画を表示させるためには、EXPLAIN 文を使用します。EXPLAIN 文に ANALYZE 句を指定すると、実行計画作成時に計算された見積もり統計と実際に SQL 文を実行した結果の統計情報を並べて表示することができます。

例 131 EXPLAIN 文の実行

```
postgres=> EXPLAIN ANALYZE SELECT * FROM data1 WHERE c1 BETWEEN 1000 AND 2000 ;
                                         QUERY PLAN
-----
Index Scan using pk_data1 on data1  (cost=0.29..8.31 rows=1 width=11)
(actual time=0.033..0.033 rows=0 loops=1)
Index Cond: ((c1 >= '1'::numeric) AND (c1 <= '1000'::numeric))
Planning time: 9.849 ms
Execution time: 1.691 ms
(4 rows)
```

表 58 EXPLAIN 文の出力結果の例

出力	説明	備考
Index Scan using	実行計画と対象のオブジェクト名	
cost	計算されたコスト（詳細は後述）	
rows	推定されたレコード数	
width	推定された一般的な出力バイト数（1 レコードの幅）	
actual time	実際の実行時間（詳細は後述）	
rows	出力されたレコード数	
loops	処理の繰り返し回数	
Index Cond:	インデックスの部分検索を行ったことを示す	
Planning time:	予想時間（ms）	9.4 追加
Execution time:	総実行時間（ms）	

EXPLAIN 文では、実行計画作成に使用されたプランナーが動的計画法（DP）なのか、遺伝的最適化（GEQO）なのかは表示されません。また PL/pgSQL 等で記述されたストアド・プロシージャ内部から実行された SQL 文の実行計画は表示されません。

5.3.2 コスト

EXPLAIN 文で表示される cost 部分は、SQL 文を実行するために必要なコストです。コストはシーケンシャル I/O で 1 ページ (8 KB) を読み込むコストを 1.0 とした際の相対的な推定値です。

例 132 EXPLAIN 文によるコストの表示

```
cost=0.00..142.10
```

最初の数字はスタートアップ・コスト、次の数字はトータル・コストです。スタートアップ・コストはいくつかの演算子を使う際に使用されます。チューニングで注意すべきはトータル・コストです。

表 59 実行計画とコストのデフォルト値

パラメーター	説明	デフォルト値
seq_page_cost	シーケンシャル・ページ読み込み	1.0
cpu_index_tuple_cost	インデックスの処理 1 回	0.005
cpu_operator_cost	計算 1 回	0.0025
cpu_tuple_cost	1 レコードの操作	0.01
random_page_cost	ランダム・ページ読み込み	4.0
parallel_setup_cost	パラレル・クエリー初期コスト	1000
parallel_tuple_cost	パラレル・クエリーのタプルコスト	0.1

例 133 コストの表示

```
postgres=> EXPLAIN SELECT * FROM data1 ;
          QUERY PLAN
-----
Seq Scan on data1  (cost=0.00..2133.98 rows=89998 width=41)
(1 row)

postgres=> SELECT reltuples, relpages FROM pg_class WHERE relname='data1' ;
      reltuples |      relpages
-----+-----
      89998 |        1234
(1 row)
```

上記例におけるコストの計算値は、relpages (1,234) × seq_page_cost (1.0) + reltuples (89,998) × cpu_tuple_cost (0.01) = 2,133.98 となります。

5.3.3 実行計画

EXPLAIN コマンドで出力される実行計画には以下のクエリー・オペレーターがあります。このリストは、PostgreSQL 9.6.2 のソースコード (src/backend/commands/explain.c) から検索しました。

表 60 実行計画

クエリー・オペレーター	動作	スタートアップ・コスト
Result	非テーブル問い合わせ	無
Insert	INSERT 文の実行	
Delete	DELETE 文の実行	
Update	UPDATE 文の実行	
Append	データの追加処理	無
Merge Append	マージ処理	
Recursive Union	再帰ユニオン	
BitmapAnd	ビットマップ検索 AND	
BitmapOr	ビットマップ検索 OR	
Nested Loop	ネスティッド・ループ	無
Merge Join	マージ結合	有
Hash Join	ハッシュ結合	有
Seq Scan	全件検索	無
Sample Scan	サンプル検索	
Index Scan	インデックス部分検索	無
Index Only Scan	インデックスのみの部分検索	
Bitmap Index Scan	インデックスのビットマップ・スキアン	
Bitmap Heap Scan	ヒープのビットマップ・スキアン	有
Tid Scan	TID 走査プラン	無
Subquery Scan	サブクエリー検索	無
Function Scan	関数スキアン	無
Values Scan	値スキアン	
CTE Scan	WITH 句を使った CTE スキアン	
WorkTable Scan	一時テーブル検索	
Foreign Scan	外部テーブル検索	
Custom Scan	カスタム検索	
Materialize	副問い合わせ	有

表 60 実行計画（続）

クエリー・オペレーター	動作	スタートアップ・コスト
Sort	ソート処理	有
Group	GROUP BY 句の処理	有
Aggregate	集計処理の利用	有
GroupAggregate	グループ化	
HashAggregate	ハッシュ集計処理の利用	
WindowAgg	ウインドウ集計処理	
Unique	DISTINCT / UNION 句の処理	有
SetOp	INTERCEPT / EXCEPT 句の処理	有
HashSetOp	ハッシュ処理	
LockRows	ロック	
Limit	LIMIT 句の処理	有 (OFFSET > 0)
Hash	ハッシュ処理	有
Parallel Seq Scan	パラレル全件検索	
Finalize Aggregate	パラレル処理の最終集計	
Gather	パラレル・ワーカーの集約	
Partial Aggregate	パラレル処理の集計	
Partial HashAggregate		
Partial GroupAggregate		
Single Copy	単一プロセスで実行	

以下に代表的なクエリー・オペレーターの説明を記述します。

□ Sort

ORDER BY 句で明示的に指定される場合や Unique 処理、Merge Join 処理などによる暗黙のソートでも実行される可能性があります。

□ Index Scan

インデックスからテーブルを検索します。Index Cond 実行計画が表示されない場合はインデックスのフルスキヤンを指します。

□ Index Only Scan

インデックスから必要な情報をすべて取得し、テーブルに検索を行いません。ただしビギビリティ・マップを参照した結果、テーブルにアクセスする場合もあります。

□ Bitmap Scan

BitmapOr と BitmapAnd を使ってリレーション全体のビットマップをメモリー内で作成します。

□ Result

テーブルにアクセスせずに結果を返す場合に表示されます。

□ Unique

重複値を排除します。UNION 句、DISTINCT 句の処理で使用されます。

□ Limit

ORDER BY 句と共に LIMIT 句が指定された場合に使用されます。

□ Aggregate

集計関数、GROUP BY 句で使用されます。HashAggregate、GroupAggregate が使用される場合があります。

□ Append

UNION (ALL) によるデータの追加処理で使用されます。

□ Nested Loop

INNER JOIN、LEFT OUTER JOIN で使用されます。外部テーブルをスキャンし、内部テーブルにマッチするレコードを検索します。

□ Merge Join

Merge Right Join と Merge In Join があります。ソートされたレコードセットを結合させます。

□ Hash, Hash Join

ハッシュ・テーブルを作成し、2つのテーブルを比較します。ハッシュ・テーブルの作成のために初期コストが必要です。

□ Tid Scan

Tuple Id (ctid)を指定した検索で使用されます。

□ Function

関数がレコードを生成する場合に使用されます (SELECT * FROM func()等)。

□ SetOp

INTERSECT 句、INTERSECT ALL 句、EXCEPT 句、EXCEPT ALL 句の処理で使用されます。

実行計画を作成するプランナーが選択するクエリー・オペレーターを制御するパラメーターは以下の通りです。これらのパラメーターの設定値はデフォルトで on になっています。これらのパラメーターを off に設定しても、指定したクエリー・オペレーターが完全に禁止されるわけではありません。

パラメーターを off に指定すると、スタートアップ・コストに 1.0e10 (10,000,000,000) が追加されて実行計画が比較されます。

表 61 実行計画を制御するパラメーター

パラメーター	説明	デフォルト値
enable_bitmapscan	ビットマップ・スキヤンの実行	on
enable_indexscan	インデックス・スキヤンの実行	on
enable_tidscan	TID スキヤンの実行	on
enable_seqscan	シーケンシャル・スキヤンの実行	on
enable_hashjoin	ハッシュ結合の実行	on
enable_mergejoin	マージ結合の実行	on
enable_nestloop	ネストループ結合の実行	on
enable_hashagg	ハッシュ集約の実行	on
enable_material	問い合わせプランナーの具体化の実行	on
enable_sort	ソート処理の実行	on
enable_indexonlyscan	インデックスのみで検索を完了	on

下記の例ではシーケンシャル・スキヤンのみが実行される環境で、パラメーター enable_seqscan の値を off に設定しています。初期コストが大きくなりますが、シーケンシャル・スキヤン (Seq Scan) が行われていることがわかります。

例 134 パラメーターenable_seqscan の変更

```
postgres=> SHOW enable_seqscan ;
enable_seqscan
-----
on
(1 row)

postgres=> EXPLAIN SELECT * FROM data1 ;
          QUERY PLAN
-----
Seq Scan on data1  (cost=0.00.. 7284.27 rows=466327 width=11)
(1 row)

postgres=> SET enable_seqscan = off ;
SET
postgres=> EXPLAIN SELECT * FROM data1 ;
          QUERY PLAN
-----
Seq Scan on data1  (cost=10000000000.00.. 10000007284.27 rows=466327 width=11)
(1 row)
```

5.3.4 実行時間

EXPLAIN 文に ANALYZE 句を指定すると、実際の実行時間が表示されます。出力された actual 部分には、時間が 2 つ存在します。最初の数字は、最初のレコードが出力された時間、2 つめの数字がトータル実行時間です。

例 135 実行時間の表示

```
actual_time=0.044..0.578
```

5.3.5 空テーブルのコスト計算

EXPLAIN 文を実行してレコード数 0 のテーブルに対して検索を行うと、cost 項目、rows 項目に実際とは異なる数字が表示されます。空テーブルは 10 ブロックとして計算され、10 ブロックに格納可能な最大レコード件数を見積もってコストが計算されます。

例 136 空ブロック・テーブルの rows, cost 表示

```
postgres=> CREATE TABLE data1 (c1 NUMERIC, c2 NUMERIC) ;
CREATE TABLE
postgres=> ANALYZE data1 ;
ANALYZE
postgres=> SELECT reltuples, relpages FROM pg_class WHERE relname='data1' ;
   reltuples |   relpages
   +-----+
   0 |          0
(1 row)
postgres=> EXPLAIN SELECT * FROM data1 ;
           QUERY PLAN
-----
Seq Scan on data1  (cost=0.00..18.60 rows=860 width=64)
(1 row)
postgres=>
```

5.3.6 ディスクソート

ソート処理は、パラメーター `work_mem` で指定されたメモリー内で実行されますが、レコードがメモリー内に格納できない場合はディスク上でソートが行われます。ディスクソート用の一時データは、テーブルが所属するデータベースが保存されるテーブル空間の `pgsql_tmp` ディレクトリに作成されます。データベースの保存先がテーブル空間 `pg_default` の場合は `{PGDATA}/base/pgsql_tmp`、その他のテーブル空間の場合は、`{TABLESPACEDIR}/PG_9.6_201608131/pgsql_tmp` ディレクトリが使用されます。ソートに使用されるファイル名は、「`pgsql_tmp{PID}.{9}`」です。`{PID}`はバックエンドのプロセスID、`{9}`は、0から始まる一意の番号です。

例 137 ディスクソート用ファイル

```
$ pwd
/usr/local/pgsql/data/base/pgsql_tmp
$ ls -l
total 34120
-rw----- 1 postgres postgres 34897920 Feb 11 17:02 pgsql_tmp6409.0
```

以下にディスクソートの確認方法を記述します。

□ 実行計画

EXPLAIN ANALYZE 文で実行計画を取得すると、ディスクソートを示す「Sort Method: external merge」または「Sort Method: external sort」および使用されたディスク容量がが表示されます。

例 138 ディスクソートの実行計画

```
postgres=> EXPLAIN ANALYZE SELECT * FROM data1 ORDER BY 1, 2 ;
                                         QUERY PLAN
-----
Sort  (cost=763806.52..767806.84 rows=1600128 width=138)
      (actual time=7600.693..9756.909 rows=1600128 loops=1)
  Sort Key: c1, c2
  Sort Method: external merge Disk: 34080kB
->  Seq Scan on data1  (cost=0.00..24635.28 rows=1600128 width=138)
      (actual time=1.239..501.092 rows=1600128 loops=1)
Total runtime: 9853.630 ms
(5 rows)
```

ソートを行うタプル数により、ディスクソートの方法は Replacement Selection または Quicksort が選択されます。ソート対象のタプル数がパラメーター replacement_sort_tuples (デフォルト値 150000) の指定以下の場合は、Replacement Selection が選択されます。

□ パラメーターtrace_sort

パラメーターtrace_sort を on に指定すると、ソート関連のイベントがログに出力されます (デフォルト値 off)。このパラメーターはメモリー・ソート実行時にもログを出力するため、商用環境で設定しないでください。

例 139 trace_sort=on 設定時のログ (Replacement Selection による外部ソート実行時の抜粋)

```
LOG: statement: SELECT * FROM data1 ORDER BY 1;
LOG: begin tuple sort: nkeys = 1, workMem = 4096, randomAccess = f
LOG: numeric_abbrev: cardinality 10049.436974 after 10240 values (10240 rows)
LOG: switching to external sort with 15 tapes: CPU 0.03s/0.01u sec elapsed 0.04
sec
LOG: replacement selection will sort 58253 first run tuples
LOG: performsort starting: CPU 0.99s/0.51u sec elapsed 1.51 sec
LOG: finished incrementally writing only run 1 to tape 0: CPU 1.01s/0.54u sec
elapsed 1.55 sec
LOG: performsort done: CPU 1.01s/0.54u sec elapsed 1.56 sec
LOG: external sort ended, 2687 disk blocks used: CPU 1.50s/0.95u sec elapsed
2.48 sec
```

例 140 trace_sort=on 設定時のログ (Quicksort による外部ソート実行時の抜粋)

```
LOG: begin tuple sort: nkeys = 1, workMem = 4096, randomAccess = f
LOG: numeric_abbrev: cardinality 10049.436974 after 10240 values (10240 rows)
LOG: switching to external sort with 15 tapes: CPU 0.01s/0.01u sec elapsed 0.02
sec
LOG: starting quicksort of run 1: CPU 0.01s/0.01u sec elapsed 0.02 sec
LOG: finished quicksort of run 1: CPU 0.01s/0.01u sec elapsed 0.03 sec
LOG: finished writing run 1 to tape 0: CPU 0.02s/0.01u sec elapsed 0.04 sec
LOG: performsort starting: CPU 0.31s/0.25u sec elapsed 0.56 sec
LOG: starting quicksort of run 20: CPU 0.31s/0.25u sec elapsed 0.56 sec
LOG: finished quicksort of run 20: CPU 0.31s/0.25u sec elapsed 0.56 sec
LOG: finished writing run 20 to tape 5: CPU 0.31s/0.25u sec elapsed 0.57 sec
LOG: finished 7-way merge step: CPU 0.45s/0.36u sec elapsed 0.82 sec
LOG: grew memtuples 1.29x from 58253 (1366 KB) to 74896 (1756 KB) for final
merge
LOG: tape 0 initially used 144 KB of 144 KB batch (1.000) and 4581 out of 5348
slots (0.857)
LOG: performsort done (except 14-way final merge): CPU 0.45s/0.37u sec elapsed
0.82 sec
LOG: external sort ended, 2679 disk blocks used: CPU 2.07s/0.40u sec elapsed
2.48 sec
```

□ pg_stat_database カタログ

pg_stat_database カタログには、一時ファイルに関する情報が記録されています。これらの値は ORDER BY によるディスクソートだけでなく、CREATE INDEX 文によるインデックス作成によるディスクソートもカウントされています。

表 62 pg_stat_database カタログの一時ファイル関連データ

列名	説明
datname	データベース名
temp_files	作成された一時ファイル数
temp_bytes	作成された一時ファイルの合計サイズ

5.3.8 テーブル・シーケンシャル・スキャンとインデックス・スキャン

テーブル全体にアクセスするシーケンシャル・スキャンと、インデックス・スキャンを postgres プロセスが発行するシステムコールで比較しました。インスタンス起動直後で共有バッファにデータが無い状態で検証しました。下記の例では、シーケンシャル・スキャン用に SELECT * FROM data1 を、インデックス・スキャン用に SELECT * FROM data1 BETWEEN c1 10000 AND 2000 を実行してシステムコールをトレースしています。テーブル data1 のファイルは「base/16499/16519」、インデックス idx1_data1 は「base/16499/16535」になります。

□ シーケンシャル・スキャン

シーケンシャル・スキャンでは、テーブルの先頭から 1 ブロックずつ読み込み、数ブロック読んだ後、クライアントに送信しています。複数ブロックをまとめて読むことはありません。インデックスを利用しないにもかかわらずインデックスの先頭を読み込んでいるのは実行計画決定のためと思われます。

例 141 シーケンシャル・スキャン

```

open("base/16499/16519", 0_RDWR)      = 27           ← テーブルのオープン
lseek(27, 0, SEEK_END)                = 88563712

open("base/16499/16525", 0_RDWR)      = 29           ← インデックスの読み込み
lseek(29, 0, SEEK_END)                = 67477504
lseek(29, 0, SEEK_SET)                = 0

read(29, "¥0¥0¥0¥0h¥342¥251e¥0¥0¥60¥37¥4 ¥0¥0¥0¥0b1¥5¥0¥2¥0¥0¥0"..., 8192) =
8192

lseek(27, 0, SEEK_END)                = 88563712
lseek(27, 0, SEEK_SET)                = 0           ← テーブル先頭に移動
read(27, "¥1¥0¥0¥0¥020¥374¥2¥30¥3¥0 ¥4 ¥0¥0¥0¥0¥330¥0¥260¥237D¥0"..., 8192) =
8192
read(27, "¥1¥0¥0¥0¥200S¥270¥50¥3¥0 ¥4 ¥0¥0¥0¥0¥330¥237D¥0¥237D¥0"..., 8192) =
8192
read(27, "¥1¥0¥0¥0¥360s¥270¥5¥0¥0¥5¥0 ¥4 ¥0¥0¥0¥0¥330¥230¥237D¥0"..., 8192) =
8192
    ↑ テーブルの読み込み
sendto(10, "T¥0¥0¥0000¥0¥2c1¥0¥0¥0@¥207¥0¥1¥0¥0¥67¥0"..., 8192, 0, NULL, 0) =
8192
    ↑ クライアントへ送信
read(27, "¥1¥0¥0¥0`¥224¥20¥0¥74¥2¥30¥3¥0 ¥4 ¥0¥0¥0¥0¥3260¥237D¥0"..., 8192) =
8192
read(27, "¥1¥0¥264¥270¥5¥0¥0¥4¥2¥30¥3¥0 ¥4 ¥0¥0¥0¥0¥330¥23237D¥0"..., 8192) =
8192
    ↑ テーブルの読み込み
sendto(10, "¥0¥25¥0¥2¥0¥0¥0¥0¥003556¥0¥01D¥0¥0¥0¥00355"..., 8192, 0, NULL, 0) =
8192
    ↑ クライアントへ送信

```

□ インデックス・スキャン

インデックス・スキャンでは、インデックスの読み込み→テーブルの読み込みを繰り返します。このため lseek システムコールと read システムコールが繰り返されることになります。

例 142 インデックス・スキャン

```
open("base/16499/16519", O_RDWR)      = 36 ← テーブルのオープン
lseek(36, 0, SEEK_END)                = 88563712

open("base/16499/16525", O_RDWR)      = 38 ← インデックスの読み込み
lseek(38, 0, SEEK_END)                = 67477504
lseek(38, 0, SEEK_SET)                = 0
read(38, "¥0¥0¥0¥0h¥342¥251e¥0¥00¥360¥37¥360¥37¥4 ¥05¥0¥2¥0¥0¥0"..., 8192) =
8192

lseek(38, 2375680, SEEK_SET)          = 2375680 ← インデックスの移動／読込
read(38, "¥0¥0¥033¥304¥¥0¥260¥230¥35¥360¥37¥4 0¥0¥350¥20H¥237 ¥0"..., 8192) =
8192
lseek(38, 24576, SEEK_SET)            = 24576
read(38, "¥0¥0¥0¥0¥210if¥0L¥3 (¥23¥360¥37¥4 ¥¥340¥237 ¥0¥337¥20¥0"..., 8192) =
8192
lseek(38, 237568, SEEK_SET)          = 237568
read(38, "¥1¥0¥0¥0¥330¥2302¥v200¥26¥360¥37¥4 340¥237 ¥0¥0¥237 ¥0"..., 8192) =
8192

lseek(36, 434176, SEEK_SET)          = 434176 ← テーブルの移動／読込
read(36, "¥1¥¥0X¥352¥276¥¥374¥2¥30¥3¥0 ¥4 ¥330¥237D¥0¥260¥237D¥0"..., 8192) =
8192

sendto(10, "T¥0¥0¥0¥33¥02¥0¥0¥4¥23¥377¥¥16¥0¥0D¥¥0¥0"..., 8192, 0, NULL, 0) =
8192
↑ クライアントに対する送信
```

5.3.9 BUFFERS 指定

EXPLAIN 文に ANALYZE オプションと合わせて BUFFERS オプションを指定すると、実行時に取得したバッファ情報が出力されます。出力される情報は、共有バッファ (shared)、ローカル・バッファ (local)、一時セグメント (temp) のカテゴリーごとのバッファ I/O 情報です。

例 143 BUFFERS オプション

```
postgres=> EXPLAIN (ANALYZE true, BUFFERS true) SELECT * FROM stat1 s1
          ORDER BY c1, c2 ;
                                         QUERY PLAN
-----
Sort  (cost=1041755.43..1057768.57 rows=6405258 width=10)
      (actual time=20067.420..25192.887 rows=6406400 loops=1)
  Sort Key: c1, c2
  Sort Method: external merge Disk: 130288kB
  Buffers: shared hit=16153 read=18477, temp read=33846 written=33846
->  Seq Scan on stat1 s1  (cost=0.00..98682.58 rows=6405258 width=10)
      (actual time=0.290..751.019 rows=6406400 loops=1)
      Buffers: shared hit=16153 read=18477
Planning time: 0.079 ms
Execution time: 25535.583 ms
(8 rows)
```

Buffers: 以降に出力される項目は以下の通りです。出力される数字はすべてブロック数（8 KB 単位）です。

表 63 Buffers: 出力項目

カテゴリー	項目名	説明
shared	hit	共有バッファのキャッシュ・ヒット
	read	共有バッファのキャッシュ・ミス
	dirtied	ダーティ・バッファの読み込み
	written	共有バッファのキャッシュ書き込み
local	hit	ローカル・バッファのキャッシュ・ヒット
	read	ローカル・バッファのキャッシュ・ミス
	dirtied	ダーティ・バッファの読み込み
	written	ローカル・バッファのキャッシュ書き込み
temp	read	一時セグメントの読み込み
	written	一時セグメントの書き込み

5.4 パラメーター

5.4.1 パフォーマンスに関連するパラメーター

PostgreSQL のパフォーマンスに関連する主なパラメーターは以下の通りです。

表 64 関連パラメーター

パラメーター	説明	バージョン
autovacuum_work_mem	自動 Vacuum の一時メモリー	9.4~
effective_cache_size	単一の問い合わせで利用できるディスクキャッシュの実効容量	
effective_io_concurrency	同時実行ディスク I/O 作業の数	
huge_pages	Huge Pages を使用するか	9.4~
maintenance_work_mem	VACUUM、CREATE INDEX 等の保守作業用メモリー	
shared_buffers	共有バッファのサイズ	
temp_buffers	一時テーブル用メモリーのサイズ	
wal_buffers	WAL 情報が格納される共有バッファ	
work_mem	ソート、ハッシュ用の一時メモリー	
max_wal_size	チェックポイント発生契機となる WAL 書き込み量	9.5~
replacement_sort_tuples	外部ソートで Replacement Selection を行う最大タプル数	9.6~

5.4.2 effective_cache_size

このパラメーターは共有バッファと、OS が使用するキャッシュの合計を指定します。主に実行計画決定時のインデックス・スキヤンのコストを計算するパラメーターとして使用されます。 `index_pages_fetched` 関数 (`src/backend/optimizer/path/costsize.c`)、`gistInitBuffering` 関数 (`src/backend/access/gist/gistbuild.c`) で使用されます。

5.4.3 effective_ioConcurrency

このパラメーターの指定値は GUC `target_prefetch_pages` にコピーされます。指定された値は実行計画ビットマップ・ヒープ・スキヤン実行時のプリフェッチ数の計算に使用されます。マニュアルの記述はわかりにくく、変更による効果は検証できていません。パラメー



ターを大きくするとプリフェッチ数が大きくなります。ビットマップ・ヒープ・スキャン以外には使用されません。PostgreSQL 9.6 のマニュアルには以下の記述が追加されました。

「SSDs and other memory-based storage can often process many concurrent requests, so the best value might be in the hundreds.」

5.5 システム・カタログ

5.5.1 システム・カタログの実体

システム・カタログ（pg_で始まる名前のビュー）は、マニュアル上では「通常のテーブル」とされていますが、実行統計を取得するカタログ（pg_stat で始まるカタログ）は異なるデータソースから情報を提供しています。実行計画から、実体を検証しました。

表 65 システム・カタログの実体

システム・カタログ	情報取得元
pg_statio_*_indexes	pg_namespace, pg_class, pg_index, pg_stat_*()
pg_statio_*_sequences	pg_namespace, pg_class, pg_stat_*()
pg_statio_*_tables	pg_namespace, pg_index, pg_class, pg_stat_*()
pg_stat_activity	pg_database, pg_authid, pg_stat_get_activity()
pg_stat_archiver	pg_database, pg_stat_get_db_*
pg_stat_bgwriter	pg_stat_get_*
pg_stat_database	pg_database, pg_stat_get_*
pg_stat_database_conflicts	pg_database, pg_authid, pg_stat_get_*
pg_stat_replication	pg_authid, pg_stat_get_activity(), pg_stat_get_wal_senders()
pg_stat_*_indexes	pg_namespace, pg_class, pg_index, pg_stat_get_*
pg_stat_*_tables	pg_namespace, pg_class, pg_index, pg_stat_get_*
pg_stat_*_functions	pg_namespace, pg_proc, pg_stat_get_function_*
pg_stat_xact_*_tables	pg_namespace, pg_class, pg_index, pg_stat_get_xact_*
pg_stat_xact_*_functions	pg_namespace, pg_proc, pg_index, pg_stat_get_xact_*
pg_stat_xact_*_tables	pg_namespace, pg_index, pg_stat_get_xact_*
pg_statistic	table
pg_stats	pg_namespace, pg_class, pg_statistic, pg_attribute, has_column_privilege()
pg_stat_progress_vacuum	pg_stat_get_progress_info(), pg_database
pg_stat_wal_receiver	pg_stat_get_wal_receiver()

6. SQL 文の仕様

6.1 ロック

6.1.1 ロックの種類

PostgreSQL はテーブル上のレコードの整合性を保つため自動的にロックを取得します。LOCK TABLE 文や SELECT FOR UPDATE 文によりアプリケーションが明示的にロックを行う場合もあります。通常ロックはトランザクションが確定（COMMIT または ROLLBACK）するまで保持されます。ロックに関する詳しい情報はマニュアル¹²に記載されています。

表 66 ロックの種類

ロック	説明
ACCESS SHARE	SELECT 文を実行すると対象テーブルに対して取得されます。最も弱いロックになります。
ROW SHARE	SELECT FOR UPDATE / SELECT FOR SHARE 文を実行するとテーブルに対して取得されます。
ROW EXCLUSIVE	UPDATE, DELETE, INSERT 文が取得するロックです。参照されるテーブルには ACCESS SHARE ロックも取得されます。
SHARE UPDATE EXCLUSIVE	VACUUM, ANALYZE, CREATE INDEX CONCURRENTLY, ALTER TABLE 文等で取得されます。
SHARE	CONCURRENTLY 句が無い CREATE INDEX 文で取得されます。
SHARE ROW EXCLUSIVE	自動的には取得されません。
EXCLUSIVE	自動的には取得されません。
ACCESS EXCLUSIVE	ALTER TABLE, DROP TABLE, TRUNCATE, REINDEX, CLUSTER, VACUUM FULL 文により取得されます。

¹² マニュアルは <http://www.postgresql.org/docs/9.6/static/explicit-locking.html>

6.1.2 ロックの取得

Oracle Database 等とは異なり、PostgreSQLは単純なSELECT文でもACCESS SHAREロックを取得します。LOCK TABLE 文でIN句を指定しない場合、ACCESS EXCLUSIVEロックを取得します。このためLOCK TABLE文を実行したテーブルにはSELECT文も含めてアクセスができないになります。現在のロック状況はpg_locksカタログから確認できます。

例 144 LOCK TABLE 文とロック

```
postgres=> BEGIN ;  
BEGIN  
postgres=> LOCK TABLE data1 ;  
LOCK TABLE  
postgres=>  
別セッションから  
postgres=> SELECT locktype, relation, mode FROM pg_locks ;  
locktype | relation | mode  
-----+-----+-----  
virtualxid | | ExclusiveLock  
relation | 16519 | AccessExclusiveLock
```

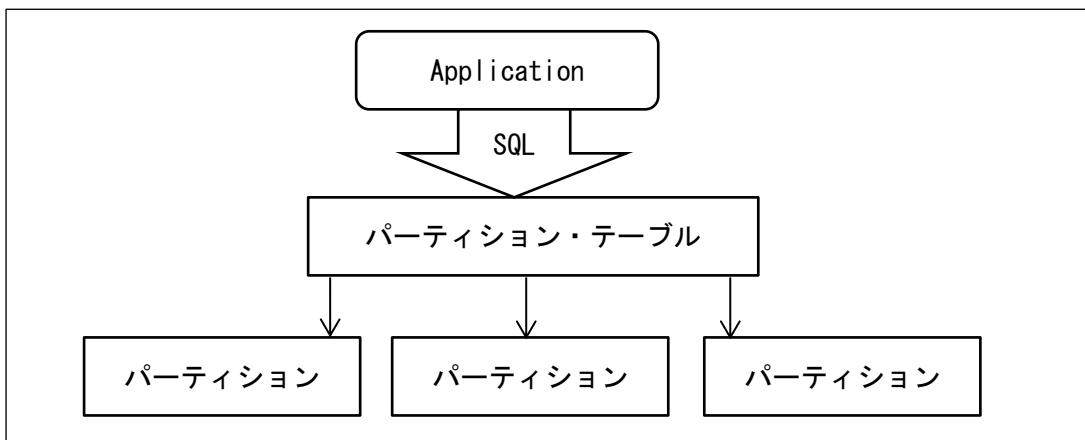
ACCESS EXCLUSIVE LOCKは他の全てのロックと競合します。このためVACUUM FULLのようにこのロックを取得する処理を実行中のテーブルには検索を含めてアクセスできなくなります。

6.2 パーティション・テーブル

6.2.1 パーティション・テーブルとは

パーティション・テーブルは大規模なテーブルを物理的に複数のパーティションに分割し、I/O 範囲を減少させることでパフォーマンスやメンテナンス性を向上させる機能です。アプリケーションからはパーティションは隠ぺいされるため、単一のテーブルが存在しているように見えます。

図 15 パーティション・テーブル



一般的に、パーティション・テーブルのレコードは列値の範囲や固定値によってどのパーティションに格納されるかが自動的に判断されます。

6.2.2 パーティション・テーブルの実装

Oracle Database や Microsoft SQL Server 等とは異なり、PostgreSQL はネイティブに使用できるパーティション・テーブルの機能を持っていません。テーブルの継承、チェック制約、トリガー等を組み合わせることでパーティション・テーブルの機能を実装します。

PostgreSQLにおけるパーティション・テーブルは以下の手順で作成します。

- 親テーブルを作成（アプリケーションが SQL アクセスするテーブル）
- 親テーブルを継承（inherits）し、チェック制約を含む継承テーブルを作成（パーティション）
- トリガー用関数を作成
- 親テーブルに INSERT トリガーを作成し、トリガー用関数を登録
- 親テーブルに対して SQL 文を実行（アプリケーション発行 SQL）

以下の例は main1 テーブルを key1 列の範囲によって、3 つのパーティション（main1_part100, main1_part200, main1_part300）に分割する例です。

例 145 親テーブル作成

```
postgres=> CREATE TABLE main1 (key1 NUMERIC, val1 VARCHAR(10), val2  
VARCHAR(10)) ;  
CREATE TABLE
```

パーティションとなる継承テーブルを作成します。INHERITS 句に親テーブルを指定し、パーティションに含まれるレコードに対する CHECK 制約を指定します。

例 146 継承テーブル（パーティション）作成

```
postgres=> CREATE TABLE main1_part100 (  
          CHECK(key1 < 100)  
      ) INHERITS (main1) ;  
CREATE TABLE  
postgres=> CREATE TABLE main1_part200 (  
          CHECK(key1 >= 100 AND key1 < 200)  
      ) INHERITS (main1) ;  
CREATE TABLE  
postgres=> CREATE TABLE main1_part300 (  
          CHECK(key1 >= 200 AND key1 < 300)  
      ) INHERITS (main1) ;  
CREATE TABLE
```

トリガーに使用するファンクションを作成します。

例 147 トリガー関数作成

```
postgres=> CREATE OR REPLACE FUNCTION func_main1_insert()
              RETURNS TRIGGER AS $$

              BEGIN
                  IF      (NEW.key1 < 100) THEN
                      INSERT INTO main1_part100 VALUES (NEW.*);
                  ELSIF (NEW.key1 >= 100 AND NEW.key1 < 200) THEN
                      INSERT INTO main1_part200 VALUES (NEW.*);
                  ELSIF (NEW.key1 < 300) THEN
                      INSERT INTO main1_part300 VALUES (NEW.*);
                  ELSE
                      RAISE EXCEPTION 'ERROR! key1 out of range.' ;
                  END IF ;
                  RETURN NULL ;
              END ;
              $$

              LANGUAGE plpgsql ;

CREATE FUNCTION
```

CREATE FUNCTION 文により作成されたファンクションを、トリガーに登録します。

例 148 トリガー作成

```
postgres=> CREATE TRIGGER trg_main1_insert
              BEFORE INSERT ON main1
              FOR EACH ROW EXECUTE PROCEDURE func_main1_insert() ;
CREATE TRIGGER
```

6.2.3 実行計画の確認

パーティション・テーブルに対する実行計画を検証しました。

□ SELECT 文によるパーティション選択

WHERE 句にパーティションを特定できる構文がある場合は、自動的に継承テーブルのみにアクセスします。ただしパーティション化列の特定に計算が必要な場合（左辺が計算値等）は全パーティション・テーブルにアクセスされます。

例 149 SELECT 文の実行計画

```
postgres=> EXPLAIN SELECT * FROM main1 WHERE key1 = 10 ;
                                                     QUERY PLAN
-----
Append  (cost=0.00..8.17 rows=2 width=108)
-> Seq Scan on main1  (cost=0.00..0.00 rows=1 width=108)
    Filter: (key1 = 10::numeric)
-> Index Scan using pk_main1_part100 on main1_part100  (cost=0.15..8.17 rows=1 width=108)
    Index Cond: (key1 = 10::numeric) ↑ 1つのパーティション・テーブルのみ
Planning time: 0.553 ms
(6 rows)

postgres=> EXPLAIN SELECT * FROM main1 WHERE key1 + 1 = 11 ;
                                                     QUERY PLAN
-----
Append  (cost=0.00..20.88 rows=6 width=108)
-> Seq Scan on main1  (cost=0.00..0.00 rows=1 width=108)
    Filter: ((key1 + 1::numeric) = 11::numeric)
-> Seq Scan on main1_part100  (cost=0.00..18.85 rows=3 width=108)
    Filter: ((key1 + 1::numeric) = 11::numeric)
-> Seq Scan on main1_part200  (cost=0.00..1.01 rows=1 width=108)
    Filter: ((key1 + 1::numeric) = 11::numeric)
-> Seq Scan on main1_part300  (cost=0.00..1.01 rows=1 width=108)
    Filter: ((key1 + 1::numeric) = 11::numeric)
Planning time: 0.167 ms ↑ 全パーティション・テーブルにアクセス
(10 rows)
```

□ INSERT 文の実行

INSERT 文はトリガーによるパーティション・テーブルへの振り分けが行われます。実行計画上はトリガーが起動されたことのみ出力されます。

例 150 INSERT 文の実行計画

```
postgres=> EXPLAIN ANALYZE VERBOSE INSERT INTO main1 VALUES (101, 'val1', 'val2') ;
                                         QUERY PLAN
-----
Insert on public.main1  (cost=0.00..0.01 rows=1 width=0) (actual time=0.647..0.647 rows=0
loops=1)
    -> Result  (cost=0.00..0.01 rows=1 width=0) (actual time=0.001..0.001 rows=1 loops=1)
        Output: 101::numeric, 'val1'::character varying(10), 'val2'::character varying(10)
Planning time: 0.046 ms
Trigger trg_main1_insert: time=0.635 calls=1
Execution time: 0.675 ms
(6 rows)
```

□ DELETE 文の実行

DELETE 文は WHERE 句によるパーティション特定が可能であれば特定のパーティションのみにアクセスします。パーティション特定の条件は SELECT 文と同じです。

例 151 DELETE 文の実行計画

```
postgres=> EXPLAIN DELETE FROM main1 WHERE key1 = 100 ;
                                         QUERY PLAN
-----
Delete on main1  (cost=0.00..8.17 rows=2 width=6)
    -> Seq Scan on main1  (cost=0.00..0.00 rows=1 width=6)
        Filter: (key1 = 100::numeric)
    -> Index Scan using pk_main1_part200 on main1_part200  (cost=0.15..8.17 rows=1 width=6)
        Index Cond: (key1 = 100::numeric)
Planning time: 0.973 ms
(6 rows)
```

□ UPDATE 文の実行

UPDATE 文は WHERE 句によるパーティション特定が可能であれば特定のパーティションのみにアクセスします。条件は SELECT 文と同じです。

例 152 UPDATE 文の実行計画

```
postgres=> EXPLAIN UPDATE main2 SET val1='upd' WHERE key1 = 100 ;
QUERY PLAN
-----
Update on main2 (cost=0.00..8.30 rows=2 width=46)
-> Seq Scan on main2 (cost=0.00..0.00 rows=1 width=76)
  Filter: (key1 = 100::numeric)
-> Index Scan using pk_main2_part1 on main2_part1 (cost=0.29..8.30 rows=1 width=15)
  Index Cond: (key1 = 100::numeric)
Planning time: 1.329 ms
(6 rows)
```

□ JDBC PreparedStatement

PreparedStatement オブジェクトを使ってパーティション・キー列をバインド変数化した場合の実行計画を確認しました。JDBC Driver は postgresql-9.3-1101.jdbc41.jar、JRE は 1.7.0_09-icedtea を使用しました。

パーティション・キーにバインド変数を使用している場合でも、パーティションの自動選択機能が動作することを確認しました。

例 153 Java アプリケーションの一部

```
PreparedStatement st = cn.prepareStatement("SELECT * FROM main1 WHERE key1=?") ;
st.setInt(1, 200) ;
ResultSet rs = st.executeQuery() ;
```

例 154 実行計画

```
Append (cost=0.00..188.99 rows=2 width=62) (actual time=0.018..2.947 rows=1 loops=1)
  -> Seq Scan on main1 (cost=0.00..0.00 rows=1 width=108)
      (actual time=0.003..0.003 rows=0 loops=1)
      Filter: (key1 = 200::numeric)
  -> Seq Scan on main1_part200 (cost=0.00..188.99 rows=1 width=16)
      (actual time=0.014..2.943 rows=1 loops=1)
      Filter: (key1 = 200::numeric)
      Rows Removed by Filter: 9998
Planning time: 1.086 ms
Execution time: 3.005 ms
```

6.2.4 制約

パーティション・テーブルの親テーブルに作成された制約は機能しません。制約は継承テーブルに指定します。以下の例では親テーブルに主キーを指定していますが、主キー違反レコードが格納できています。

例 155 主キー制約違反のレコード格納

```
postgres=> ALTER TABLE main1 ADD CONSTRAINT pk_main1 PRIMARY KEY (key1) ;
ALTER TABLE
postgres=> INSERT INTO main1 VALUES (100, 'val1', 'val2') ;
INSERT 0 0
postgres=> INSERT INTO main1 VALUES (100, 'val1', 'val2') ;
INSERT 0 0
```

6.2.5 パーティション間のレコード移動

パーティション・キー列の値を、別のパーティションに格納すべき値に変更する UPDATE 文は実行できません。CHECK 制約によるエラーが発生します。

例 156 パーティション・キー列の更新

```
postgres=> \d+ main1_part1
                                         Table "public.main1_part1"
  Column |      Type       | Modifiers | Storage | Stats target | Description
-----+----------------+-----+-----+-----+
  key1  | numeric        |          | main   |              |
  val1  | character varying(10) |          | extended |              |
  val2  | character varying(10) |          | extended |              |
Check constraints:
    "main1_part1_key1_check" CHECK (key1 < 10000::numeric)
Inherits: main1

postgres=> UPDATE main1 SET key1 = 15000 WHERE key1 = 100 ;
ERROR:  new row for relation "main1_part1" violates check constraint
"main1_part1_key1_check"
DETAIL: Failing row contains (15000, val1, val2).
postgres=>
```

6.2.6 パーティション・テーブルと統計情報

パーティション・テーブルにおける統計情報は、親テーブルおよび継承テーブルごとに独立して格納されています。親テーブルの pg_class カタログの reltuples 列と relopages 列の値は 0 になっています。pg_stats カタログの親テーブルの統計情報は全継承テーブルの合計が格納されます。

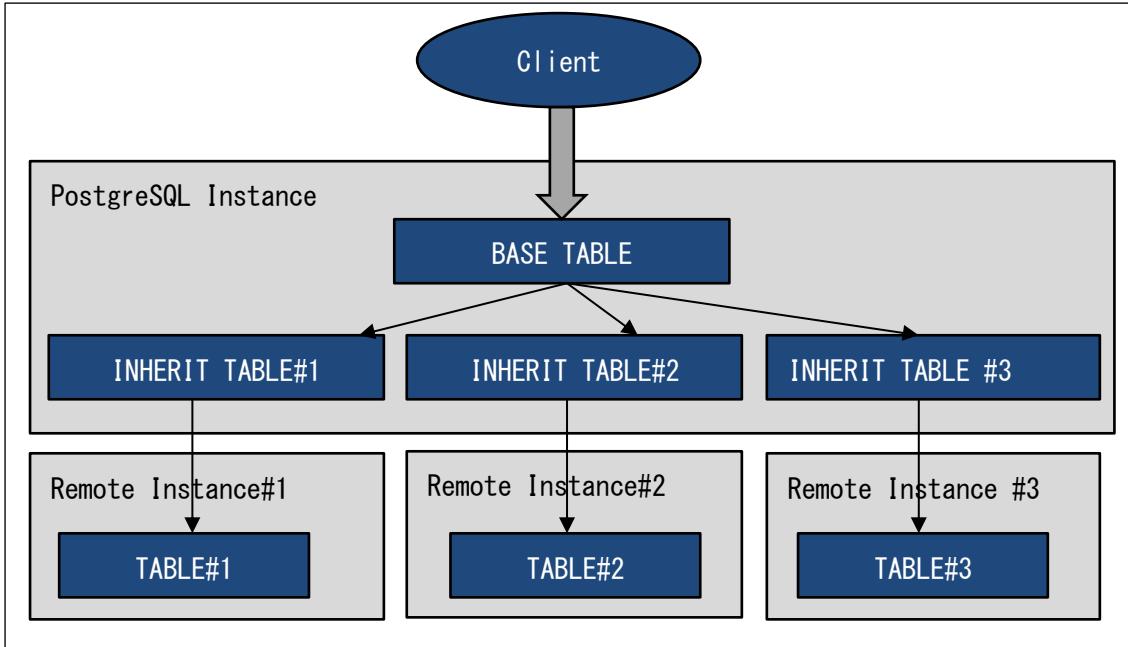
親テーブルに対する ANALYZE 文では継承テーブルの統計情報は更新されません。継承テーブルの統計情報を更新する場合には、継承テーブル単位に ANALYZE 文を実行します。

親テーブルの統計情報が自動 VACUUM により更新されるのは、親テーブルに対する更新 DML が実行された場合のみです。継承テーブルに対して直接更新 DML を実行した場合、親テーブルに対する ANALYZE は実行されません。この結果、親テーブルの統計情報と継承テーブルの統計情報の合計に差異が発生する可能性があります。

6.2.7 パーティション・テーブルと外部テーブル

PostgreSQL 9.5 では既存テーブルの継承テーブルとして、外部テーブル (FOREIGN TABLE) を作成できるようになりました。この機能により、パーティショニングと外部テーブルを組み合わせて処理を複数ホストに分散することが可能になります。

図 16 FOREIGN TABLE INHERIT



構文 4 CREATE FOREIGN TABLE 文

```
CREATE FOREIGN TABLE table_name (check_constraints ...)
INHERITS (parent_table)
SERVER server_name
OPTIONS (option = 'value' ...)
```

PostgreSQL 9.4 と異なる部分は INHERITS 句です。ここで親テーブルを指定します。以下に実装例と、実行計画を検証します。

例 157 親テーブルの作成

```
postgres=> CREATE TABLE parent1(key NUMERIC, val TEXT) ;
CREATE TABLE
```

リモート・インスタンス上でテーブル (inherit1、inherit2、inherit3) を作成します。

例 158 リモート・インスタンスで子テーブルの作成

```
postgres=> CREATE TABLE inherit1(key NUMERIC, val TEXT) ;
CREATE TABLE
```

CREATE FOREIGN TABLE 文を実行して、各リモート・インスタンス上のテーブル (inherit1, inherit2, inherit3) に対する外部テーブルを作成します。

例 159 外部テーブルの作成

```
postgres=# CREATE EXTENSION postgres_fdw ;
CREATE EXTENSION
postgres=# CREATE SERVER remsvr1 FOREIGN DATA WRAPPER postgres_fdw
    OPTIONS (host 'remsvr1', dbname 'demodb', port '5432') ;
CREATE SERVER
postgres=# CREATE USER MAPPING FOR public SERVER remsvr1
    OPTIONS (user 'demo', password 'secret') ;
CREATE USER MAPPING
postgres=# GRANT ALL ON FOREIGN SERVER remsvr1 TO public ;
GRANT
postgres=> CREATE FOREIGN TABLE inherit1(CHECK(key < 1000))
    INHERITS (parent1) SERVER remsvr1 ;
CREATE FOREIGN TABLE
```

PostgreSQL 9.4 と異なる部分は CREATE FOREIGN TABLE 文に列定義ではなく、CHECK 制約を指定している部分と INHERITS 句で継承元テーブルを指定している点です。上記は 1 サーバーのみの例ですが、複数インスタンスに対して CREATE SERVER 文、CREATE USER MAPPING 文、CREATE FOREIGN TABLE 文を作成します。

例 160 INHERITS 句を指定した FOREIGN TABLE 定義

```
postgres=> \d+ inherit2
              Foreign table "public.inherit2"
 Column | Type   | Modifiers | FDW Options | Storage | Stats target | Description
-----+-----+-----+-----+-----+-----+
 key   | numeric |           |           | main   |             |
 val   | text    |           |           | extended |             |
Check constraints:
    "inherit2_key_check" CHECK (key >= 1000::numeric AND key < 2000::numeric)
Server: remsvr2
Inherits: parent1
```

実行計画を確認すると、CHECK 制約により特定のインスタンスにのみアクセスしていることがわかります。

例 161 実行計画の確認

```
postgres=> EXPLAIN SELECT * FROM parent1 WHERE key = 1500 ;
          QUERY PLAN
-----
Append  (cost=0.00..121.72 rows=6 width=64)
->  Seq Scan on parent1  (cost=0.00..0.00 rows=1 width=64)
      Filter: (key = '1500'::numeric)
->  Foreign Scan on inherit2  (cost=100.00..121.72 rows=5 width=64)
(4 rows)
```

6.3 シーケンス

6.3.1 シーケンスの使い方

シーケンス (SEQUENCE) は自動的に一意な数値を生成するオブジェクトで、CREATE SEQUENCE 文を使って作成します。詳しい使用方法はマニュアル¹³を参照してください。

シーケンスを使って値を取得するには、nextval 関数にシーケンス名を指定します。現在の値を取得するためには currval 関数を指定します。

セッション内で nextval 関数を実行しない状態で currval 関数を実行するとエラーになります。

例 162 シーケンスの利用

```
postgres=> CREATE SEQUENCE seq01 ;
CREATE SEQUENCE
postgres=> SELECT currval('seq01') ;
ERROR: currval of sequence "seq01" is not yet defined in this session
postgres=> SELECT nextval('seq01') ;
nextval
-----
1
(1 row)

postgres=> SELECT currval('seq01') ;
currval
-----
1
(1 row)
```

シーケンス一覧は、psql ユーティリティの `\$d` コマンドまたは、`information_schema.sequences` ビューを参照します。またシーケンスの個別の情報を取得するには、シーケンス名をテーブル名に指定して `SELECT` 文を実行します。

¹³ マニュアルは <https://www.postgresql.org/docs/9.6/static/sql-createsequence.html>

例 163 シーケンスの情報取得

```
postgres=> \ds+
              List of relations
 Schema | Name   | Type  | Owner | Size    | Description
-----+-----+-----+-----+-----+-----+
 public | seq01 | sequence | data1 | 8192 bytes |
(1 row)
postgres=> SELECT sequence_schema, sequence_name, start_value FROM
            information_schema.sequences ;
sequence_schema | sequence_name | start_value
-----+-----+-----+
 public          | seq01        | 1
(1 row)
postgres=> SELECT sequence_name, start_value, cache_value FROM seq01 ;
sequence_name | start_value | cache_value
-----+-----+-----+
 seq01         |           1 |           1
(1 row)
```

6.3.2 キャッシュ

CREATE SEQUENCE 文には CACHE 句を指定できます。この属性にはシーケンス値をキャッシュする個数を指定できます。CACHE 句のデフォルト値は 1 で、キャッシュは生成されません。マニュアルには「The optional clause CACHE cache specifies how many sequence numbers are to be preallocated and stored in memory for faster access. The minimum value is 1 (only one value can be generated at a time, i.e., no cache), and this is also the default.」と記載されていますが、memory がどのメモリー領域を指すのかは記載されていません。

実際にキャッシュが行われるメモリー領域はバックエンド・プロセス postgres の仮想メモリー領域です。あるシーケンスに対して複数のセッションがシーケンス値を取得すると、それぞれのセッションに対応するキャッシュが生成されます。同一の値が取得されることはありませんが、シーケンス値の大小関係と時系列は一致しなくなります。

例 164 シーケンス値のキャッシュと時系列

```
(SESSION#1) postgres=> CREATE SEQUENCE seq01 CACHE 10 ;
CREATE SEQUENCE
(SESSION#1) postgres=> SELECT nextval('seq01') ;
nextval
-----
 1
(1 row)
(SESSION#2) postgres=> SELECT nextval('seq01') ;
nextval
-----
 11
(1 row)

(SESSION#1) postgres=> SELECT nextval('seq01') ;
nextval
-----
 2
(1 row)

(SESSION#2) postgres=> SELECT nextval('seq01') ;
nextval
-----
 12
(1 row)
```

上記の例ではまず、CACHE 10 を指定してシーケンスを作成し、nextval 関数で値を取得しています。この時点では SESSION#1 セッションにはキャッシュとして 1~10 が作成されます。次に SESSION#2 セッションで nextval 関数を実行すると、別のキャッシュ 11~20 が作成され、nextval 関数には 11 が返ります。

PostgreSQL 9.4 ではセッション内のシーケンス・キャッシュを削除する DISCARD SEQUENCES 文が利用できるようになりました。

6.3.3 トランザクション

シーケンス値はトランザクションとは独立しています。トランザクション中に取得したシーケンス値はロールバックできません。

例 165 シーケンスとトランザクション

```
postgres=> BEGIN ;
BEGIN
postgres=> SELECT nextval('seq01') ;
nextval
-----
3
(1 row)
postgres=> ROLLBACK ;
ROLLBACK
postgres=> SELECT nextval('seq01') ;
nextval
-----
4
(1 row)
postgres=>
```

6.4 バインド変数とPREPARE文

PREPARE文を実行すると、セッション上にプリペアド文オブジェクトを作成することができます。プリペアド文を再利用することで、構文解析と書き換えの負荷を削減することができます。auto_explain Contrib モジュールを使って実行計画はEXECUTE文実行時に作成されることを確認しました。PREPARE文で作成したSQL文の実行計画がバインド変数によって変化するかを確認しました。

以下の例ではPREPARE文でbind1testオブジェクトを作成後、異なるパラメーターで2回SELECT文を実行しています。初回の実行ではテーブル内のほとんどのデータを取得し、2回目の実行ではテーブル内的一部を取得しています。列c2にはインデックスが付与されています。

例 166 PREPAREオブジェクトの作成と実行

```
postgres=> PREPARE bind1test(VARCHAR) AS SELECT COUNT(*) FROM bind1 WHERE
c2=$1 ;
PREPARE
postgres=> EXECUTE bind1test('maj') ;
      count
-----
10000000
(1 row)

postgres=> EXECUTE bind1test('min') ;
      count
-----
101
(1 row)
```

auto_explain Contrib モジュールにより実行計画を出力しました。初回の実行ではSeq Scanが採用されています。

例 167 初回実行の実行計画

```
LOG: duration: 1583.140 ms plan:  
    Query Text: PREPARE bind1test(VARCHAR) AS SELECT COUNT(*) FROM bind1  
    WHERE c2=$1 ;  
        Aggregate (cost=204053.52..204053.53 rows=1 width=0)  
            -> Seq Scan on bind1 (cost=0.00..179055.85 rows=9999068 width=0)  
                Filter: ((c2)::text = 'maj'::text)
```

2回目の実行では、Index Only Scan が採用されています。

例 168 2回目実行の実行計画

```
LOG: duration: 0.046 ms plan:  
    Query Text: PREPARE bind1test(VARCHAR) AS SELECT COUNT(*) FROM bind1  
    WHERE c2=$1;  
        Aggregate (cost=41.44..41.45 rows=1 width=0)  
            -> Index Only Scan using idx1_bind1 on bind1 (cost=0.43..38.94  
rows=1000 width=0)  
                Index Cond: (c2 = 'min'::text)
```

結果として実行計画が変更されたことがわかります。このため PREPARE 文を実行した場合でも実行計画は都度作成されることが確認できました。この動作は PostgreSQL 9.2 から採用されています。Oracle Database や Microsoft SQL Server のように実行計画までキャッシュするわけではありません。

6.5 INSERT ON CONFLICT

6.5.1 INSERT ON CONFLICT 文の基本構文

PostgreSQL 9.5 では制約違反となる INSERT 文実行時に自動的に UPDATE 文に切り替えること(いわゆる UPSERT 文)ができるようになりました。INSERT 文に ON CONFLICT 句を指定します。

構文 5 INSERT INTO ON CONFLICT 文

```
INSERT INTO ...
ON CONFLICT [ { (column_name, ...) | ON CONSTRAINT constraint_name } ]
{ DO NOTHING | DO UPDATE SET column_name = value }
[ WHERE ... ]
```

ON CONFLICT 部分には制約違反が発生する場所を指定します。

- 列名のリストまたは、「ON CONSTRAINT 制約名」の構文で制約名を指定します。
- 複数列で構成される制約を指定する場合は、制約に含まれる全ての列名を指定する必要があります。
- ON CONFLICT 以降を省略すると全ての制約違反がチェックされます。省略できるのは DO NOTHING を使用する場合のみです。
- ON CONFLICT 句で指定された列または制約以外の制約違反が発生すると、INSERT 文はエラーになります。

ON CONFLICT 句以降には制約違反が発生した場合の動作を記述します。DO NOTHING 句を指定すると、制約違反が発生しても何もしません(制約違反も発生しません)。DO UPDATE 句を指定すると、特定の列を UPDATE します。以下に実行例を記載します。

例 169 テーブルの準備

```
postgres=> CREATE TABLE upsert1 (key NUMERIC, val VARCHAR(10)) ;
CREATE TABLE
postgres=> ALTER TABLE upsert1 ADD CONSTRAINT pk_upsert1 PRIMARY KEY (key) ;
ALTER TABLE
postgres=> INSERT INTO upsert1 VALUES (100, 'Val 1') ;
INSERT 0 1
postgres=> INSERT INTO upsert1 VALUES (200, 'Val 2') ;
INSERT 0 1
postgres=> INSERT INTO upsert1 VALUES (300, 'Val 3') ;
INSERT 0 1
```

以下は ON CONFLICT 句の記述例です。処理部分には DO NOTHING を指定しているので、制約違反が発生しても何もしません。

例 170 ON CONFLICT 句

```
postgres=> INSERT INTO upsert1 VALUES (200, 'Update 1')
          ON CONFLICT DO NOTHING ;           ← 制約名や列を省略
INSERT 0 0
postgres=> INSERT INTO upsert1 VALUES (200, 'Update 1')
          ON CONFLICT(key) DO NOTHING ;   ← 制約違反が発生する列を記述
INSERT 0 0
postgres=> INSERT INTO upsert1 VALUES (200, 'Update 1')
          ON CONFLICT(val) DO NOTHING ;  ← 制約が無い列を指定するとエラー発生
ERROR: there is no unique or exclusion constraint matching the ON CONFLICT
specification
postgres=> INSERT INTO upsert1 VALUES (200, 'Update 1')
          ON CONFLICT ON CONSTRAINT pk_upsert1 DO NOTHING ;  ← 制約名を指定
INSERT 0 0
```

DO UPDATE 句には、更新処理を記述します。基本的には UPDATE 文の SET 句以降と同じです。EXCLUDED というエイリアスを使用すると、INSERT 文を実行しようとして格納できなかったレコードにアクセスできます。

例 171 DO UPDATE 句

```

postgres=> INSERT INTO upsert1 VALUES (400, 'Upd4')
      ON CONFLICT DO UPDATE SET val = EXCLUDED.val ; ← 制約を省略してエラー
ERROR:  ON CONFLICT DO UPDATE requires inference specification or constraint
       name
LINE 2: ON CONFLICT DO UPDATE SET val = EXCLUDED.val ;
          ^
HINT:  For example, ON CONFLICT ON CONFLICT (<column>).
postgres=> INSERT INTO upsert1 VALUES (400, 'Upd4')
      ON CONFLICT(key) DO UPDATE SET val = EXCLUDED.val ; ← EXCLUDED エイリアスを
       使用
UPSERT 0 1
postgres=> INSERT INTO upsert1 VALUES (400, 'Upd4')
      ON CONFLICT(key) DO UPDATE SET val = EXCLUDED.val WHERE upsert1.key = 100 ;
UPSERT 0 0      ↑ WHERE 句を指定して UPDATE 条件を決定できる

```

6.5.2 ON CONFLICT 句とトリガーの関係

INSERT ON CONFLICT 文の実行時にトリガーがどのように動作するかを検証しました。 BEFORE INSERT トリガーは常に動作しました。 DO UPDATE 文によりレコードが更新される場合は、 BEFORE INSERT トリガー、 BEFORE / AFTER UPDATE トリガーが動作しました。 WHERE 句により UPDATE が行われなかつた場合は BEFORE INSERT トリガーのみが実行されました。

表 67 トリガーの起動 (ON EACH ROW トリガー／数字は実行順)

トリガー	INSERT 成功	DO NOTHING	DO UPDATE (更新あり)	DO UPDATE (更新なし)
BEFORE INSERT	①実行	①実行	①実行	①実行
AFTER INSERT	②実行	-	-	-
BEFORE UPDATE	-	-	②実行	-
AFTER UPDATE	-	-	③実行	-

表 68 トリガーの起動 (ON EACH STATEMENT トリガー／数字は実行順)

トリガー	INSERT 成功	DO NOTHING	DO UPDATE (更新あり)	DO UPDATE (更新なし)
BEFORE INSERT	①実行	①実行	①実行	①実行
AFTER INSERT	④実行	②実行	④実行	④実行
BEFORE UPDATE	②実行	-	②実行	②実行
AFTER UPDATE	③実行	-	③実行	③実行

6.5.3 ON CONFLICT 句と実行計画

ON CONFLICT 句の部分が実行されることで、実行計画が変化します。EXPLAIN 文を実行すると、実行計画内に Conflict Resolution, Conflict Arbiter Indexes, Conflict Filter 等が表示されます。具体的な出力は以下の例の通りです。

例 172 ON CONFLICT 句と実行計画

```
postgres=> EXPLAIN INSERT INTO upsert1 VALUES (200, 'Update 1')
      ON CONFLICT(key) DO NOTHING ;
          QUERY PLAN
-----
Insert on upsert1  (cost=0.00..0.01 rows=1 width=0)
Conflict Resolution: NOTHING
Conflict Arbiter Indexes: pk_upsert1
-> Result  (cost=0.00..0.01 rows=1 width=0)
(4 rows)
postgres=> EXPLAIN INSERT INTO upsert1 VALUES (400, 'Upd4')
      ON CONFLICT(key) DO UPDATE SET val = EXCLUDED.val ;
          QUERY PLAN
-----
Insert on upsert1  (cost=0.00..0.01 rows=1 width=0)
Conflict Resolution: UPDATE
Conflict Arbiter Indexes: pk_upsert1
-> Result  (cost=0.00..0.01 rows=1 width=0)
(4 rows)
postgres=> EXPLAIN INSERT INTO upsert1 VALUES (400, 'Upd4')
      ON CONFLICT(key) DO UPDATE SET val = EXCLUDED.val WHERE upsert1.key = 100 ;
          QUERY PLAN
-----
Insert on upsert1  (cost=0.00..0.01 rows=1 width=0)
Conflict Resolution: UPDATE
Conflict Arbiter Indexes: pk_upsert1
Conflict Filter: (upsert1.key = '100'::numeric)
-> Result  (cost=0.00..0.01 rows=1 width=0)
(5 rows)
```

現在のバージョンでは、postgres_fdw モジュールを使ったリモート・インスタンスに対しては ON CONFLICT DO UPDATE 文はサポートされていません。

6.5.4 ON CONFLICT 句とパーティション・テーブル

INSERT トリガーを使ったパーティション・テーブルに対する ON CONFLICT 句は無

視されます。

例 173 パーティション・テーブルに対する INSERT ON CONFLICT(1)

```
postgres=> CREATE TABLE main1 (key1 NUMERIC, val1 VARCHAR(10)) ;
CREATE TABLE
postgres=> CREATE TABLE main1_part100 (CHECK(key1 < 100)) INHERITS (main1) ;
CREATE TABLE
postgres=> CREATE TABLE main1_part200 (CHECK(key1 >= 100 AND key1 < 200))
    INHERITS (main1) ;
CREATE TABLE
postgres=> ALTER TABLE main1_part100 ADD CONSTRAINT pk_main1_part100
    PRIMARY KEY (key1);
ALTER TABLE
postgres=> ALTER TABLE main1_part200 ADD CONSTRAINT pk_main1_part200
    PRIMARY KEY (key1);
ALTER TABLE
postgres=> CREATE OR REPLACE FUNCTION func_main1_insert()
    RETURNS TRIGGER AS $$%
BEGIN
    IF      (NEW.key1 < 100) THEN
        INSERT INTO main1_part100 VALUES (NEW.*);
    ELSIF (NEW.key1 >= 100 AND NEW.key1 < 200) THEN
        INSERT INTO main1_part200 VALUES (NEW.*);
    ELSE
        RAISE EXCEPTION 'ERROR! key1 out of range.' ;
    END IF ;
    RETURN NULL ;
END ;
$$ LANGUAGE 'plpgsql';
CREATE FUNCTION
```

例 174 パーティション・テーブルに対する INSERT ON CONFLICT(2)

```
postgres=> CREATE TRIGGER trg_main1_insert BEFORE INSERT ON main1
              FOR EACH ROW EXECUTE PROCEDURE func_main1_insert() ;
CREATE TRIGGER
postgres=> INSERT INTO main1 VALUES (100, 'DATA100') ;
INSERT 0 0
postgres=> INSERT INTO main1 VALUES (100, 'DATA100') ;
ERROR:  duplicate key value violates unique constraint "pk_main1_part200"
DETAIL:  Key (key1)=(100) already exists.
CONTEXT: SQL statement "INSERT INTO main1_part200 VALUES (NEW.*)"
PL/pgSQL function func_main1_insert() line 6 at SQL statement
postgres=> INSERT INTO main1 VALUES (100, 'DATA100')
          ON CONFLICT DO NOTHING ;
ERROR:  duplicate key value violates unique constraint "pk_main1_part200"
DETAIL:  Key (key1)=(100) already exists.
CONTEXT: SQL statement "INSERT INTO main1_part200 VALUES (NEW.*)"
PL/pgSQL function func_main1_insert() line 6 at SQL statement
```

6.6 TABLESAMPLE

6.6.1 概要

PostgreSQL 9.5 からテーブルから一定割合のレコードをサンプリングする TABLESAMPLE 句が利用できるようになりました。

構文 6 SELECT TABLESAMPLE 文

```
SELECT ... FROM table_name ...
  TABLESAMPLE {SYSTEM | BERNoulli} (percent)
  [ REPEATABLE (seed) ]
```

サンプリング方法として SYSTEM と BERNOUlli を指定できます。percent にはサンプリングを行うパーセンテージを指定します (0.0~100.0)。0.0~100.0 以外の値を指定すると SELECT 文はエラーになります。

REPEATABLE 句はオプションです。サンプリングにしようする乱数のシードを指定します。REPEATABLE 句を省略した場合には random(3) 関数で生成された乱数がしようされます。

6.6.2 SYSTEM と BERNOUlli

SYSTEM はサンプリングしたブロック全体のタプルを使用します。BERNOULLI はブロック内から一定割合のタプルを選択します。サンプリングを行うために、まずテーブル全体のレコード数を推計します。

テーブルのスキャン方法、可視レコードの確認方法はサンプリング方法とサンプリング割合により変化します。

□ SYSTEM

サンプリング割合が 1%を超える場合、一括読み込み (Bulk Read) が行われます。可視化のチェックはページ・モードで行われます。以下は該当部分のソースコードです。

例 175 system_beginsamplescan 関数(src/backend/access/tablesample/system.c)

```
/*
 * Bulkread buffer access strategy probably makes sense unless we're
 * scanning a very small fraction of the table. The 1% cutoff here is a
 * guess. We should use pagemode visibility checking, since we scan all
 * tuples on each selected page.
 */
node->use_bulkread = (percent >= 1);
node->use_pagemode = true;
```

□ BERNOULLI

常に一括読み込み (Bulk Read) が行われます。可視化のチェックはサンプリング割合が 25 パーセントを超える場合にページ・モードで行われます。以下は該当部分のソースコードです。

例 176 bernoulli_beginsamplescan 関数(src/backend/access/tablesample/202ernoulli.c)

```
/*
 * Use bulkread, since we're scanning all pages. But pagemode visibility
 * checking is a win only at larger sampling fractions. The 25% cutoff
 * here is based on very limited experimentation.
 */
node->use_bulkread = true;
node->use_pagemode = (percent >= 25);
```

以下の例では、同一構造のテーブルを作成し、読み込みブロック数を確認しています。テーブルのサイズはどちらも 5,000 ブロックです。

例 177 読み込みブロック数の確認

```
postgres=> SELECT COUNT(*) FROM samplesys TABLESAMPLE SYSTEM (10) ;
postgres=> SELECT COUNT(*) FROM sampleber TABLESAMPLE BERNOULLI (10) ;
postgres=> SELECT relname, heap_blkls_read FROM pg_statio_user_tables ;
   relname  | heap_blkls_read
-----+-----
samplesys |          533
sampleber |         4759
(2 rows)
```

6.6.3 実行計画

サンプリングを行った場合の実行計画は以下の通りとなります。BERNOULLI を指定するとコストが大きくなることがわかります。

例 178 サンプリング時の実行計画 (TABLESAMPLE SYSTEM)

```
postgres=> EXPLAIN ANALYZE SELECT COUNT(*) FROM data1 TABLESAMPLE SYSTEM (10) ;
               QUERY PLAN
-----
Aggregate  (cost=341.00..341.01 rows=1 width=0) (actual time=4.914..4.915 rows=1
loops=1)
    ->  Sample Scan (system) on data1  (cost=0.00..316.00 rows=10000 width=0)
        (actualtime=0.019..3.205 rows=10090 loops=1)
Planning time: 0.106 ms
Execution time: 4.977 ms
(4 rows)
```

例 179 サンプリング時の実行計画 (TABLESAMPLE BERNOULLI)

```
postgres=> EXPLAIN ANALYZE SELECT COUNT(*) FROM data1 TABLESAMPLE
          BERNOULLI (10) ;
          QUERY PLAN
-----
Aggregate  (cost=666.00..666.01  rows=1  width=0)  (actual  time=13.654..13.655
rows=1  loops=1)
->  Sample Scan (beroulli) on data1  (cost=0.00..641.00  rows=10000  width=0)
(actual  time=0.013..12.121  rows=10003  loops=1)
Planning time: 0.195 ms
Execution time: 13.730 ms
```

6.7 テーブルの属性変更

テーブル定義は ALTER TABLE 文で変更できます。ここでは ALTER TABLE 文による影響について記載しています。

6.7.1 ALTER TABLE SET UNLOGGED

ALTER TABLE SET LOGGED 文または ALTER TABLE SET UNLOGGED 文を実行することで、通常のテーブルと UNLOGGED テーブルを切り替えることができます。

例 180 UNLOGGED TABLE への切り替え

```
postgres=> CREATE TABLE logtbl1 (c1 NUMERIC, c2 VARCHAR(10)) ;
CREATE TABLE
postgres=> \d+ logtbl1
                                         Table "public.logtbl1"
  Column |          Type           | Modifiers | Storage  | Stats target | Description
-----+---------------------+-----+-----+-----+
  c1    | numeric            |          | main    |             |
  c2    | character varying(10) |          | extended |             |

postgres=> ALTER TABLE logtbl1 SET UNLOGGED ;
ALTER TABLE
postgres=> \d+ logtbl1
                                         Unlogged table "public.logtbl1"
  Column |          Type           | Modifiers | Storage  | Stats target | Description
-----+-----+-----+-----+-----+
  c1    | numeric            |          | main    |             |
  c2    | character varying(10) |          | extended |             |
```

psql コマンドの \d+ コマンドにより、通常のテーブルが UNLOGGED テーブルに変更されたことがわかります。内部的には、同一構造を持つ新規の UNLOGGED TABLE (または TABLE) を作成し、データのコピーを行っています。pg_class カタログの relfilenode 列と relopersistence 列が変更されます。

例 181 pg_class カタログの relfilenode 列の変化

```
postgres=> SELECT relname, relfilenode, relpersistence FROM pg_class WHERE
      relname='logtbl1' ;
      relname | relfilenode | relpersistence
-----+-----+
      logtbl1 |      16483 | p
(1 row)

postgres=> ALTER TABLE logtbl1 SET UNLOGGED ;
ALTER TABLE
postgres=> SELECT relname, relfilenode, relpersistence FROM pg_class WHERE
      relname='logtbl1' ;
      relname | relfilenode | relpersistence
-----+-----+
      logtbl1 |      16489 | u
(1 row)
```

□ レプリケーション環境における切り替え

ストリーミング・レプリケーション環境のスレーブ・インスタンスでは UNLOGGED TABLE にはアクセスできません。このため LOGGED TABLE から UNLOGGED TABLE に変更したテーブルは、スレーブ・インスタンスからはアクセスできなくなります。

例 182 スレーブ・インスタンスで UNLOGGED TABLE の参照

MASTER

```
postgres=> ALTER TABLE logtest1 SET UNLOGGED ;
ALTER TABLE
```

SLAVE

```
postgres=> SELECT * FROM logtest1 ;
ERROR:  cannot access temporary or unlogged relations during recovery
```

一方で、UNLOGGED TABLE から LOGGED TABLE への変換では WAL が出力されるため、LOGGED TABLE に変換したテーブルの内容はスレーブ・インスタンスでアクセス

可能になります。

例 183 スレーブ・インスタンスで TABLE の参照

MASTER

```
postgres=> CREATE UNLOGGED TABLE logtest1(c1 NUMERIC) ;  
CREATE TABLE  
postgres=> INSERT INTO logtest1 VALUES (generate_series(1, 10000)) ;  
INSERT 0 10000  
postgres=> ALTER TABLE logtest1 SET LOGGED ;  
ALTER TABLE
```

SLAVE

```
postgres=> SELECT COUNT(*) FROM logtest1 ;  
count  
-----  
10000  
(1 row)
```

6.7.2 ALTER TABLE SET WITH OIDS

ALTER TABLE SET WITH OIDS 文または ALTER TABLE SET WITHOUT OIDS 文を実行すると、属性が異なる一時テーブルが作成され、データがコピーされます。このためテーブルを構成するファイル名が変更されます。またテーブルに付与されたインデックスも再作成されます。

例 184 ALTER TABLE SET WITH OIDS 実行時のファイル

```
postgres=> SELECT relname, relfilenode FROM pg_class WHERE relname = 'data1' ;
      relname | relfilenode
-----+-----
 data1    |      16468
(1 row)

postgres=> ALTER TABLE data3 SET WITH OIDS ;
ALTER TABLE
postgres=> SELECT relname, relfilenode FROM pg_class WHERE relname = 'data1' ;
      relname | relfilenode
-----+-----
 data1    |      17489
(1 row)
```

6.7.3 ALTER TABLE MODIFY COLUMN TYPE

ALTER TABLE ALTER COLUMN TYPE 文を実行すると列のデータ型を変更することができます。ALTER TABLE 文の実行により列データのサイズを縮小する場合はテーブルが再作成されます。このためテーブルを構成するファイル名が変更され、テーブルに付与された全てのインデックスも再作成されます。以下はパラメーター log_min_messages を debug5 に設定し、テーブル data1 列の c2 列を NUMERIC 型から NUMERIC(10)型に変更した場合のログです。インデックス idx1_data1 が再作成されていることがわかります。

例 185 ALTER TABLE ALTER COLUMN TYPE 文実行時のログ

```
DEBUG: StartTransactionCommand
DEBUG: StartTransaction
DEBUG: name: unnamed; blockState:      DEFAULT; state: INPROGR, xid/subid/cid:
0/1/0, nestlvl: 1, children:
DEBUG: ProcessUtility
DEBUG: EventTriggerTableRewrite(16415)
DEBUG: building index "pg_toast_16447_index" on table "pg_toast_16447"
DEBUG: rewriting table "data1"
DEBUG: building index "idx1_data1" on table "data1"
DEBUG: drop auto-cascades to type pg_temp_16415
DEBUG: drop auto-cascades to type pg_temp_16415[]
DEBUG: drop auto-cascades to toast table pg_toast.pg_toast_16415
DEBUG: drop auto-cascades to index pg_toast.pg_toast_16415_index
DEBUG: drop auto-cascades to type pg_toast.pg_toast_16415
DEBUG: CommitTransactionCommand
DEBUG: CommitTransaction
```

6.8 ECPG

ECPG は C 言語プログラム内に SQL 文を直接記述するためのプリプロセッサです。ここでは SQL 文の実行結果を取得するホスト変数について検証しています。

6.8.1 ホスト変数のフォーマット

文字列データをホスト変数で取得した場合の出力フォーマットについて検証しました。

□ CHAR 型列値

テーブルの CHAR 型列を char 型の配列に書き込みを行いました。CHAR(5)列型の列に'ABC'を格納し、C 言語変数 char[7]に格納した場合のフォーマットを検証しています。SP はスペース (0x20)、NL は NULL (0x00)、OR は変更されていないことを示します。

表 69 列型 CHAR(5) → ホスト変数 char[7]

配列	char[0]	char[1]	char[2]	char[3]	char[4]	char[5]	char[6]
格納データ	A	B	C	SP	SP	NL	OR

文字列の後にスペースが付与され、最後に NULL が格納されることがわかります。

□ VARCHAR 型列値

テーブルの VARCHAR 型列を char 配列に書き込みを行いました。VARCHAR(5)列型の列に'ABC'を格納し、C 言語変数 char[7]に格納した場合のフォーマットを検証しています。

表 70 列型 VARCHAR(5) → ホスト変数 char[7]

配列	char[0]	char[1]	char[2]	char[3]	char[4]	char[5]	char[6]
格納データ	A	B	C	NL	OR	OR	OR

文字列の後 NULL が格納され、NULL 以降の値は変更されていないことがわかります。

□ VARCHAR 型ホスト変数

ホスト変数に VARCHAR 型を指定することができます。この変数は ecpg を通すと、メンバー int len と char arr[99]を持つ構造体に変換されます。

表 71 列型 VARCHAR(5) → ホスト変数 VARCHAR(7)

配列	arr[0]	arr[1]	arr[2]	arr[3]	arr[4]	arr[5]	arr[6]
格納データ	A	B	C	NL	NL	NL	NL

データが格納される `char arr` 配列はすべて `NULL` (`0x00`)で初期化されていることがわかります。

6.8.2 領域不足時の動作

文字列型列の値をホスト変数に出力する場合、ホスト変数の領域が不足していると結果は切り詰められ、指示子 (Indicator) に正の値が書き込まれます。マニュアルには以下のように示されており、具体的な動作についての記述がありません。

例 186 指示子のマニュアル

This second host variable is called the indicator and contains a flag that tells whether the datum is null, in which case the value of the real host variable is ignored.

□ 領域不足時のホスト変数 1

テーブルの `VARCHAR` 型列を `char` 配列に書き込みを行いますが、`char` 型の領域が明らかに不足している場合のフォーマットを検証しました。`VARCHAR(5)` 列型の列に'ABCDE'を格納し、C 言語変数 `char[3]` に格納した場合のフォーマットを検証しています。

表 72 列型 `VARCHAR(5)` → ホスト変数 `char[3]`

配列	<code>char[0]</code>	<code>char[1]</code>	<code>char[2]</code>
格納データ	A	B	C

上記のように格納できる部分まで格納されますが、`NULL` 値は付与されません。指示子には 5 が格納されます。

□ 領域不足時のホスト変数 2

`VARCHAR` 型列を `char` 配列に書き込みを行いますが、`char` 型の領域が `NULL` 値分のみ不足している場合のフォーマットを検証しました。`VARCHAR(5)` 列型の列に'ABCDE'を格納し、C 言語変数 `char[5]` に格納した場合のフォーマットを検証しています。

表 73 列型 `VARCHAR(5)` → ホスト変数 `char[5]`

配列	<code>char[0]</code>	<code>char[1]</code>	<code>char[2]</code>	<code>char[3]</code>	<code>char[4]</code>
格納データ	A	B	C	D	E

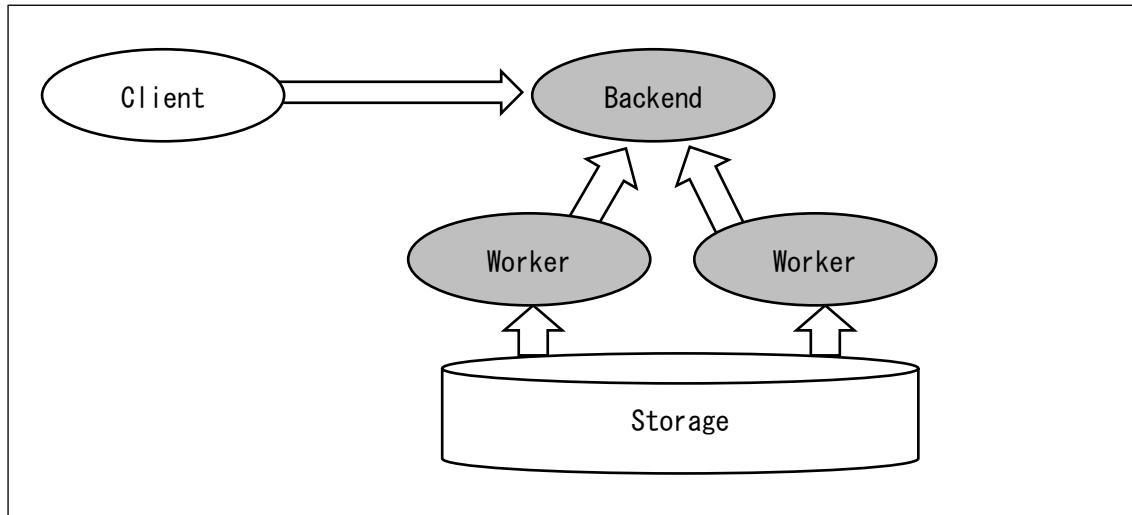
上記のようにデータベース格納文字はそのまま保存されますが、NULL 値は付与されません。指示子には 0 が指定されます。このため NULL 用の領域が不足していても指示子では判定できないことがわかります。

6.9 Parallel Query

6.9.1 概要

従来の PostgreSQL では、SQL 文の実行は接続を受け付けたバックエンド・プロセスがすべて実行していました。PostgreSQL では複数のワーカー・プロセスによる並列処理を行うことができるようになりました。

図 17 Parallel Seq Scan / Parallel Aggregate



並列処理が行われるのは Seq Scan および Aggregate のみで、並列数はテーブルのサイズに依存します。並列処理を行うプロセスは Background Worker の仕組みを利用します。並列数の最大値はパラメーター `max_parallel_workers_per_gather` または `max_worker_processes` のいずれか小さい方で決定されます。パラメーター `max_parallel_workers_per_gather` は一般ユーザーがセッション単位に変更できます。

表 74 関連するパラメーター

パラメータ名	説明 (context)	デフォルト値
max_parallel_workers_per_gather	実行計画作成時の並列度の最大値 (user)	0
parallel_setup_cost	並列処理の開始コスト (user)	1000
parallel_tuple_cost	並列処理のタプル処理コスト (user)	0.1
max_worker_processes	ワーカー・プロセスの最大値 (postmaster)	8
force_parallel_mode	並列処理を強制 (user)	off
min_parallel_relation_size	並列処理を検討する最小テーブル・サイズ	8MB

□ パラメーター force_parallel_mode

並列処理は通常のシリアル処理よりもコストが低いとみなされる場合にのみ実行されます。パラメーター force_parallel_mode を on に指定すると、並列処理を強制することができます（同様の動作で再帰テスト用に設定値 regress を指定することができます）。ただし並列処理が行われるのはパラメーター max_parallel_workers_per_gather が 1 以上の場合に限ります。

□ テーブル・オプション

テーブル・オプション parallel_workers は並列度をテーブル単位に決定することができます。設定値を 0 に指定すると並列処理が禁止されます。設定されない場合はセッションのパラメーター max_parallel_workers_per_gather が上限になります。

parallel_workers の設定値を大きくしても実際の並列度の上限は max_parallel_workers_per_gather を超えることはできません。

例 187 テーブル・オプション parallel_degree の設定

```
postgres=> ALTER TABLE data1 SET (parallel_degree = 2) ;
ALTER TABLE
postgres=> \d+ data1
              Table "public.data1"
   Column |          Type          | Modifiers | Storage | Stats target | Description
-----+----------------+-----+-----+-----+-----+
    c1   | numeric          |           | main   |             |
    c2   | character varying(10) |           | extended |             |
Options: parallel_workers=5
```

6.9.2 実行計画

下記は並列処理を行う SELECT 文の実行計画出力例です。大規模なテーブルの COUNT 处理を 3 並列で処理しています。

例 188 並列処理の実行計画

```
postgres=> SET max_parallel_degree = 10 ;
SET
postgres=> EXPLAIN (ANALYZE, VERBOSE) SELECT COUNT(*) FROM data1 ;
                                QUERY PLAN
-----
Finalize Aggregate  (cost=29314.09..29314.10 rows=1 width=8)
    (actual time=662.055..662.055 rows=1 loops=1)
        Output: pg_catalog.count(*)
-> Gather  (cost=29313.77..29314.08 rows=3 width=8)
    (actual time=654.818..662.043 rows=4 loops=1)
        Output: (count(*))
Workers Planned: 3
Workers Launched: 3
-> Partial Aggregate  (cost=28313.77..28313.78 rows=1 width=8)
    (actual time=640.330..640.331 rows=1 loops=4)
        Output: count(*)
Worker 0: actual time=643.386..643.386 rows=1 loops=1
Worker 1: actual time=645.587..645.588 rows=1 loops=1
Worker 2: actual time=618.493..618.494 rows=1 loops=1
-> Parallel Seq Scan on public.data1  (cost=0.00..25894.42
rows=967742 width=0) (actual time=0.033..337.848 rows=750000 loops=4)
    Output: c1, c2
Worker 0: actual time=0.037..295.732 rows=652865 loops=1
Worker 1: actual time=0.026..415.235 rows=772230 loops=1
Worker 2: actual time=0.042..359.622 rows=620305 loops=1
Planning time: 0.130 ms
Execution time: 706.955 ms
(18 rows)
```

並列処理を実行するにあたり、EXPLAIN 文を実行すると以下のような実行計画が表示される可能性があります。

表 75 EXPLAIN 文の出力

実行計画	説明	出力される構文
Parallel Seq Scan	並列検索処理	全て
Partial Aggregate	ワーカー・プロセスが行う集計処理	全て
Partial HashAggregate		
Partial GroupAggregate		
Gather	ワーカー・プロセスを集約する処理	全て
Finalize Aggregate	最終的な集約処理	全て
Workers Planned:	計画されたワーカー数	全て
Workers Launched:	実際に実行されたワーカー数	ANALYZE
Worker N (N=0,1,...)	各ワーカーの処理時間等	ANALYZE, VERBOSE
Single Copy	単一プロセスで実行する処理	すべて

6.9.3 並列処理と関数

並列処理実行時に使用できる関数と使用できない関数があります。pg_proc カタログの proparallel 列が'u'になっている関数 (PARALLEL UNSAFE) が SQL 文内に使用されていると、並列処理は行われません。以下の表は主な PARALLEL UNSAFE 標準関数です。

表 76 主な PARALLEL UNSAFE 標準関数

分類	関数例
シーケンス関連	nextval, currval, setval, lastval
Large Object 関連	lo_*, loread, lowrite
レプリケーション関連	pg_create_*_slot, pg_drop_*_slot, pg_logical_*, pg_replication_*
その他	make_interval, parse_ident, pg_extension_config_dump, pg_*_backup, set_config, ts_debug, txid_current, query_to_xml*

下記の例では、WHERE 句の条件部分のみ異なる 2 つの SQL 文を実行しています。WHERE 句にリテラルを指定した SELECT 文は並列処理が行われますが、シーケンス操作関数の currval を指定した SELECT 文はシリアルに実行されます。

例 189 PARALLEL UNSAFE 関数の使用による実行計画の違い

```
postgres=> EXPLAIN SELECT COUNT(*) FROM data1 WHERE c1=10 ;
               QUERY PLAN
-----
Aggregate  (cost=29314.08..29314.09 rows=1 width=8)
->  Gather  (cost=1000.00..29314.07 rows=3 width=0)
    Workers Planned: 3
        ->  Parallel Seq Scan on data1  (cost=0.00..28313.78 rows=1 width=0)
            Filter: (c1 = '10'::numeric)
(5 rows)

postgres=> EXPLAIN SELECT COUNT(*) FROM data1 WHERE c1=currval('seq1') ;
               QUERY PLAN
-----
Aggregate  (cost=68717.01..68717.02 rows=1 width=8)
->  Seq Scan on data1  (cost=0.00..68717.00 rows=3 width=0)
    Filter: (c1 = (currval('seq1'::regclass))::numeric)
(3 rows)
```

pg_proc カタログの proparallel 列が'r'になっている関数は、パラレル処理のリーダーとなるプロセスでのみ実行可能です。

表 77 主な RESTRICTED PARALLEL SAFE 標準関数

分類	関数例
日付関連	age, now
乱数関連	random, setseed
アップグレード関連	binary_upgrade*
XML 関連	cursor_to_xml*, database_to_xml*, schema_to_xml*, table_to_xml*
その他	pg_start_backup, inet_client*, current_query, pg_backend_pid, pg_conf*, pg_cursor, pg_get_viewdef, pg_prepared_statement, etc

□ ユーザー一定義関数と PARALLEL SAFE

ユーザー一定義関数に対して並列処理を行うことができるかどうかを示すため、CREATE FUNCTION 文または ALTER FUNCTION 文に PARALLEL SAFE 句または PARALLEL UNSAFE 句を使用することができます。デフォルトは PARALLEL UNSAFE です。

例 190 ユーザー一定義関数と PARALLEL SAFE の定義

```
postgres=> CREATE FUNCTION add(integer, integer) RETURNS integer
postgres->   AS 'select $1 + $2;'
postgres->   LANGUAGE SQL IMMUTABLE RETURNS NULL ON NULL INPUT ;
CREATE FUNCTION
postgres=> SELECT proname, proparallel FROM pg_proc WHERE proname = 'add' ;
proname | proparallel
-----+-----
add     | u
(1 row)
postgres=> ALTER FUNCTION add(integer, integer) PARALLEL SAFE ;
ALTER FUNCTION
postgres=> SELECT proname, proparallel FROM pg_proc WHERE proname='add' ;
proname | proparallel
-----+-----
add     | s
(1 row)
```

ユーザー一定義関数が Parallel Safe なのか Parallel Unsafe なのかの決定は関数作成者の責任です。Parallel Safe 指定のユーザー一定義関数内で Parallel Unsafe 関数が呼ばれても、オプティマイザはパラレル・クエリーの実行計画を作成する可能性があります。nextval 関数等、一部の標準関数はパラレル・クエリーが行われていることを検知するとエラーを発生させます。

下記の例では、Parallel Unsafe 関数 unsafe1 を作成しています。また Parallel Unsafe 関数を実行する Parallel Safe 関数 safe1 を作成しています。

例 191 Parallel Unsafe 関数を実行する Parallel Safe 関数

```
postgres=> CREATE FUNCTION unsafe1() RETURNS numeric AS $$  
postgres$> BEGIN RETURN 100; END;  
postgres$> $$ PARALLEL UNSAFE LANGUAGE plpgsql ;  
CREATE FUNCTION  
postgres=>  
postgres=> CREATE FUNCTION safe1() RETURNS numeric AS $$  
BEGIN RETURN unsafe1(); END;  
$$ PARALLEL SAFE LANGUAGE plpgsql ;  
CREATE FUNCTION  
postgres=>  
postgres=> SELECT proname, proparallel FROM pg_proc WHERE proname like  
'%safe%' ;  
proname | proparallel  
-----+-----  
unsafe1 | u  
unsafe1insafe | s  
(2 rows)
```

下記の例では Parallel Safe 関数 safe1 を含む SELECT 文を実行すると、パラレル・クエリーが実行されることがわかります。

例 192 Parallel Unsafe 関数を実行する Parallel Safe 関数

```
postgres=> EXPLAIN ANALYZE VERBOSE SELECT * FROM data1 WHERE c1 = safe1() ;
                                         QUERY PLAN
-----
Gather  (cost=1000.00..115781.10 rows=1 width=11)
    (actual time=1.354..2890.075 rows=1 loops=1)
        Output: c1, c2
        Workers Planned: 2
        Workers Launched: 2
        ->  Parallel Seq Scan on public.data1   (cost=0.00..114781.00 rows=0
width=11) (actual time=1904.255..2866.980 rows=0 loops=3)
            Output: c1, c2
            Filter: (data1.c1 = unsafe1insafe())
            Rows Removed by Filter: 333333
            Worker 0: actual time=2844.205..2844.205 rows=0 loops=1
            Worker 1: actual time=2867.581..2867.581 rows=0 loops=1
Planning time: 0.083 ms
Execution time: 2890.790 ms
(12 rows)
```

6.9.4 並列度の計算

並列度は検索対象のテーブルのサイズを元に計算されます。テーブルのサイズがパラメーター `min_parallel_relation_size` テーブルを基準に、3倍ごとに並列度を増加させます。並列度の上限はパラメーター `max_parallel_workers_per_gather` またはパラメーター `max_worker_processes` を超えない範囲で決定されます。実際の計算処理はソースコード `src/backend/optimizer/path/allpaths.c` 内の `create_plain_partial_paths` 関数内に記述されています。

例 193 `create_plain_partial_paths` 関数の一部

```
int parallel_threshold;

/*
 * If this relation is too small to be worth a parallel scan, just
 * return without doing anything ... unless it's an inheritance child.
 * In that case, we want to generate a parallel path here anyway. It
 * might not be worthwhile just for this relation, but when combined
 * with all of its inheritance siblings it may well pay off.
 */
if (rel->pages < (BlockNumber) min_parallel_relation_size &&
    rel->reloptkind == RELOPT_BASEREL)
    return;

/*
 * Select the number of workers based on the log of the size of the
 * relation. This probably needs to be a good deal more
 * sophisticated, but we need something here for now. Note that the
 * upper limit of the min_parallel_relation_size GUC is chosen to
 * prevent overflow here.
 */
parallel_workers = 1;
parallel_threshold = Max(min_parallel_relation_size, 1);
while (rel->pages >= (BlockNumber) (parallel_threshold * 3))
{
    parallel_workers++;
    parallel_threshold *= 3;
    if (parallel_threshold > INT_MAX / 3)
        break; /* avoid overflow */
}
```

7. 権限とオブジェクト作成

7.1 オブジェクト権限

PostgreSQL のオブジェクト権限について、オブジェクトの所有者とは別の一般ユーザー (login 権限のみ) が実行可能な操作をまとめました。

7.1.1 テーブル空間の所有者

接続ユーザーとテーブル空間の所有者が異なる場合、オブジェクトが作成できるか検証しました。以下の結果から、テーブル空間に接続ユーザーが所有者となるデータベースが作成されている場合にはオブジェクトが作成できることがわかりました。

表 78 テーブル空間の所有者と接続ユーザーが異なる場合

操作	データベースなし	データベースあり
CREATE TABLE	×	○
CREATE INDEX	×	○

7.1.2 データベースの所有者

データベースの所有者と接続ユーザーが異なる場合、public スキーマにオブジェクトが作成できるかを検証しました。以下のようにスキーマ以外の主なオブジェクトはデータベースの所有者が異なっても作成可能です。public スキーマに対するアクセスを禁止したい場合には public ロールから一旦すべてのアクセス権限を剥奪し、必要な権限のみを該当ユーザーに追加する必要があります。

表 79 pg_default テーブル空間 (postgres ユーザー所有) に対するオブジェクト作成

操作	実行可否
CREATE TABLE	○
CREATE INDEX	○
CREATE SEQUENCE	○
CREATE SCHEMA	×
CREATE FUNCTION	○
CREATE TYPE	○

7.2 Row Level Security

7.2.1 Row Level Security とは

PostgreSQL では、テーブルや列に対してアクセスする権限を GRANT 文で指定しました。PostgreSQL 9.5 には従来の方法に加えて Row Level Security の機能が追加されました。Row Level Security は、GRANT 文で許可したレコードを更にレコード（タプル）レベルで制限することができる機能です。Row Level Security によるアクセス制限には POLICY と呼ばれるオブジェクトを作成します。

図 18 Row Level Security によるアクセス制限



7.2.2 準備

Row Level Security を利用するためには、ポリシーによる制限を行うテーブルに対して ALTER TABLE ENABLE ROW LEVEL SECURITY 文を実行します。標準ではテーブルに対する Row Level Security 設定は有効になっていません。テーブルに対する設定を無効にするには ALTER TABLE DISABLE ROW LEVEL SECURITY 文を実行します。

例 194 テーブルに対する機能の有効化

```
postgres=> ALTER TABLE poltbl1 ENABLE ROW LEVEL SECURITY ;
ALTER TABLE
postgres=> \d+ poltbl1
          Table "public.poltbl1"
  Column |      Type       | Modifiers | Storage | Stats target | Description
-----+---------------+-----+-----+-----+-----+
  c1    | numeric        |           | main   |           |
  c2    | character varying(10) |           | extended |           |
  uname | character varying(10) |           | extended |           |
Policies (Row Security Enabled): (None)
```

7.2.3 ポリシーの作成

テーブルに対してアクセス権限を指定するにはポリシーを作成します。ポリシーは CREATE POLICY 文で作成します。POLICY の作成は一般ユーザーでも行うことができます。

構文 7 CREATE POLICY 文

```
CREATE POLICY policy_name ON table_name
[ FOR { ALL | SELECT | INSERT | UPDATE | DELETE } ]
[ TO { roles | PUBLIC, ... } ]
USING (condition)
[ WITH CHECK (check_condition) ]
```

表 80 CREATE POLICY 文の構文

構文	説明
<i>policy_name</i>	ポリシーの名前を指定
ON	ポリシーを適用するテーブル名を指定
FOR	ポリシーを適用する操作または ALL
TO	ポリシーを許可する対象ロール名または PUBLIC
USING	タプルに対する許可を行う条件文 (WHERE 句と同一構文) を記述します。USING 句で指定された条件が TRUE になるタプルのみが利用者に返されます。
WITH CHECK	UPDATE 文により更新できる条件を記述します。SELECT 文に対するポリシーでは CHECK 句は指定できません。

下記の例では、テーブル poltbl1 に対するポリシーを作成しています。TO 句を省略しているため、対象は全ユーザー (PUBLIC)、操作はすべての SQL (FOR ALL)、許可を行うタプルは uname 列が現在のユーザー名 (current_user 関数) と同じタプルのみになります。

例 195 CREATE POLICY 設定

```
postgres=> CREATE POLICY pol1 ON poltbl1 FOR ALL USING (uname = current_user) ;
CREATE POLICY
postgres=> \d+ poltbl1
                                         Table "public.poltbl1"
 Column |          Type           | Modifiers | Storage | Stats target | Description
-----+---------------------+-----+-----+-----+
 c1    | numeric            |          | main   |             |
 c2    | character varying(10) |          | extended |             |
 uname | character varying(10) |          | extended |             |
 Policies:
 POLICY "pol1" FOR ALL
 USING ((uname)::name = "current_user"())
```

作成したポリシーは pg_policy カタログから確認することができます。またポリシーを設定されたテーブルの情報は pg_policies カタログから確認できます。

以下の例では、ポリシーの効果を検証しています。

- テーブル poltbl1 のオーナーである user1 ユーザーが 3 レコードを格納しています (2 ~12 行)。
- ユーザー user2 の権限でテーブル tblpol1 を検索していますが、uname 列の値が user2 であるレコード 1 件のみしか参照できません (15~19 行)。
- uname 列の値を変更しようとしていますが、CREATE POLICY1 文の USING 句で指定した条件から逸脱するため、UPDATE 文が失敗しています (20~21 行)。

例 196 ポリシーの効果

```
1 $ psql -U user1
2 postgres=> INSERT INTO tblpol1 VALUES (100, 'Val100', 'user1') ;
3 INSERT 0 1
4 postgres=> INSERT INTO tblpol1 VALUES (200, 'Val200', 'user2') ;
5 INSERT 0 1
6 postgres=> INSERT INTO tblpol1 VALUES (300, 'Val300', 'user3') ;
7 INSERT 0 1
8 postgres=> SELECT COUNT(*) FROM tblpol1 ;
9 count
10 -----
11      3
12 (1 row)
13
14 $ psql -U user2
15 postgres=> SELECT * FROM poltbl1 ;
16   c1  |   c2  |  uname
17   -----+-----+
18 200 | val200 | user2
19 (1 row)
20 postgres=> UPDATE poltbl1 SET uname='user3' ;
21 ERROR: new row violates row level security policy for "poltbl1"
```

ポリシーを作成する CREATE POLICY 文は、ENABLE ROW LEVEL SECURITY 句を指定されていないテーブルに対して実行してもエラーにはなりません。この場合、該当テーブルに対しては ROW LEVEL SECURITY 機能は有効にならず、GRANT 文による許可のみが有効になります。

ポリシー設定の変更や削除はそれぞれ ALTER POLICY 文、DROP POLICY 文で行います。テーブルに対して複数のポリシーを指定した場合には、論理和（OR）に合致したレコードが取得されます。

例 197 複数ポリシーの効果

```
1 $ psql -U user1
2 postgres=> CREATE TABLE pol1(c1 NUMERIC, c2 NUMERIC) ;
3 CREATE TABLE
4 postgres=> ALTER TABLE pol1 ENABLE ROW LEVEL SECURITY ;
5 ALTER TABLE
6 postgres=> CREATE POLICY p1_pol1 ON pol1 FOR ALL USING (c1 = 100) ;
7 CREATE POLICY
8 postgres=> CREATE POLICY p2_pol1 ON pol1 FOR ALL USING (c2 = 100) ;
9 CREATE POLICY
10 postgres=> GRANT SELECT ON pol1 TO user2 ;
11 GRANT
12
13 $ psql -U user2
14 postgres=> EXPLAIN SELECT * FROM pol1 ;
15
16
17 Seq Scan on pol1  (cost=0.00..23.20 rows=9 width=64)
18   Filter: ((c2 = '100'::numeric) OR (c1 = '100'::numeric))
19 (2 rows)
```

7.2.4 パラメーターの設定

Row Level Security の機能はパラメーター row_security で制御されます。以下の設定値をとることができます。

表 81 パラメーターrow_security

パラメーター値	説明
on	Row Level Security の機能を有効にします。この値はデフォルト値です。
off	Row Level Security の機能を無効にします。ポリシーによるアクセス制御が無効になるため、GRANT 文による許可のみが有効になります。
force	Row Level Security の機能を強制します。ポリシーを設定されたテーブルに対してはポリシーの許可が強制されます。このためテーブル所有者でもポリシー違反のデータにはアクセスできなくなります。

□ ユーザー権限

ユーザーの権限 BYPASSRLS を持っているスーパーユーザーは、ポリシーの設定をバイパスすることができます。BYPASSRLS 権限のみ持っている一般ユーザーはポリシーを無視することはできません。CREATE USER 文のデフォルトは NOBYPASSRLS が指定されます。

8. ユーティリティ

8.1 ユーティリティ使用方法

特徴的なコマンドの使用方法について説明します。

8.1.1 pg_basebackup コマンド

pg_basebackup コマンド¹⁴は、データベース・クラスターの完全なコピーを作成するため開発されました。使用には以下の点に注意します。内部的にはオンライン・バックアップと同じ処理を実施しています。

- -x オプションを指定して、バックアップ完了後にログスイッチを実施します。
- データベース・クラスター以外のテーブル空間ディレクトリは同一パスに保存されます。パスを変更するためには—tablespace-mapping パラメーターで新旧のパスを指定する必要があります (PostgreSQL 9.4 から指定可能)。
- バックアップ先のディレクトリは空にしておく必要があります。
- WAL 書込みディレクトリは{PGDATA}/pg_xlog になります。異なるディレクトリを指定する場合は—xlogdir パラメーターを指定します (PostgreSQL 9.4 から指定可能)。

例 198 pg_basebackup コマンド実行

```
$ pg_basebackup -D back -h hostsrc1 -p 5432 -x -v
Password: {PASSWORD}
transaction log start point: 0/7E000020
transaction log end point: 0/7E0000A8
pg_basebackup: base backup completed
$
```

□ データ転送動作

pg_basebackup コマンドの処理は、大部分は接続先インスタンス上の wal sender プロセス内で実行されます。wal sender プロセスは以下の処理を行っています。

- pg_start_backup 関数の実行
- データベース・クラスター内のファイルの転送
- 外部表スペースの実体検索とファイルの転送

¹⁴ マニュアルは <https://www.postgresql.org/docs/9.6/static/app-pgbasebackup.html>

- pg_stop_backup 関数の実行
- WAL 転送設定が行われていた場合は、WAL ファイルの転送

バックアップ元となるファイルのデータは 32 KB¹⁵単位で読み込まれ、クライアントに送信されます。

□ コピーされるファイル

`pg_basebackup` コマンドはデータベース・クラスターのコピーを行いますが、すべてのファイルをコピーするわけではありません。以下のファイル（およびディレクトリ）はコピー元のデータベース・クラスターとは異なります。

表 82 `pg_basebackup` コマンドとコピー元の差分

ファイル/ディレクトリ	差分	備考
backup_label	新規作成	
backup_label.old	新規作成	旧 backup_label ファイル
pg_replslot	コピーされない	レプリケーション・スロット
pg_stat_tmp	コピーされない	
pg_xlog	コピーされない	-x が指定されていない場合
	一部がコピーされない	-x が指定されている場合
postmaster.opts	コピーされない	
postmaster.pid	コピーされない	
外部表領域内の PostgreSQL 管理外ファイル	コピーされない	

□ 転送量の制限

`pg_basebackup` コマンド実行時に—max-rate パラメーターを指定すると、時間当たりのネットワーク転送量を制限できます（PostgreSQL 9.4 新機能）。転送量の制御は wal sender プロセス内で実行されます。データ転送量がパラメーターで指定されたバイト数の 1 / 8 を超える度にラッチのタイムアウトによる待機を行い、一定時間内のデータ転送量を抑制します。

□ recovery.conf ファイル

-R パラメーターを指定すると、recovery.conf ファイルを自動的に作成します。作成された recovery.conf ファイルには standby_mode='on' と primary_conninfo パラメーターのみが含まれます。バックアップ元のデータベース・クラスター内に既に recovery.conf ファイル

¹⁵ src/backend/replication/basebackup.c の TAR_SEND_SIZE で定義

が存在する場合（スレーブ・インスタンスからバックアップを取得する場合）でも、-R パラメーターを指定すると新規の recovery.conf ファイルを作成します。-R パラメーターを指定しない場合、既存の recovery.conf ファイルがバックアップ先ディレクトリにコピーされます。

□ Replication Slot のコピー

pg_basebackup コマンドでは pg_create_physical_replication_slot 関数で作成したスロット情報はコピーされません。このためコマンド終了時点ではコピー先の {PGDATA}/pg_replslot ディレクトリには空になります。

□ ユーザー作成ファイルとディレクトリ

データベース・クラスター内に PostgreSQL とは関係ないディレクトリおよびファイルを作成した場合の動作を検証しました。

例 199 ユーザー作成ファイルのコピー（データベース・クラスター）

```
$ touch data/test_file.txt
$ mkdir data/test_dir/
$ touch data/test_dir/test_file.txt
$ pg_basebackup -D back -p 5432 -v -x
transaction log start point: 0/2000028 on timeline 1
transaction log end point: 0/20000C0
pg_basebackup: base backup completed
$ ls back/test*
back/test_file.txt

back/test_dir:
test_file.txt
```

データベース・クラスターに作成された PostgreSQL が管理しないディレクトリおよびファイルは pg_basebackup コマンドによってバックアップされることを確認しました。次に CREATE TABLESPACE 文で作成されたテーブル空間内に作成されたディレクトリ／ファイルがコピーされるかを確認しました。

例 200 ユーザー作成ファイルのコピー（表スペース）

```
$ mkdir ts1
postgres=# CREATE TABLESPACE ts1 OWNER demo LOCATION '/usr/local/pgsql/ts1' ;
CREATE TABLESPACE
$ touch ts1/test_file.txt
$ mkdir ts1/test_dir/
$ touch ts1/test_dir/test_file.txt
$ pg_basebackup -D back -p 5432 -v -x
      --tablespace-mapping=/usr/local/pgsql/ts1=/usr/local/pgsql/back_ts1
transaction log start point: 0/2000028 on timeline 1
transaction log end point: 0/20000C0
pg_basebackup: base backup completed
$ ls back_ts1/
PG_9.6_201608131
```

上記例の通り、表スペース内に作成されたユーザー作成ファイルはコピーされないことを確認しました。

□ —xlog-method パラメーターの動作

--xlog-method（または-X）パラメーターには WAL ファイルの転送方法を指定します。このパラメーターを指定すると—xlog（または-x）パラメーターは指定できませんが、WAL ファイルが転送元インスタンスからバックアップ先に転送されます。--xlog-method=streaming を指定した場合、PostgreSQL インスタンスには同時に 2 つの wal sender プロセスが起動します。

表 83 pg_basebackup コマンドにより起動するプロセス

プロセス名	説明
postgres: wal sender process ... sending backup ...	ファイル転送用プロセス
postgres: wal sender process ... streaming ...	WAL 転送用プロセス

8.1.2 pg_archivecleanup コマンド

pg_archivecleanup コマンド¹⁶はバックアップが完了し、不要になったアーカイブログ・ファイルを削除します。

¹⁶ PostgreSQL 9.5 で Contrib モジュールから PostgreSQL 9.5 本体に移動されました。

通常はストリーミング・レプリケーション環境で recovery.conf ファイルの archive_cleanup_command のパラメーター値として使用します。最初のパラメーターにはアーカイブログ出力ディレクトリを、2番目のパラメーターには最終の WAL ファイルを示す「%r」を指定します。

例 201 recovery.conf ファイルの指定

```
archive_cleanup_command = 'pg_archivecleanup /usr/local/pgsql/archive %r'
```

pg_archivecleanup コマンドはスタンダードアロン環境でも使用できます。第2パラメーターには、オンライン・バックアップで作成されたラベル・ファイルを指定します。以下のようなプログラムを作成することで、最終のラベル・ファイルを取得することができます。

例 202 pg_archivecleanup コマンド実行スクリプト

```
#!/bin/sh

export PATH=/usr/local/pgsql/bin:${PATH}

ARCHDIR=/usr/local/pgsql/archive
LASTWALPATH=`/bin/ls $ARCHDIR/*.backup | /bin/sort -r | /usr/bin/head -1`

if [ $LASTWALPATH = '' ]; then
    echo 'NO label file found.'
    exit 1
fi

LASTWALFILE=`/bin/basename $LASTWALPATH`

pg_archivecleanup $ARCHDIR $LASTWALFILE
stat=$?

echo 'Archivelog cleanup complete'
exit $stat
```

8.1.3 psql コマンド

psql コマンドは会話的に SQL 文を実行するクライアント・ツールです。psql コマンド¹⁷が使用する環境変数は以下の通りです。

表 84 psql コマンドが使用する環境変数

環境変数	説明	デフォルト
COLUMNS	改行幅の制限値 \$pset columns のデフォルト	ターミナルの幅から計算
PAGER	ページヤ・コマンド名	Cygwin 環境では less それ以外では more
PGCLIENTENCODING	クライアント・エンコード	auto
PGDATABASE	デフォルト・データベース名	OS ユーザー名
PGHOST	デフォルト・ホスト名	localhost
PGPORT ¹⁸	デフォルト・ポート番号	5,432
PGUSER	デフォルト・ユーザ名	OS ユーザー名
PSQL_EDITOR	\$e コマンドで使用するエディタ名。リストを上から検索する。	Linux / Unix では vi Windows では notepad.exe
EDITOR		
VISUAL		
PSQL_EDITOR_LINEN	エディタに行番号を渡すコマンド	Linux / UNIX では '+' Windows ではなし
NUMBER_ARG		
COMSPEC	! コマンド用シェル (Windows)	cmd.exe
SHELL	!コマンド用シェル (Linux / UNIX)	/bin/sh
PSQL_HISTORY	履歴保存ファイル	Linux / UNIX では {HOME}/.psql_history Windows では {HOME}\psql_history
PSQLRC	初期化コマンド用ファイルのパス	Linux / UNIX では {HOME}/.psqlrc Windows では {HOME}\psqlrc.conf

¹⁷ マニュアルは <https://www.postgresql.org/docs/9.6/static/app-psql.html>

¹⁸ インスタンス起動時の接続待ちポート番号のデフォルト値としても使用される。

表 84 (続) psql コマンドが使用する環境変数

環境変数	説明	デフォルト
TMPDIR	ファイル編集用一時ディレクトリ	Linux / UNIX では /tmp Windows では GetTempPath API による取得
PGPASSFILE	パスワードファイル	Linux では\$HOME/.pgpass Windows では %APPDATA%\postgresql\pgpass.conf

8.1.4 pg_resetxlog コマンド

pg_resetxlog コマンド¹⁹は WAL ファイルの再作成を行います。このコマンドはインスタンス起動中には実行できません。マニュアルにあるとおり、{PGDATA}/postmaster.pid ファイルの存在をチェックしています。存在のみをチェックしており、インスタンスの起動をチェックしているわけではありません。

pg_resetxlog コマンドは以下の処理を行っています。

1. オプションのチェック
2. データベース・クラスターのチェックと、ディレクトリ移動
3. postmaster.pid ファイルの存在チェック
4. pg_control ファイルの読み込み
 - a. 読込不可→プログラム終了
 - b. バージョン・チェックと CRC のチェック
5. pg_control ファイルに不整合があった場合は正しい値を予測
6. pg_xlog ディレクトリから最終の WAL ファイルを検索
7. 直前のインスタンスが正常終了 (DB_SHUTDOWNED (1)) かどうかをチェックし、正常終了では無い場合-f オプションが指定されていなければ終了。
8. pg_control ファイルの削除と再作成
9. WAL ファイルの削除
10. archive_status ディレクトリ内のファイル削除
11. 新規 WAL ファイルの作成

以下は、pg_resetxlog コマンド実行前後で pg_controldata コマンドの出力結果の比較例です。表示が異なる部分のみ記載しています。

¹⁹ マニュアルは <https://www.postgresql.org/docs/9.6/static/app-pgresetxlog.html>

表 85 pg_controldata コマンドの比較

項目	pg_resetxlog 実行前	pg_resetxlog 実行後
pg_control last modified	Fri Feb 11 15:22:13 2017	Fri Feb 11 16:02:40 2017
Latest checkpoint location	2/A4000028	2/AC000028
Prior checkpoint location	2/A3002D20	0/0
Latest checkpoint's REDO location	2/A4000028	2/AC000028
Latest checkpoint's REDO WAL file	0000000100000002000000A4	0000000100000002000000AC
Time of latest checkpoint	Fri Feb 11 15:22:01 2017	Fri Feb 11 16:02:26 2017
Backup start location	0/E1000028	0/0

8.1.5 pg_rewind コマンド

pg_rewind²⁰コマンドは PostgreSQL 9.5 で追加されました。

□ 概要

pg_rewind コマンドはレプリケーション環境を構築するツールです。pg_basebackup コマンドと異なり、既存のデータベース・クラスターに対して同期を行うことができます。プロモーションされたスレーブ・インスタンスと旧マスター・インスタンスの再同期を行う場面を想定しています。

□ パラメーター

pg_rewind コマンドには以下のパラメーターを指定できます。

²⁰ マニュアルは <https://www.postgresql.org/docs/9.6/static/app-pgrewind.html>

表 86 パラメーター

パラメーター	説明
-D / --target-pgdata	更新を行うデータベース・クラスターのディレクトリ
--source-pgdata	データ取得元のディレクトリ
--source-server	データ取得元の接続情報（リモート・インスタンス）
-P / --progress	実行状況の出力
-n / --dry-run	実行シミュレーションを行う
--debug	デバッグ情報の表示
-V / --version	バージョン情報表示
-? / --help	使用方法のメッセージ表示

□ 条件

`pg_rewind` コマンドを実行するためには、いくつかの条件があります。`pg_rewind` コマンドは、実行する条件をソースとターゲットの `pg_control` ファイルの内容からチェックしています。

まず PostgreSQL インスタンスのパラメーター `wal_log_hints` を `on`（デフォルト値 `off`）に指定するか、データ・チェックサムの機能を有効にする必要があります。またパラメータ `full_page_writes` を `on` に設定する必要があります（デフォルト値 `on`）。

例 203 パラメーター設定上のエラー・メッセージ

```
$ pg_rewind --source-server=' host=remhost1 port=5432 user=postgres'
      --target-pgdata=data -P
connected to remote server

target server need to use either data checksums or "wal_log_hints = on"
Failure, exiting
```

またターゲットとなるデータベース・クラスターのインスタンスは正常に停止している必要があります。

例 204 ターゲット・インスタンス起動中のエラー・メッセージ

```
$ pg_rewind --source-server=' host=remhost1 port=5432 user=postgres'
      --target-pgdata=data -P
target server must be shut down cleanly
Failure, exiting
```

データのコピー処理は pg_basebackup コマンドと同様に wal sender プロセスに対する接続を使用します。接続先（データ提供元）インスタンスの pg_hba.conf ファイル設定や、max_wal_senders パラメーターの設定が必要です。

□ 実行手順

pg_rewind は以下の手順で実行します。下記の例は、プロモーションを行ったスレーブ・インスタンスに接続し、旧マスター・インスタンスを新しいスレーブ・インスタンスに設定しています。

1. パラメーター設定

現在のマスター・インスタンスのパラメーター、及び pg_hba.conf ファイルの設定を行います。必要に応じてファイルの情報をリロードします。

2. ターゲット・インスタンス停止

同期を取る（旧マスター）インスタンスを停止します。

3. pg_rewind 実行

旧マスター・インスタンス（データを更新する側）で pg_rewind コマンドを実行します。最初に -n パラメーターを指定して、テストを行った後、-n パラメーターを取って実行します。

例 205 pg_rewind -n コマンドの実行

```
$ pg_rewind --source-server='host=remhost1 port=5432 user=postgres'  
--target-pgdata=data -n
```

```
The servers diverged at WAL position 0/50000D0 on timeline 1.  
Rewinding from last common checkpoint at 0/5000028 on timeline 1  
Done!
```

例 206 pg_rewind コマンドの実行

```
$ pg_rewind --source-server='host=remhost1 port=5432 user=postgres'  
--target-pgdata=data
```

```
The servers diverged at WAL position 0/50000D0 on timeline 1.  
Rewinding from last common checkpoint at 0/5000028 on timeline 1  
Done!
```

4. recovery.conf ファイルの編集

pg_rewind コマンドでは更新先データベース・クラスターに recovery.conf ファイルは作成されません。このため新スレーブ・インスタンス用に recovery.conf ファイルを作成します。前項で作成したレプリケーション・スロット名の指定を行います。

5. postgresql.conf ファイルの編集

pg_rewind コマンドの実行により、postgresql.conf ファイルはリモート・ホストからコピーされて上書きされています。必要に応じてパラメーターを編集します。

6. スレーブ・インスタンスの起動

新しいスレーブ・インスタンスを起動します。

□ 終了ステータス

pg_rewind コマンドは、処理が正常に終了すると 0 を、失敗すると 1 を返して終了します。

8.1.6 vacuumdb コマンド

vacuumdb コマンドは強制的に Vacuum 处理を実行します。複数コアを積極的に使用するために--jobs パラメーターを指定することができます。--jobs パラメーターには並列処理させるジョブ数を指定します。ここでは--jobs パラメーターの詳細な実装について検証しました。ジョブ数は 1 以上、「マクロ FD_SETSIZE - 1」以下 (Red Hat Enterprise Linux 7 では 1,023 以下) です。

例 207 --jobs パラメーターの上限と下限

```
$ vacuumdb --jobs=-1
vacuumdb: number of parallel "jobs" must be at least 1
$ vacuumdb --jobs=1025
vacuumdb: too many parallel jobs requested (maximum: 1023)
```

□ --jobs パラメーターとセッション数

--jobs パラメーターに数値を指定すると、パラメーターで指定された数と同数のセッションが作成されます。全データベースに対して VACUUM を行う場合 (--all 指定) は、データベース単位で、单一のデータベースに対して VACUUM を行う場合は、テーブル単位で並列に処理を行います。このパラメーターのデフォルト値は 1 で、従来バージョンと同じ

動作になります。

下記の例では--jobs=10 を指定したため、postgres プロセスが 10 個起動しています。

例 208 --jobs パラメーターの指定とセッション

```
$ vacuumdb --jobs=10 -d demodb &
vacuumdb: vacuuming database "demodb"

$ ps -ef|grep postgres
postgres 14539      1  0 10:59 pts/2  00:00:00 /usr/local/pgsql/bin/postgres -D data
postgres 14540 14539  0 10:59 ?        00:00:00 postgres: logger process
postgres 14542 14539  0 10:59 ?        00:00:00 postgres: checkpointer process
postgres 14543 14539  0 10:59 ?        00:00:00 postgres: writer process
postgres 14544 14539  0 10:59 ?        00:00:00 postgres: wal writer process
postgres 14545 14539  0 10:59 ?        00:00:00 postgres: stats collector process
postgres 14569 14539  6 11:00 ?        00:00:00 postgres: postgres demodb [local] VACUUM
postgres 14570 14539  0 11:00 ?        00:00:00 postgres: postgres demodb [local] idle
postgres 14571 14539  5 11:00 ?        00:00:00 postgres: postgres demodb [local] VACUUM
postgres 14572 14539  7 11:00 ?        00:00:00 postgres: postgres demodb [local] VACUUM
postgres 14573 14539  0 11:00 ?        00:00:00 postgres: postgres demodb [local] idle
postgres 14574 14539  0 11:00 ?        00:00:00 postgres: postgres demodb [local] idle
postgres 14575 14539  9 11:00 ?        00:00:00 postgres: postgres demodb [local] VACUUM
postgres 14576 14539  5 11:00 ?        00:00:00 postgres: postgres demodb [local] VACUUM
postgres 14577 14539  0 11:00 ?        00:00:00 postgres: postgres demodb [local] idle
postgres 14578 14539  1 11:00 ?        00:00:00 postgres: postgres demodb [local] idle
```

--jobs パラメーターで指定した値が、--table パラメーター数以上だった場合、並列度の上限はテーブル数になります。またセッション数の計算には PostgreSQL パラメーター max_connections は考慮されないため、セッション数の超過が検知されると「FATAL: sorry, too many clients already」エラーが発生します。

8.2 ユーティリティの終了ステータス

PostgreSQL に付属する各種ユーティリティの終了ステータスを確認しました。

8.2.1 pg_ctl コマンド

`pg_ctl` コマンドは処理が成功した場合 0 を返します。インスタンス起動時 (`pg_ctl start`) に -w オプションを指定していない場合、`system` 関数によるバックグラウンド起動が成功した時点で 0 を返して終了します。このため `pg_ctl` コマンドはインスタンス起動の成功をチェックしているわけではありません。

表 87 pg_ctl コマンドの終了ステータス

ステータス	説明	備考
0	処理が成功した場合	
1	処理が失敗した場合 標準エラー出力にメッセージを出力 (-l オプション指定時はログ・ファイルに)。	
3	<code>pg_ctl status</code> コマンド実行時にインスタンスが起動していない場合	
4	<code>pg_ctl status</code> コマンド実行時に、データベース・クラスターが存在しない場合	9.4 追加

8.2.2 psql コマンド

`psql` コマンドは処理が成功した場合 0 を返します。`-c` オプションで SQL 文を指定し、処理が失敗すると 1 を返します。しかし、`-f` オプションで SQL 文を指定し、処理が失敗した場合には 0 を返します。`-f` オプションの動作は `ON_ERROR_STOP` 属性を `true` (または 1) に設定することで変更されます。`ON_ERROR_STOP` 属性は、`\$set` コマンドで実行するか、`psql` コマンドの`--set` オプションで指定します。

表 88 psql コマンドの終了ステータス

ステータス	説明	備考
0	処理成功	
1	処理失敗	
2	接続失敗または flex 関連エラー	
3	<code>\\$set ON_ERROR_STOP true</code> 実行後にエラーが発生した場合。	

例 209 psql のエラー検知

```
$ psql -c 'SELECT * FROM notexists'  
Password: {PASSWORD}  
ERROR: relation "notexist" does not exist  
LINE 1: SELECT * FROM notexist  
          ^  
  
$ echo $?  
1  
  
$ cat error.sql  
SELECT * FROM notexists ;  
$ psql -f error.sql  
Password: {PASSWORD}  
psql:error.sql:1: ERROR: relation "notexists" does not exist  
LINE 1: SELECT * FROM notexists ;  
          ^  
  
$ echo $?  
0  
  
$ psql -f error.sql--set=ON_ERROR_STOP=true  
Password: {PASSWORD}  
psql:error.sql:1: ERROR: relation "notexists" does not exist  
LINE 1: SELECT * FROM notexists ;  
  
$ echo $?  
3  
  
$ cat stop.sql  
$set ON_ERROR_STOP true  
SELECT * FROM notexists ;  
$ psql -f error.sql  
Password: {PASSWORD}  
psql:error.sql:1: ERROR: relation "notexists" does not exist  
LINE 1: SELECT * FROM notexists ;  
          ^  
  
$ echo $?  
3
```

8.2.3 pg_basebackup コマンド

pg_basebackup コマンドは処理が成功した場合 0 を返します。処理失敗時は標準出力にメッセージを出力して、1 を返して終了します。

表 89 pg_basebackup コマンドの終了ステータス

ステータス	説明	備考
0	処理成功	
1	処理失敗	

8.2.4 pg_archivecleanup コマンド

pg_archivecleanup コマンドは処理が成功した場合 0 を返します。処理失敗時は標準エラーにメッセージを出力して、2 を返して終了します。

表 90 pg_archivecleanup コマンドの終了ステータス

ステータス	説明	備考
0	処理成功（またはヘルプ表示）	
2	処理失敗	

8.2.5 initdb コマンド

initdb コマンドは処理が成功した場合 0 を返します。処理失敗時は標準エラーにメッセージを出力して、1 を返して終了します。

表 91 initdb コマンドの終了ステータス

ステータス	説明	備考
0	処理成功	
1	処理失敗	

8.2.6 pg_isready コマンド

pg_isready コマンドはパラメーターのチェックを行い、不正なパラメーターが指定された場合は 3 を返します。その後 PQpingParams 関数 (src/interfaces/libpq/fe-connect.c) を実行して関数の戻り値で終了します。以下の値が返ります。

表 92 pg_isready コマンドの終了ステータス

ステータス	説明	備考
0	インスタンスは稼働中で接続受付け可能	PQPING_OK
1	インスタンスは稼働中だが接続受付不可	PQPING_REJECT
2	インスタンスと通信不可	PQPING_NO_RESPONSE
3	パラメーター不正、接続文字列不正	PQPING_NO_ATTEMPT

備考のマクロはヘッダ（src/interfaces/libpq/fe-connect.h）で定義されています。

8.2.7 pg_receivexlog コマンド

pg_receivexlog コマンドは処理が成功した場合 0 を返します。処理失敗時は標準エラーにメッセージを出力して、1 を返して終了します。

表 93 pg_receivexlog コマンドの終了ステータス

ステータス	説明	備考
0	処理成功	以下の場合もステータスは 0 になります。 <ul style="list-style-type: none"> • SIGINT シグナルを受けた場合 • --help を指定した場合 • --version を指定した場合
1	処理失敗	

9. システム構成

9.1 パラメーターのデフォルト値

PostgreSQL で使用するパラメーターはデータベース・クラスター内の `postgresql.conf` ファイルに記載されます。先頭が#の行はコメントとして扱われます。`initdb` コマンド実行直後で変更されているパラメーターを調査しました。

9.1.1 initdb コマンド実行時に導出されるパラメーター

いくつかのパラメーターは `initdb` コマンド実行時の環境変数やホストの設定状況から値を導出し、`postgresql.conf` ファイルに設定します。

表 94 initdb コマンド実行時に設定されるパラメーター

パラメーター	設定値	デフォルト値
<code>max_connections</code>	100	100
<code>shared_buffers</code>	128MB	8MB
<code>dynamic_shared_memory_type</code>	<code>posix</code>	<code>posix</code>
<code>log_timezone</code>	環境変数から導出	GMT
<code>datestyle</code>	環境変数から導出	ISO,MDY
<code>timezone</code>	環境変数から導出	GMT
<code>lc_messages</code>	環境変数から導出	-
<code>lc_monetary</code>	環境変数から導出	C
<code>lc_numeric</code>	環境変数から導出	C
<code>lc_time</code>	環境変数から導出	C
<code>default_text_search_config</code>	環境変数から導出	'pg_catalog.simple'

9.2 推奨構成

PostgreSQLには多くのパラメーターや属性が定義されています。必要に応じて変更しますが、まず初期状態として以下の値を使用することをお勧めします。

9.2.1 ロケール設定

データベース・クラスター作成時に指定する `initdb` コマンドのパラメーター推奨値は以下の通りです。ロケール関連機能が明らかに必要である場合以外は使用しないことを推奨します。また、エンコードは文字集合が大きい UTF-8 をお勧めします。

表 95 `initdb` コマンドの推奨パラメーター

パラメーター	推奨値	備考
<code>--encoding</code>	UTF8	または EUC_JIS_2004
<code>--locale</code>	指定しない	
<code>--no-locale</code>	指定する	
<code>--username</code>	postgres	
<code>--data-checksums</code>	指定する	アプリケーション要件で決定

9.2.2 推奨パラメーター

一般的なシステムにおける推奨パラメーター設定は以下の通りです。

表 96 postgresql.conf ファイルに設定する推奨パラメーター

パラメーター	推奨値	備考
archive_command	'test ! -f {ARCHIVEDIR}/%f && cp %p {ARCHIVEDIR}/%f'	
archive_mode	on	
autovacuum_max_workers	データベース数以上	
max_wal_size	2GB	
checkpoint_timeout	30min	
checkpoint_warning	30min	
client_encoding	utf8	
effective_cache_size	搭載メモリー量	
log_autovacuum_min_duration	60	
log_checkpoints	on	
log_line_prefix	'%t %u %d %r '	
log_min_duration_statement	30s	
log_temp_files	on	
logging_collector	on	
maintenance_work_mem	32MB	
max_connections	予想接続数 × 110%	
max_wal_senders	レプリケーション数+1 以上	
server_encoding	utf8	
shared_buffers	搭載メモリー量の 3 分の 1	
tcp_keepalives_idle	60	
tcp_keepalives_interval	5	
tcp_keepalives_count	5	
temp_buffers	8MB	
timezone	Japan	
wal_buffers	16MB	
work_mem	8MB	
wal_level	replica	
max_replication_slots	スレーブ・インスタンス数 + 1 以上	レプリケーション 時

10. ストリーミング・レプリケーション

PostgreSQL 9.0 から利用できるようになったストリーミング・レプリケーションについて簡単に説明しています。

10.1 ストリーミング・レプリケーションの仕組み

PostgreSQL にはリモート・ホストで稼働するインスタンスとデータを同期するレプリケーション機能が標準で提供されています。ここでは PostgreSQL 標準のレプリケーション機能であるストリーミング・レプリケーション機能を説明します。

10.1.1 ストリーミング・レプリケーションとは

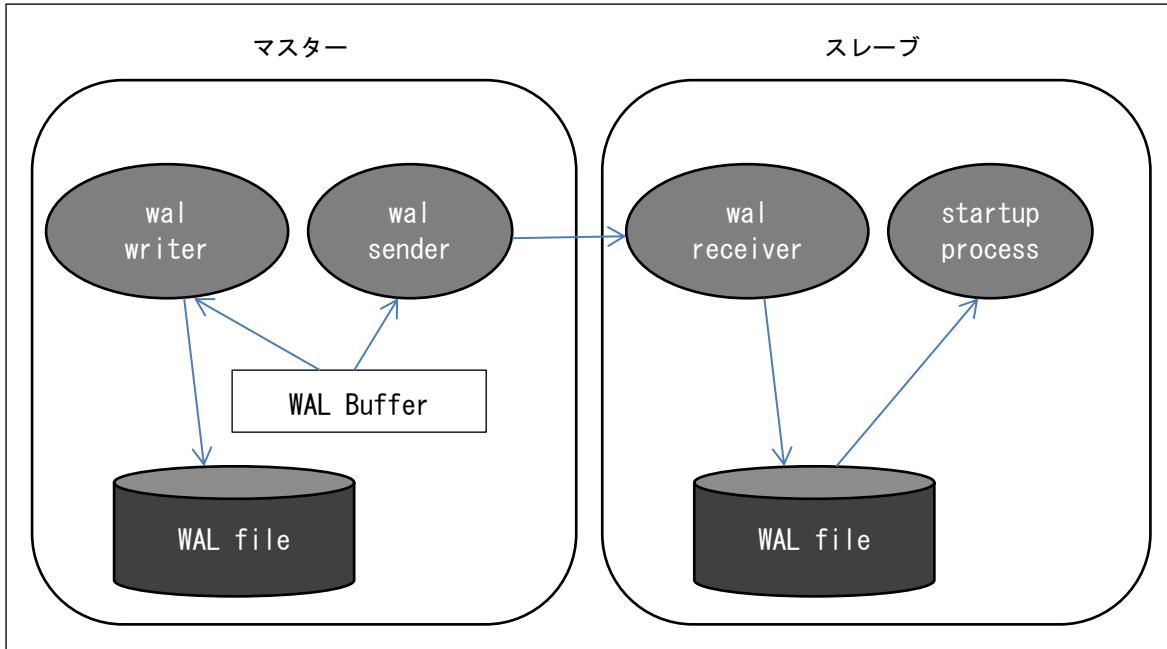
PostgreSQL 9.0 よりも古いバージョンでは、データベースのレプリケーションには PostgreSQL 本体とは独立したツールである Slony-I や pgpool-II を使用していました。これらのツールは現在でも有効ですが、PostgreSQL 9.0 からトランザクション・ログ (WAL) を転送することでレプリケーションを行うストリーミング・レプリケーション機能が標準で提供されるようになりました。

ストリーミング・レプリケーション環境では、更新処理は常に 1 つのインスタンスのみで実行されます。更新を行なうインスタンスをマスター・インスタンスと呼びます。データベースに対する更新で発生した WAL 情報はスレーブ・インスタンスに転送されます。スレーブ・インスタンスでは受信した WAL 情報をデータベースに適用することで複数データベースの同一性を保障します。スレーブ・インスタンスではリアルタイムにデータベースのリカバリーを実施している状態になります。スレーブ・インスタンスは読み込み専用 (Read Only) 状態で起動することができ、テーブルに対して検索 (SELECT) を実行することができます。このため、レプリケーション環境を検索負荷の分散用に利用することができます。

10.1.2 ストリーミング・レプリケーションの構成

ストリーミング・レプリケーションは WAL を転送し、適用することでレプリカを更新する機能であるため、WAL 適用のためのベースとなるデータベースが必要です。マスターとなるデータベース・クラスターのコピーを取得し、レプリケーションのベースとします。マスター・インスタンスに対して更新トランザクションが発生すると、ローカルの WAL が更新されます。次に wal sender プロセスがスタンバイ側に WAL 情報を転送します。スタンバイ・インスタンスは wal receiver が WAL 情報を受け取り、WAL をストレージに書き込みます。書き込まれた WAL は非同期にデータベース・クラスターに適用され、スレーブ・インスタンスが更新されます。

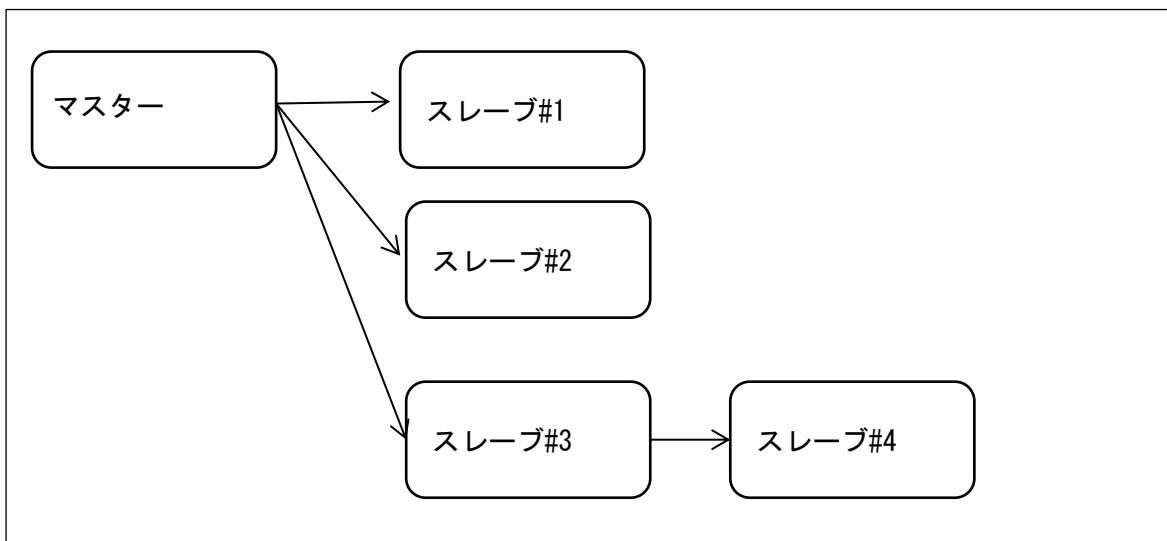
図 19 ストリーミング・レプリケーション



□ カスケード・レプリケーション構成

ストリーミング・レプリケーションの最も単純な構成はマスターに対してスタンバイを1つだけ用意するものです。複数のスレーブに対してレプリケーションを行うことも可能です。また、スレーブ・インスタンスを他のインスタンスのマスターと見なすカスケード・レプリケーション構成を取ることもできます。

図 20 カスケード・レプリケーション



10.2 レプリケーション環境の構築

ここではレプリケーション環境の構築方法について説明しています。

10.2.1 レプリケーション・スロット

PostgreSQL 9.4 以降のストリーミング・レプリケーションは、マスター・インスタンスにレプリケーション・スロットと呼ばれるオブジェクトを作成し、スレーブ・インスタンスはレプリケーション・スロット名を参照することでレプリケーションの進捗状況を管理することができるようになりました。PostgreSQL 9.3 までのレプリケーションは、スレーブ・インスタンスが停止している場合でも WAL ファイルはパラメーター wal_keep_segments で指定された個数までしかファイルを保持しませんでした。レプリケーション・スロットを使うことでスレーブに必要な WAL ファイルが自動的に管理され、スレーブが WAL ファイルを受け取らない限りマスターの WAL が削除されないように変更されました。基本的なレプリケーションの構造は変更が無いため、引き続き wal sender プロセス用パラメーターや pg_hba.conf ファイルの設定等は必要です。

□ レプリケーション・スロットの管理

レプリケーション・スロットは以下の関数で管理を行います。使用中のレプリケーション・スロットは削除できません。従来から利用できるストリーミング・レプリケーションは Physical Replication と呼ばれます。

構文 8 スロット作成関数

```
pg_create_physical_replication_slot(スロット名)  
pg_create_logical_replication_slot(スロット名, プラグイン名)
```

構文 9 スロット削除関数

```
pg_drop_replication_slot(スロット名)
```

データベース・クラスターに作成できるレプリケーション・スロット数の最大値はパラメーター max_replication_slots で指定します。このパラメーターのデフォルト値は 0 であるため、レプリケーションを行う場合には変更する必要があります。インスタンス起動時には、パラメーター max_replication_slots で指定された値を元に共有バッファ上にレプリケーション関連の情報が展開されます。

作成したレプリケーション・スロットの情報は pg_replication_slots カタログから確認することができます。

例 210 レプリケーション・スロットの作成と確認

```
postgres=# SELECT pg_create_physical_replication_slot('slot_1') ;
pg_create_physical_replication_slot
-----
(slot_1, )
(1 row)

postgres=# SELECT slot_name, active, active_pid FROM pg_replication_slots ;
slot_name | active | active_pid
-----+-----+
slot_1    | f      |
(1 row)
```

マスター・インスタンスで作成したレプリケーション・スロットは、スレーブ・インスタンスの recovery.conf から参照します。

例 211 recovery.conf ファイル内のレプリケーション・スロット参照

```
primary_slot_name = 'slot_1'
primary_conninfo = 'host=hostmstr1 port=5433 application_name=prim5433'
standby_mode = on
```

レプリケーションが成功するとマスター側の pg_stat_replication カタログ、pg_replication_slots カタログは以下の表示になります。

例 212 レプリケーション状況の確認

```
postgres=# SELECT pid, state, sync_state FROM pg_stat_replication ;
      pid      |   state    | sync_state
-----+-----+-----+
 12847 | streaming | async
(1 row)

postgres=# SELECT slot_name, slot_type, active, active_pid FROM
            pg_replication_slots ;
      slot_name | slot_type | active | active_pid
-----+-----+-----+-----+
 slot_1 | physical | t       |      12847
(1 row)
```

PostgreSQL 9.4 の pg_stat_replication カタログには backend_xmin 列が追加されています。PostgreSQL 9.5 の pg_replication_slots カタログには active_pid 列が追加されています。スレーブ・インスタンスでは pg_stat_wal_receiver カタログでレプリケーション状況を確認することができます。

□ 設定ミスによる挙動

現状の実装では、スレーブ側に primary_slot_name パラメーターの指定が存在しない場合でもレプリケーションは成功します。primary_slot_name が記述されているにも関わらずマスター側でレプリケーション・スロットが作成されていないと以下のエラーが発生します。

例 213 存在しないレプリケーション・スロット名を指定した場合のエラー

```
FATAL: could not start WAL streaming: ERROR: replication slot "slot_1" does
not exist
```

レプリケーション・スロットが見つからない場合、レプリケーションはできませんが、スレーブ側のインスタンスは起動します。

複数のスレーブ・インスタンスを起動する場合、primary_slot_name パラメーターに既に使用中のレプリケーション・スロット名を指定すると、以下のエラーが発生します。この場合、レプリケーションは実行できませんが、スレーブ側のインスタンスは起動します。

例 214 使用中のレプリケーション・スロット名を指定した場合のエラー

```
FATAL: could not start WAL streaming: ERROR: replication slot "slot_1" is already active
```

□ レプリケーション・スロットの実体

レプリケーション・スロットの実体は{PGDATA}/pg_replslotディレクトリ内に作成されたレプリケーション・スロット名と同じ名前のディレクトリとファイルです。

例 215 レプリケーション・スロットの実体

```
$ ls -l data/pg_replslot/
total 4
drwx-----. 2 postgres postgres 4096 Feb 11 15:42 slot_1
$ ls -l data/pg_replslot/slot_1/
total 4
-rw-----. 1 postgres postgres 176 Feb 11 15:42 state
```

カスケード化されたレプリケーション環境では、WAL を提供するすべてのインスタンスでレプリケーション・スロットを作成します。スレーブ・インスタンスは読み取り専用ですが、レプリケーション・スロットの作成関数は実行できます。

10.2.2 同期と非同期

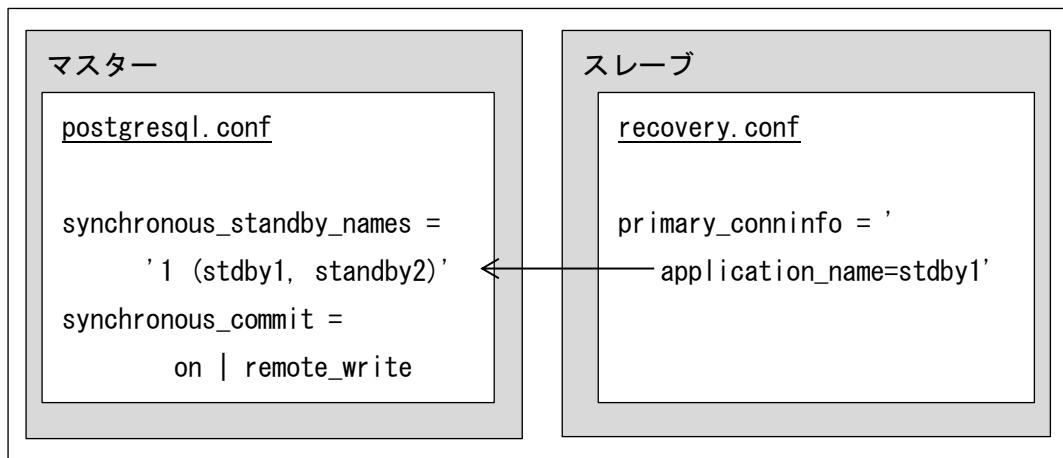
ストリーミング・レプリケーションはマスター・インスタンスから WAL 情報を転送することによりスレーブ・インスタンスと同期を取ります。マスター・インスタンスの WAL と、スレーブ・インスタンスの WAL の両方を書き込んだことを確認してからトランザクションの完了をユーザーに通知すれば信頼性は向上しますが、パフォーマンスは大幅に低下します。一方でスレーブ・インスタンスに対する WAL 書き込みを非同期に行えばパフォーマンスは向上するかもしれません、スレーブ・システムに WAL が到着する前にマスター・インスタンスが異常終了すると書き込んだはずのトランザクションが失われる恐れがあります。このため PostgreSQL のストリーミング・レプリケーションには、信頼性とパフォーマンスのバランスを選択するために 5 つのモードを用意しています。モードの選択はマスター・インスタンスのパラメーター `synchronous_commit` で決定されます。

表 97 パラメーター `synchronous_commit` 設定値

設定値	プライマリ WAL	スタンバイ WAL	スタンバイ 適用	備考
on	同期	同期	非同期	ストレージ書き込みまで同期
remote_write	同期	同期	非同期	メモリー書き込みまで同期
local	同期	非同期	非同期	
off	非同期	非同期	非同期	
remote_apply	同期	同期	同期	PostgreSQL 9.6～

同期レプリケーションを行うには、マスター・インスタンスのパラメーター `synchronous_standby_names` に同期レプリケーションを行うインスタンス数と、任意の名前のリスト（カンマ区切り）を指定します。スレーブ・インスタンスは `recovery.conf` ファイルの `primary_conninfo` パラメーター内の `application_name` 項目に同じ名前を指定します。

図 21 同期レプリケーション設定



レプリケーションの同期方法は、`pg_stat_replication` カタログの `sync_state` 列で確認できます。この列は以下の値を持つ可能性があります。

表 98 パラメーター `synchronous_commit` 設定値

列値	説明
<code>sync</code>	同期レプリケーション
<code>async</code>	非同期レプリケーション
<code>potential</code>	現状では非同期レプリケーションだが、より優先順位が高い同期レプリケーション・インスタンスが停止した場合には同期レプリケーションに変更される。

同期レプリケーションが行われるスレーブ・インスタンス数はパラメーター `synchronous_standby_names` の先頭に記述された値です。優先順位が高いインスタンス順に同期レプリケーションが行われます。優先順位はマスター・インスタンスのパラメーター `synchronous_standby_names` に指定した名前の左から順に決定されます。現在の優先順位は `pg_stat_replication` カタログの `sync_priority` 列で確認できます。非同期レプリケーションではこの列の値は常に 0 です。

停止していたより高い優先順位を持つスレーブ・インスタンスがレプリケーション環境に復帰すると、同期レプリケーションが行われるインスタンスは自動的に切り替わります。

□ スレーブ・インスタンス停止時の動作

非同期レプリケーション環境では、スレーブ・インスタンスが停止してもマスター・インスタンスは稼働を続けます。スレーブ・インスタンスがレプリケーションを再開し、マスター・インスタンスに追いつくためには停止中の WAL を適用する必要があります。PostgreSQL 9.3 までは、パラメーター `wal_keep_segments` に `pg_xlog` ディレクトリに保持する WAL ファイル数を指定します。PostgreSQL 9.4 以降ではスレーブに適用できなかつた WAL ファイルはレプリケーション・スロットにより管理されるため、自動的に保持されるようになりました。

同期レプリケーションでは、スレーブ・インスタンスがすべて停止するとマスター・インスタンスはトランザクションを実行できずにハング状態に陥ります。

10.2.3 パラメーター

レプリケーションに関連するパラメーターは以下の通りです。

表 99 マスター・インスタンスのパラメーター

パラメーター	説明	設定値
wal_level	WAL 出力レベル	hot_standby
archive_mode	アーカイブ出力モード	on (または always)
archive_command	アーカイブ出力コマンド	cp コマンド等
max_wal_senders	wal sender プロセス最大数	スレーブ数以上
max_replication_slots	レプリケーション・スロット 最大数	スレーブ数以上
synchronous_commit	同期コミット	選択
synchronous_standby_names	同期コミット名称	同期の場合設定

スレーブに対して SELECT 文を実行する場合は以下のパラメーターを指定します。 hot_standby パラメーターがデフォルト値 (off) の場合、スレーブ側のインスタンスに対して SELECT 文を実行することはできません。この値はマスター・インスタンスで指定しても影響はありませんが、スイッチオーバー時に有効にするためにあらかじめ設定しておくこともできます。

表 100 スレーブ・インスタンスのパラメーター

パラメーター	説明	設定値
hot_standby	スレーブを参照可能に設定	on
archive_mode	アーカイブ出力モード	off (または always)

10.2.4 recovery.conf ファイル

スレーブ・インスタンスは WAL 情報を受信し、リカバリーを行っている状態です。このためデータベースのリカバリーと同じように、データベース・クラスター内に recovery.conf ファイルを作成します。以下のパラメーターを指定することができます。 primary_slot_name パラメーターは PostgreSQL 9.4 から指定できます。

表 101 スレーブ・インスタンスの recovery.conf ファイル

パラメーター	説明	設定	備考
standby_mode	スタンバイ・モード	on	
primary_slot_name	接続先レプリケーション・スロット名	レプリケーション・スロット名	9.4～
primary_conninfo	接続先インスタンス	接続情報	
restore_command	アーカイブ・リスト アコマンド	scp コマンドによるアーカイブのコピー設定	
recovery_target_timeline	タイムライン設定	選択	
min_recovery_apply_delay	遅延リカバリー設定	時間設定	9.4～
trigger_file	トリガー・ファイル	ファイル名	

例 216 recovery.conf の作成例

```
standby_mode = 'on'
primary_slot_name = 'slot_1'
primary_conninfo = 'host=hostmstr1 port=5432 user=postgres password=secret
                   application_name=standby1'
restore_command = 'scp hostmstr1:/usr/local/pgsql/archive/%f %p'
recovery_target_timeline='latest'
```

□ trigger_file パラメーター

trigger_file パラメーターは必須ではありません。指定すると startup process が 5 秒ごとにファイルのチェックを行い、ファイルが存在するとスレーブ・インスタンスがマスターに昇格されます。

□ primary_conninfo パラメーター

primary_conninfo パラメーターにはマスター・インスタンスに接続するための情報を記述します。記述は「パラメータ名=値」をスペースで区切って複数記述することができます。user 句には REPLICATION 権限を持つユーザー名を指定します。

表 102 primary_conninfo に指定できるパラメーター

パラメーター	説明	備考
service	サービス名	
user	接続ユーザー名	
password	接続パスワード	
connect_timeout	接続タイムアウト (秒)	
dbname	データベース名	
host	接続ホスト名	マスター・インスタンスのホスト名
hostaddr	接続ホスト IP アドレス	
port	接続ポート番号	
client_encoding	クライアント文字コード	
options	オプション	
application_name	アプリケーション名	同期レプリケーションの場合

□ restore_command パラメーター

リカバリーに必要なアーカイブログ・ファイルを取得するためのコマンドを指定します。スレーブ・インスタンスが長時間停止していた場合には、指定されたコマンドが実行されてリカバリーに必要なアーカイブログ・ファイルを取得します。スレーブ・インスタンスがリモート・ホストで稼働している場合にはパスワードを必要としない `scp` コマンド等を使ってファイルをコピーします。

`%p` パラメーターは `{PGDATA}/pg_xlog/RECOVERYXLOG` に展開されます。`RECOVERYXLOG` ファイルはコピーが完了すると、ファイル名が変更され、次いで `{PGDATA}/pg_xlog/archive_status/{WALFILE}.done` ファイルが作成されます。

10.3 フェイルオーバーとスイッヂオーバー

マスター・インスタンスに異常が発生し、スレーブ・インスタンスを昇格させることをフェイルオーバーと呼び、マスター・インスタンスとスレーブ・インスタンスの役割を交換する動作をスイッヂオーバーと呼びます。

10.3.1 スイッヂオーバー

スイッヂオーバーを実施するためには、以下の手順で実行します。

1. マスター・インスタンスを正常停止
2. スレーブ・インスタンスを正常停止
3. 必要であれば両インスタンスのパラメーターを変更
4. 旧スレーブ・インスタンスの recovery.conf ファイルを削除
5. 旧マスター・インスタンスの recovery.conf ファイルを作成
6. 両インスタンスの起動

10.3.2 pg_ctl promote コマンド

スレーブ・インスタンスからマスター・インスタンスへ昇格するためには、スレーブ・インスタンスに対して pg_ctl promote コマンドを実行します。コマンドを実行した直後からスレーブ・インスタンスはマスター・インスタンスとして活動します。下記例ではマスター側で pg_ctl promote コマンドを実行していますが、スタンバイ・モードではないためエラーになっています。スレーブ・インスタンスに対するコマンドの実行では「server promoting」メッセージが出力されて昇格が行われたことがわかります。

例 217 pg_ctl promote コマンドの実行

```
$ pg_ctl -D data.master promote
pg_ctl: cannot promote server; server is not in standby mode
$ echo $?
1
$
$ pg_ctl -D data.slave promote
server promoting
```

スレーブ・インスタンスが昇格した後でも、これまでマスターだったインスタンスはそのまま稼働を続けます。

□ pg_ctl promote コマンドの動作

PostgreSQL のフェイルオーバーは、スレーブ・インスタンスに対する pg_ctl promote コマンドにより実行されます。pg_ctl promote コマンドは以下の処理を実行しています。

1. postmaster のプロセス ID 取得とチェック
 - postmaster.pid ファイルの解析を行うことで取得
2. Single-User サーバーでないかのチェック
3. recovery.conf ファイルの存在チェック
4. {PGDATA}/promote ファイルの作成
5. postmaster プロセスに対して SIGUSR1 シグナルの送信
6. 「server promoting」メッセージ出力

この操作は「src/bin/pg_ctl/pg_ctl.c」ファイルの do_promote 関数内で実行しています。postmaster プロセスは SIGUSR1 を受信すると startup プロセスに対して SIGUSR2 シグナルを送信します。

10.3.3 トリガー・ファイルによるマスター昇格

recovery.conf ファイルの trigger_file パラメーターに指定されたファイルが作成されると、スレーブ・インスタンスはマスターに昇格します。マスター・インスタンスに昇格が完了すると、ファイルは削除されます。ファイルを削除できなかった場合でもマスターへの昇格はエラーになりません。

トリガー・ファイルによるマスター昇格時は以下のログが出力されます。

例 218 トリガー・ファイルによるマスター昇格時のログ

```
LOG: trigger file found: /tmp/trigger.txt
FATAL: terminating walreceiver process due to administrator command
LOG: invalid record length at 0/6000060: wanted 24, got 0
LOG: redo done at 0/6000028
LOG: selected new timeline ID: 2
LOG: archive recovery complete
LOG: MultiXact member wraparound protections are now enabled
LOG: database system is ready to accept connections
LOG: autovacuum launcher started
```

10.3.4 障害発生時のログ

レプリケーション環境で、マスターまたはスレーブ・インスタンスが停止した場合のログ出力について検証しました。

□ スレーブ・インスタンスの停止

スレーブ・インスタンスを smart または fast モードで停止した場合、マスター・インスタンスのログには何も出力されません。スレーブ・インスタンスが異常終了した場合には、以下のログが出力されます。

例 219 スレーブ・インスタンスの異常終了ログ

```
LOG: unexpected EOF on standby connection
```

スレーブ・インスタンスが再起動した場合には以下のログが出力されます（同期レプリケーションの場合のみ）。

例 220 同期レプリケーションの再開ログ

```
LOG: standby "standby_1" is now the synchronous standby with priority 1
```

□ マスター・インスタンスの停止

マスター・インスタンスが停止した場合、スレーブ・インスタンスには以下のログが出力されます。

例 221 マスター・インスタンス停止ログ

```
FATAL: could not send data to WAL stream: server closed the connection  
unexpectedly
```

```
This probably means the server terminated abnormally  
before or while processing the request.
```

マスター・インスタンスが再開すると、以下のログが出力されます。

例 222 レプリケーションの再開ログ

```
LOG: started streaming WAL from primary at 0/5000000 on timeline 1
```

11. ソースコード構造

PostgreSQL はオープンソース・ソフトウェアであるため、ソースコードが公開されています。PostgreSQL のソースコードはほとんどの部分が C 言語で記述されています。ソースコードは PostgreSQL ホームページ (<http://www.postgresql.org/ftp/source/>) からダウンロードすることができます。

11.1 ディレクトリ構造

11.1.1 トップ・ディレクトリ

ダウンロードしたソースコードを展開すると、`postgresql-{VERSION}` ディレクトリが作成されます。作成されたディレクトリは以下のファイルやディレクトリが作成されます。

表 103 トップレベルのディレクトリとファイル

ファイル/ディレクトリ	説明
<code>aclocal.m4</code>	<code>configure</code> 用ファイルの一部
<code>config</code>	<code>configure</code> 用ファイル格納ディレクトリ
<code>config.log</code>	<code>configure</code> 実行ログ
<code>config.status</code>	<code>configure</code> コマンドが生成するスクリプト
<code>configure</code>	<code>configure</code> プログラム本体
<code>configure.in</code>	<code>configure</code> プログラムの雛形
<code>contrib</code>	Contrib モジュール用ディレクトリ
<code>doc</code>	ドキュメント保存用ディレクトリ
<code>src</code>	ソースコード用ディレクトリ
<code>COPYRIGHT</code>	著作権情報
<code>GNUmakefile</code>	トップレベルの Makefile
<code>GNUmakefile.in</code>	Makefile の雛形
<code>HISTORY</code>	リリースノートを表示する URL が記載
<code>INSTALL</code>	インストール方法の概要資料
<code>Makefile</code>	ダミーの Makefile
<code>README</code>	概要説明資料

11.1.2 src ディレクトリ

src ディレクトリは階層構造を持ったディレクトリにソースコードが格納されています。

表 104 src ディレクトリ内の主なディレクトリ

ディレクトリ	説明
backend	バックエンド・プロセス群のソースコード
bin	pg_ctl 等のコマンド類のソースコード
common	共通使用のソースコード
include	ヘッダ・ファイル
interfaces	ECPG, libpq ライブラリのソースコード
makefiles	Makefile
pl	PL/Perl, PL/pgSQL, PL/Python, PL/tcl のソースコード
port	libpgport ライブラリのソースコード
template	各種 OS 用のシェルスクリプト
test	ビルド・テスト
timezone	タイムゾーン関連
tools	ビルド・ツール
tutorial	SQL チュートリアル

11.2 ビルド環境

11.2.1 configure コマンド・パラメータ

サーバー・ログに日本語メッセージを出力する場合には、configure コマンドの--enable-
nl パラメーターを有効にする必要があります。標準では英語メッセージのみが出力されます。

11.2.2 make コマンド・パラメータ

configure コマンドで環境設定を行った後、make コマンドを使ってバイナリのコンパイルやインストールを行います。make コマンドのターゲットとして指定できる項目は以下の通りです。

表 105 make コマンドの主なオプション

ターゲット	説明
指定なし	PostgreSQL バイナリのビルド
world	Contrib モジュール、HTML ドキュメント等もビルド
check	リグレッション・テスト実行
install	PostgreSQL バイナリのインストール
install-docs	HTML, man ドキュメントのインストール
install-world	Contrib モジュール、HTML ドキュメント等もインストール
clean	バイナリのクリア

12. Linux オペレーティング・システム設計

PostgreSQL を稼働させるために Linux 環境で変更が推奨される項目について記述しています。

12.1 カーネル設定

12.1.1 メモリー・オーバーコミット

Red Hat Enterprise Linux は標準でメモリー・オーバーコミットの機能が動作しています。メモリーが不足する状況になった場合にメモリー使用量が多い PostgreSQL インスタンスが強制終了されることを防ぐため、メモリー・オーバーコミットの機能は停止すべきと考えます。

表 106 オーバーコミット設定

カーネル・パラメーター	デフォルト値	推奨値
vm.overcommit_memory	0	2
vm.overcommit_ratio	50	99

12.1.2 I/O スケジューラ

I/O スケジューラは deadline に変更することでパフォーマンスのばらつきが解消したという事例の発表がありました。また SSD ストレージを使ったシステムでは noop に変更することも検討すべきです。

例 223 I/O スケジューラを noop に変更

```
# cat /sys/block/sda/queue/scheduler
noop [deadline] cfq
# grubby --update-kernel=ALL --args="elevator=noop"
```

Red Hat Enterprise Linux 7 ではデフォルトの I/O スケジューラが deadline に変更されたため通常は設定不要になりました。

12.1.3 SWAP

できるだけスワップアウトせずにメモリー内にプロセスを維持するため、カーネル・パラメーター vm.swappiness は 5 以下にすることが推奨されます。

表 107 スワップ設定

パラメーター	デフォルト値	推奨値
vm.swappiness	30	0

12.1.4 Huge Pages

大規模メモリー環境では Huge Pages を利用する設定を行います。PostgreSQL 9.4 以降のデフォルト設定では Huge Pages が利用できれば利用します。カーネル・パラメーター `vm.nr_hugepages` には共有バッファで使用される領域以上のサイズ（2 MB 単位）を指定します。パラメーター `huge_pages=on` の場合、必要量が不足する場合はインスタンス起動エラーになります。

表 108 Huge Pages 設定

パラメーター	デフォルト値	推奨値
<code>vm.nr_hugepages</code>	0	shared_buffers + wal_buffers 以上

12.1.5 セマフォ

同時セッション数が 1,000 を超えるシステムでは、インスタンス起動時に確保するセマフォ・セットが不足する可能性があります。パラメーター `max_connections` を拡大する場合には、カーネル・パラメーター `kernel.sem` を更新してください。

12.2 ファイルシステム設定

PostgreSQL はストレージとしてファイルシステムを使用し、多数の小規模ファイルを自動的に作成します。このためファイルシステムのパフォーマンスがシステムの性能に大きく影響します。Linux 環境では ext4 または XFS（Red Hat Enterprise Linux 7 の標準）が推奨されます。

12.2.1 ext4 使用時

データベース・クラスター用ファイルシステムのマウント・オプションとして `noatime`、`nodiratime` を指定します。

12.2.2 XFS 使用時

データベース・クラスター用ファイルシステムのマウント・オプションとして `nobarrier`、`noatime`、`noexec`、`nodiratime` を指定します。

12.3 Core ファイル

トラブルの解析には障害発生時に生成された core ファイルが役に立ちます。ここでは Red Hat Enterprise Linux のトラブル解析ツールを使って PostgreSQL が core ファイルを生成する設定について記述しています。

12.3.1 Core ファイル出力設定

標準では core ファイルのサイズ制限が 0 になっているため、制限を解除します。

- limits.conf ファイルの編集

/etc/security/limits.conf ファイルに以下のエントリーを追加します。postgres は PostgreSQL インスタンス実行ユーザー名です。

例 224 core ファイル制限

```
postgres - core unlimited
```

- .bashrc ファイルの編集

postgres ユーザーの{HOME}/.bashrc ファイルに以下のエントリーを追加します。

例 225 ユーザー制限

```
ulimit -c unlimited
```

12.3.2 ABRT による Core 管理

Red Hat Enterprise Linux 6 以降にはバグ・レポートに必要な情報を自動的に収集する Auto Bug Reporting Tool (ABRT) が搭載されました。ABRT は標準的なインストールで自動的に動作しています。

- カーネル・パラメーターの設定

ABRT がインストールされた環境では、カーネル・パラメーター kernel.core_pattern が以下の設定に変更されています。

例 226 core_pattern カーネル・パラメーター

```
!/usr/libexec/abrt-hook-ccpp %s %c %p %u %g %t e
```

このため Core ファイルが生成されると ABRT にファイル内容が渡されます。

□ ディレクトリの作成と出力先の設定

Core ファイルは標準では /var/spool/abrt ディレクトリに出力されます。ディレクトリを変更するには /etc/abrt/abrt.conf ファイルの DumpLocation パラメーターを変更します。

例 227 ディレクトリの作成と ABRT 設定

```
# mkdir -p /var/crash/abrt
# chown abrt:abrt /var/crash/abrt
# chmod 755 /var/crash/abrt
# cat /etc/abrt/abrt.conf
DumpLocation = /var/crash/abrt ← 出力ディレクトリ
MaxCrashReportsSize = 0          ← ファイル・サイズの制限なし
```

□ ABRT パッケージ設定

標準では署名されていないプログラムの Core ファイルは生成されません。この制限を解除するためには /etc/abrt/abrt-action-save-package-data.conf ファイルの OpenGPGCheck パラメーターを no に設定します。

例 228 署名されていないプログラムの Core を出力する

```
# cat /etc/abrt/abrt-action-save-package-data.conf
OpenGPGCheck = no
```

□ その他の設定

/etc/abrt/plugins/CCpp.conf ファイルには Core ファイルの生成ルールやフォーマットを指定します。

例 229 その他の Core ファイル設定

```
# cat /etc/abrt/plugins/CCpp.conf
MakeCompatCore = no
SaveBinaryImage = yes
```

ABRT の 詳 細 は 以 下 の URL を 参 照 し て く だ さ い 。

https://access.redhat.com/documentation/ja-JP/Red_Hat_Enterprise_Linux/7/html/SystemAdministrators_Guide/ch-abrt.html

12.4 ユーザー制限

Linux 上の PostgreSQL インスタンスは一般的に Linux ユーザー `postgres` の権限で動作します。Red Hat Enterprise Linux 6 環境で同時接続数が 1,000 を超えるシステムでは `postgres` ユーザーのプロセス制限を拡張します。プロセス数の上限は `/etc/security/limits.conf` ファイルに記述します。

例 230 `limits.conf` ファイル設定

<code>postgres</code>	<code>soft</code>	<code>nproc</code>	<code>1024</code>
<code>postgres</code>	<code>hard</code>	<code>nproc</code>	<code>1024</code>

Red Hat Enterprise Linux 7 ではこの制限のデフォルト値が 4096 に変更されたため、上記の対処は不要になりました。

12.5 `systemd` 対応

Red Hat Enterprise Linux 7 では、オペレーティング・システムのサービス管理に `systemd` が使われるようになりました。Linux ブート時に PostgreSQL インスタンスを自動起動させるためには PostgreSQL データベースを `systemd` に対応させる必要があります。

12.5.1 サービス登録

`systemd` に対応するため、スクリプトを作成し、`systemd` デーモンに登録します。以下の例ではサービス名を `postgresql-9.6.2.service` にしています。`systemctl` コマンドでサービスの登録を行います。

例 231 systemd 登録

```
# vi /usr/lib/systemd/system/postgresql-9.6.2.service
#
# systemctl enable postgresql-9.6.2.service
In           -s          '/usr/lib/systemd/system/postgresql-9.6.2.service'
'/etc/systemd/system/multi-user.target.wants/postgresql-9.6.2.service'
#
# systemctl --system daemon-reload
#
```

例 232 systemd スクリプト

```
[Unit]
Description=PostgreSQL 9.6.2 Database Server
After=syslog.target network.target

[Service]
Type=forking
TimeoutSec=120

User=postgres

Environment=PGDATA=/usr/local/pgsql/data
PIDFile=/usr/local/pgsql/data/postmaster.pid

ExecStart=/usr/local/pgsql/bin/pg_ctl start -D "/usr/local/pgsql/data" -l
"/usr/local/pgsql/data/pg_log/startup.log" -w -t ${TimeoutSec}
ExecStop=/usr/local/pgsql/bin/pg_ctl stop -m fast -w -D "/usr/local/pgsql/data"
ExecReload=/usr/local/pgsql/bin/pg_ctl reload -D "/usr/local/pgsql/data"

[Install]
WantedBy=multi-user.target
```

12.5.2 サービス起動と停止

サービスの起動、停止は `systemctl` コマンドを使って行います。`systemctl` コマンドの詳細はオンライン・マニュアルを参照してください。

表 109 systemctl コマンドによるサービスの制御

操作	コマンド
サービス起動	systemctl start {SERVICENAME}
サービス停止	systemctl stop {SERVICENAME}
サービス状態確認	systemctl status {SERVICENAME}
サービス再起動	systemctl restart {SERVICENAME}
サービス再読み込み	systemctl reload {SERVICENAME}

例 233 systemctl コマンドによるサービスの制御

```
# systemctl start postgresql-9.6.2.service
# systemctl status postgresql-9.6.2.service
postgresql-9.6.2.service - PostgreSQL 9.6.2 Database Server
   Loaded: loaded (/usr/lib/systemd/system/postgresql-9.6.2.service; enabled)
   Active: active (running) since Tue 2017-02-11 12:02:00 JST; 5s ago
     Process: 12655 ExecStop=/usr/local/pgsql/bin/pg_ctl stop -m fast -w -D /home/postgres/data (code=exited, status=1/FAILURE)
     Process: 12661 ExecStart=/usr/local/pgsql/bin/pg_ctl -w start -D /home/postgres/data -l /home/postgres/data/pg_log/startup.log -w -t ${TimeoutSec} (code=exited, status=0/SUCCESS)
   Main PID: 12663 (postgres)
      CGroup: /system.slice/postgresql-9.6.2.service
              + /usr/local/pgsql/bin/postgres -D /home/postgres/data
              + postgres: logger process
              + postgres: checkpointer process
              + postgres: writer process
              + postgres: wal writer process
              + postgres: autovacuum launcher process
              + postgres: archiver process
              + postgres: stats collector process

Feb 11 12:02:00 rel71-2 systemd[1]: Starting PostgreSQL 9.6.2 Database Server...
Feb 11 12:02:00 rel71-2 systemd[1]: PID file /home/postgres/data/postmaster.pid ... t.
Feb 11 12:02:00 rel71-2 pg_ctl[12661]: waiting for server to start... stopped
waiting
Feb 11 12:02:00 rel71-2 pg_ctl[12661]: server is still starting up
Feb 11 12:02:00 rel71-2 systemd[1]: Started PostgreSQL 9.6.1 Database Server.
Hint: Some lines were ellipsized, use -l to show in full.
```

systemctl コマンドは起動したサービスのプロセスの状態管理を行っています。このため systemctl コマンドで起動したインスタンスを pg_ctl コマンドで停止すると、systemd デーモンはサービスが異常終了したと判断します。

12.6 その他

12.6.1 SSH

スロットを使用しないレプリケーション環境では、アーカイブログのギャップを解消するために `recovery.conf` ファイルの `restore_command` パラメーターを使用します。このパラメーターにはプライマリー・インスタンスのアーカイブログ・ファイルをコピーするコマンドを記述します。プライマリ・ホストに対して `scp` コマンドでパスワードなしで接続できるように設定を行います。

12.6.2 Firewall

ファイア・ウォールを使用する場合はローカル TCP ポート 5,432 (パラメータ `port`) に対する接続を許可します。下記の例ではサービス `postgresql` に対する接続を許可しています。

例 234 firewalld 設定

```
# firewall-cmd --permanent --add-service=postgresql  
success
```

12.6.3 SE-Linux

現状では SE-Linux と PostgreSQL の組み合わせについて明確な指針が無いように見えます。Permissive モードまたは Disabled モードに設定することが一般的です。

12.6.4 systemd

Red Hat Enterprise Linux 7 Update 2 ではユーザーがログオフを行うとユーザーが作成した共有メモリーを削除する設定がデフォルトになりました。デフォルト状態では、ログオフと同時に PostgreSQL インスタンスが使用する共有メモリーが削除されるため、この設定を旧バージョンの設定に戻します。`/etc/systemd/logind.conf` ファイルを以下のように修正します。

例 235 logind.conf 設定

```
[login]  
RemoveIPC=no           ← 追加
```

付録. 参考文献

付録 1. 書籍

PostgreSQLについて参考になる書籍の情報です。

表 110 書籍の情報

書籍名	著者（敬称略）	出版社
PostgreSQL 徹底入門 第3版	笠原 辰仁 北川 俊広 坂井 潔 坂本 昌彦 佐藤 友章 石井 達夫（監修）	翔泳社
PostgreSQL 全機能バイブル	鈴木 啓修	技術評論社
内部構造から学ぶ PostgreSQL 設計・運用計画の鉄則	勝俣 智成 佐伯 昌樹 原田 登志	技術評論社
PostgreSQL Replication - Second Edition	Hans-Jurgen Schonig	PACKT
PostgreSQL 9 Administration Cookbook - Second Edition	Simon Riggs Gianni Ciolfi	PACKT
Troubleshooting PostgreSQL	Hans-Jurgen Schonig	PACKT
PostgreSQL for Data Architects	Jayadevan Maymala	PACKT
PostgreSQL Server Programming - Second Edition	Usama Dar Hannu Krosing	PACKT
PostgreSQL Developer's Guide	Ahmed, Ibrar Fayyaz, Asif	PACKT
PostgreSQL Cookbook	Chitij Chauhan	PACKT
PostgreSQL: Up and Running	Regina O. Obe Leo S. Hsu	O'Reilly

付録 2. URL

PostgreSQLについて参考になるURLの情報です。

表 111 参考になる URL

サイト	URL
PostgreSQL 日本語版 ドキュメント	http://www.postgresql.jp/document/
PostgreSQL 公式 ドキュメント	http://www.postgresql.org/docs/
PostgreSQL JDBC Driver	http://jdbc.postgresql.org/
PostgreSQL Internals	http://www.postgresqlinternals.org/index.php
PostgreSQL Deep Dive	http://pgsqldeepdive.blogspot.jp/
オープンソースデータベース標準教科書	http://www.oss-db.jp/ossdbtext/text.shtml
日本 PostgreSQL ユーザー会	https://www.postgresql.jp/
Let's Postgres	http://lets.postgresql.jp/
PostgreSQL エンタープライズ・コンソーシアム	https://www.pgecons.org/
EnterpriseDB	http://www.enterprisedb.com/
Michael Paquier - Open source developer based in Japan	http://michael.otacoo.com/
PostgreSQL は、SELECT もロックを獲得する	http://d.hatena.ne.jp/chiheisen/20100310/1268238033
PostgreSQL SQL チューニング入門 入門編	http://www.slideshare.net/MikiShimogai/sql-42453213
PostgreSQL SQL チューニング入門 実践編	http://www.slideshare.net/satoshiyamada71697/postgresql-sql
より深くオプティマイザとそのチューニング	http://www.slideshare.net/hayamiz/ss-42415350
PostgreSQL のリカバリー超入門	http://www.slideshare.net/suzuki_hironobu/recovery-11
PostgreSQL 9.5 WAL format	https://wiki.postgresql.org/images/a/af/FOSDEM-2015-New-WAL-format.pdf
日々の記録 別館（ぬこ@横浜さん）	http://d.hatena.ne.jp/nuko_yokohama/
PostgreSQL GitHub	https://github.com/postgres/postgres
PostgreSQL Commitfests	https://commitfest.postgresql.org/

変更履歴

変更履歴

版	日付	作成者／更新者	説明
1.0	16-Jul-2014	篠田典良	公開版を作成
1.0.1	04-Aug-2014	篠田典良	誤字修正
1.1	16-May-2015	篠田典良	<p>PostgreSQL 9.4 正式版に対応</p> <p>2.1.2 情報追記</p> <p>2.3.2 パラメーター設定方法を追記</p> <p>2.3.3 外部ライブラリの登録を追記</p> <p>3.1.3 TOAST に関するミスを修正</p> <p>3.3.8 archiver の動作追記</p> <p>3.5.4 pg_control のデータ情報追記</p> <p>3.9.1 ロケール詳細情報追記</p> <p>3.10.3 ログ・ローテーション追記</p> <p>5.1.4 統計情報に関する記述追記</p> <p>5.2.3 autovacuum 情報追記</p> <p>5.3.8 EXPLAIN の説明追記</p> <p>6.2.6 統計情報に説明追加</p> <p>6.3 postgres_fdw を(2)へ移動</p> <p>6.4 PREPARE 文追記</p> <p>12.1.6 セマフォ情報追記</p> <p>12.4 ユーザー制限追記</p> <p>参考文献、参考 URL 更新</p>
1.1.1	18-May-2015	篠田典良	誤字修正
1.2	11-Feb-2017	篠田典良	<p>PostgreSQL 9.6 に対応</p> <p>インストール先を/usr/local/pgsql に統一</p> <p>2.1.5 Microsoft Windows 環境のプロセス構成追加</p> <p>2.3.1/2.3.2/2.3.3 情報追加</p> <p>2.3.7 Windows Server 停止時の動作追加</p> <p>3.3.3 Visibility Map の情報追加</p>

変更履歴（続）

版	日付	作成者／更新者	説明
1.2	11-Feb-2017	篠田典良	<p>3.4.1/3.5.4 情報追加</p> <p>3.5.5 pg_filenode.map 追加</p> <p>3.3.9 LOGGED / UNLOGGED テーブルの切り替え追記</p> <p>3.5.4 pg_control の詳細追記</p> <p>3.5.5 pg_filenode.map を追記</p> <p>3.10.5 ログのエンコード詳細を追記</p> <p>4.1.2 情報追加</p> <p>4.3.2 プロセス異常終了時のトランザクション追加</p> <p>5.2.4 使用メモリー容量追記</p> <p>5.3.2 コストの計算とパラメーターの関係を追加</p> <p>5.3.4 情報追加</p> <p>6.2.7 パーティション・テーブルと外部テーブル追加</p> <p>6.5 INSERT ON CONFLICT 追加</p> <p>6.6 TABLESAMPLE 追加</p> <p>6.9 Parallel Query 追加</p> <p>7.2 Row Level Security 追加</p> <p>8.1.1 pg_basebackup の情報追記</p> <p>8.1.5 pg_rewind 追加</p> <p>8.1.6 vacuumdb 追加</p> <p>12 Red Hat Enterprise Linux 7 対応</p> <p>12.1.6 cstate 設定の記述削除</p> <p>参考書籍を追記</p> <p>参考 URL を追記</p>

以上

