2014 年 7 月 16 日

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した。

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 デリバリー統括本部）にレビューいただきました。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。

2014 年 7 月 16 日

篠田典良

目次

用語集

表 1　略語／用語

略語/用語	説明
ACID 特性	データベースが保持すべき特性（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。テーブルのレコードを一意に既定することもできる。
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

[1] http://www.postgresql.org/docs/9.4/static/contrib.html

表 1（続）　略語／用語

略語/用語	説明
アーカイブログ	リカバリに使用される WAL のコピー
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	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 9.4 beta 1	9.4.0beta
オペレーティング・システム	Red Hat Enterprise Linux 6 Update 4 (x86-64)	

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@hp.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	000000010000000000000A8
{ARCHIVEDIR}	アーカイブログ出力用ディレクトリ	/opt/PostgreSQL/9.4/arch
{BGWORKER}	カスタム Worker プロセス名	custom_worker
{DATE}	日付	2014-06-17
{HOME}	psql コマンド実行ユーザーのホーム・ディレクトリ	/home/postgres
{OID}	任意の OID 番号	12993
{INSTALL}	PostgreSQL インストール・ディレクトリ	/opt/PostgreSQL/9.4
{PGDATA}	データベース・クラスタ用ディレクトリ	/opt/PostgreSQL/9.4/data
{PGDATABASE}	データベース名	demodb
{PGUSER}	接続ユーザー名	demo1
{PID}	プロセス ID	3468
{PORT}	接続待ちポート番号	5432

表 3 表記（続）

表記	説明	例
{RELFILENODE}	テーブルに対応するファイル名、pg_class ビューの relfilenode 列に対応	16531
{SLOT}	レプリケーション・スロット名	slot_1
{SOURCE}	パラメータ設定元のマクロ	
{SQL}	任意の SQL 文	SELECT * FROM table1
{TABLESPACEDIR}	テーブル空間用ディレクトリ	/opt/PostgreSQL/9.4/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	9.4
{WALFILE}	WAL ファイル名	000000010000000000000B0
{WALOFFSET}	WAL オフセット	5225832
{YYYYMMDDN}	フォーマット番号	201305111
${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	備考
#	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>	スタンドアロン・モードのプロンプト	

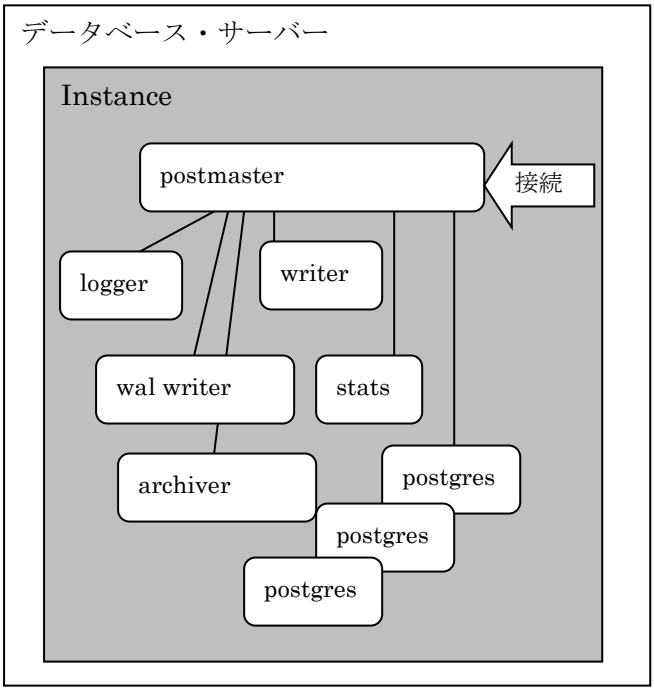
2. プロセスとメモリー

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、postmaster[2]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。その他のファイルはすべて 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 /opt/PostgreSQL/9.4/bin/postgres -D
/opt/PostgreSQL/9.4/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 シグナルを受け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 2 異常終了後の再起動ログ

```
LOG:  database system was interrupted; last known up at 2014-05-28 10:18:07 JST
LOG:  database system was not properly shut down; automatic recovery in progress
LOG:  record with zero length at 0/1715178
LOG:  redo is not required
LOG:  database system is ready to accept connections
LOG:  autovacuum launcher started
```

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。

□ 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 5　autovacuum launcher プロセスの動作

シグナル	ハンドラー	動作
SIGHUP	avl_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	リカバリ処理
SIGUSR2	avl_sigusr2_handler	自動 Vacuum 処理失敗のリカバリ
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 6　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 7　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	
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 8　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	
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。ただしこの方法はパラメータ log_checkpoints を on に指定してもログが出力されま

せん。pg_stat_bgwriter ビューは更新されます。

□ 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 9 stas 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 10　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	handle_sig_alarm	タイムアウト発生通知
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 3 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 11　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 12　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 13 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	

2.1.3 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 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 14 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}
wal receiver	postgres: wal receiver process streaming {WALFILE}
startup process	postgres: startup process recovering {WALFILE}
bgworker	postgres: bgworker: {BGWORKER}

2.1.4 プロセスの起動と停止

checkpointer、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 15　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 logging_collector = on の場合に起動（デフォルト off）
autovacuum launcher	パラメータ 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 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	パラメータ archive_mode = on の場合に起動（デフォルト off）
postgres (local)	クライアントの接続時に起動、切断時に停止
postgres (remote)	クライアントの接続時に起動、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側で起動。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
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

□ 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 autovacuum_max_workers（デフォルト値 3）で決まります。複数の worker プロセスが単一のデータベースの VACUUM 処理を行うことはありません。このため多数のデータベースが構成されたインスタンスではパラメータ autovacuum_max_workers の値を拡大しないと VACUUM 処理が全データベースで並列に行われない可能性があります。

□ 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 max_connections（デフォルト値 100）に制限されます。
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 4 一般ユーザーの接続数超過

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

例 5 パラメータ max_connections で指定された接続数超過

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

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 IPC 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 6 共有バッファの状況

```
$ 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が、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

```
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、共有バッファのキーは 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、メモリーマップドファイル（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 Page

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 Page を利用することができます。Huge Page 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 Page を使用する場合のページ・サイズは 2 MB です（2 * 1024 * 1024 バイト）。 Huge Page 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 が指定されます。

□ パラメータ設定

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

表 16　パラメータ huge_pages に指定できる値

パラメータ値	説明	備考
on	Huge Page を使用する	
off	Huge Page を使用しない	
try	Huge Page 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、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。このパラメータを on に指定すると強制的に Huge Page を使用します。プラットフォームが Huge Page 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。

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

□　Huge Page の設定方法

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

例 7 Linux の Huge Page 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 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 8 Huge Page ページ不足エラー

```
$ pg_ctl -D data start
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.
```

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.  */
```

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の使用されています。PostgreSQL ではインスタンス起動時に以下のパラメータから計算された数のセマフォ集合が作成されます。

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 9 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 ）。 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

PostgreSQL はメモリーを各種データのキャッシュとして使用します。メモリー上のデータには永続性が無いため、メモリー上で更新されたページはストレージに書き込まれる必要があります。メモリーとストレージを同期し、永続化を保障する点またはこの同期処理を開始することをチェックポイントと呼びます。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

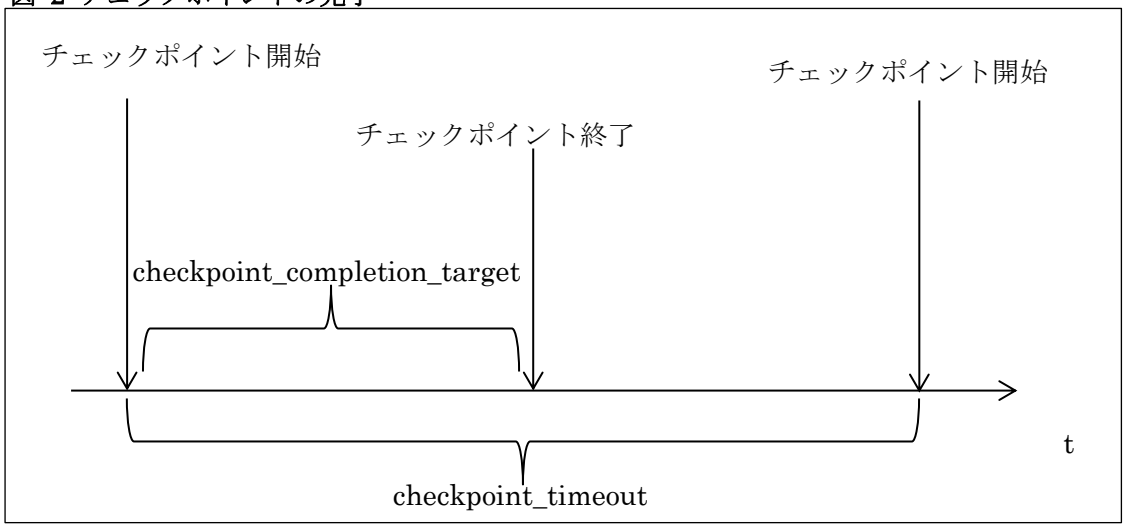
管理者が CHECKPOINT 文を実行した場合。

- パラメータ 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 ファイル数がパラメーターcheckpoint_segments に達した場合
  16 MB の WAL ファイルがパラメーターで指定された数（デフォルト 3）だけ書き込まれた場合。
- オンライン・バックアップ開始時
  pg_start_backup 関数実行時
- 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

□ チェックポイントの完了

チェックポイントには種類が 2 つあります。一定時間間隔や WAL ファイルの数により発生する 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 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 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 2 チェックポイントの完了



書込みが必要なブロック数に対する書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 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

チェックポイントに関するパラメータは以下の通りです。

表 17 チェックポイントに関係するパラメータ

パラメータ	説明	デフォルト値
checkpoint_timeout	チェックポイント間隔	300sec
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計算	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

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 IN 文による一括ロードが行われると、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の一部を循環して使用するリング・バッファを使用します。これら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 18 循環バッファのサイズ

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

実際に作成される循環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）。

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時の動作、入出力ファイルおよび共有ライブラリの利用状況をまとめています。

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／待機を行わない-W パラメータ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 パラメータがデフォルトで、インスタンスの停止時は-w パラメータがデフォルトです（http://www.postgresql.org/docs/9.4/static/app-pg-ctl.html）。

表 19　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

インスタンス起動時は-w パラメータ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 10 インスタンス起動失敗時の動作

```
$ pg_ctl -D data start
server starting
2014-06-13 10:16:26 JST  00000 LOG:  redirecting log output to logging collector
process
  2014-06-13 10:16:26 JST  00000 HINT:  Future log output will appear in directory
"pg_log".
$ pg_ctl -D data start  ← 同じクラスタに対して２回起動（エラーになる）
pg_ctl: another server might be running; trying to start server anyway
server starting
2014-06-13 10:16:36 JST  F0001 FATAL:  lock file "postmaster.pid" already exists
2014-06-13 10:16:36 JST  F0001 HINT:  Is another postmaster (PID 3950) running in
data directory "/opt/PostgreSQL/9.4/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（デフォルト）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ていてもインスタンスは停止できます。

例 11 レプリケーション時の -m smart パラメータ

```
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
```

2.3.2 パラメータの設定

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。

パラメータの一覧を取得するにはは pg_settings ビューを検索するか、psql ユーティリティから show all コマンドを実行します。pg_settings ビューの source 列は、パラメータ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 20　pg_settings ビューの 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	
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 コマンドによる変更	

□ パラメータの動的変更

PostgreSQL 9.4 からは ALTER SYSTEM 文により、パラメータ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 文で変更したパラメータの値は「{PGDATA}/postgresql.auto.conf」ファイルに記載されます。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 12　ALTER SYSTEM 文によるパラメータ変更

```
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 文のパラメータ値として DEFAULT を指定すると、postgresql.auto.conf ファイルからパラメータが削除されます。

例 13　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.
$
```

2.3.3 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 14 インスタンス終了タイムアウト

```
$ pg_ctl -D /opt/PostgreSQL/9.4/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 demo
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
$
```

2.3.4 インスタンス起動時のライブラリ

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 21　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	備考
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	

2.3.5 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。またパラメータ等はデフォルト値を使用しています。

表 22　入出力ファイル

ファイル	パス	備考
postgresql.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	
pg_internal.init	{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	

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

データベース・クラスタは、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、インスタンスの起動単位にもなります。

例 15 データベース・クラスタ内のファイル構造

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

データベース・クラスタ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 23　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	備考
{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	
pg_filenode.map	pg_class.filerelnode に対応する。オブジェクトとファイルの対応を定義する	
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、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。	

□ テーブルの特定

テーブルやインデックスは pg_class ビューの relfilenode 列の値が、オペレーティング・システムのファイルと対応しています。これらの関係は 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 テーブルの 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 16 ファイルの特定

```
$ oid2name -d demodb
From database "demodb":
  Filenode  Table Name
----------------------
     16437       demo1

postgres=> SELECT relname, relfilenode, reltablespace FROM pg_class
    WHERE relname IN ('demo2', 'demo3') ;
 relname | relfilenode | reltablespace
---------+-------------+---------------
 demo2   |       34115 |             0          <- テーブル空間 pg_default
 demo3   |       34119 |         32778          <- テーブル空間 tbl2
```

□　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 17 セグメント・ファイル

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

$ ls -l 16495*
-rw-------. 1 postgres postgres 1073741824 Nov 29 14:06 16495
-rw-------. 1 postgres postgres   96550912 Nov 29 14:06 16495.1
```

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

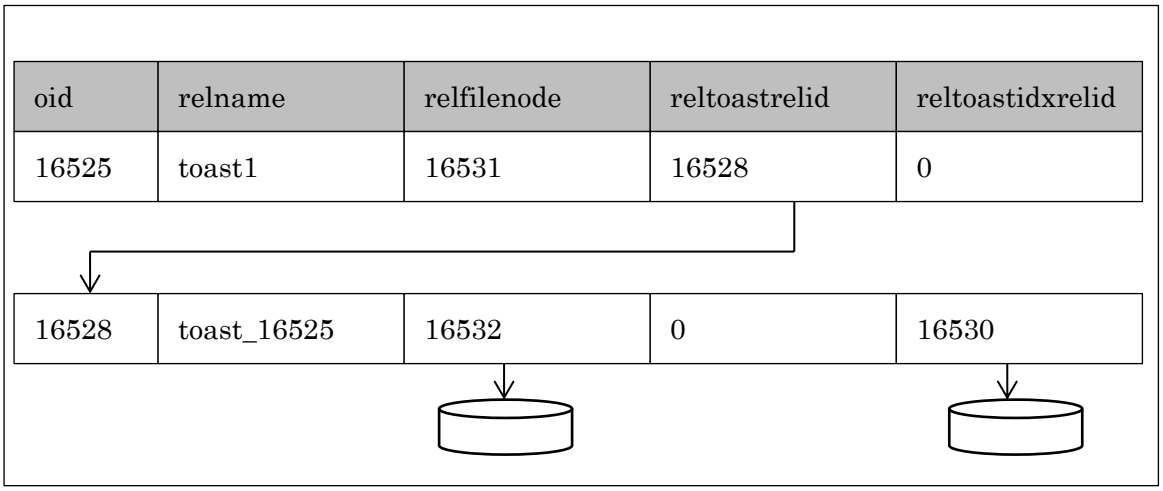
たがって格納されることはありません。このため大規模なレコードはページに含めることができません。より大規模なレコードを格納するために TOAST (The Oversized-Attribute Storage Technique)と呼ばれる機能が提供されています。TOAST データは圧縮済の列データが TOAST_TUPLE_THRESHOLD（コンパイル時に決定）で決められたサイズを超える場合に作成されます。また TOAST_TUPLE_TARGET 以下に縮小されるまで TOAST テーブルにデータを格納します。

□ TOAST テーブル

TOAST データは pg_class ビューの 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 ビューの reltoastrelid 列には、TOAST テーブルの oid が保存されます。pg_class ビューから TOAST テーブルのファイル名（relfilenode）および TOAST インデックスのファイル名（reltoastidxnode）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 テーブルは pg_tables ビューや pg_indexes ビューには表示されません。

図 3 はテーブルと TOAST テーブル、TOAST インデックスの関係を表しています。テーブル toast1 を作成しすると、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 16530 ファイルに格納されます。

図 3 pg_class ビューと 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 24　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 18 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 19 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 ビューの relfilenode 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 20 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 Jul 23 11:23 25782
-rw------- 1 postgres postgres  24576 Jul 23 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 Jul 23 11:25 25782      ←  旧ファイル
-rw------- 1 postgres postgres 0 Jul 23 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 Jul 23 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 の値をでデフォルト値から下げることを推奨します。

□ 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 ビューの reloptions 列を確認します。

例 21 FILLFACTOR の設定

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

postgres=>
```

□ 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 22 FILLFACTOR の設定

```
postgres=# INSERT INTO fill1 VALUES (generate_series(1, 1000), 'data') ;
INSERT 0 1000
postgres=# SELECT MAX(lp) 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(lp) 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_items[3]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。

[3] 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[4]に格納されます。データベース・クラスタを作成すると、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の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 テーブル空間が指定されます。

□ パラメータ default_tablespace

パラメータ 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。

このパラメータのデフォルト値は''（空文字列）で、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。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 25　テーブル空間の指定をしなかった場合の保存先

オブジェクト	パラメータ default_tablespace の指定	
	パラメータ指定あり	パラメータ指定なし（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[5]	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

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

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

SET 文を使ってセッション単位で指定した場合、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 ファイルでインスタンス単位で指定した場合、指定されたテーブル空間が存在するか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 23 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
            Table "public.data6"
 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
postgres=> 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.
↑ SET 文によるパラメータ default_tablespace 変更時はチェックが行われる。
```

□ パラメータ 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 random 関数で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

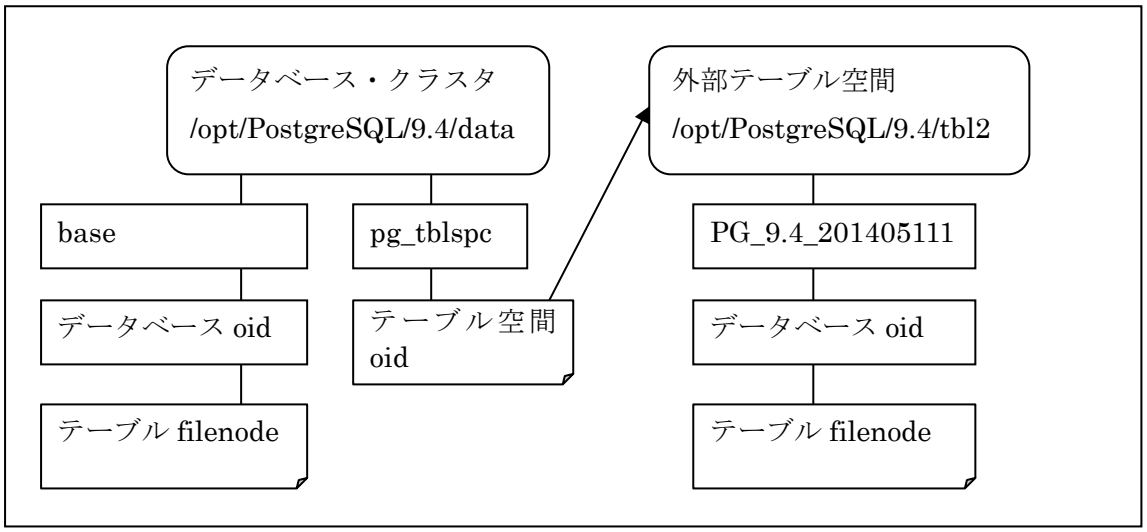
PostgreSQL ではデータベースやテーブル等のオブジェクトはオペレーティング・システムのディレクトリやファイルと対応しています。

□　テーブル空間（TABLESPACE）の特定

pg_default テーブル空間はデータベース・クラスタ内の 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 ビューの oid 列に対応する名前です。

図 4 ディレクトリとテーブル空間



例 24 テーブル空間の対応

```
postgres=# CREATE TABLESPACE tbl2 LOCATION '/opt/PostgreSQL/9.4/tbl2' ;
CREATE TABLESPACE
postgres=# SELECT oid, spcname FROM pg_tablespace ;
  oid  |  spcname
-------+------------
  1663 | pg_default
  1664 | pg_global
 32788 | tbl2
(3 rows)

$ ls -l /opt/PostgreSQL/9.4/data/pg_tblspace
total 0
lrwxrwxrwx 1 postgres postgres 26 Jun 23 11:15 32788 -> /opt/PostgreSQL/9.4/tbl2
$
```

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 25 テーブル空間の内部

```
$ ls -l /opt/PostgreSQL/9.4/tbl2
total 4
drwx------ 3 postgres postgres 4096 Jun 23 11:16 PG_9.4_201405111
$
```

□　データベースの特定

データベースにはデータベース・クラスタ全体で一意の ID（oid）が付与されます。この oid は pg_database ビューの oid 列として確認することができます（または pg_stat_database ビューの 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 26 データベースの対応

```
postgres=# SELECT oid, datname FROM pg_database ;
  oid  |  datname
-------+-----------
     1 | template1
 12783 | template0
 12788 | postgres
 16385 | demodb
(4 rows)

$ oid2name
All databases:
    Oid  Database Name  Tablespace
----------------------------------
  12788       postgres  pg_default
  16385         demodb  pg_default
  12783      template0  pg_default
      1      template1  pg_default

$ ls -l /opt/PostgreSQL/9.4/data/base
total 32
drwx------ 2 postgres postgres 12288 Apr 16 12:59 1
drwx------ 2 postgres postgres  4096 Apr 16 12:58 12783
drwx------ 2 postgres postgres  4096 Apr 23 11:09 12788
drwx------ 2 postgres postgres 12288 Apr 23 11:09 16385
$
```

□　オブジェクト名からファイルを特定

pg_class ビューを検索する以外に、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 27 オブジェクトとファイルの対応

```
postgres=> CREATE TABLE demo1(c1 NUMERIC, c2 CHAR(10)) ;
CREATE TABLE
postgres=> SELECT pg_relation_filepath('public.demo1') ;
 pg_relation_filepath
----------------------
 base/16394/16447
(1 row)
postgres=> CREATE TABLE demo2 (c1 NUMERIC, c2 CHAR(10)) TABLESPACE ts1 ;
CREATE TABLE
postgres=> SELECT pg_relation_filepath('public.demo2') ;
             pg_relation_filepath
-----------------------------------------------
 pg_tblspc/16453/PG_9.4_201405111/16394/16454
(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・クラスタの保護モード

データベース・クラスタ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 28 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
server starting
FATAL:  data directory "/opt/PostgreSQL/9.4/data" has group or world access
DETAIL:  Permissions should be u=rwx (0700).
```

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 29 保護モードの変更

```
$ mkdir data1
$ ls -ld data1
drwxrwxr-x 2 postgres postgres May 28 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 May 28 12:59 data1
$
$ mkdir ts1
$ ls -ld ts1
drwxr-xr-x. 2 postgres postgres 4096 May 28 12:59 ts1
$ psql
postgres=# CREATE TABLESPACE ts1 LOCATION '/opt/PostgreSQL/9.4/ts1' ;
CREATE TABLESPACE
postgres=# ¥q
$ ls -ld ts1
drwx------. 3 postgres postgres 4096 May 28 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エラーにはなりません。

例 30 テーブル空間の保護モード変更

```
postgres=# CREATE TABLESPACE ts1 LOCATION '/opt/PostgreSQL/9.4/ts1' ;
CREATE TABLESPACE
postgres=# ¥q
$ ls -ld ts1
drwx------. 3 postgres postgres 4096 May 28 12:59 ts1
$ chmod a+r ts1
drwxr--r--. 3 postgres postgres 4096 May 28 12:59 ts1
$ pg_ctl -D data restart  -m fast
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 ビューの relfilenode 列で確認できます。

例 31 テーブルの作成とファイル

```
postgres=> CREATE TABLE demo1(c1 varchar(10), c2 varchar(10)) ;
CREATE TABLE
postgres=> SELECT relfilenode FROM pg_class WHERE relname='demo1' ;
relfilenode
-------------
       16446
(1 row)

$ cd data/base/16424/
$ ls -l 16446
-rw------- 1 postgres postgres 0 Apr 24 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 32 チェックポイント前のファイル

```
postgres=> INSERT INTO demo1 VALUES('ABC', '123') ;
INSERT 01
postgres=> ¥q
$ ls -l 16446
-rw------- 1 postgres postgres 0 Apr 24 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 33 チェックポイント後のファイル

```
postgres=# CHECKPOINT ;
CHECKPOINT
postgres=#

$ ls -l 16446
-rw------- 1 postgres postgres 8192 Apr 24 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 34 UPDATE の実行とファイルの状況

```
postgres=> UPDATE demo1 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 nul nul nul nul nul nul nul nul nul nul nul nul nul nul
*
0017700  ff dc4 etx nul nul nul nul nul nul nul nul nul nul nul 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 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、単一（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

Visibility Map はガベージが存在する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 1 ビットで管理します。ファイル名は「{RELFILENODE}_vm」です。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

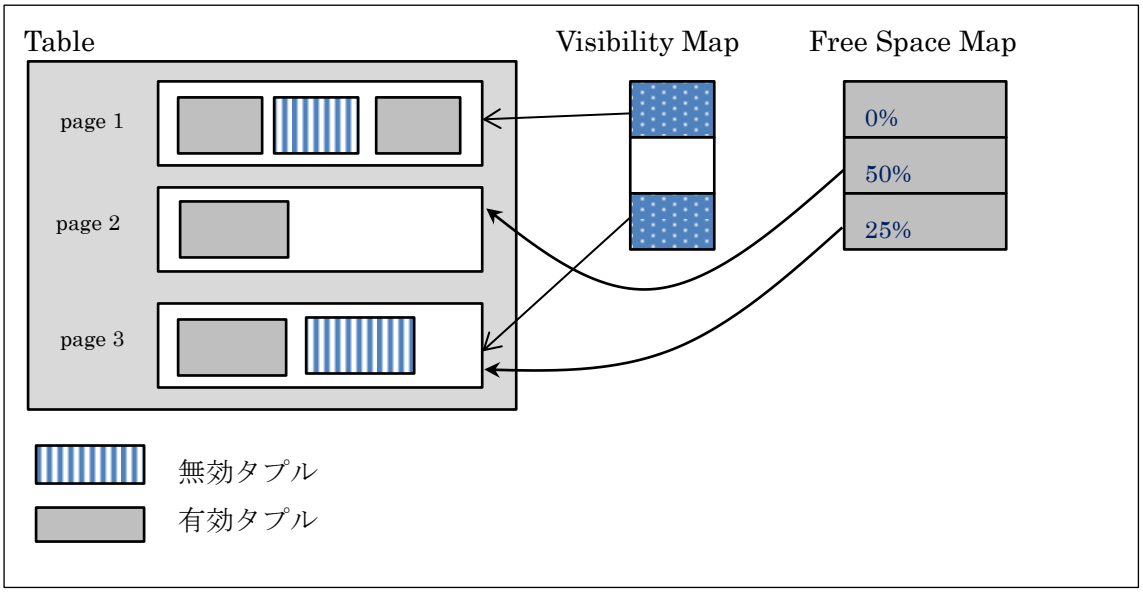
キップし、VACUUM 処理の I/O 負荷を削減することができます。初期サイズは 8 KB です。テーブル作成後、最初のチェックポイントまたは VACUUM 時に作成されます。

□ 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 5 Visibility Map と Free Space Map



例 35 Visibility Map と Free Space Map

```
postgres=> SELECT relname, relfilenode FROM pg_class
        WHERE relname='demo1' ;
relname  | relfilenode
---------+-------------
 demo1   |       16409
(1 row)

$ cd data/base/16385
$ ls 16409*
-rw------- 1 postgres postgres  8192 Jun  6 16:46 16409 ← Table
-rw------- 1 postgres postgres 24576 Jun  6 16:46 16409_fsm ← Free Space Map
-rw------- 1 postgres postgres  8192 Jun  6 16:46 16409_vm ← Visibility Map
$
```

3.3.4 VACUUM 動作

ここでは VACUUM 処理により、ファイルの内容がどのように変化するかを確認します。

□ VACUUM CONCURRENT

VACUUM CONCURRENT 処理は、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 36 データ準備（12 件挿入）

```
postgres=> CREATE TABLE demo1 (c1 CHAR(500) NOT NULL, c2 CHAR(500) NOT NULL) ;
CREATE TABLE
postgres=> INSERT INTO demo1 VALUES ('AAA', '111') ;
postgres=> INSERT INTO demo1 VALUES ('BBB', '222') ;
postgres=> INSERT INTO demo1 VALUES ('CCC', '333') ;
postgres=> INSERT INTO demo1 VALUES ('DDD', '444') ;
postgres=> INSERT INTO demo1 VALUES ('EEE', '555') ;
postgres=> INSERT INTO demo1 VALUES ('FFF', '666') ;
postgres=> INSERT INTO demo1 VALUES ('GGG', '777') ;
postgres=> INSERT INTO demo1 VALUES ('HHH', '888') ;
postgres=> INSERT INTO demo1 VALUES ('III', '999') ;
postgres=> INSERT INTO demo1 VALUES ('JJJ', '000') ;
postgres=> INSERT INTO demo1 VALUES ('AAA', 'aaa') ;
postgres=> INSERT INTO demo1 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 37 データ準備（ファイルの特定）

```
$ oid2name -d demodb
From database "demodb":
  Filenode  Table Name
----------------------
     16470       demo1

$ cd /opt/PostgreSQL/9.4/data/base/16424
$ ls -l 16470
-rw------- 1 postgres postgres 16384 Apr 26 10:56 16470
```

例 38 ブロック初期状態（第１ブロック）

```
0000000 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
* 略
0002720  sp  sp  sp  sp  sp  sp  sp  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
* 略
0005760   `bel nul nul   E   E   E  sp  sp  sp  sp  sp  sp  sp  sp  sp
* 略
0006740  sp  sp  sp  sp  sp  sp  sp  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
* 略
0012000   `bel nul nul   C   C   C  sp  sp  sp  sp  sp  sp  sp  sp  sp
* 略
0012760  sp  sp  sp  sp  sp  sp  sp  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
* 略
0016020   `bel nul nul   A   A   A  sp  sp  sp  sp  sp  sp  sp  sp  sp
* 略
0017000  sp  sp  sp  sp  sp  sp  sp  sp   `bel nul nul   1   1   1  sp
*
```

例 39 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 nul nul nul nul nul nul nul nul nul nul nul nul 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  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  sp
* 略
0032000   `bel nul nul   J   J   J  sp  sp  sp  sp  sp  sp  sp  sp  sp
* 略
0032760  sp  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  sp
* 略
0036020   `bel nul nul   H   H   H  sp  sp  sp  sp  sp  sp  sp  sp  sp
* 略
0037000  sp  sp  sp  sp  sp  sp  sp  sp   `bel nul nul   8   8   8  sp
0037020  sp  sp  sp  sp  sp  sp  sp  sp  sp  sp  sp  sp  sp  sp  sp  sp
*
0040000
```

各ブロックの中間レコードを削除し、VACUUM 処理を実行します。

例 40 レコード削除と VACUUM 実行

```
postgres=> DELETE FROM demo1 WHERE c1 IN ('CCC', 'JJJ') ;
DELETE 2
postgres=# CHECKPOINT ;
CHECKPOINT
postgres=> VACUUM demo1 ;
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開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。

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 41 VACUUM 処理後（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  sp  sp  sp  sp  sp  sp  sp  sp
*略
0002720  sp  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  sp
*略
0005760   `bel nul nul   F   F   F  sp  sp  sp  sp  sp  sp  sp  sp  sp
*略
0006740  sp  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  sp  sp
*略
0012000   `bel nul nul   D   D   D  sp  sp  sp  sp  sp  sp  sp  sp  sp
*略
0012760  sp  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  sp  sp
*略
0016020   `bel nul nul   A   A   A  sp  sp  sp  sp  sp  sp  sp  sp  sp
*略
0017000  sp  sp  sp  sp  sp  sp  sp  sp   `bel nul nul   1   1   1  sp
*
```

例 42 VACUUM 処理後（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
*略
0026740  sp  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  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  sp
*略
0036020   `bel nul nul   H   H   H  sp  sp  sp  sp  sp  sp  sp  sp  sp
*略
0037000  sp  sp  sp  sp  sp  sp  sp  sp   `bel nul nul   8   8   8  sp
*
0040000
```

□ VACUUM 後の空間利用

VACUUM 処理で空き領域を作成できたため、データを格納します。

例 43 VACUUM 処理（データ格納）

```
postgres=> INSERT INTO demo1 VALUES ('MMM', 'ccc') ;
INSERT 0 1
postgres=# CHECKPOINT ;
CHECKPOINT
```

例 44 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  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  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 45 VACUUM 処理（VACUUM FULL 実行）

```
$ ls -li 16470  ← VACUUM FULL 前のファイル
558969 -rw------- 1 postgres postgres 16384 Apr 26 11:39 16470

$ oid2name -d demodb
From database "demodb":
  Filenode  Table Name
----------------------
     16476       demo1  ← 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 Apr 26 11:47 16476
```

□ 一括更新と部分更新

1,000 レコードのテーブルを一括で UPDATE 行う場合と、単一レコードを 1,000 回 UPDATE を行う場合では、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 46 更新方法によるブロック数の差異

```
postgres=> CREATE TABLE demo1(c1 NUMERIC, c2 VARCHAR(100), c3 VARCHAR(100)) ;
CREATE TABLE
-- insert 1000 records
postgres=> SELECT relpages, reltuples FROM pg_class WHERE relname='demo1' ;
 relpages | reltuples
----------+-----------
        8 |      1000
(1 row)
postgres=> UPDATE demo1 SET c1=c1+1 ;   ← 一括更新
UPDATE 1000
postgres=> SELECT relpages, reltuples FROM pg_class where relname='demo1' ;
 relpages | reltuples
----------+-----------
       15 |      1000   ←  ブロック数が増えている
(1 row)
postgres=> TRUNCATE TABLE demo1 ;
-- insert 1000 records
postgres=> UPDATE demo1 SET C2='TEST' WHERE c1=100 ; -- 1,000 回実行
UPDATE 1
postgres=> SELECT relpages, reltuples FROM pg_class WHERE relname='demo1' ;
 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ログファイルと autovacuum launcher プロセスが pg_database ビューに対応するファイルのみオープンします。

表 26　オープン・ファイル

プロセス	オブジェクト／ファイル	備考
postmaster	/tmp/.s.PGSQL.{PORT}	
logger process	{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、サーバー・プロセス（postgres）が、pg_am ビューをオープンします。またユーザーの接続が契機か不明ですが、checkpointer プロセスがカレントの WAL ファイルをオープンします。

表 27　追加オープン・ファイル

プロセス	オブジェクト／ファイル	備考
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、サーバー・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 28　追加オープン・ファイル

プロセス	オブジェクト／ファイル	備考
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、サーバー・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 29　オープン・ファイル

プロセス	オブジェクト／ファイル	備考
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は十分な検証を行っていません。

□ パラメータ 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 47 postgres プロセスが発行するシステムコール

```
 1: recvfrom(10, "Q¥0¥0¥0,INSERT INTO data1 values (1"..., 8192, 0, NULL, NULL) = 45
 2: open("base/16394/91338_fsm", O_RDWR)    = -1
 3: open("base/16394/91338", O_RDWR)        = 16
 4: lseek(16, 0, SEEK_END)                  = 0
 5: lseek(16, 0, SEEK_END)                  = 0
 6: kill(7487, SIGUSR1)                     = 0
 7: write(16, "¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0¥0"..., 8192) = 8192
 8: open("pg_xlog/000000010000000D000000DA", O_RDWR) = 17
 9: lseek(17, 6381568, SEEK_SET)            = 6381568
10: write(17, "u¥320¥5¥0¥1¥0¥0¥0¥0`a¥332¥r¥0¥0005¥4¥0¥0¥0¥0¥0¥0"..., 8192) =  8192
11: fdatasync(17)                           = 0
12: sendto(9, "¥2¥0¥0¥0¥230¥2¥0¥0¥n@¥0¥0¥6¥0¥0¥0¥0¥0¥0¥0"..., 664, 0, NULL, 0) = 664
13: sendto(10, "C¥0¥0¥0¥17INSERT 0 1¥0Z¥0¥0¥0¥5I", 22, 0, NULL, 0) = 22
```

表 30　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) へシグナル送信
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 synchronous_commit = off の場合

パラメータ 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 48 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"..., 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"..., 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"..., 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 49 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/000000010000000100000017", O_RDWR) = 5
write(5, "}¥320¥6¥0¥1¥0¥¥27¥1¥0¥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（7～10 行目）。

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、ダーティ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 50 checkpointer による書き込み処理（一部）

```
pen("pg_clog/0000", O_RDWR|O_CREAT, 0600) = 5
lseek(5, 0, SEEK_SET)                     = 0
write(5, "@UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU"..., 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"..., 8192) = 8192
fdatasync(14)                             = 0
open("global/pg_control", O_RDWR)         = 11
write(11, 02x¥322cuS¥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 プロセスが一定の間隔でチェックポイントによる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の発生による 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 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 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 / backend/port/pg_latch.c）。

書き込みブロック数の最大値は、パラメータ 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 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 bgwriter_lru_maxpages を超えることはありません。パラメータ 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 31 writer プロセスに関係するパラメータ

パラメータ	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

3.4 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（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 51 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 52 pg_start_backup 関数実行時の backup_label ファイル

```
$ cat data/backup_label
START WAL LOCATION: 0/59000028 (file 000000010000000000000059)
CHECKPOINT LOCATION: 0/59000028
BACKUP METHOD: pg_start_backup
BACKUP FROM: master
START TIME: 2014-05-28 13:36:25 JST
LABEL: 2014-05-28 13:36:25.256729+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 53 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 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 54 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)
```

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・バックアップの終了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。

ラベル・ファイルのフォーマット

```
{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。

例 55 backup_label ファイルが残った状態でインスタンス起動

```
2013-05-28 11:47:57 JST   LOG:  database system was shut down at 2013-05-28 11:47:26
JST
2013-05-28 11:47:57 JST   FATAL:  invalid data in file "backup_label"
2013-05-28 11:47:57 JST   LOG:  startup process (PID 9240) exited with exit code 1
2013-05-28 11:47:57 JST   LOG:  aborting startup due to startup process failure
```

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 32　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	
8	STOP TIME:	終了時刻	

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 56 ラベル・ファイル

```
START WAL LOCATION: 0/5B000028 (file 000000010000000000000005B)
STOP WAL LOCATION: 0/5B0000B8 (file 000000010000000000000005B)
CHECKPOINT LOCATION: 0/5B000028
BACKUP METHOD: pg_start_backup
BACKUP FROM: master
START TIME: 2014-05-28 13:39:39 JST
LABEL: 2014-05-28 13:39:38.764743+09
STOP TIME: 2014-05-28 13:39:46 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 57 スレーブ・インスタンスでオンライン・バックアップ

```
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 58 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･バックアップ中に､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 59 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
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....  2014-06-13 12:37:27 JST  00000
DEBUG:  logger shutting 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ように書かれていますが、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。これらの定義はソースコード（src/include/miscadmin.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 33　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 に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。

□ パラメータ external_pid_file

パラメータ 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 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。

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がそのまま出力されます。

例 60 postmaster.opts ファイルの内容

```
$ pg_ctl start -D data
$ cat data/postmaster.opts
/opt/PostgreSQL/9.4/bin/postgres "-D" "data"
$ pg_ctl stop
$ pg_ctl start
$ cat data/postmaster.opts
/opt/PostgreSQL/9.4/bin/postgres
$
```

3.5.3 PG_VERSION

PG_VERSION ファイルは、データベース・クラスタ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 61 PG_VERSION ファイルの内容

```
$ cd /opt/PostgreSQL/9.4/data
$ cat PG_VERSION
9.4
$
```

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 コマンドで確認することができます。

例 62 pg_controldata コマンドの実行

```
$ pg_controldata data
pg_control version number:            937
Catalog version number:               201405111
Database system identifier:           6013822633043442764
Database cluster state:               shut down
pg_control last modified:             Wed 02 Jul 2014 05:25:25 PM JST
Latest checkpoint location:           2/B1000028
Prior checkpoint location:            2/B0000028
Latest checkpoint's REDO location:    2/B1000028
Latest checkpoint's REDO WAL file:    000000010000000200000B1
Latest checkpoint's TimeLineID:       1
<<途中省略>>
Latest checkpoint's oldestMulti's DB: 1
Time of latest checkpoint:            Wed 02 Jul 2014 05:25:25 PM JST
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
Current wal_level setting:            archive
<<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
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_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・クラスタの状態が出力されます。

表 34　Database cluster state

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？

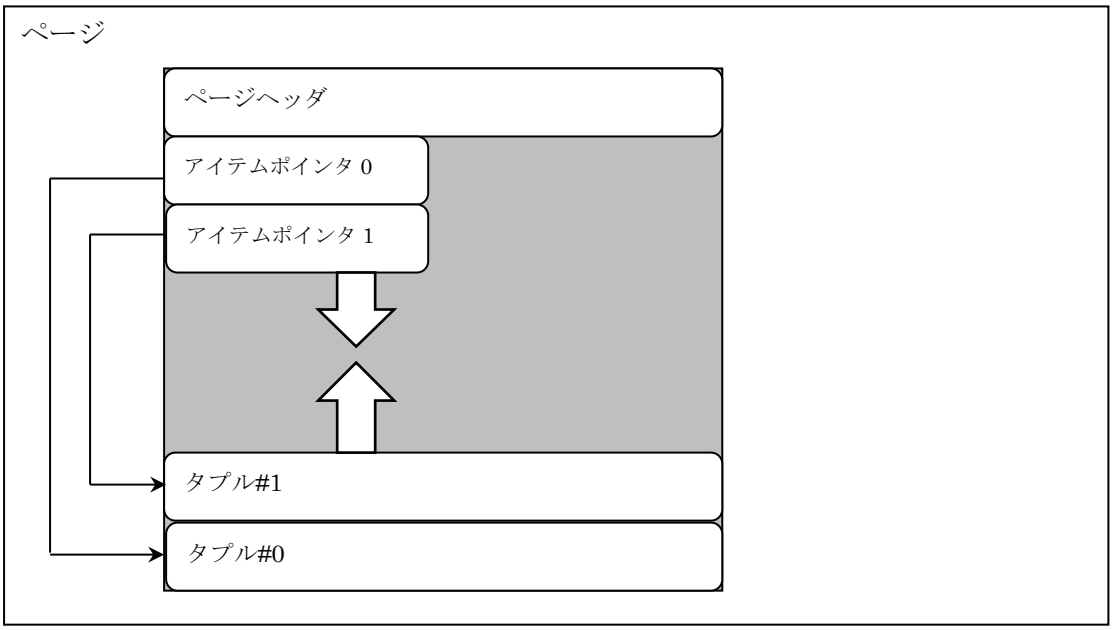
3.6 ブロックのフォーマット

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 ではブロック=ページになるため、同じ意味と考えることができます。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 KB * 131,072 = 1 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 6 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 63 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[1];     /* beginning of line pointer array */
} PageHeaderData;
```

表 35　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。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 64 HeapTupleHeaderData

```
struct HeapTupleHeaderData
{
        union
        {
                HeapTupleFields t_heap;
                DatumTupleFields t_datum;
        }                       t_choice;
        ItemPointerData t_ctid;       /* current TID of this or newer tuple */
        /* 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[1];      /* bitmap of NULLs -- VARIABLE LENGTH */
        /* MORE DATA FOLLOWS AT END OF STRUCT */
};
```

例 65 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 36　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 は削除されたタプルのトランザクション ID です。このため有効なタプルの 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 66 タプルのトランザクション ID

```
postgres=> SELECT XMAX, XMIN, c1 FROM demo1 ;
  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 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 VACUUM 処理が発生します。

自動 VACUUM は対象ブロックを Visibility Map を使って発見しますが、更新されなかったブロックに古いトランザクション ID が残っていても気づかないことになります。このため、パラメータ 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 を無視して全ブロックがチェックされます。

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 ビューの relfrozenxid 列で定義されています。この列はテーブル内のタプルの XMIN の最小値です（FrozenXID を除く）。また pg_class ビューの relfrozenxid の最小値が pg_database ビューの 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 処理が行われなかった場合、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。

```
WARNING:  database "データベース名" must be vacuumed within 残り数 transactions
HINT:  To avoid a database shutdown, execute a database-wide VACUUM in "データベース名".
```

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。

```
ERROR:  database is not accepting commands to avoid wraparound data loss in database "データベース名"
HINT:  Stop the postmaster and use a standalone backend to VACUUM in "データベース名".
```

このような状態に陥ったデータベースはスタンドアロン・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 67 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 9.4beta1
backend> VACUUM
backend>
```

3.7.2 FREEZE 処理に関するパラメータ

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

□ 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 がテーブルスキャ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 はテーブル全体の走査を行います。通常の VACUUM は Visibility Map からガベージの有無を検索しますが、その場合古いトランザクション ID を持つタプルが発見できない恐れがあります。このパラメータの時期に到達すると、VACUUM 処理は Visibility Map を無視して、テーブル全体をスキャンします。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。

3.8 ロケール指定

3.8.1 ロケールの指定とエンコーディング

デフォルトのロケール設定はinitdbコマンドの--localeパラメータで指定します。--localeパラメータ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 68 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 69 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 ビューの 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 70 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)
```

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 71 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。

CREATE INDEX 文のオプション指定

```
CREATE INDEX インデックス名 ON テーブル名（列名 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 37　バイナリ比較オプション

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

例 72 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 73 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 74 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=# ¥l
                                 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 |
 demodb    | demo     | 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)
```

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[6]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・クラスタを作成することができます。

例 75 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 フィールドの後ろに 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、src/include/storage/bufpage.h で定義されています。

[6] または--data-checksums オプション

例 76 ページヘッダ

```
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ファイルからページ読み取り時に行われます。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 77 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 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または、パラメータ data_checksums を確認します。パラメータ data_checksums は PostgreSQL 9.3.4 から利用できます。

例 78 チェックサムの確認

```
$ 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)
```

3.10 ログファイル

PostgreSQL が出力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 が出力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 log_destination（デフォルト値 stderr）で決定されます。このパラメータの値を 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 log_destination を stderr または csv に設定し、パラメータ 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 38　ログファイルの出力先（パラメータ log_destination）

パラメータ値	説明	備考
stderr	標準エラー出力	
csvlog	CSV ファイル	logging_collector=on 必須
syslog	SYSLOG 転送	
eventlog	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、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で決定されます。

表 39　ログファイルの決定

パラメータ	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 log_directory は絶対パスまたはデータベース・クラスタ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 log_directory に「/var」を指定した場合のインスタンス起動エラーです。

例 79 ディレクトリ作成エラー

```
$ pg_ctl -D data start
server starting
FATAL:  could not open log file "/var/postgresql-2014-06-10_134931.log":
Permission denied
```

パラメータ log_filename に%A を含めるとクラスタ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 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 にすることができます。

例 80 CSV ファイル

```
2014-06-10　14:17:04.415　JST,,,8226,,539694d0.2022,1,,2014-06-10　14:17:04
JST,,0,LOG,00000,"autovacuum launcher started",,,,,,,,,""
2014-06-10　14:17:04.415　JST,,,8220,,539694d0.201c,2,,2014-06-10　14:17:04
JST,,0,LOG,00000,"database system is ready to accept connections",,,,,,,,,""
```

パラメータ 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 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 81 SYSLOG に転送された情報

```
Jun 11 13:40:24 rel64-1 postgres[7054]: [3-1] LOG:  database system was shut down
at 2014-06-11 13:40:09 JST
Jun 11 13:40:24 rel64-1 postgres[7051]: [3-1] LOG:  database system is ready to
accept connections
Jun 11 13:40:24 rel64-1 postgres[7058]: [3-1] LOG:  autovacuum launcher started
```

表 40 SYSLOG 出力に関係するパラメータ

パラメータ	説明	デフォルト値
log_destination	syslog に設定することで SYSLOG 転送を有効化	stderr
syslog_facility	SYSLOG のファシリティ	LOCAL0
syslog_ident	ログに出力されるアプリケーション名	postgres

SYSLOG に出力されたログにはパラメータ 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

logger プロセスに USR1 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 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。

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出力され、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 41 エラー・ログのカテゴリーと監視対象

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 log_line_prefix を指定します。以下の文字を指定できます。

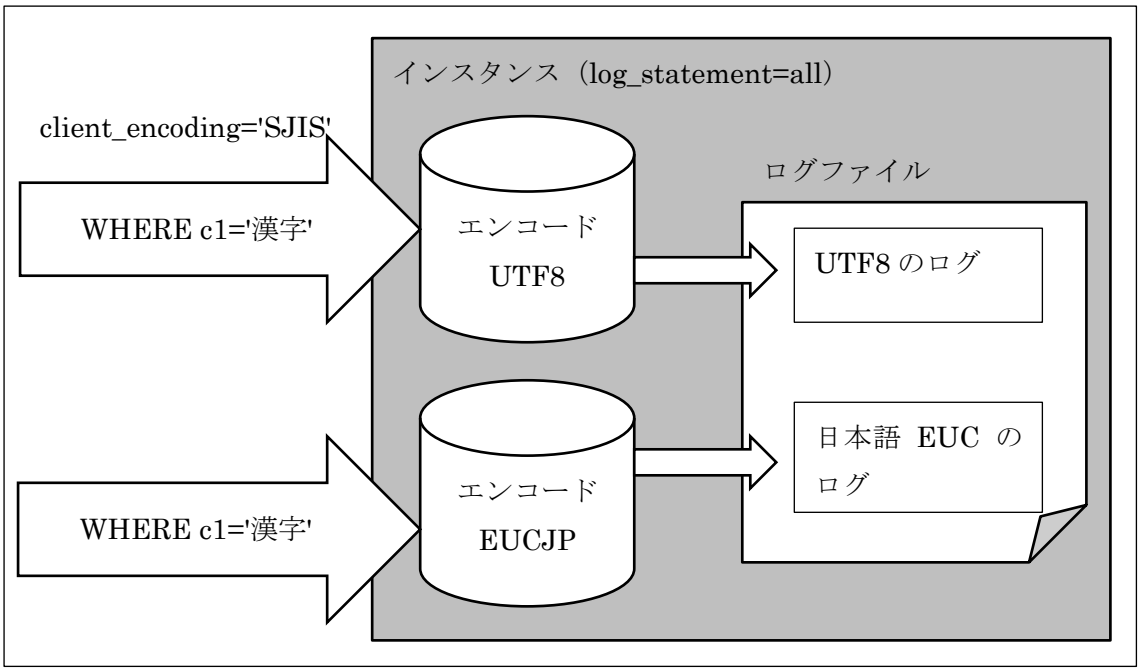
表 42 パラメータ log_line_prefix に指定できる文字

文字列	内容
%a	アプリケーション名（set application_name 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

3.10.5 ログのエンコード

ログファイルに出力される文字の文字コードは、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、SQL 文に指定されたリテラル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 CLIENT_ENCODING は SJIS なので、SQL 文は Shift_JIS で送信されます。パラメータ 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 7 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

インスタンス起動時に 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 82 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 "/opt/PostgreSQL/9.4/data",
but could not open file "/opt/PostgreSQL/9.4/data/global/pg_control": No such
file or directory
```

pg_control ファイルをリカバリするためには、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 83 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:  database system was shut down at 2014-06-09 14:31:28 JST
LOG:  invalid primary checkpoint record
LOG:  invalid secondary checkpoint record
PANIC:  could not locate a valid checkpoint record
LOG:  startup process (PID 7707) 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。

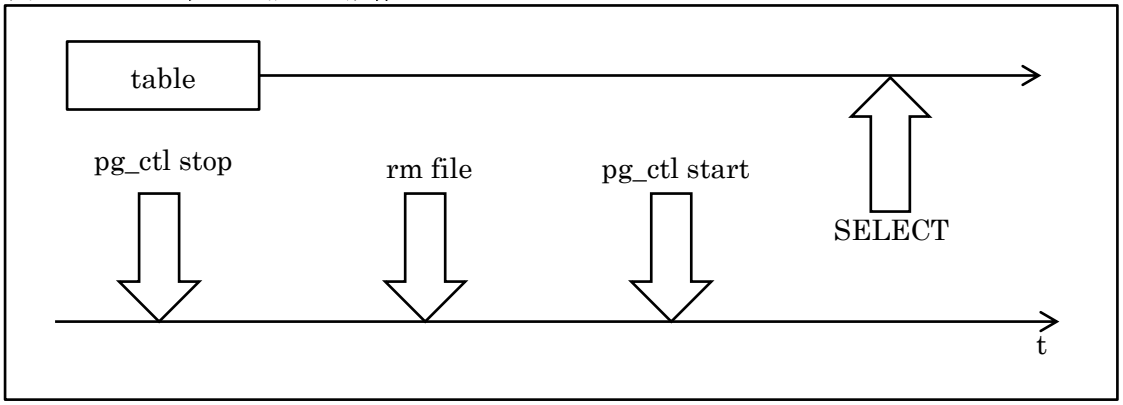
例 84 インスタンスの起動失敗ログ

```
LOG:  could not open file "pg_xlog/000000010000000000000002" (log file 0, segment
2): No such file or directory
LOG:  invalid primary checkpoint record
LOG:  could not open file "pg_xlog/000000010000000000000002" (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
```

4.1.3 データファイル消滅時の動作（正常終了時）

インスタンス正常終了後、次回インスタンスが起動するまでにテーブルを構成するデータファイルが削除された場合の動作を検証しました。

図 8 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稼働しました。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。

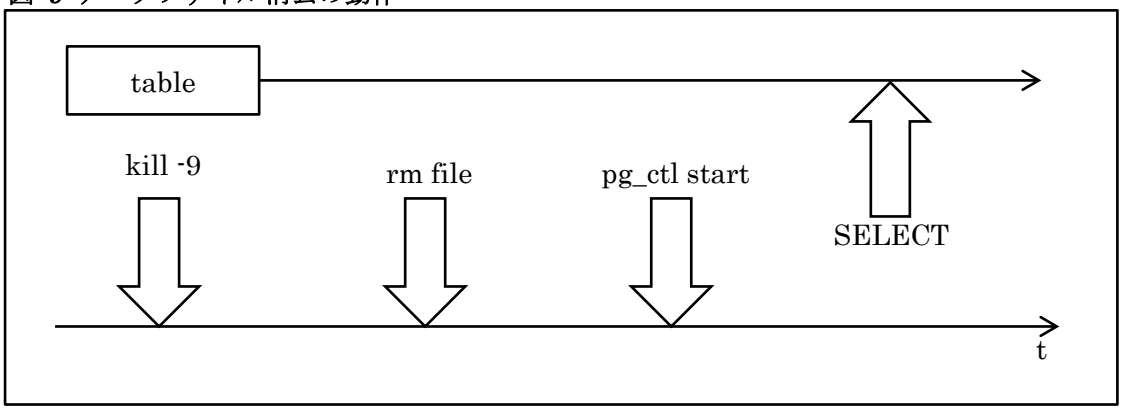
表 43　検索時のログ

```
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・ファイル消滅時の動作（クラッシュ時／変更なし）

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から変更が無い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 9 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稼働しました。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。

表 44　起動時のログ

```
LOG:  database system was interrupted; last known up at 2014-07-01 19:26:19
JST
LOG:  database system was not properly shut down; automatic recovery in
progress
LOG:  redo starts at 0/170AC508
LOG:  record with zero length at 0/1766E888
LOG:  redo done at 0/1766E858
LOG:  last completed transaction was at log time 2014-07-01 19:26:23.657567+09
LOG:  autovacuum launcher started
LOG:  database system is ready to accept connections
```

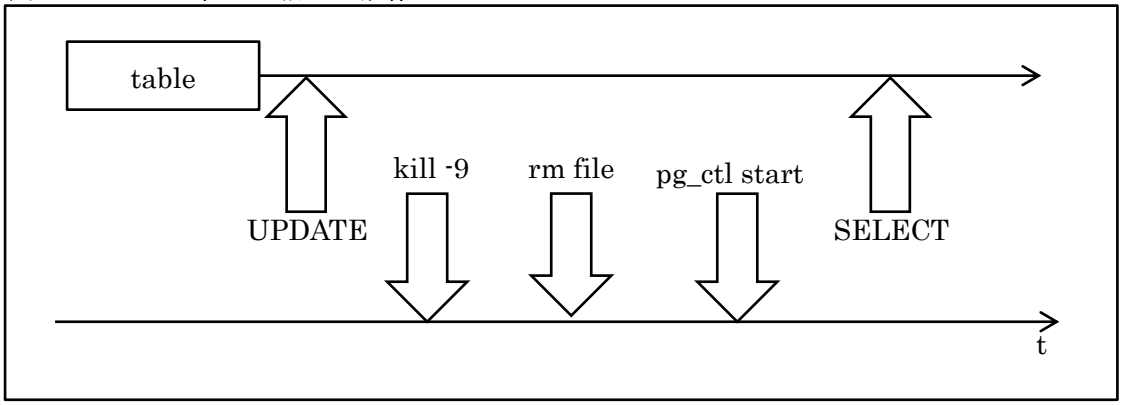
表 45　検索時のログ

```
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 ;
```

4.1.5　データファイル消滅時の動作（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 10 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稼働しました。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

  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。

表 46　起動時のログ

```
LOG:  database system was interrupted; last known up at 2014-07-01 19:37:33
JST
LOG:  database system was not properly shut down; automatic recovery in
progress
LOG:  redo starts at 0/180000E0
LOG:  record with zero length at 0/18008190
LOG:  redo done at 0/18008160
LOG:  last completed transaction was at log time 2014-07-01 19:38:05.152216+09
LOG:  autovacuum launcher started
LOG:  database system is ready to accept connections
```

4.1.6　その他のファイル

□　VM/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、オブジェクト名と実際のファイルが特定できなくなるため SQL 文の実行がエラーになります。

例 85 pg_filenode.map 削除時のログ

```
psql: 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 86 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 87 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 異常

□　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 88 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/000000010000000000000017": Permission
denied
STATEMENT:  SELECT pg_switch_xlog() ;
LOG:  server process (PID 8542) was terminated by signal 6: Aborted
DETAIL:  Failed process was running: SELECT pg_switch_xlog() ;
LOG:  terminating any other active server processes
PANIC:  could not open file "pg_xlog/000000010000000000000017": Permission
denied
PANIC:  could not open file "pg_xlog/000000010000000000000017": Permission
denied
The connection to the server was lost. Attempting reset: 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
FATAL:  the database system is in recovery mode
LOG:  database system was interrupted; last known up at 2014-06-09 15:25:53 JST
LOG:  creating missing WAL directory "pg_xlog/archive_status"
Failed.
!> FATAL:  could not create missing directory "pg_xlog/archive_status":
Permission denied
LOG:  startup process (PID 8547) exited with exit code 1
LOG:  aborting startup due to startup process failure
```

4.3 プロセス障害

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 47 プロセス異常終了時の動作

プロセス	動作
postmaster[7]	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。
wal writer	postmaster により writer、autovacuum launcher も同時に再起動されます。
autovacuum launcher	postmaster により writer、wal writer も同時に再起動されます。
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。
checkpointer	postmaster により再起動されます。
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。
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。
startup process	スレーブ・インスタンス全体が終了します。

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

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（パラメータ 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 89 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
/opt/PostgreSQL/9.4/bin/postgres
$ kill -KILL 6016
$ pg_ctl -D data start
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。

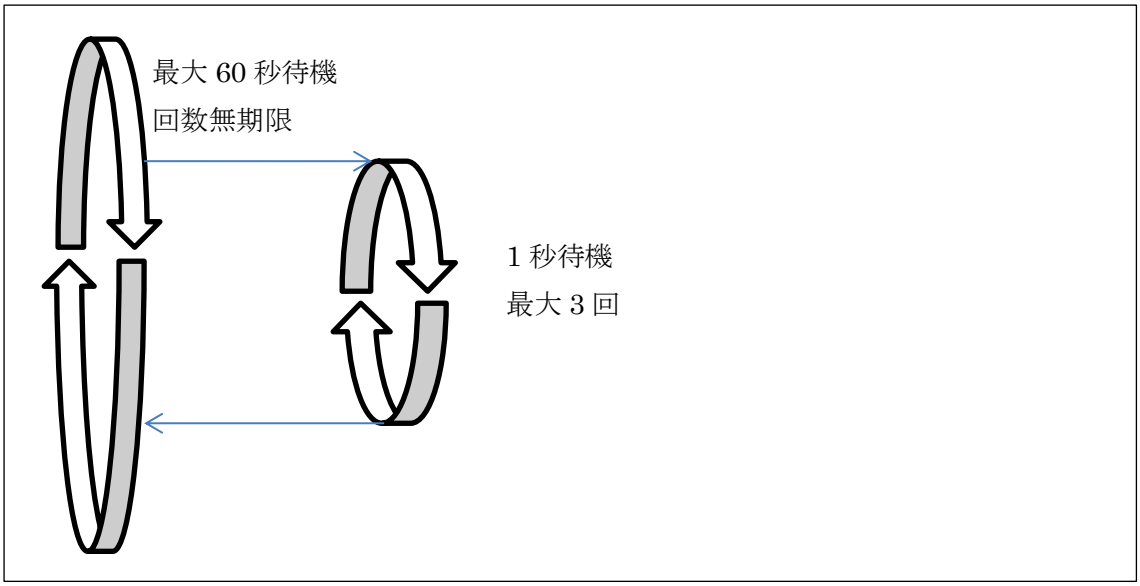
4.4.3 アーカイブ処理の失敗

WAL ファイルが完全に書き込まれると、パラメータ 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 11　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 ビュー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 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 90　アーカイブ処理失敗のログ

```
cp: accessing `/arch/000000010000000000000070': Permission denied
LOG:  archive command failed with exit code 1
DETAIL:  The failed archive command was: cp pg_xlog/000000010000000000000070
/arch/000000010000000000000070
cp: accessing `/arch/000000010000000000000070': Permission denied
LOG:  archive command failed with exit code 1
DETAIL:  The failed archive command was: cp pg_xlog/000000010000000000000070
/arch/000000010000000000000070
cp: accessing `/arch/000000010000000000000070': Permission denied
LOG:  archive command failed with exit code 1
DETAIL:  The failed archive command was: cp pg_xlog/000000010000000000000070
/arch/000000010000000000000070
WARNING:  archiving transaction log file "000000010000000000000070" 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 48 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 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 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 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 / DELET / 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

```
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
  インスタンス･パラメータ 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
  インスタンス・パラメータ 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。サンプリングのレコード数は、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。

サンプル・レコード数 = MAX(各列 STATISTICS) * 300

各列の STATSITICS 値のデフォルト値は、パラメータ default_statistics_target が使用されます。パラメータ default_statistics_target はセッションの値が優先されます。

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_typanalyze 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。

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 49 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 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、パラメータ log_min_messages を DEBUG2 に設定します(ANALYZE VARBOSE 文を指定する場合には INFO)。以下の例は「ANALYZE VARBOSE demo1」文を実行した場合のログです。約 90,000 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 91 ANALYZE 処理のログ

```
2013-05-09 08:00:22 MMT INFO:  analyzing "public.demo1"
2013-05-09 08:00:22 MMT INFO:  "demo1": scanned 1234 of 1234 pages, containing
89998 live rows and 10001 dead rows; 30000 rows in sample, 89998 estimated total
rows
```

列の STATISTICS 設定は、ALTER TABLE ALTER COLUMN 文で実行します。

例 92 列の STATISTICS 設定と確認

```
postgres=> ALTER TABLE demo1 ALTER COLUMN col1 SET STATISTICS 200 ;
ALTER TABLE
postgres=> ¥d+ demo1

Column |     Type                | Modifiers | Storage  | Stats target | Desc
--------+-------------------------+-----------+----------+--------------+-----
 c1     | numeric                 |           | main     | 200          |
 c2     | character varying(100) |           | extended |              |
Has OIDs: no
```

5.1.4 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_*ビューや pg_statio_*ビュー[8]で確認することができます。

表 50 統計情報ビュー

ビュー名	内容
pg_stat_activity	インスタンス全体の統計情報
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|}_indexes	インデックスに関する I/O 統計情報
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 ビューの整形ビュー
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

統計情報の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 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 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。

[8] マニュアルは http://www.postgresql.org/docs/9.4/static/monitoring-stats.html

5.2 自動 VACUUM

5.2.1 タイミング

自動 VACUUM は autovacuum launcher プロセスから、パラメータ autovacuum_naptime（デフォルト 1min）で指定された間隔で実際に処理を行う worker プロセスが起動されます。

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

```
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
  インスタンス･パラメータ 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
  インスタンス・パラメータ 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 ビューの reltuples 列で確認できます。

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 93 EXPLAIN 文の実行

```
postgres=> EXPLAIN ANALYZE SELECT * FROM data1 WHERE c1 BETWEEN 1000 AND 2000 ;
                                                        QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------
 Index Scan using idx1_data1 on data1  (cost=0.42..40.60 rows=959 width=14)
(actual time=0.052..0.375 rows=1001 loops=1)
   Index Cond: ((c1 >= 1000::numeric) AND (c1 <= 2000::numeric))
 Planning time: 9.849 ms
 Execution time: 1.691 ms
(4 rows)
```

表 51 EXPLAIN 文の出力結果

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）	

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 94 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、右の数字はトータル・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 52 実行計画とコストの既定値

パラメータ	説明	既定値	備考
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	

例 95 コストの表示

```
postgres=> EXPLAIN SELECT * FROM demo1 ;
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on demo1  (cost=0.00..2133.98 rows=89998 width=41)
(1 row)
postgres=> SELECT reltuples, relpages FROM pg_class WHERE relname='demo1' ;
 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・オペレータがあります。このリストは、PostgreSQL 9.4.1 のソースコード（src/backend/commands/explain.c）から検索しました。

表 53 実行計画

クエリー・オペレータ	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	
BitmapOr	ビットマップ検索	
Nested Loop	ネステッド・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
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・スキャン	
Bitmap Heap Scan	ヒープのビットマップ・スキャン	有
Tid Scan	TID 走査プラン	無
Subquery Scan	サブクエリー検索	無
Function Scan	関数スキャン	無
Values Scan	値スキャン	
CTE Scan	WITH 句を使った CTE スキャン	
WorkTable Scan	一時テーブル検索	
Foreign Scan	外部テーブル検索	
Materialize	副問い合わせ	有
Sort	ソート処理	有
Group	GROUP BY 句の処理	有

表 53 実行計画（続）

クエリー・オペレータ	動作	スタートアップ・コスト
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

□ Sort クエリー・オペレータ

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 クエリー・オペレータ

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ャンを指します。

□ Bitmap Scan クエリー・オペレータ

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 クエリー・オペレータ

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 クエリー・オペレータ

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

□ LIMIT クエリー・オペレータ

ORDER BY 句と共に LIMIT 句が指定された場合に使用されます。

□ Aggregate クエリー・オペレータ

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 クエリー・オペレータ

UNION (ALL) によるデータの追加処理で使用されます。

□ Nested Loop クエリー・オペレータ

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 クエリー・オペレータ

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 クエリー・オペレータ

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 クエリー・オペレータ

Tuple Id (ctid)を指定した検索で使用されます。

□ Function クエリー・オペレータ

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 クエリー・オペレータ

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

5.3.4 実行時間

EXPLAIN 文に ANALYZE 句を指定すると、実際の実行時間が出力されます。
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 96 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値が計算されます。

例 97 空ブロック・テーブルの rows, cost 表示

```
postgres=> CREATE TABLE demo1 (c1 NUMERIC, c2 NUMERIC) ;
CREATE TABLE
postgres=> ANALYZE demo1 ;
ANALYZE
postgres=> SELECT reltuples, relpages FROM pg_class WHERE relname='demo1' ;
 reltuples | relpages
-----------+----------
         0 |        0
(1 row)
postgres=> EXPLAIN SELECT * FROM demo1 ;
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on demo1  (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 work_mem で指定されたメモリー内で実行されますが、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_9.4_201405111/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 98 ディスクソート用ファイル

```
$ pwd
/opt/PostgreSQL/9.4/data/base/pgsql_tmp
$ ls -l
total 34120
-rw------- 1 postgres postgres 34897920 Jun 17 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出力されます。

例 99 ディスクソートの実行計画

```
postgres=> EXPLAIN ANALYZE SELECT * FROM demo1 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 demo1  (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)
```

□　パラメータ trace_sort

パラメータ 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)。ディスクソートが行われた場合は「external sort ended」ログにブロック数が出力されます。

例 100 trace_sort=on のログ（一部）

```
LOG:  begin tuple sort: nkeys = 2, workMem = 64, randomAccess = f
LOG:  switching to external sort with 7 tapes: CPU 0.00s/0.00u sec elapsed 0.00 sec
LOG:  finished writing run 1 to tape 0: CPU 0.00s/0.00u sec elapsed 0.00 sec
LOG:  performsort starting: CPU 1.00s/3.43u sec elapsed 4.44 sec
LOG:  finished writing run 48 to tape 0: CPU 1.00s/3.43u sec elapsed 4.44 sec
LOG:  performsort done (except 6-way final merge): CPU 1.08s/6.20u sec elapsed 7.33
sec
LOG:  external sort ended, 4260 disk blocks used: CPU 1.11s/7.77u sec elapsed 12.03
sec
```

このパラメータはメモリー・ソート実行時にもログを出力するので、本番環境で設定しないでください。

□ pg_stat_database ビュー

pg_stat_database ビューには、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 18 pg_stat_database ビューの一時ファイル関連データ

列名	説明	備考
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	

5.3.7 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８KB 単位で送信しています。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 101 シーケンシャル・スキャン

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

open("base/16499/16525", O_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¥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。シーケンシャル・スキャンと同様、クライアントには８KB 単位で送信されます。

例 102 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
```

5.4 パラメータ

5.4.1 パフォーマンスに関連するパラメータ

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

表 18 関連パラメータ

パラメータ	説明	備考
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 Page 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	
wal_buffers	WAL 情報が格納される共有メモリー	
work_mem	ソート、ハッシュ用の一時メモリー	

5.4.2 effective_cache_size

このパラメータは共有バッファと 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・スキャンのコストを計算するパラメータ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_io_concurrency

このパラメータの指定値は 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・スキャ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。

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ビュー（pg_stat で始まるビュー）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 18 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()

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[9]に記載されています。

表 18 ロックの種類

ロック	説明
ACCESS SHARE	SELECT 文を実行すると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。

[9] マニュアル http://www.postgresql.org/docs/9.4/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くなります。現在のロック状況は pg_locks ビューから確認できます。

例 103 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
```

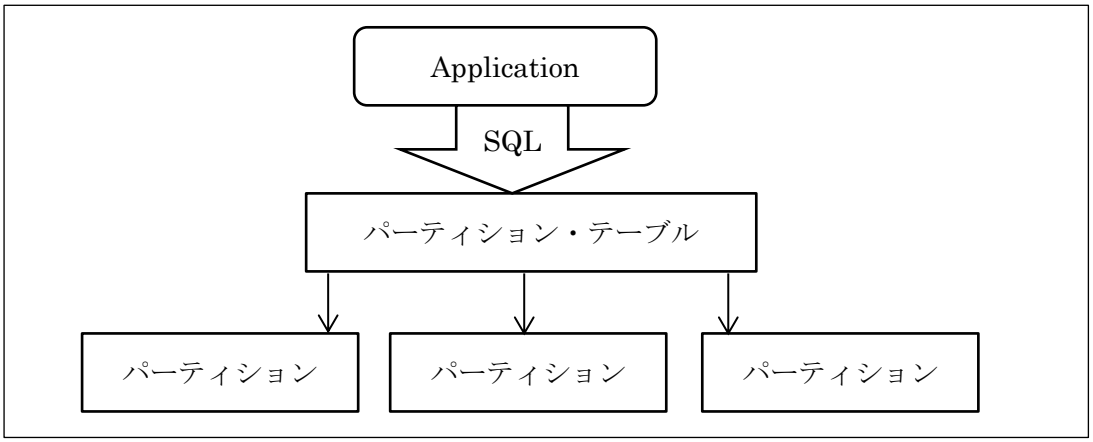
ACCESS EXCLUSIVE LOCK は他の全てのロックと競合します。このため VACUUM FULL のようにこのロックを取得する処理を実行中のテーブルには検索を含めてアクセスできなくなります。

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 12　パーティション・テーブル



一般的に、パーティション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 104 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。親テーブルを INHERITS 句で指定し、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 105 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指定します。

例 106 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 107 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す。

□ 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 108 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 109 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。条件は SELECT 文と同じです。

例 110 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 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 111 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

Java アプリケーションから、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 112 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 113 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・テーブルを作成すると、親テーブルの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 114 UPDATE 文の実行計画

```
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 115 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=>
```

6.3 postgres_fdw

6.3.1 postgres_fdw とは

postgres_fdw は外部システムに対して SQL 文を投入する FOREIGN DATA WRAPPER に対応する contrib モジュール[10]で、リモート上の PostgreSQL に SQL 文を投入することができます。古いバージョンの PostgreSQL には、ネットワーク上に稼働する PostgreSQL インスタンスに SQL 文を投入する機能として dblink 関数が提供されてきました。しかし、この関数は扱いにくいため、より簡単で洗練された実装が開発されました。PostgreSQL 9.3 からは FOREIGN DATA WRAPPER に対する書込みを行うことが可能になりました。

6.3.2 postgres_fdw の使用方法

postgres_fdw を使用するためには以下の手順で実行します。

- CREATE EXTENSION 文で postgres_fdw をロード
- CREATE SERVER 文でリモート・インスタンスを特定
- CREATE USER MAPPING 文でリモート・アクセスするユーザーを特定
- CREATE FOREIGN TABLE 文でリモート・テーブルを特定
- ローカル・テーブルと同様に DML を実行

例 116 postgres_fdw の使用方法

```
postgres=# CREATE EXTENSION postgres_fdw ;
CREATE EXTENSION
postgres=# CREATE SERVER remsvr FOREIGN DATA WRAPPER postgres_fdw
        OPTIONS (host 'remhost1', dbname 'dbappl1', port '5432') ;
CREATE SERVER
postgres=# CREATE USER MAPPING FOR public SERVER remsvr
        OPTIONS (user 'postgres', password 'secret');
CREATE USER MAPPING
postgres=# CREATE FOREIGN TABLE data1 (c1 NUMERIC, c2 VARCHAR(10))
        SERVER remsvr;
postgres=# SELECT * FROM data1 ;
```

6.3.3 実行される SQL

リモート・インスタンスで SQL 文が投入されると、postgres_fdw 経由で SQL 文が送信

[10] マニュアルは http://www.postgresql.org/docs/9.4/static/postgres-fdw.html

され、結果が返されます。ここでは postgres_fdw が送信する SQL 文について検証しています。SQL 文は以下のように変更されて送信されます。

- 列名の「*」は、列名リストに変換
- コメントは削除
- 予約語は大文字に変換
- 集計（SUM, COUNT 等）／グルーピング（GROUP BY）／順番（ORDER BY）は削除
- 結合は２つの SELECT 文に分解

以下の例は、リモート・インスタンスに対して「SELECT * FROM data1」文を投入した場合に実行される SQL 文です。下記のように複数の SQL 文に分解されて実行されます。

- SART TRANSACTION ISOLATION LEVEL REPEATABLE READ
- DECLARE CURSOR
- FETCH
- CLOSE
- COMMIT TRANSACTION

例 117 実行される SQL 文の例

```
START TRANSACTION ISOLATION LEVEL REPEATABLE READ
DECLARE c1 CURSOR FOR
        SELECT c1, c2 FROM public.data1
FETCH 100 FROM c1
FETCH 100 FROM c1
..
CLOSE c1
COMMIT TRANSACTION
```

次の表は、投入した SQL 文と実行された SQL 文の対比です。START TRANSACTION 文、FETCH 文、CLOSE 文、COMMIT 文は省略しています。

表 18 ローカル投入 SQL とリモート実行 SQL（検索）

ローカル投入 SQL	リモート実行 SQL
select * from data1	DECLARE c1 CURSOR FOR SELECT c1, c2 FROM public.data1
select c1 from data1	DECLARE c1 CURSOR FOR SELECT c1 FROM public.data1
select /* comment */ * from data1	DECLARE c1 CURSOR FOR SELECT c1, c2 FROM public.data1
select c1 from data1 order by 1	DECLARE c1 CURSOR FOR SELECT c1 FROM public.data1
select count(*) from data1	DECLARE c1 CURSOR FOR SELECT NULL FROM public.data1
select count(c1) from data1	DECLARE c1 CURSOR FOR SELECT c1 FROM public.data1
select c1 from data1 group by c1	DECLARE c1 CURSOR FOR SELECT c1 FROM public.data1
select * from data1 where c1 = 10	DECLARE c1 CURSOR FOR SELECT c1, c2 FROM public.data1 WHERE ((c1 < 10::numeric))
select * from data1 where octet_length(c2) = 2	DECLARE c1 CURSOR FOR SELECT c1, c2 FROM public.data1 WHERE ((octet_length(c2) = 2))
select d1.c1, d2.c2 from data1 d1, data2 d2 where d1.c1 = d2.c1 ;	DECLARE c1 CURSOR FOR SELECT c1 FROM public.data1 WHERE ((c1 < 100::numeric)) DECLARE c2 CURSOR FOR SELECT c1, c2 FROM public.data2 WHERE ((c1 < 10::numeric))

SELECT 文を投入すると、カーソルを作成し、100 レコードずつフェッチを行っています。フェッチ数の 100 はソースコード（contrib/postgres_fdw/postgres_fdw.c）にハードコードされているため現状では変更できません。次の例は更新文で実行される SQL 文です。UPDATE 文、DELETE 文ではカーソルを定義して、1 レコードずつ UPDATE / DELETE 文を実行していることがわかります。

表 18 ローカル投入 SQL とリモート実行 SQL（更新）

ローカル投入 SQL	リモート実行 SQL
update data1 set c1 = c1 + 1 where c1 < 10	DECLARE c1 CURSOR FOR SELECT c1, c2, ctid FROM public.data1 WHERE ((c1 < 10::numeric)) FOR UPDATE FETCH 100 FROM c1 UPDATE public.data1 SET c1 = $2 WHERE ctid = $1 … UPDATE public.data1 SET c1 = $2 WHERE ctid = $1 DEALLOCATE pgsql_fdw_prep_2
insert into data1 values (20000, 'after')	INSERT INTO public.data1(c1, c2) VALUES ($1, $2) DEALLOCATE pgsql_fdw_prep_3
delete from data1 where c1 = 20000	DECLARE c1 CURSOR FOR SELECT ctid FROM public.data1 WHERE ((c1 = 20000::numeric)) FOR UPDATE FETCH 100 FROM c1 … DELETE FROM public.data1 WHERE ctid = $1 DEALLOCATE pgsql_fdw_prep_4

6.4 シーケンス

6.4.1 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[11]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 118 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¥ds 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。

[11] マニュアルは http://www.postgresql.org/docs/9.4/static/sql-createsequence.html

例 119 シーケンスの情報取得

```
postgres=> ¥ds+
                    List of relations
 Schema | Name  |   Type   | Owner |    Size    | Description
--------+-------+----------+-------+------------+-------------
 public | seq01 | sequence | scott | 8192 bytes |
(1 row)
postgres=> SELECT 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)
```

6.4.2 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 120 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 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。

6.4.3 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 121 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=>
```

6.5 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。

6.5.1 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 54 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 55 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 56 列型 VARCHAR(5) → ホスト変数 VARCHAR(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	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。

6.5.2 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。

> 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 57 列型 VARCHAR(5) → ホスト変数 char(3)

配列	char(0)	char(1)	char(2)
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 58 列型 VARCHAR(5) → ホスト変数 char(5)

配列	char(0)	char(1)	char(2)	char(3)	char(4)
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。

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 59 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	備考
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 60 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	備考
CREATE TABLE	○	
CREATE INDEX	○	
CREATE SEQUENCE	○	
CREATE SCHEMA	×	
CREATE FUNCTION	○	
CREATE TYPE	○	

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[12]は、データベース・クラスタの完全なコピーを作成するために開発されました。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 パラメータを指定します(PostgreSQL 9.4 から指定可能)。

例 122 pg_basebackup コマンド実行

```
$ pg_basebackup -D back -h hostsrc1 -p 5432 -x -v
Password: << パスワード >>
transaction log start point: 0/7E000020
transaction log end point: 0/7E0000A8
pg_basebackup: base backup completed
$
```

8.1.2 pg_archivecleanup コマンド

pg_archivecleanup コマンドは、contrib モジュール[13]に含まれます。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。

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 のパラメータ値として使用します。最初のパラメータにはアーカイブログ出力ディレクトリを、2 番目のパラメータには最終の WAL ファイルを示す「%r」

[12] マニュアルは http://www.postgresql.org/docs/9.4/static/app-pgbasebackup.html

[13] マニュアルは http://www.postgresql.org/docs/9.4/static/pgarchivecleanup.html

を指定します。

例 123 recovery.conf ファイルの指定

```
archive_cleanup_command = 'pg_archivecleanup /opt/PostgreSQL/9.4/arch %r'
```

pg_archivecleanup コマンドはスタンドアロン環境でも使用できます。第２パラメータには、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 124 pg_archivecleanup コマンド実行スクリプト

```
#! /bin/sh

. /opt/PostgreSQL/9.4/pg_env.sh

ARCHDIR=/opt/PostgreSQL/9.4/arch
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[14]が使用する環境変数は以下の通りです。

[14] マニュアルは http://www.postgresql.org/docs/9.4/static/app-psql.html

表 61　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	ページャ・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 ユーザ名
PGHOST	デフォルト・ホスト名	localhost
PGPORT	デフォルト・ポート番号	5,432
PGUSER	デフォルト・ユーザ名	OS ユーザ名
PSQL_EDITOR EDITOR VISUAL	¥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
PSQL_EDITOR_LINENUMBER_ARG	エディタに行番号を渡すコマンド	Linux / UNIX では'+' Windows ではなし
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
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

8.1.4 pg_resetxlog コマンド

pg_resetxlog コマンド[15]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。

[15] マニュアルは http://www.postgresql.org/docs/9.4/static/app-pgresetxlog.html

表 62　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	Mon 30 Jun 2014 03:38:37 PM JST	Mon 30 Jun 2014 05:45:09 PM JST
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	00000001000000020000000A4	00000001000000020000000AC
Time of latest checkpoint	Mon 30 Jun 2014 03:38:36 PM JST	Mon 30 Jun 2014 05:43:46 PM JST
Backup start location	0/E1000028	0/0

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処理が成功した時点で 0 を返して終了します。このためインスタンス起動の成功を確認しているわけではありません。

表 63　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	pg_ctl status 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	pg_ctl status コマンド実行時に、データベース・クラスタが存在しない場合	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 64　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	¥set ON_ERROR_STOP true　実行後にエラーが発生した場合。	

例 125 psql のエラー検知

```
$ psql -c 'SELECT * FROM notexists' -h localhost -U scott -p 5432
Password for user scott: <<password>>
ERROR:  relation "notexist" does not exist
LINE 1: SELECT * FROM notexist
                      ^
$ echo $?
1
$ cat error.sql
SELECT * FROM notexists ;
$ psql -f error.sql -h localhost -U scott -p 5432
Password for user scott: <<password>>
psql:error.sql:1: ERROR:  relation "notexists" does not exist
LINE 1: SELECT * FROM notexists ;
                      ^
$ echo $?
0
$ psql -f error.sql -h localhost -U scott -p 5432 --set=ON_ERROR_STOP=true
Password for user scott: << 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 -h localhost -U scott -p 5432
Password for user scott: <<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 65　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 66　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 67　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のチェックを行い、不正なパラメータ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 68　pg_isready コマンドの終了ステータス

ステータス	説明	備考
0	サーバーは稼働中で接続受付け可能	PQPING_OK
1	サーバーは稼働中だが接続受付不可	PQPING_REJECT
2	サーバーと通信不可	PQPING_NO_RESPONSE
3	パラメータ不正、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。

9. システム構成

9.1 パラメータのデフォルト値

PostgreSQL で使用するパラメータはデータベース・クラスタ内の 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 コマンド実行直後の状態で変更されているパラメータを調査しました。

9.1.1 initdb コマンド実行時に導出されるパラメータ

いくつかのパラメータは 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 69 initdb コマンド実行時に設定されるパラメータ

パラメータ	設定値	デフォルト値
max_connections	100	100
shared_buffers	128MB	8MB
dynamic_shared_memory_type	posix	posix
log_timezone	環境変数から導出	GMT
datestyle	環境変数から導出	ISO,MDY
timezone	環境変数から導出	GMT
lc_messages	環境変数から導出	-
lc_monetary	環境変数から導出	C
lc_numeric	環境変数から導出	C
lc_time	環境変数から導出	C
default_text_search_config	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

クラスタ作成時に指定する initdb コマンドのパラメータ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 70 initdb コマンドの推奨パラメータ

パラメータ	推奨値	備考
--encoding	UTF8	
--locale	指定しない	
--no-locale	指定する	
--username	postgres	

9.2.2 推奨パラメータ

推奨パラメータ設定は以下の通りです。

表 71 postgresql.conf ファイルに設定する推奨パラメータ

パラメータ	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	
checkpoint_segments	128	
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	hot_standby	レプリケーション時
max_replication_slots	レプリケーション数+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。

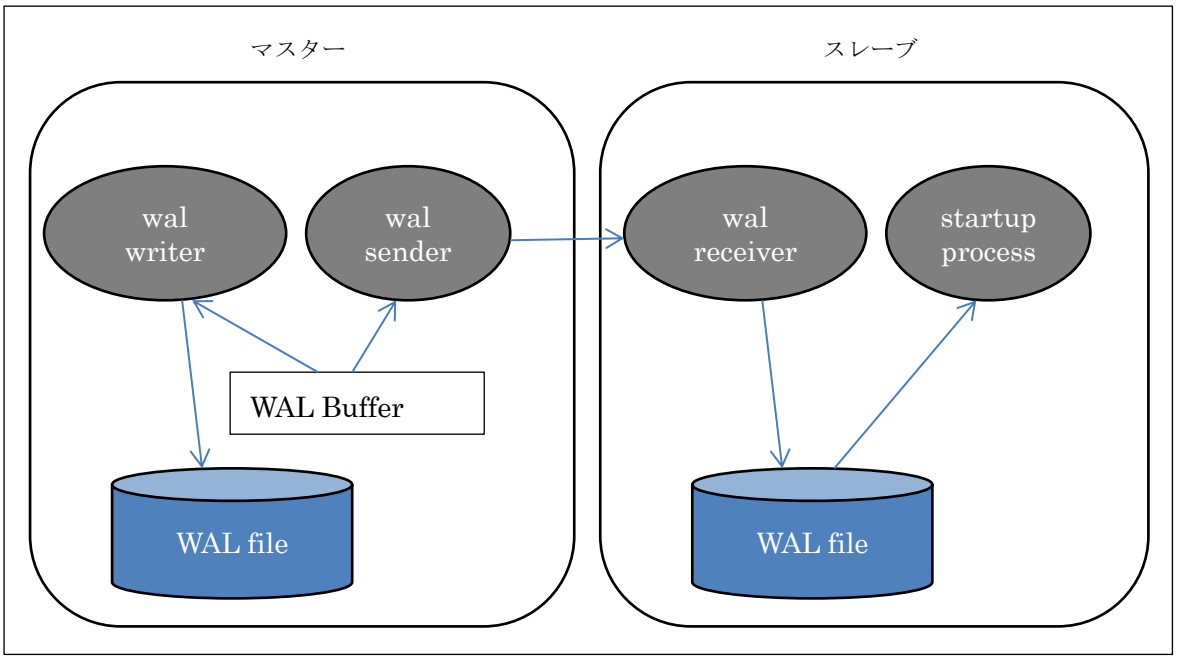
ストリーミング・レプリケーション環境では、更新処理は常に１つのインスタンスのみで実行されます。更新を行うインスタンスをマスター・インスタンスと呼びます。データベースに対する更新で発生した WAL 情報はスレーブ・インスタンスに転送されます。スレーブ・インスタンスでは受信した WAL 情報をデータベースに適用することで複数データベースの同一性を保障します。スレーブ・インスタンスではリアルタイムにデータベースのリカバリを実施している状態になります。スレーブ・インスタンスは読み込み専用（Read Only）状態で起動することができ、テーブルに対して検索（SELECT）を実行することができます。このため、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・クラスタに適用され、スレーブ・イ

ンスタンスが更新されます。

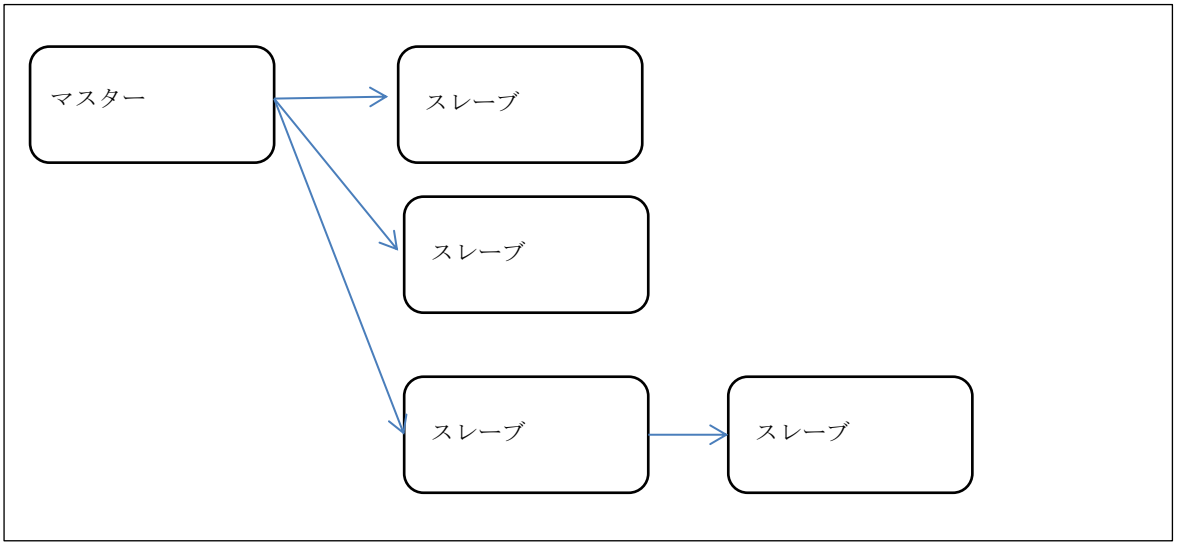
図 13 ストリーミング・レプリケーション



□ 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もっと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 14 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 スロット

PostgreSQL 9.4 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・インスタンスはスロット名を参照することで実現されます。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 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。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や pg_hba.conf ファイルの設定等は必要です。

□ スロットの管理

スロットは以下の関数で管理を行います。使用中の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。

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)
```

スロット削除関数

```
pg_drop_replication_slot(スロット名)
```

クラスタに作成できるスロット数の最大値はパラメータ max_replication_slots で指定します。このパラメータのデフォルト値は 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 max_replication_slots で指定された値を元に共有メモリー上にレプリケーション関連の情報が展開されます。

作成したスロットの情報は pg_replication_slots ビューから確認することができます。

例 126 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 * FROM pg_replication_slots ;
 slot_name | plugin | slot_type | datoid | database | active | xmin | catalog_xmin | restart_lsn
-----------+--------+-----------+--------+----------+--------+------+--------------+-------------
 slot_1    |        | physical  |        |          | f      |      |              |
(1 row)
```

マスター・インスタンスで作成したスロットは、スレーブ・インスタンスの recovery.conf から参照します。

例 127 recovery.conf ファイル内のスロット参照

```
primary_slotname = 'slot_1'
primary_conninfo = 'host=hostmstr1 port=5433 application_name=prim5433'
standby_mode = on
```

現状の実装では、スレーブ側に primary_slotname 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_slotname が記述されているにも関わらずマスター側で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。

```
FATAL:  could not start WAL streaming: ERROR:  replication slot "slot_1" does not exist
```

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。

レプリケーションが成功するとマスター側の pg_stat_replication ビュー、pg_replication_slots ビューは以下の表示になります。

例 128 レプリケーション状況の確認

```
postgres=# SELECT * FROM pg_stat_replication ;
  pid  | usesysid | usename | application_name | client_addr | client_hostname | client_port |
 backend_start         | backend_xmin |   state   | sent_location | write_location | flush_location |
replay_location | sync_priority | sync_state
-------+----------+---------+------------------+-------------+-----------------+-------------+-------
----------------------+-------------+-----------+---------------+----------------+----------------+-
---------------+---------------+------------
 18481 |       10 | postgres | stdby1           | 127.0.0.1   |                 |       45345 | 2014-05
-16 13:04:27.344758+09 |              | streaming | 0/30C6468     | 0/30C6468      | 0/30C6468      |
0/30C6468       |             0 | async
(1 row)
postgres=# SELECT * FROM pg_replication_slots ;
slot_name | plugin | slot_type | datoid | database | active | xmin | catalog_xmin | restart_lsn
-----------+--------+-----------+--------+----------+--------+------+--------------+-------------
 slot_1    |        | physical  |        |          | t      |      |              | 0/30C6468
(1 row)
```

PostgreSQL 9.4 の pg_stat_replication ビューには backend_xmin 列が追加されています。

□　スロットの実体

スロットの実体は{PGDATA}/pg_replslot ディレクトリ内に作成された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 129 スロットの実体

```
$ ls -l data/pg_replslot/
total 4
drwx------. 2 postgres postgres 4096 May  16 15:42 slot_1
$ ls -l data/pg_replslot/slot_1/
total 4
-rw-------. 1 postgres postgres 176 May  16 15:42 state
```

10.2.2 同期と非同期

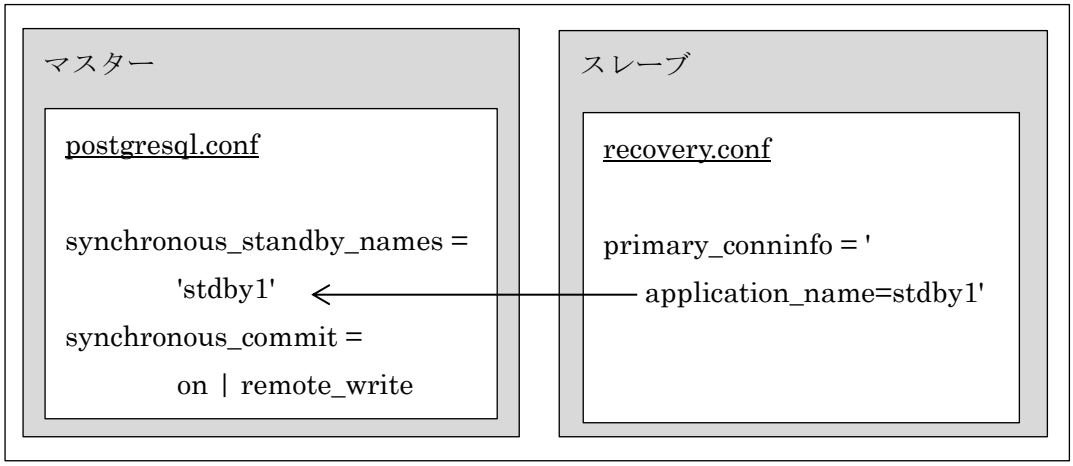
ストリーミング・レプリケーションはマスター・インスタンスから WAL 情報を転送することによりスレーブ･インスタンスと同期を取ります。マスター･インスタンスの WAL と、スレーブ･インスタンスの WAL の両方を書き込んだことを確認してからトランザクションの完了をユーザーに通知すれば信頼性は向上しますが、パフォーマンスは大幅に低下します。一方でスレーブ・インスタンスに対する WAL 書き込みを非同期に行えばパフォーマンスは向上するかもしれませんが、スレーブ・システムに WAL が到着する前にマスター・インスタンスが異常終了すると書き込んだはずのトランザクションが失われる恐れがあります。このため 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４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 synchronous_commit で決定されます。

表 72 パラメータ synchronous_commit 設定値

設定値	プライマリ WAL	スタンバイ WAL	備考
on	同期	同期	ストレージ書き込みまで同期
remote_write	同期	同期	メモリー書き込みまで同期
local	同期	非同期	
off	非同期	非同期	

同期レプリケーションを行うには、パラメータ synchronous_standby_names に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 パラメータ内の application_name 項目に同じ名前を指定します。

図 15 同期レプリケーション設定



□ 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 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 ではスレーブに適用できなかった WAL ファイルは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

レプリケーションに関連するパラメータは以下の通りです。

表 73 マスター・インスタンスのパラメータ

パラメータ	説明	設定値	備考
wal_level	WAL 出力レベル	hot_standby	
archive_mode	アーカイブ出力モード	on	
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	1 以上	
max_replication_slots	スロット最大数	1 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を指定します。デフォルト値の場合、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、スイッチオーバー用にあらかじめ設定しておくこともできます。

表 74 スレーブ・インスタンスのパラメータ

パラメータ	説明	設定値	備考
hot_standby	スレーブを参照可能に設定	on	スレーブ

10.2.4 recovery.conf

スレーブ・インスタンスは WAL 情報を受信し、リカバリ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と同じように、データベース・クラスタ内に recovery.conf ファ

イルを作成します。以下のパラメータを指定することができます。primary_slotname パラメータは PostgreSQL 9.4 から指定できます。

表 75　スレーブ・インスタンスの recovery.conf

パラメータ	説明	設定	備考
standby_mode	スタンバイ・モード	on	
primary_slotname	接続先スロット名	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	遅延リカバリ設定	時間設定	9.4～
trigger_file	トリガー・ファイル	ファイル名	

例 130 recovery.conf の作成例

```
standby_mode = 'on'
primary_slotname = 'slot_1'
primary_conninfo = 'host=hostmstr1 port=5432 user=postgres password=secret
        application_name=stdby1'
restore_command = 'scp hostmstr1:/opt/PostgreSQL/9.4/arch/%f %p'
recovery_target_timeline='latest'
```

□ trigger_file パラメータ

trigger_file パラメータ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

primary_conninfo パラメータにはマスター・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 76 primary_conninfo に指定できるパラメータ

パラメータ	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

リカバリ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チオーバー

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チオーバーと呼びます。

10.3.1 スイッチオーバー

スイッチ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 131 pg_ctl promote

```
$ 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。

10.3.3 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。

- 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
- Single-User サーバでないかのチェック
- recovery.conf ファイルの存在チェック
- {PGDATA}/promote ファイルの作成
- postmaster プロセスに対して SIGUSR1 シグナルの送信
- 「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。

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。

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 77 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	備考
aclocal.m4	configure 用ファイルの一部	
config	configure 用ファイル格納ディレクトリ	
config.log	configure 実行ログ	
config.status	configure コマンドが生成するスクリプト	
configure	configure プログラム本体	
configure.in	configure プログラムの雛形	
contrib	contrib モジュール用ディレクトリ	
COPYRIGHT	著作権情報	
doc	ドキュメント保存用ディレクトリ	
GNUmakefile	トップレベルの Makefile	
GNUmakefile.in	Makefile の雛形	
HISTORY	リリースノートを表示する URL が記載	
INSTALL	インストール方法の概要資料	
Makefile	ダミーの Makefile	
README	概要説明資料	
src	ソースコード用ディレクトリ	

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 78 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	備考
backend	バックエンド・プロセス群のソース	
bin	pg_ctl 等のコマンド類のソース	
common	共通使用のソース	
include	ヘッダファイル	
interfaces	ECPG, libpq ライブラリのソース	
pl	PL/Perl, PL/pgSQL, PL/Python, PL/tcl のソース	
port	libpgport ライブラリのソース	
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 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 79 make コマンドの主なオプション

ターゲット	説明	備考
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 80 オーバーコミット設定

カーネル・パラメータ	デフォルト値	推奨値	備考
vm.overcommit_memory	0	2	
vm.overcommit_ratio	50	99	

12.1.2 I/O スケジューラ

一般的なシステムでは I/O スケジューラの変更は必要ないと思われます。一括読み込みが多いシステムでは deadline に変更することも検討します。

例 132 I/O スケジューラを deadline に変更（/etc/grub.conf ファイル）

```
kernel /vmlinuz-2.6.18-274.3.1.el5 ro root=/dev/vg00/lvol1 elevetor=deadline
```

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 vm.swappiness は 5 以下にすることが推奨されます。

表 81 スワップ設定

パラメータ	デフォルト値	推奨値	備考
vm.swappiness	60	0	

12.1.4 Huge Page

大規模メモリー環境では Huge Page を利用する設定を行います。PostgreSQL 9.4 のデフォルト設定では Huge Page が利用できれば利用します。カーネル・パラメータ vm.nr_hugepages には共有メモリーで使用される領域以上のサイズ（2 MB 単位）を指定します。必要量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 82 Huge Pages

パラメータ	デフォルト値	推奨値	備考
vm.nr_hugepages	0	shared_buffers 以上	

12.1.5 省電力モード

パフォーマンス上の理由から省電力モードは使用しないことが推奨されます。/etc/grub.conf ファイルに以下のエントリを指定します。あわせて BIOS 設定が必要な場合があります。

表 83 省電力モード

パラメータ	推奨値	備考
intel_idle.max_cstate	0	
processor.max_cstate	0	

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 が推奨されます。

12.2.1 ext4 使用時

データベース・クラスタ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 - 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。

```
ulimit -c unlimited
```

12.3.2 ABRT による Core 管理

Red Hat Enterprise Linux 6 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 kernel.core_pattern が以下の設定に変更されています。

```
|/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 パラメータを変更し

ます。

例 133 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/spool/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 パラメータを no に設定します。

例 134 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 135 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/site/documentation/ja-JP/Red_Hat_Enterprise_Linux/6/html/Deployment_Guide/sect-abrt-configuration.html

12.4 その他

12.4.1 SSH

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 パラメータが使用されます。ここにはアーカイブログ・ファイルのコピー用コマンドを記述します。プライマリ・ホストに 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。

12.4.2 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。

12.4.3 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。

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 84 書籍の情報

書籍名	著者（敬称略）	出版社	備考
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	
PostgreSQL 9.0 High Performance	Gregory Smith	PACKT	
PostgreSQL Replication	Zoltan Boszormenyi Hans-Jurgen Schonig	PACKT	
PostgreSQL 9 Admin Cookbook	Simon Riggs Hannu Krosing	PACKT	

付録 2. URL

PostgreSQL について参考になる URL の情報です。

表 85 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 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

変更履歴

変更履歴

版	日付	作成者	説明
1.0	16-Jul-2014	篠田典良	公開版を作成

以上