JPUG 仕組み分科会　2009-02-14 補足資料

補足資料：PostgreSQL の WAL と PITR

鈴木啓修@InterDB.jp

1. PostgreSQL の簡単な紹介

"PostgreSQL"はオープンソースのデータベースシステム。

1.1. 歴史

PostgreSQL の起源は、カリフォルニア大学バークレー校で作られた"Postgres"。

Postgres の開発は 1986 年から。しかし、研究プロジェクトだったため、保守とユーザサポートの負担が大きくなったことを理由に、バージョン 4.2 をもって開発が終了。

1994 年に Andrew Yu 氏と Jolly Chen 氏が Postgres に改良を加え、"Postgres95"としてリリース。1996 年には"PostgreSQL"と改名され、機能拡張と改良を加えながら現在に至る。

1.2. リリース状況

PostgreSQL のバージョン番号はコンマで区切った 3 つの数字からなり、最初の 2 つがメジャーバージョン、末尾がマイナーバージョン。例えば"PostgreSQL 8.3.5"はメジャーバージョンが"8.3"、マイナーバージョンが"5"。

メジャーバージョンが上がるのは機能追加や大きな変更があったとき、マイナーバージョンが上がるのはバグフィックスされたとき。

バージョン	リリース	主な機能
8.3	08/02	HOT(HEAP Only Tuple)、チェックポイント時の負荷分散、wal writer
8.2	06/12	内部ロックの改良、シーケンシャルスキャンの効率化、バキューム処理効率
8.1	05/11	2相コミット(two-phase commit),自動VACUUM,ビットマップスキャン
8.0	05/01	Windows対応，アーカイブログ機能，バックグランドライタ機能，テーブルスペースのサポート，PITR(Point-In-Time Recovery)，Save Pointのサポート
7.4	03/11	IPv6対応
7.3	02/11	スキーマ，動的SQL文実行
7.2	02/02	並行VACUUM，MD5によるパスワード暗号化
7.1	01/04	WAL(Write Ahead Logging)，外部結合(Outer Joins)
7.0	00/05	外部キー制約(Foreign Keys)，各種結合(Join)
6.5	99/06	多版型同時実行制御(MVCC)，ホットバックアップ
6.4	98/10	PL/pgSQL，マルチバイト文字
6.3	98/03	副問い合わせ
6.2	97/10	JDBC，トリガ
6.1	97/06	遺伝的アルゴリズムによる問い合わせ最適化，シーケンス
6.0	97/01	PostgreSQLとしての初リリース

より詳細な情報は以下の URL を参照のこと：

http://www.postgresql.org/

http://www.postgresql.jp/

1.3. プロセス構造

PostgreSQL サーバの本体は"postgres"というデーモンプロセス。postgres はクライアントから接続要求を受けるとバックエンドプロセス"postgres"を生成(fork)し、そのバックエンドプロセスがクライアントの SQL 文を処理する(図 1)。

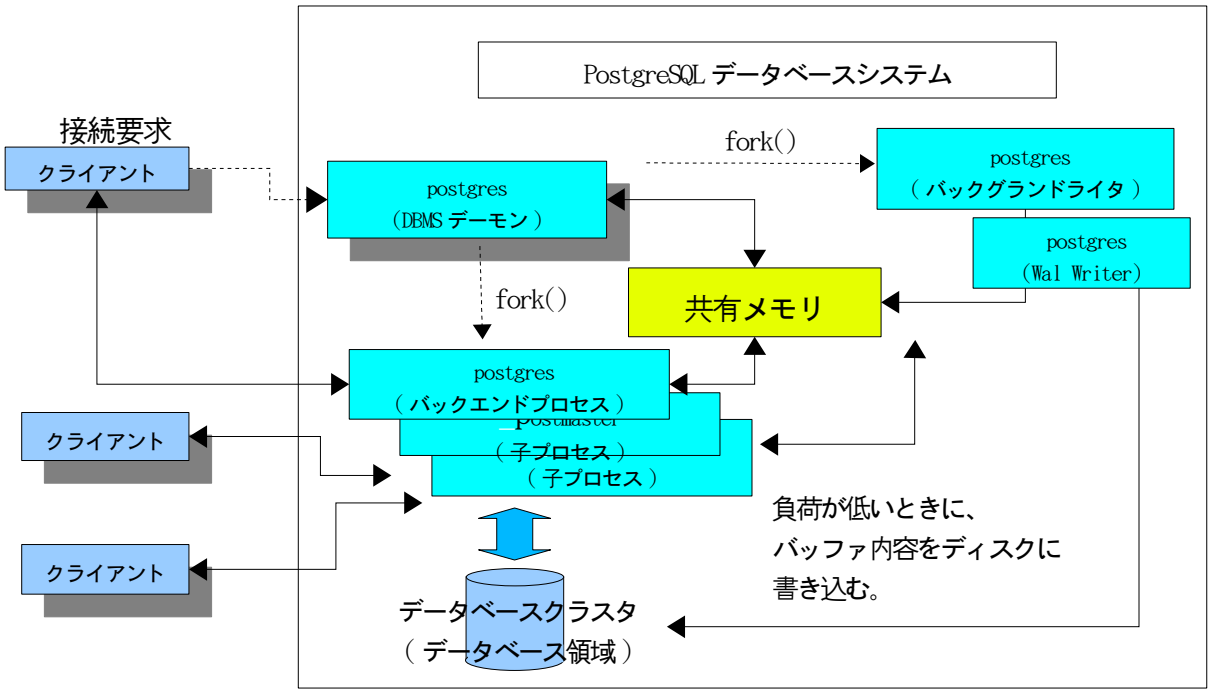


図 1 プロセス構造

実際に動作しているプロセスは以下のとおり。これは、PostgreSQL サーバ起動後、1 つのクライアントが接続している状態。

```
26233 pts/7    S      0:00 /usr/local/pgsql/bin/postgres            ← postgres デーモン
26238 ?        Ss     0:00 postgres: logger process                 ← 統計情報収集
26242 ?        Ss     0:00 postgres: writer process                 ← Background Writer
26243 ?        Ss     0:00 postgres: wal writer process             ← WAL Writer
26244 ?        Ss     0:00 postgres: autovacuum launcher process    ← autovacuum
26245 ?        Ss     0:00 postgres: archiver process               ← アーカイブログの管理
26246 ?        Ss     0:00 postgres: stats collector process        ← 統計情報収集
26248 ?        Ss     0:00 postgres: postgres testdb [local] idle   ← クライアント psql との接続
```

バックグランドライタは ver8.0 から追加されたプロセスで、高負荷時の CHECKPOINT 実行による性能低下を避けるために、バッファの変更内容を少しずつハードディスクに書き込む。

ver8.1 からバックグランドで VACUUM 処理を行なうために周期的に起動する、自動 VACUUM 機能のためのプロセスも追加された。

ver8.3 から周期的に(デフォルトでは 200msec 毎)WAL ログバッファ情報を WAL ログに書き込む、WAL Writer プロセスも追加された。

1.4. メモリ構造

PostgreSQL は起動時に、データ処理の効率化と信頼性向上のため、共有メモリ上に 3 つのメモリ領域を確保する(図 2)。また、バックエンドプロセスごとにワークメモリ領域(work_mem, ver7.4 までは sort_mem)とメンテナンスワークメモリ領域(maintenance_work_mem, ver7.4 までは vacuum_mem)を確保する。

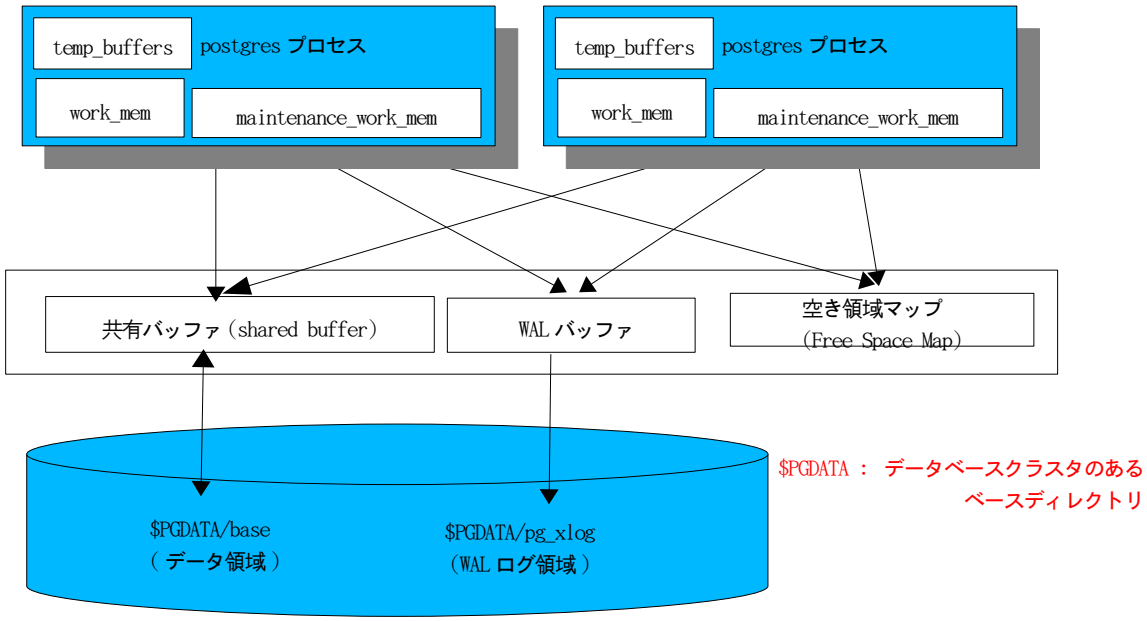


図 2 メモリ構造

1.4.1 共有メモリ上に確保するメモリ領域

(1)共有バッファ(shared buffer)

PostgreSQL は共有バッファ上にデータを読み込み、更新や検索などのデータ操作を行う。

(2)WAL バッファ(WAL buffer)

WAL バッファはトランザクションログをバッファリングする。

(3) 空き領域マップ(Free Space Map)

PostgreSQL は追記型のデータ管理方式を採用しているため、定期的にデータ領域中の不要領域を回収(または開放)する必要がある。空き領域マップは、不要となったデータ領域を記録する。

1.4.2 バックエンドプロセスが確保するメモリ領域

(1)ワークメモリ領域(work_mem)

プランナが問い合わせ実行計画を作成するときに使う、マージソート結合とハッシュ結合のためのメモリ領域。

(2)メンテナンスワークメモリ領域(maintenance_work_mem)

VACUUM 処理や CREATE INDEX 実行時に一時的に確保するメモリ領域。

(3)一時バッファ(temp_buffers)

一時テーブルにアクセスする時にのみ使用するメモリ領域。

1.4.3 データベースクラスタの構造

物理的なデータ本体や設定ファイルなど、データベースシステムの全データを保存する領域を、PostgreSQL では"**データベースクラスタ**"と呼ぶ。

以下にベースディレクトリのディレクトリ構造を示す。ベースディレクトリは図 2 中"$PGDATA"で示したディレクトリと同じ。入門書などでは"/usr/local/pgsql/data"が使われることが多い。

PG_VERSION	PostgreSQLのバージョン番号ファイル
pg_hba.conf	ホスト認証設定ファイル
pg_ident.conf	identによる認証ファイル
postgresql.conf	実行時パラメータ設定ファイル
postmaster.opts	起動オプション記録
base/	データ領域(データベースのデータを格納する)
global/	(コントロールファイルやパスワードファイルなど)共通オブジェクトの保存ディレクトリ
pg_clog/	ミットログをおくディレクトリ(コミットログはすべてのトランザクションのコミット状態を記録する)
pg_xlog/	WALログ(トランザクションログ)をおくディレクトリ.
pg_subtrans/	サクブトランザションの状態を記録する(バージョン8.0から)
pg_tblspc/	テーブルスペースへのシンボリックリンクを記録する(バージョン8.0から)
pg_twophase/	準備されたトランザクション(2相コミット)の状態を記録する(バージョン8.1から)
pg_multixact/	マルチトランザクションの状態を記録する。共有行ロックで使用(バージョン8.1から)

```
postgres> pwd
/usr/local/pgsql/data
postgres> ls
PG_VERSION  pg_clog        pg_log        pg_tblspc     postgresql.conf
base        pg_hba.conf    pg_multixact  pg_twophase   postmaster.opts
global      pg_ident.conf  pg_subtrans   pg_xlog       postmaster.pid
```

2. WAL、アーカイブログ、PITR(Point In Time Recovery)

2.1. WAL とアーカイブログ

2.1.1. WAL

WAL ログはすなわち、REDO ログのことである。

WAL の根本目的は「(1)コストのかかるデータ領域の更新を極力抑え」、且つ「(2)データ変更部分は WAL ログにシリアルに書き込む」ことで、書き込み性能と対障害性の両立を目指したものである。

後述するが、WAL ログは 16[Mbyte]のファイルで、ここにシリアルに REDO データを追加していく。

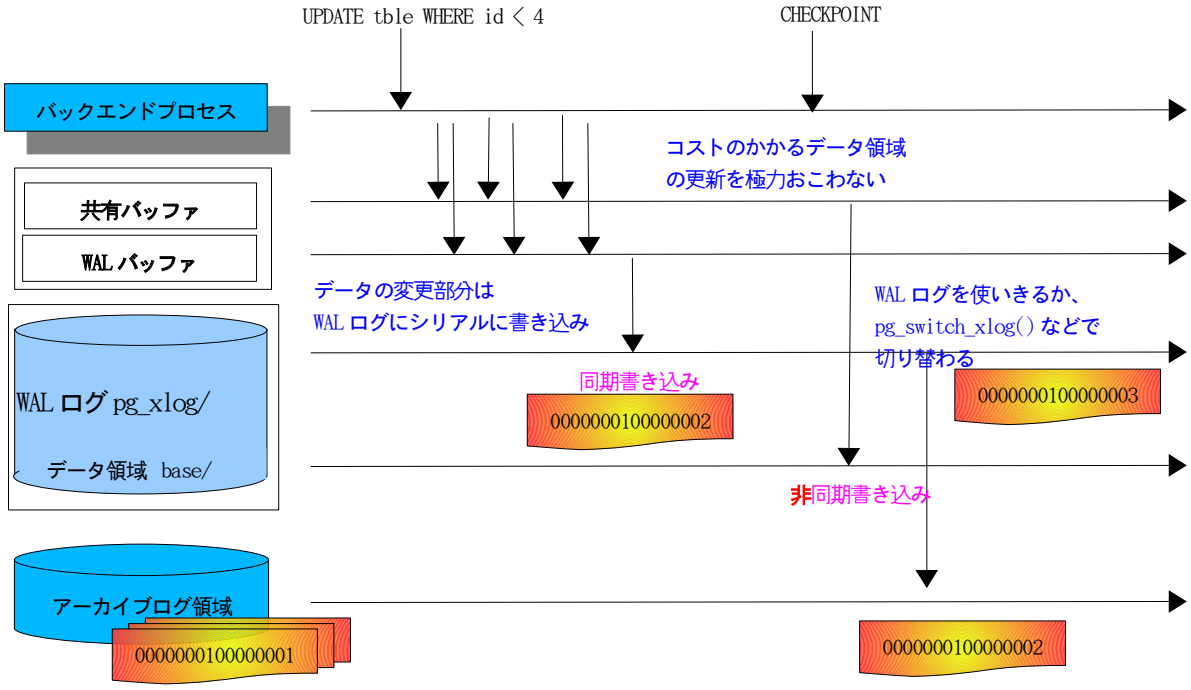


図 3 WAL とアーカイブログの概略

2.1.2. アーカイブログ

使い終わった WAL ログは「アーカイブログ」として、別ディレクトリに保存される。アーカイブログとは、本来なら消去される WAL ログに他ならない。

2.1.3. PITR(Point In Time Recovey)

ある時点でのデータベースクラスタと、それ以降のアーカイブログがあれば、任意の別サーバ上でデータベースのリカバリが可能となる。さらに、任意の時刻までのリカバリなど、PITR 機能は臨機応変な復旧手段を与える。

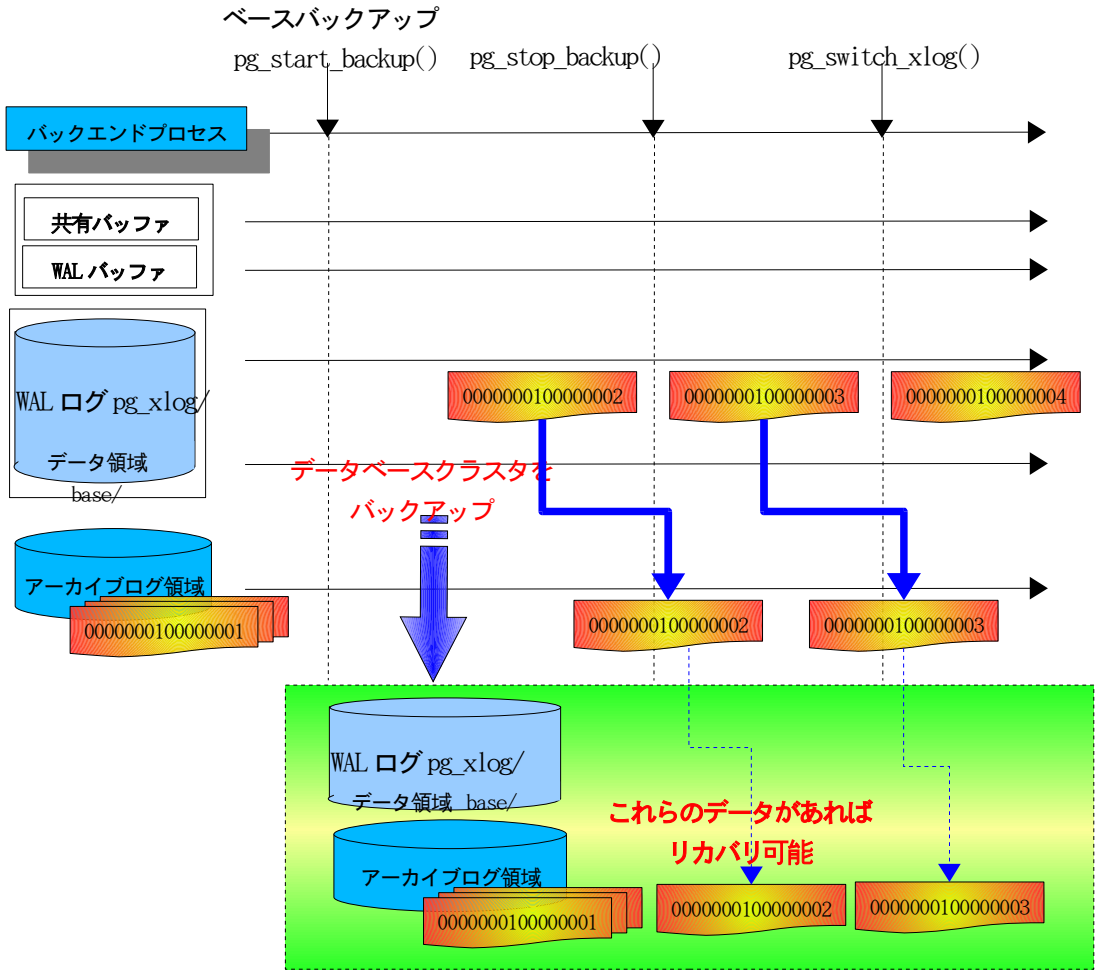


図 4 PITR の概略

2.1.4. PITR 以前のデータバックアップとリカバリ

PITR 以前のデータバックアップやリカバリは：

(1)PostgreSQL サーバを停止してデータベースクラスタ領域をコピーするか、

(2)pg_dump コマンドで(稼働中の PostgreSQL サーバから)SQL 文のリストを出力させるしかなかった(図 5)。

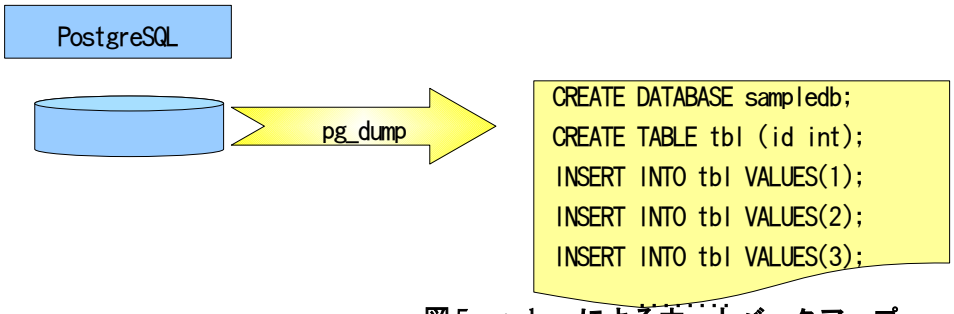


図 5 pg_dump によるホットバックアップ

これは、MySQL の mysqldump コマンドと同じであるが、ダンプデータは CSV 形式か INSERT 文の羅列で、しかも変更分だけをバックアップする差分バックアップにも対応していないので、 データサイズが大きくなり、大規模システムではとても扱い難かった。

3. WAL のおさらい

3.1.WAL

WAL の情報は(tli, xlogid, xrecoff)の３組の数値で管理する。ここで:

tli = *TimeLineId*[1]

xlogid = *ログ ID*

xrecoff = *ログ ID 毎のオフセット*

つまり、内部的に WAL ログは(図 4)のような管理がなされている(TimeLineId については省略)。

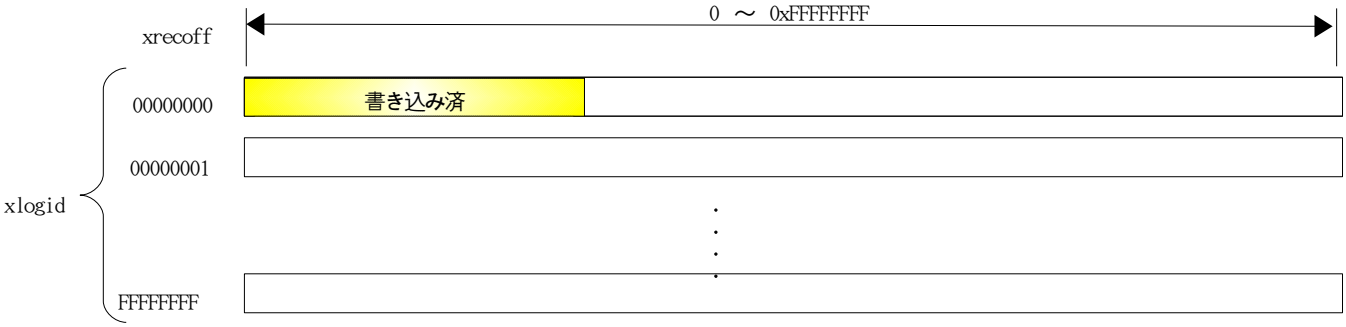


図４ WAL ログの内部管理

PostgreSQL には、現時点で WAL ログに書き込んだ位置(xlogid, xrecoff)表示する関数：pg_current_xlog_location()がある。

```
testdb=# SELECT pg_current_xlog_location();
 pg_current_xlog_location
--------------------------
  0/14FBD68                                    ←  xlogid = 0,  xrefoff = 14FBD68
(1 row)
```

内部管理においては xrefoff の値(0～0xFFFFFFFF)で問題ないが、pg_xlog ディレクトリ下やアーカイブログ領域に保存するには、ファイルサイズが巨大すぎる。

よって、16[Mbyte]毎に WAL ログを WAL ログの**セグメントファイル**に区切って管理する。

セグメントファイルの命名規則は次のとおり：

```
snprintf(fname, MAXFNAMELEN, "%08X%08X%08X", tli, xlogid, xrecoff / XlogSegSize);  /* XlogSegSize = 16[Mbyte] */
```

先ほど求めた WAL ログの位置(0,14FBD68)がどのセグメントに当たるか、関数 pg_xlogfile_name_offset()で求める。

1　今回は省略

```
testdb=# SELECT * FROM pg_xlogfile_name_offset('0/14FBD68');
        file_name         | file_offset
--------------------------+-------------
 00000001000000000000001 |     5225832
(1 row)
```

これによれば、'0/14FBD68'はWALログセグメントファイル名”00000001000000000000001”のオフセット5225832の部分にあることが分かる。

念のため、ログセグメントについて説明すると、”00000001000000000000001”はTimeLineID = 1、xlogID = 0で、セグメント番号が”00000001”である。

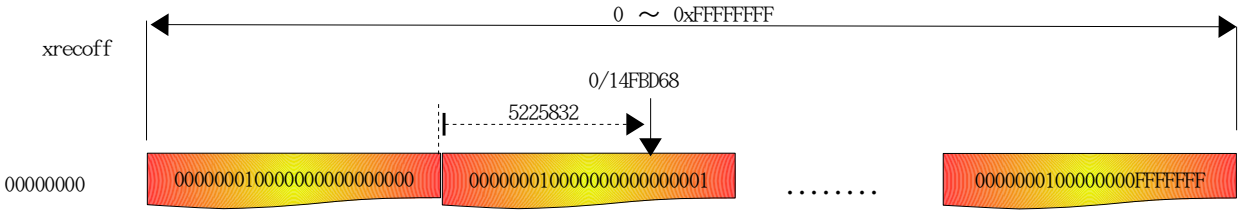


図5 WALログとセグメント

3.2. ソースコードの抜粋からみる、WAL の書き込みシーケンス

図3で示した WAL ログの書き込みシーケンスについて、関連するソースコードの一部を示す。ご覧のとおり、非常に簡略化したものである。図６と併せて参照のこと。なお、今回は CLOG についての記述は省略する。

```
exec_simple_query() @postgres.c                                    ……(1)
    /* parse, plan, portal */
    ExecUpdate() @execMain.c                    ← 1行目更新
        ExtendCLOG() @clog.c                    ← CLOGに”IN_PROGRESS”記述   ……(2)
        XLogInsert() @xlog.c                    ← WALログバッファに書き込み  ……(3)
            XLogCheckBuffer() @xlog.c
    ExecUpdate() @execMain.c                    ← 2行目更新
        XLogInsert() @xlog.c                    ← WALログバッファに書き込み  ……(4)
            XLogCheckBuffer() @xlog.c
    ExecUpdate() @execMain.c                    ← 3行目更新
        XLogInsert() @xlog.c                    ← WALログバッファに書き込み  ……(5)
            XLogCheckBuffer() @xlog.c
    finish_xact_command() @postgres.c
    XLogInsert() @xlog.c
    XLogFlush() @xlog.c
    XLogWrite() @xlog.c                         ← WALログに書き込み          ……(6)
    issue_xlog_fsync() @xlog.c                  ← 同期書き込みfsync() || fdatasync()実行
    TransactionIdSetStatus() @clog.c            ← CLOGに”COMMITED”記述       ……(7)
    finish_xact_command() @postgres.c
```

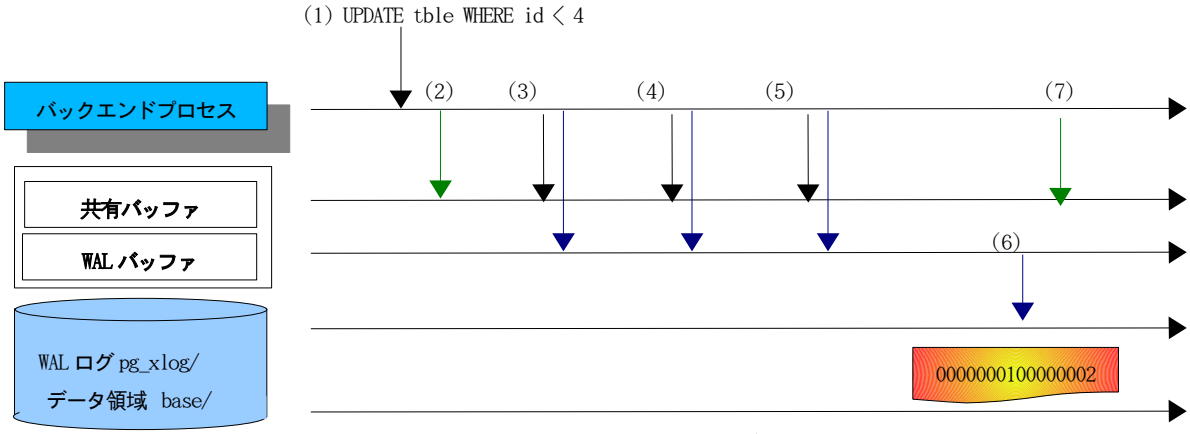


図６　ソースコード理解の参照図

3.2.1. wal writer プロセス

ver8.3から実装されたwal writerプロセスは、周期的にXLogBackgroundFlush()を実行するだけのバックグランドプロセスである。

```
walwriter.c
for (;;) {
        XlogBackgroundFlush()  / * @xlog.c */
        sleep(200msec);
}
```

4. PITRのおさらい

4.1. pg_start_backup()の動き

pg_start_backup()は、CHECKPOINTを実行し、backup_labelというファイルを作成。

ファイルdata/backup_labelはpg_stop_backup()が利用。

```
1)CHECKPOINT実行
        RequestCheckpoint(CHECKPOINT_FORCE | CHECKPOINT_WAIT);
2)ControlFileからCHECKPOINT情報を取得
        checkpointloc = ControlFile->checkPoint;
        startpoint = ControlFile->checkPointCopy.redo;
3)XLog, Segを計算
        XLByteToSeg(startpoint, _logId, _logSeg);
        XLogFileName(xlogfilename, ThisTimeLineID, _logId, _logSeg);
4)backup_label書き込み
       fp = AllocateFile(BACKUP_LABEL_FILE, "w");
       fprintf(fp, "START WAL LOCATION: %X/%X (file %s)\n", startpoint.xlogid, startpoint.xrecoff, xlogfilename);
       fprintf(fp, "CHECKPOINT LOCATION: %X/%X\n", checkpointloc.xlogid, checkpointloc.xrecoff);
       fprintf(fp, "START TIME: %s\n", strfbuf);
       fprintf(fp, "LABEL: %s\n", backupidstr);
       fflush(fp); ferror(fp) ;FreeFile(fp);
```

ここで、実際にpg_start_backup()関数を実行し、その結果生成されたbackup_labelを示す。

```
testdb=# SELECT pg_start_backup('walcheck');
 pg_start_backup
-----------------
 0/14FBD28
(1 row)
```

```
postgres> cat /usr/local/pgsql/data/backup_label
START WAL LOCATION: 0/14FBD28 (file 000000010000000000000001)
CHECKPOINT LOCATION: 0/14FBD28
START TIME: 2009-02-01 03:17:00 JST
LABEL: walcheck
```

pg_start_backup()関数を実行した時刻と、WALログの位置(0/14FBD28)が記入されている。

WAL ログのセグメント(ファイル名)とオフセット値は(00000001000000000000001, 5225768)である。

```
testdb=# SELECT * FROM pg_xlogfile_name_offset('0/14FBD28');
        file_name         | file_offset
--------------------------+-------------
 00000001000000000000001 |     5225768
```

念のため、WAL ログ領域とアーカイブログ領域のファイルを表示しておく。

WAL ログ領域では、使用中のセグメント“00000001000000000000001”と未使用の“00000001000000000000002”がある。

アーカイブログ領域には、使用済の“00000001000000000000000”が保存されている。

```
postgres> ls -1 pg_xlog/
00000001000000000000001          ← 使用中
00000001000000000000002          ← 作成済、未使用
archive_status

postgres> ls archive_log/
00000001000000000000000          ← アーカイブログとして保存済
```

4.1.1. オンラインリカバリ

今回の本題である pgpool-II のオンラインリカバリは、pg_start_backup()関数実行直後である現時点でのデータベースクラスタを元に行われる。

つまり、ここで作成した backup_label の情報から、確実に CHECKPOINT が実行された時刻や WAL の位置を求め、これをベースとしてリカバリを行うことになる[2]。

2　PostgreSQL のリカバリは、src/backend/access/transam/xlog.c 中の StartupXLOG()関数が行う。今回、リカバリのシーケンスを gdb で辿ったが、話をまとめきれなかった。機会があれば次回以降に説明したいと思う。今回はご容赦願いたい。

4.2. pg_stop_backup()の動き

pg_stop_backup()は、WAL ログ(セグメント)を切り替え、新規に backup_label を作成する。

```
(1)WAL ログ切り替え
      stoppoint = RequestXLogSwitch();
(2)backup_label 読み込み
        "START WAL LOCATION", "CHECKPOINT LOCATION", "START TIME", "LABEL"を読み込み。
(3)新規 backup_label 生成
        XLByteToSeg(stoppoint, _logId, _logSeg);
        XLogFileName(stopxlogfilename, ThisTimeLineID, _logId, _logSeg);
        XLByteToSeg(startpoint, _logId, _logSeg);
        BackupHistoryFilePath(histfilepath, ThisTimeLineID, _logId, _logSeg,
                                              startpoint.xrecoff % XLogSegSize);
        fp = AllocateFile(histfilepath, "w");
        fprintf(fp, "START WAL LOCATION: %X/%X (file %s)\n",
                    startpoint.xlogid, startpoint.xrecoff, startxlogfilename);
        fprintf(fp, "STOP WAL LOCATION: %X/%X (file %s)\n",
                    stoppoint.xlogid, stoppoint.xrecoff, stopxlogfilename);
        /* transfer remaining lines from label to history file */
        while ((ich = fgetc(lfp)) != EOF)                fputc(ich, fp);
        fprintf(fp, "STOP TIME: %s\n", strfbuf);
        fflush(fp);  ferror(fp); FreeFile(fp);
(4)旧 backup_label の削除
                CleanupBackupHistory();
```

ここで、実際に pg_stop_backup()関数を実行する。

```
testdb=# SELECT pg_stop_backup();
 pg_start_backup
-----------------
 0/14FBD84
(1 row)
```

backup_label のファイル名は:

WAL ログセグメント．オフセット値 ．'backup'

生成される backup_label は、pg_xlog 以下の”000000010000000000000001.004FBD28.backup”である。

実際にバックアップラベル”000000010000000000000001.004FBD28.backup”を表示する。pg_start_backup()関数によって生成されたbackup_labelの情報に”STOP WAL LOCATION”と”STOP TIME”の2項目のみ追加されたことがわかる。

```
postgres> cat /usr/local/pgsql/data/pg_xlog/000000010000000000000001.004FBD28.backup
START WAL LOCATION: 0/14FBD28 (file 000000010000000000000001)
STOP WAL LOCATION: 0/14FBD84 (file 000000010000000000000001)
CHECKPOINT LOCATION: 0/14FBD28
START TIME: 2009-02-01 03:17:00 JST
LABEL: walcheck
STOP TIME: 2009-02-01 03:18:31 JST
```

ここで、WALログのセグメントが切り替わったかどうか確認するため、最後にデータを書き込んだWALログの位置を求めるpg_current_xlog_location()関数と、次に書き込むWALログの位置を示すpg_current_xlog_insert_location()関数を実行する。

```
testdb=# SELECT * FROM pg_xlogfile_name_offset(pg_current_xlog_location());
        file_name         | file_offset
--------------------------+-------------
 000000010000000000000001 |    16777216
(1 row)

testdb=# SELECT * FROM pg_xlogfile_name_offset(pg_current_xlog_insert_location());
        file_name         | file_offset
--------------------------+-------------
 000000010000000000000002 |          32                    ←　新しいWALログセグメントに切り替わった
(1 row)
```

これにより、pg_stop_swtich_xlog()関数内のRequestXLogSwitch()関数によってWALログセグメントが切り替わっていることが確認できる。

念のため、data/pg_xlogとアーカイブログ領域のファイル群を示す。

```
postgres> ls -1 pg_xlog/
000000010000000000000001                    ←　使用済
000000010000000000000001.004FBD28.backup    ← バックアップファイル
000000010000000000000002                    ←　使用中
archive_status
postgres> ls -1 archive_log/
000000010000000000000000
000000010000000000000001                    ←　新たに追加されたログ
000000010000000000000001.004FBD28.backup
```

4.3. pg_switch_xlog()の動き

WAL ログを切り替え、アーカイブログ領域にコピー。

```
RequestXLogSwitch() @xlog.c
```

WAL ログの切り替えに関しては、pg_stop_backup()関数と同じ RequestXLogSwitch()を実行する。挙動も同じである。

Appendix

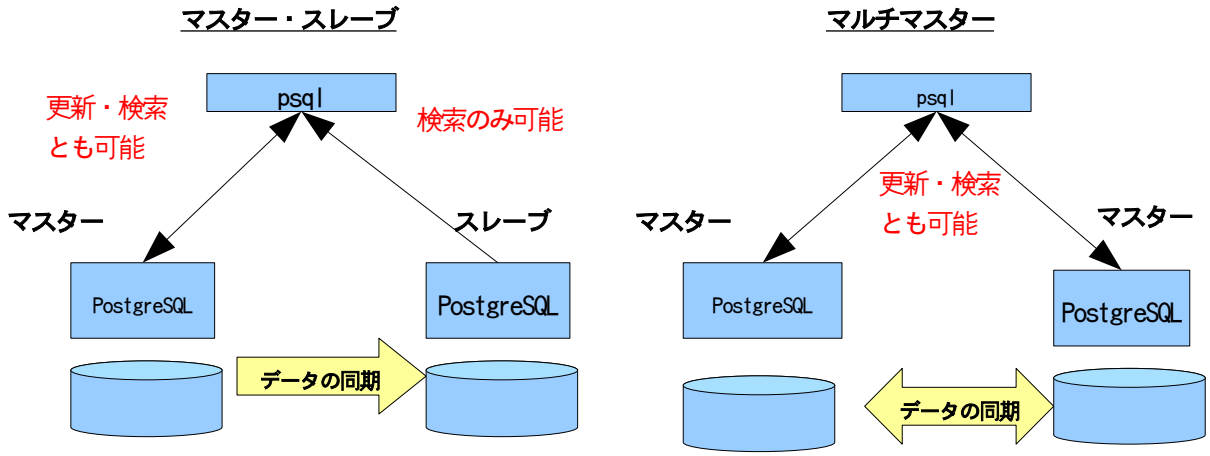
A-1. レプリケーション

A-1-1. レプリケーションの一般論

レプリケーション(Replication)とは、複数のDBMS間でデータの複製を持つこと。 レプリケーションは、次の2つの観点から分類できる。

●マスター・スレーブ/マルチマスター

マスターが更新(および検索)、スレーブは検索のみを行う‘マスター・スレーブ型’か、 すべてのサーバが更新と検索を自由に行う‘マルチマスター型’か(図A-1)。

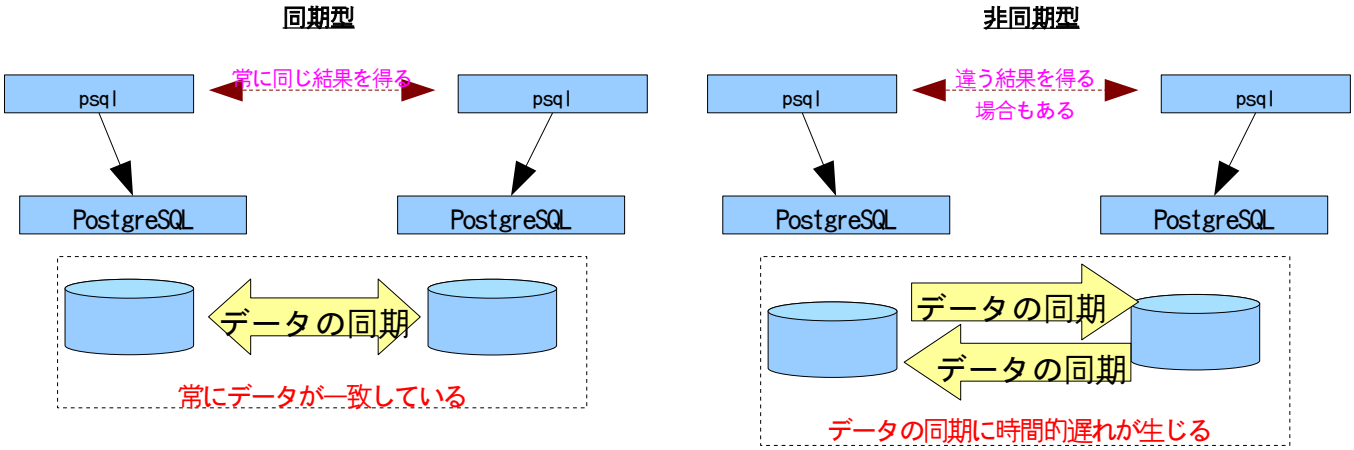


図A-1. マスター・スレーブとマルチマスター

●同期型/非同期型

データの複製は同期的に行うのか、非同期か(図A-2)。 同期型はマスター側の更新が確実にスレーブ側に反映されるまで、マスターの更新完了とならない。

非同期型はマスター側はスレーブ側の更新状況を考慮しない。 よって、データ検索のタイミングによってはマスターとスレーブのデータが一致しない瞬間もあり得る



図A-2. 同期型と非同期型

特に pgpool による同期レプリケーションは**図 A-2** と多少異なるので、改めて**図 A-3** に示す。

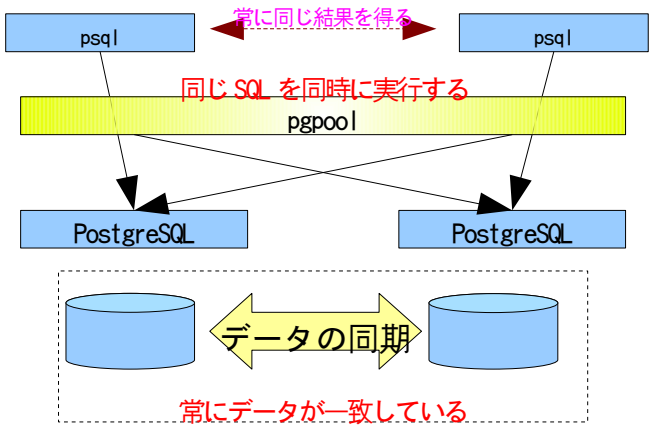


図 A-3 pgpool による同期レプリケーション

ただし、pgpool(に限らず全てのシステムにおいてであるが)の”同期レプリケーション”にはデリケートな面がある。pgpool はデフォルトでは性能を優先し、一方の SQL の完了を待たずに他方に SQL を投げる。しかし、複数のクライアントと pgpool プロセスがそれぞれ独立に稼働するので、微妙なタイミングでデッドロックやデータの不一致が起こる可能性がある。

詳細は以下の、pgpool 開発者石井さんの記事(中盤「デッドロック対策」)を参照して頂きたいが、pgpool には、確実に一方の SQL が終了するまで他方に SQL を投げない「ストリクト(strict)モード」がある。

http://itpro.nikkeibp.co.jp/members/ITPro/oss/20040422/2/

A-2. PostgreSQL のレプリケーションソフト群

現時点で開発が続行しているレプリケーションソフトの一覧を示す[3]。

プロジェクト	URL	同期/非同期	マスター・スレーブ/マルチマスター	レプリケーション単位	縮退運転	オンラインリカバリ
Slony-I	http://www.slony.info/	非同期	マスター・スレーブ	テーブル	可能	可能
pgpool-II	http://pgfoundry.org/projects/pgpool/	同期	マルチマスター	DB	可能	バージョン2から可能
PGCluster	http://pgfoundry.org/projects/pgcluster/	同期	マルチマスター	DB	可能	可能

これまでに開発されてきた PostgreSQL のレプリケーションソフト群を示す。商用ソフトの QuesryMaster は省略する。

プロジェクト	同期/非同期	マスター・スレーブ/マルチマスター	単位	備考
eRServer	非同期	マスター・スレーブ	テーブル	Slony-Iに技術移植
Rserv	非同期	マスター・スレーブ	テーブル	
DBMirror	非同期	マスター・スレーブ	テーブル	
pgReplicator	非同期	マスター・スレーブ	テーブル	
Usogres	同期	マスター・スレーブ	DB	最初期のソフト。役目を終え、自然消滅
PostgresForest	同期	マルチマスター	DB	
Postgres-R	同期	マルチマスター	DB	

3　Skype は PgBounder を開発している。詳細の把握が間に合わないので、今回は脚注に示すのみとする。
http://pgfoundry.org/projects/pgbouncer