# PostgreSQL 9.4 検証レポート

1.0.3 版
2014 年 8 月 15 日

SRA OSS, Inc. 日本支社
〒170-0022 東京都豊島区南池袋 2-32-8 8F
Tel. 03-5979-2701 Fax. 03-5979-2702
http://www.sraoss.co.jp/

# 目次

# 1. はじめに

2014 年 5 月 15 日、PostgreSQL 9.4 beta1 がリリースされ、7 月 24 日、同 beta2 がリリースされました。PostgreSQL 開発コミュニティーでは、次期メジャーバージョンアップむけに開発中ソフトウェアを順次にベータ版として評価検証用にリリースしていきます。

本ドキュメントは、9.4 に含まれる主要な新機能を説明し、実際に動作させた検証結果を報告するものです。PostgreSQL 9.4 について検証しようとしているユーザの助けとなることを目的としています。

# 2. 概要

以下に PostgreSQL 9.4 の主要な新機能を挙げます。本文書ではこれらの項目を取り上げます。

- バイナリ JSON データ型
- GIN インデックスの性能向上
- WAL 書き込みの性能向上
- postgresql.conf 設定を変更する ALTER SYSTEM 構文
- pg_prewarm 拡張モジュール
- 遅延レプリケーション
- レプリケーションスロット
- ロック競合しないマテリアライズドビューリフレッシュ
- 更新ビューの CHECK オプション
- 集約クエリーの FILTER オプション
- 行ロックエラーのメッセージ拡張
- pg_stat_archive ビュー
- 論理変更内容出力

この他にも細かな機能追加や変更が多数あります。全ての変更点の一覧については PostgreSQL 9.4 ドキュメント内のリリースノート（ベータ版にも付加されています）に記載されています。また、これらは beta バージョン段階での機能実装です。9.4 バージョンの本リリースまでにいくつか削除や変更される可能性があります。

# 3. 検証のためのセットアップ

## 3.1. ソフトウェア入手

PostgreSQL 9.4 ベータ版 は以下ＵＲＬからダウンロード可能です。本検証作業においては、beta1 を使用しています。ただし、いくつかの項目については 9.4beta2 にバージョンアップして行いました。

```
http://www.postgresql.org/download
```

## 3.2. 検証環境

検証環境として、 HP ProLiant MicroServer Gen8 マシン（Pentium G2020T 1P2C／memory:10GB／HDD:1TB 7.2krpm SATA）上の CentOS 6.5（Linux 2.6.32 x86_64）を使用しました。

```
[postgres]$ cat /etc/redhat-release
CentOS release 6.5 (Final)
```

/home/postgres をホームディレクトリとする postgres ユーザにて、ソースコードのビルドと PostgreSQL の動作を行うものとしました。

## 3.3. インストール

以下のオプションにてソースコードをビルドしました。

```
[postgres]$ cd postgresql-9.4beta1
[postgres]$ ./configure --prefix=/home/postgres/pgsql/9.4beta1 \
                        --with-libxml --enable-debug
[postgres]$ make install-world
```

比較検証のため、9.3.4 も同オプションでインストールしています。

```
[postgres]$ cd postgresql-9.3.4
[postgres]$ ./configure --prefix=/home/postgres/pgsql/9.3.4 \
                        --with-libxml --enable-debug
[postgres]$ make install-world
```

環境変数の設定をファイルにまとめ、適用します。

```
[postgres]$ cat > 9.4beta1.env <<'EOF'
export PATH=/home/postgres/pgsql/9.4beta1/bin:$PATH
export LD_LIBRARY_PATH=/home/postgres/pgsql/9.4beta1/lib:$LD_LIBRARY_PATH
export PGDATA=/home/postgres/pgdata/9.4data
EOF
[postgres]$ cat > 9.3.4.env <<'EOF'
export PATH=/home/postgres/pgsql/9.3.4/bin:$PATH
export LD_LIBRARY_PATH=/home/postgres/pgsql/9.3.4/lib:$LD_LIBRARY_PATH
export PGDATA=/home/postgres/pgdata/9.3data
EOF
[postgres]$ . 9.4beta1.env
```

以下、9.3.4 については 9.4beta1 と同じオプションにて設定を行っています。

データベースクラスタを作成します。ロケール無し（C ロケール）、UTF8 をデフォルトとし、9.3 で追加されたデータチェックサム機能を有効としました。

```
[postgres]$ initdb --no-locale --encoding=UTF8 --data-checksums
The files belonging to this database system will be owned by user "postgres".
This user must also own the server process.
 （中略）
Success. You can now start the database server using:

  postgres -D /home/postgres/pgdata/9.4data
or
  pg_ctl -D /home/postgres/pgdata/9.4data -l logfile start
```

設定ファイルに最小限の設定を与えます。

```
[postgres]$ cat >> $PGDATA/postgresql.conf << EOF
logging_collector = on
log_line_prefix = '%t[%p] '
EOF
```

PostgreSQL を起動します。

```
[postgres]$ pg_ctl -w start
waiting for server to start....2014-06-18 17:10:21 JST[30302] LOG:  redirecting
log output to logging collector process
2014-06-18 17:10:21 JST[30302] HINT:  Future log output will appear in directory
"pg_log".
 done
server started
```

## ◆ サンプルテーブルの定義とデータ

本文中で使用するサンプルデータベース、サンプルテーブルは以下のように作成しています。

```
[postgres]$ createdb db1
[postgres]$ psql -q db1 <<'EOS'

DROP TABLE IF EXISTS t_item, t_custpmer, t_sales;

CREATE TABLE t_item (id int primary key, name text);
INSERT INTO t_item SELECT g, 'ITEM' || g FROM
  generate_series(1, 100) as g;

CREATE TABLE t_customer (id int primary key, name text);
INSERT INTO t_customer SELECT g, 'CUSTOMER' || g
  FROM generate_series(1, 1000) as g;

CREATE TABLE t_sale (id int primary key, iid int references t_item (id),
  cid int references t_customer (id), amount int, ts timestamp);
INSERT INTO t_sale SELECT g, ceil(random() * 100),
  ceil(random() * 1000), (random() * 10000)::int,
  now() - ((random() * 1000)::int || 'day')::interval
  FROM generate_series(1, 100000) AS g;

EOS
```

# 4. 主要な追加機能／性能向上の検証

主要な追加機能、性能向上について動作確認をしていきます。また、合わせて機能の簡単な説明もします。

各追加機能の詳細な説明はベータ版に同梱されるマニュアルに記載されています。本インストール手順を行った場合、以下の場所（インストール先の share/doc/html）に HTML のマニュアルが生成されます。

```
/home/postgres/pgsql/9.4beta1/share/doc/html/
```

また、以下 URL にて開発中バージョンのマニュアルが公開されています。いずれも英語となります。

```
http://www.postgresql.org/docs/9.4/static/
```

## 4.1. バイナリ JSON データ型

PostgreSQL 9.4 から新たに JSONB データ型が追加されました。これはバイナリの JSON データ型です。旧来の JSON データ型とほぼ同じように JSON データの格納ができますが、テキストデータではなくバイナリデータとして格納されます。また、GIN インデックスが適用可能となります。そのため、使用用途によっては大幅な性能向上が得られます。

### ◆ JSON と JSONB の違い

JSON データ型と JSONB データ型の違いを以下表に示します。

| | JSON | JSONB |
|---|---|---|
| 格納方式 | テキスト | バイナリ |
| 構文解析 | 各処理のつど行う | 格納時に行う |
| 非正規情報の保持 | できる | できない、正規化される |
| GIN インデックス適用 | できない | できる |

非正規情報の保持とは、不要な空白や定義の重複などについて、正規化する前のデータを保持するかどうかということです。以下のようなデータが JSON では互いに区別され、JSONB では区別されません。

```
'{"a":1,"b":2,"c":3}'
'{  "a":1,  "b":2,  "c":3  }'                ← 不要な空白
'{"a":1, "b":200, "b":20,"b":2, "c":3}'      ← 定義の重複（後者が優先される）
```

JSONB でキャストする例を示します。

```
db1=# SELECT '{"a":1,"b":2,"c":3}'::jsonb =
             '{"a":1, "b":200, "b":20,"b":2, "c":3}'::jsonb;
-[ RECORD 1 ]
 ?column? |  t
```

### ◆ *JSON* 関連の新たな関数・演算子

PostgreSQL 9.3 から JSON 用に存在した関数の大部分は JSONB でも利用可能となっています。

PostgreSQL 9.4 では以下の JSONB 型専用の演算子が追加されています。これら演算子で「含まれる」というのは、第一階層のみであることに注意してください。入れ子構造の内側までは検索してくれません。

| 演算子 | 意味 |
|---|---|
| @> | 右辺の JSONB は、左辺の JSONB に含まれるか？ |
| @< | 左辺の JSONB は、右辺の JSONB に含まれるか？ |
| ? | 右辺の文字列は、左辺の JSONB のキーまたは配列要素に含まれるか？ |
| ?\| | 右辺の文字列配列の何れか要素は、左辺の JSONB のキーまたは配列要素に含まれるか？ |
| ?& | 右辺の文字列配列の全ての要素は、左辺の JSONB のキーまたは配列要素に含まれるか？ |

JSON 型用に以下のデータ生成関数が追加されています。値の並びから JSON データを生成します 。JSONB 型には結果をキャストして適用します。

| 関数 | 引数例 | 出力例 |
|---|---|---|
| `json_build_array(VARIADIC "any")` | `'AAA','BBB','CCC'` | `["AAA", "BBB", "CCC"]` |
| `json_build_object(VARIADIC "any")` | `'a',1234,'b',5678` | `{"a" : 1234, "b" : 5678}` |
| `json_object(text[])` | `'{{a,xx},{b,yy}}'` | `{"a" : "xx", "b" : "yy"}` |
| `json_object(keys text[], values text[])` | `'{a, b}', '{xx,yy}'` | `{"a" : "xx", "b" : "yy"}` |

JSON 処理用の各関数は、`json_array_length(json)` と `jsonb_array_length(jsonb)` のように JSON 型用、JSONB 型用の 2 つペアで提供されるようになりました。

従来からある JSON 関数の jsonb 版が加わったほか、以下の関数も追加されています。表では JSON 型用の関数名を示していますが、同じ働きをする JSONB 型関数も追加されています。

| 関数名 | 意味 |
|---|---|
| json_array_elements_text(json) | json_array_elemsnts(json) と同じ動作ですが、返し値型が setof json でなく setof text となります |
| json_typeof(json) | 単一要素 JSON データに対して、JSON のデータ型名を返します |
| json_to_record(json) | JSON 型データを行データに変換します |
| json_to_recordset(json) | JSON 型データを複数行データに変換します |

## 4.2. GIN インデックスの性能向上

GIN インデックスは、配列型や全文検索用テキストのように一つの値の中に複数の要素をもつようなデータ型に対して、「この要素を含むか？」といった問い合わせをする場合に適用できるインデックスです。

PostgreSQL 9.3.5 と 9.4beta2 を使って、以下の通り性能比較テストを行いました。

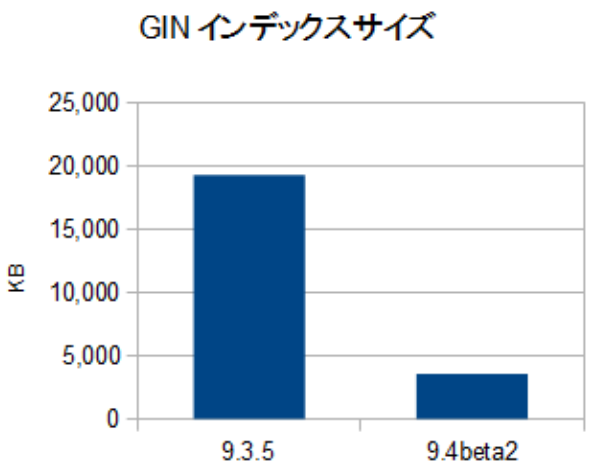
先ずは以下の手順でテスト用のテーブルとインデックスを作ります。text 型の配列の tags カラムに GIN インデックスを作成します。50 万行のデータを持ち、各々ランダムな 10 個の数字タグを持ちます。

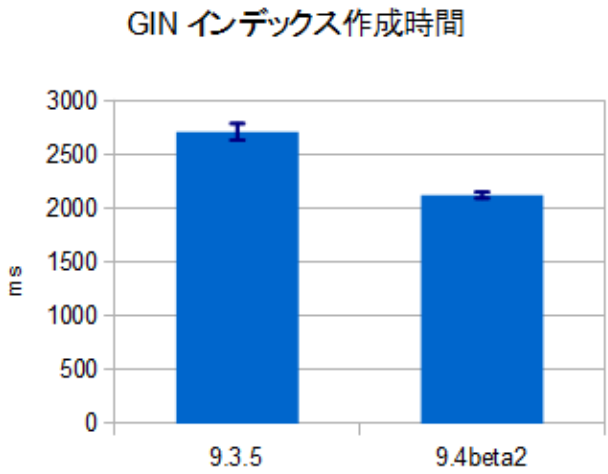
```
db1=# CREATE TABLE ttag (id int PRIMARY KEY, tags text[], dat text);
db1=# INSERT INTO ttag SELECT
  g, regexp_split_to_array(right(random()::text, 10), ''), md5(g::text)
  FROM generate_series(1, 500000) AS g;
db1=# \timing
db1=# CREATE INDEX idx_ttag_tag ON ttag USING gin (tags);
Time: 3698.789 ms
db1=# REINDEX INDEX idx_ttag_tag;
 （REINDEX を繰り返す／後略）
```

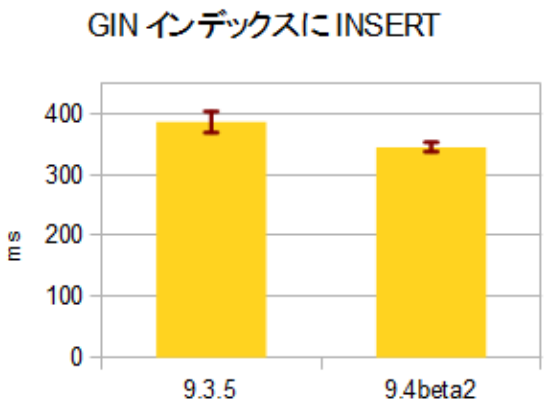
グラフ「GIN インデックスサイズ」は作成されたインデックスサイズを pg_relation_size 関数で調べた値です。サイズが 5 分の 1 程度に縮小していることがわかります。

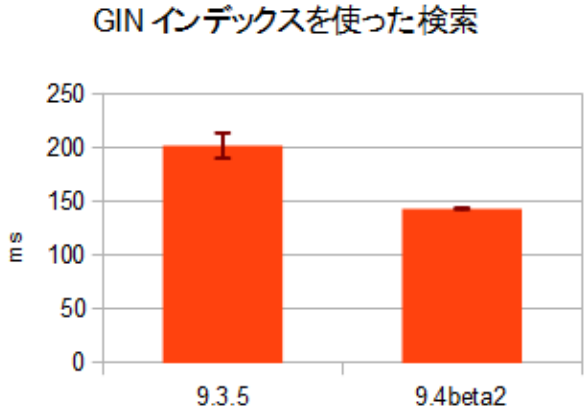
グラフ「GIN インデックス作成時間」は psql の \timing オプションで GIN インデックスに対する REINDEX コマンドの平均所要時間（エラーバーは標準偏差）です。 インデックス作成性能で 20% ほど性能がアップしていることがわかります。

グラフ「GIN インデックスに INSERT」は、50 万件のデータを投入した後、1 万件ずつデータを INSERT したときの平均所要時間（エラーバーは標準偏差）です。 本データはチェックポイント処理を避ける形で採取しています。それほど大きな差はありませんが若干、9.4beta2 が速い結果となりました。




グラフ「GIN インデックスを使った検索」は以下の SQL を実行した平均所要時間（エラーバーは標準偏差）です。9.3beta2 が 25%ほど短い所要時間となり、性能改善されていることがわかります。

```
db1=# SELECT count(*) FROM ttag
      WHERE tags @> ARRAY['1', '2', '3', '4', '5'];
 count
-------
 40304
(1 row)
Time: 166.171 ms
（後略／両バージョンで SELECT 実行を繰り返す）
```

SQL はタグ '1'、'2'、'3'、'4'、'5' がすべて付いている行を数えるというものです。なお、データはそれぞれにランダム生成していますが、該当件数は、バージョン 9.3.5 が 39865 件、バージョン 9.4beta2 が 40304 件と１％程度の違いでしたのでそのまま比較するものとしました。

なお、本 SQL はどちらのバージョンでも GIN インデックスが無いと、500ms 以上かかりました。

## 4.3. WAL 書き込みの性能向上

WAL とは Write Ahead Log の略で、PostgreSQL におけるトランザクションログファイルのことです。データ変更を伴う SQL 処理は必ず WAL への書き込みを発生させます。PostgreSQL 9.4 では WAL 書き込みの性能改善が 2 つの点で図られています。

第一には、同じ更新内容に対して必要とする WAL 書き込み量が小さくなりました。これは内部的にデータ圧縮処理が組み込まれることで実現されています。

第二には、WAL 書き込みにあたっての排他制御が改善され、同時実行処理においてより効率的に動作するようになりました。CPU コア数が多いシステムで効果が期待できます。このために設定パラメータ xloginsert_locks が新設されました。デフォルト値 8 は 8CPU コア以上あるマシンで 8 並列以上の同時更新処理をするとしても、おおむね効果が頭打ちとなる値とされています。

```
xloginsert_locks = 8    # Sets the number of locks used for
                        # concurrent xlog insertions.
```

### ◆ WAL 書き込み量縮小の検証

PostgreSQL 9.3.4 と PostgreSQL 9.4beta1 とで pgbench を行って、WAL 書き込み位置を比較してみます。このとき自動 VACUUM を無効に設定した、両バージョンの PostgreSQL に対して以下を行いました。

```
$ pgbench -i db1 -F 80          ← 最初に pgbench 用データを生成
                ↓ 開始前のトランザクションログ位置を確認
$ psql -c 'SELECT pg_current_xlog_insert_location()' db1
 pg_current_xlog_insert_location
---------------------------------
 0/7D2AB08
(1 row)

$ pgbench -c 5 -t 10000 -n    ← 更新トランザクションを計５万回実行
（出力省略）
                ↓ 終了時のトランザクションログ位置
$ psql -c 'SELECT pg_current_xlog_insert_location()' db1
 pg_current_xlog_insert_location
---------------------------------
```

```
 0/0BF0DAF8
(1 row)
```

以上のようにして、開始時点と終了時点のトランザクションログ位置を確認できます。

小さな機能追加であるため本文書では項目は立てていませんが PostgreSQL 9.4 の新機能の一つに pg_lsn データ型があります。トランザクションログ（WAL）の位置を示す '0/0CCCBFD8' という表現方法をデータ型として扱い、演算もできるようになりました。以下のようにして何バイト進んだかを計算できます。

```
postgres=# SELECT '0/0CCCBFD8'::pg_lsn - '0/86CE9A0'::pg_lsn;
 ?column?
----------
 73389624
(1 row)
```

結果は以下のようになりました。６%ほど書き込み量が減っていることがわかります。

| バージョン | 開始時点 | 終了時点 | 進行量 | 比率 |
|---|---|---|---|---|
| 9.3.4 | 0/86CE9A0 | 0/0CCCBFD8 | 73,389,624 byte | 100 |
| 9.4beta1 | 0/7D2AB08 | 0/0BF0DAF8 | 69,087,216 byte | 94.1 |

次に文字列カラムを更新するカスタムシナリオで実験した結果、以下のように 15%に出力が縮減される効果が確認できました。このことから更新データ内容によって効果が変わることが分かります。

カスタムシナリオを使った場合の結果：

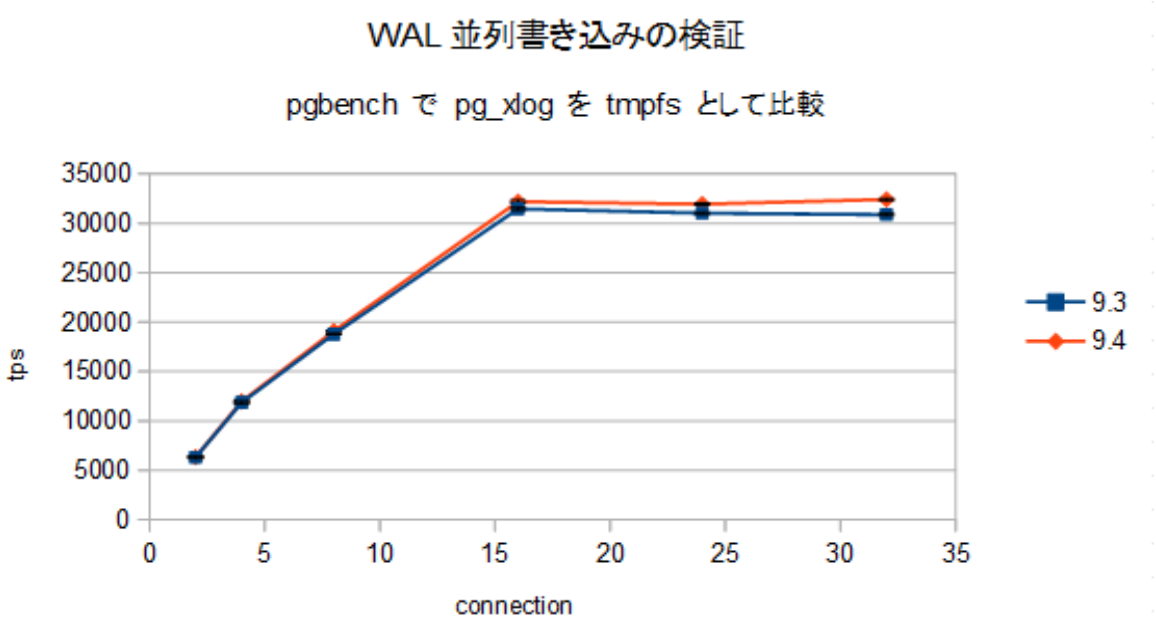
| バージョン | 開始時点 | 終了時点 | 進行量 | 比率 |
|---|---|---|---|---|
| 9.3.4 | 0/118C8960 | 0/13DA9958 | 38,670,328 byte | 100 |
| 9.4beta1 | 0/112DF3C0 | 0/13220C90 | 32,774,352 byte | 84.7 |

```
 ↓ pgbench カスタムシナリオ定義
 \setrandom aid 1 100000
 \setrandom delta -5000 5000
 UPDATE pgbench_accounts SET filler = repeat(' ', 70) || :delta WHERE aid =
:aid;
```

### ◆ 同時実行更新処理の検証

本改修項目が効果を発揮するのは以下のような状況と考えられます。

- CPU コア数が多く、同時実行数も多い
- 更新トランザクションである
- I/O 律速ではない




上記グラフ「WAL 並列書き込みの検証」は、8CPU コアの物理マシンを別途用意のうえ、WAL 出力先を RAM ディスク（tmpfs）とするなど、本改修内容に当てはまるように条件を調整して pgbench 標準テストを行った結果です（エラーバーは標準偏差）。若干の性能向上が見られます。

なお、小規模マシン（8CPU コア以下のマシン）にて様々な条件でベンチマークを行いましたが、ほとんどの場合には、性能差が出ない結果となっています。多 CPU コアと SSD 等の高速ストレージを備えたハイスペックマシンにおいて効果のある改善であると考えられます。

## 4.4. *postgresql.conf* 設定を変更する *ALTER SYSTEM* 構文

以下構文の ALTER SYSTEM コマンドが加わり、postgresql.conf の設定を SQL コマンドから変更できるようになりました。

```
ALTER SYSTEM SET parameter { TO | = } { value | 'value' | DEFAULT }
```

### ◆ 設定パラメータとコンテキスト区分

PosgreSQL 設定パラメータは、postgresql.conf における各行の設定と SET コマンドによる設定とが統一的に管理されています。GUC（Grand Unified Configuration）とも呼ばれます。

PostgreSQL 設定パラメータには、いつ誰がどのような影響範囲で設定変更できるかを区分する以下表のコンテキスト区分があります。以下の SQL コマンドで各設定パラメータのコンテキスト区分を確認できます。

```
SELECT name, context FROM pg_settings; -- パラメータ区分の確認
```

| コンテキスト | 意味 |
|---|---|
| user | 一般ユーザが SET コマンドで現在の接続について値を変更できるパラメータ。postgresql.conf の値はデフォルト値を提供する。 |
| superuser | 管理者ユーザ（postgres）が SET コマンドで現在の接続について値を変更できるパラメータ。postgresql.conf の値はデフォルト値を提供する。 |
| sighup | SET 文では変更できず、postgresql.conf 修正と pg_ctl reload （マスタプロセスへの HUP シグナル）で変更するパラメータ。現在のすでにある接続についても変更が反映されます。 |
| backend | SET 文では変更できず、postgresql.conf 修正と pg_ctl reload （マスタプロセスへの HUP シグナル）で変更するパラメータ。次の新たな接続から反映されます。 |
| postmaster | 変更を反映するために postgresql.conf の記述を変更して、PostgreSQL マスタプロセスの再起動が必要となるパラメータ。 |
| internal | インスタンス作成時点（initdb 時点）や PostgreSQL ビルド時点で固定され、変更不能なパラメータ。 |

### ◆ *ALTER SYSTEM* コマンドによる拡張

ALTER SYSTEM コマンドは、指定されたパラメータについて、データベースクラスタディレクトリ下の postgresql.auto.conf ファイルに書き込みます。postgresql.auto.conf の内容は次に PostgreSQL を 設定リロードしたり、起動・再起動するときに postgresql.conf に優先して読み込まれます。

ALTER SYSTEM コマンド自体ではリロード（pg_ctl reload） に相当する処理は行いません。しかしながら、リロードを行う関数 `pg_reload_conf()` がありますので、コンテキスト区分上可能な設定パラメータについては、データベース接続を通して設定変更と反映ができることになります。

```
db1=# ALTER SYSTEM SET shared_buffers TO '20MB';
db1=# ALTER SYSTEM SET work_mem TO '20MB';
db1=# SHOW work_mem;                                ←  直ぐには反映されません
 work_mem
----------
 2MB
```

```
(1 row)

db1=# \! cat $PGDATA/postgresql.auto.conf        ← ファイルに書き出されます。
# Do not edit this file manually!                  リロード時・次回起動時に
# It will be overwritten by ALTER SYSTEM command.  読み込まれます。もともと、
work_mem = '20MB'                                  リロードでは反映できない
shared_buffers = '20MB'                            項目は再起動が必要です。

db1=# ALTER SYSTEM SET shared_buffers TO DEFAULT;
db1=# \! cat $PGDATA/postgresql.auto.conf        ← postgresql.auto.conf
# Do not edit this file manually!                  ファイルから DEFAULT に
# It will be overwritten by ALTER SYSTEM command.  戻した項目が除去されます。
work_mem = '20MB'
```

ALTER SYSTEM コマンドで値を「DEFAULT」に設定すると、postgresql.auto.conf から当該パラメータに対する設定記述が除去されます。その場合は、通常通り postgresql.conf の値が読み込まれることになります。

## 4.5. pg_prewarm 拡張モジュール

ソースコード一式に同梱される contrib 拡張モジュールの一つとして pg_prewarm が追加されました。本モジュールをインストールすると指定したテーブル・インデックスをバッファに載せるように明示的に指示できます。本モジュールは、PostgreSQL を再起動した直後はバッファヒットしないため、極端に性能が落ちる、早期にバッファ上にデータある状態にしたい、といった用途に応えるものです。

### ◆ pg_prewarm の動作確認

以下のように pgbench で データを生成して、pg_prewarm 前後でバッファ状態を確認してみます。

```
$ pgbench -q -i -s 10 db1          ← データ生成
creating tables...
731300 of 1000000 tuples (73%) done (elapsed 5.04 s, remaining 1.85 s).
1000000 of 1000000 tuples (100%) done (elapsed 6.72 s, remaining 0.00 s).
vacuum...
set primary keys...
done.
```

```
$ pg_ctl restart                ← 再起動すると共有バッファ内容はリセットされる
$ psql -q -x db1

       ↓pg_prewarm の他、共有バッファ確認のために pg_buffercache もインストール
db1=# CREATE EXTENSION pg_prewarm;
db1=# CREATE EXTENSION pg_buffercache;

       ↓共有バッファに指定したテーブルが何ページ載っているかを確認するクエリー
db1=# SELECT count(*) FROM pg_buffercache WHERE relfilenode = (
     SELECT relfilenode FROM pg_class WHERE relname = 'pgbench_accounts');
-[ RECORD 1 ]
count | 0

       ↓pg_prewarm を実行、返し値はバッファに載ったページ数（1 ページは 8KB）
db1=# SELECT * FROM pg_prewarm('pgbench_accounts');
-[ RECORD 1 ]-----
pg_prewarm | 16394

       ↓共有バッファに大部分のページが載ったことが確認できる
db1=# SELECT count(*) FROM pg_buffercache WHERE relfilenode = (
     SELECT relfilenode FROM pg_class WHERE relname = 'pgbench_accounts');
-[ RECORD 1 ]
count | 16295
```

### ◆ *pg_prewarm* オプション

pg_prewarm 関数の第一引数にはテーブル名、インデックス名を指定しますが、オプションとして第二引数～第五引数を指定することができます。引数の意味は以下表の通りです。

| | 指定する値 | 意味 |
|---|---|---|
| 第二引数 | 'buffer'（デフォルト） | PostgreSQL の共有バッファに載せます。 |
| | 'read' | OS 上の命令で指定されたデータを読み込み、OS のバッファに載せます。共有バッファには載せません。 |
| | 'prefetch' | OS に対して指定されたデータは近い将来アクセスされると知らせます。OS 側で事前にデータ読み取りをして OS のバッファに載せてくれるかもしれません。 |
| 第三引数 | 'main'（デフォルト） | テーブル・インデックスの本体データを対象とします。<br>フリースペースマップ、ビジビリティマップは比較的小さいファイルなので、本指定 'main' 以外を必要とするケースはあまりないものと考えられます。 |
| | 'fsm' | テーブルのフリースペースマップを対象とします。 |
| | 'vm' | テーブルのビジビリティマップを対象とします。 |
| 第四引数 | 始点ページ番号 | ページ番号で範囲指定をしてテーブル・インデックスの一部だけをバッファに載せます。デフォルトは両方とも NULL で、全体をバッファに載せるという意味です。 |
| 第五引数 | 終点ページ番号 | |

なお、外部配布されているツール pgfincore でも同様の機能が実現されています。PostgreSQL 9.3 以前のバージョンでは pgfincore を使うものとなります。以下に pgfincore 説明文書の URL を示します。

http://git.postgresql.org/gitweb/?p=pgfincore.git;a=blob_plain;f=README.rst;hb=HEAD

## 4.6. 遅延レプリケーション

「レプリケーションはバックアップの代替にならない」ということがしばしば言われます。これは、操作ミスやアプリケーションプログラムの誤りに基づく SQL 実行によってデータを破壊した場合に、その破壊がレプリケーション先にも伝搬してしまうためです。

PostgreSQL9.4 では、recovery.conf にスタンバイサーバのレプリケーション適用をわざと遅延させるオプション `recovery_min_apply_delay` が追加されました。以下の動作テストに示すようにスタンバイサーバへの変更の適用（replay）が指定した時間だけ遅延されます。

```
$ cat >> $PGDATA/postgresql.conf <<'EOS'     ← レプリケーション用の最小限設定
wal_level = hot_standby
max_wal_senders = 2
hot_standby = on
hot_standby_feedback = on
EOS
$ pg_ctl restart                             ← PostgreSQL 再起動して反映
      ↓  recovery.conf 生成付きでベースバックアップ取得する
$ pg_basebackup -D ${PGDATA}s --xlog --write-recovery-conf
      ↓  遅延を 20min （20 分）に設定
$ cat >> ${PGDATA}s/recovery.conf <<'EOS'
recovery_min_apply_delay = '20min'
EOS
$ pg_ctl start -D ${PGDATA}s -o '-p 5433'    ← スタンバイ PostgreSQL を起動
$ psql db1
db1=# DELETE FROM t_sale ;                   ← プライマリでデータを削除
db1=# DROP TABLE t_customer ;                ← プライマリでテーブルを削除
db1=# \q
$ psql -q -p 5433 db1
db1=# SELECT count(*) FROM t_sale;           ← スタンバイに 20 分間は反映されない
 count                                           実際の用途としては「１日遅れ」
--------                                         くらいが有用といえる。
 100000
(1 row)
db1=# SELECT * FROM t_customer ;             ←  削除済みテーブルのアクセスも可能
 （後略）
```

なお、遅延するのはあくまで、WAL データの適用（replay）だけであって、WAL データの転送を遅延させる効果はありません。

## 4.7. レプリケーションスロット

PostgreSQL 9.4 からレプリケーションスロット機能が追加されました。これは、ストリーミングレプリケーションを構成するにあたり、プライマリサーバがスタンバイサーバの必要とする WAL ファイルを削除したり、スタンバイサーバが参照しているデータを物理レベルで削除したりしないように管理するものです。

これまでも、アーカイブ WAL の転送を併用したり、wal_keep_segments 設定で WAL ファイルを長く保持させたり、hot_standby_feedback 設定や vacuum_defer_cleanup_age 設定でデータ物理削除を避けるという方法が用意されていました。本機能はこれら手段に対して、指定した上限値に依存しない、スタンバイサーバと通信が途絶していても適切に動作する、という点で優れています。

### ◆ レプリケーションスロットを使用する

前節 4.6. でレプリケーションを構成してあるので、そこにレプリケーションスロットを追加します。

```
$ cat >> $PGDATA/postgresql.conf <<'EOS'      ← スロット数上限を設定
max_replication_slots = 2
EOS
$ pg_ctl restart                              ← プライマリ PostgreSQL 再起動
$ psql -q db1
                ↓ レプリケーションスロットを作成
db1=# SELECT * FROM pg_create_physical_replication_slot('a_slot');
                ↓ 状態確認、レプリケーションスロットは未だ active でない
db1=# SELECT * FROM pg_replication_slots;
slot_name | plugin | slot_type | datoid | database | active | xmin |
catalog_xmin | restart_lsn
---------+--------+-----------+--------+----------+--------+------+
-------------+-------------
 a_slot   |        | physical  |        |          | f      |      |
             |
(1 row)
db1=# \q

$ cat >> ${PGDATA}s/recovery.conf <<'EOS'     ← スロット名を設定に加える
primary_slot_name = 'a_slot'
EOS
$ pg_ctl restart -D ${PGDATA}s -o '-p 5433'   ← スタンバイ PostgreSQL 再起動
$ psq -q db1
db1=# SELECT * FROM pg_replication_slots;     ← スロットが active になる
slot_name | plugin | slot_type | datoid | database | active | xmin |
catalog_xmin | restart_lsn
---------+--------+-----------+--------+----------+--------+------+
```

```
-------------+-------------
 a_slot      |          | physical  |          |           | t       |       |
             |  0/5000060
(1 row)
```

以上の手順でレプリケーションスロットを使うことができます。なお、作ったレプリケーションスロットを削除するには pg_drop_replication_slot('スロット名') 関数を使います。

### ◆ WAL ファイル保持を確認

ここで WAL ファイル保持についてテストを行います。スタンバイを停止して、プライマリで pgbench -i コマンドを使って WAL 出力を沢山発生させます。このとき古い WAL ファイルが削除されてしまうと、スタンバイを次に起動してもレプリケーションが継続できなくなってしまいます。

```
$ pg_ctl stop -D ${PGDATA}s -o '-p 5433'
$ pgbench -i
$ pgbench -i
（上記を繰り返す）
$ ls $PGDATA/pg_xlog          ← WAL ファイルが削除されないでいることが確認できる
000000010000000000000005  00000001000000000000000A
000000010000000000000006  00000001000000000000000B
000000010000000000000007  00000001000000000000000C
000000010000000000000008  00000001000000000000000D
000000010000000000000009  archive_status
```

上記のようにスタンバイが必要とする WAL ファイルが、 checkpoint_segments = 3 のデフォルト設定から想定される数を超えて、長く保持されていることが確認できます。

### ◆ 行の物理削除抑止を確認

次にスタンバイで参照されている行の VACUUM による物理削除を避ける動作を確認します。

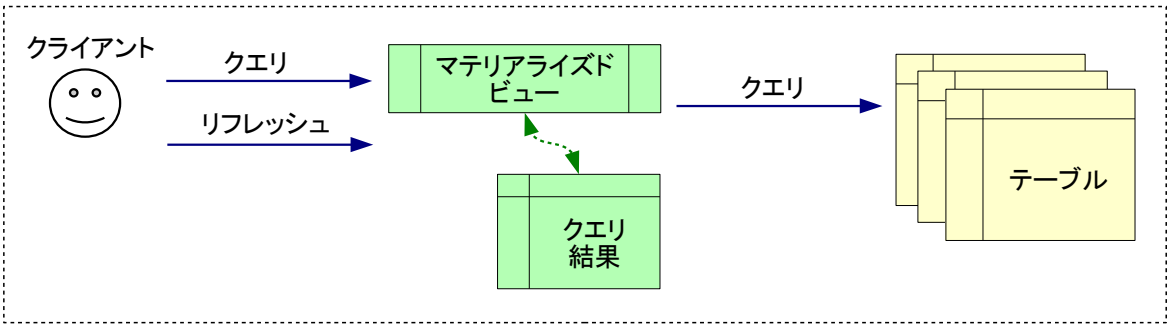
```
        ↓ 一つ前のテストで停止したスタンバイを起動しておきます
$ pg_ctl stop -D ${PGDATA}s -o '-p 5433' start
```

```
        ↓ REPEATABLE READ トランザクションで行を参照して、そのトランザクションを
          実行したままにします。この行データが物理削除されてはいけません。
$ psql -q -p 5433 db1
db1=# START TRANSACTION ISOLATION LEVEL REPEATABLE READ;
db1=# SELECT * FROM t_item ;
 id  |  name
-----+---------
  1 | ITEM1
  2 | ITEM2
     :（後略）
db1=# ^Z
[1]+  停止                 psql

        ↓ ここで pg_hba.conf で replication 接続を拒否するようにし、更に kill で
          wal receiver process を再起動させて、レプリケーション通信を途絶させます。
$ vi $PGDATA/pg_hba.conf
$ pg_ctl reload
$ kill $(pgrep -f "wal receiver process")

        ↓ プライマリで DELETE と VACUUM を行います。レプリケーションスロットを使用
          しない場合には、行データは物理削除されてしまいます。しかし、本テスト結果では
          物理削除が保留されていることが VACUUM VERBOSE 出力からわかります。
$ psql -q db1
db1=# DELETE FROM t_item;
db1=# VACUUM VERBOSE t_item;
     :（前略）
INFO:  "t_item": found 0 removable, 100 nonremovable row versions in 1 out
of 1 pages
DETAIL:  100 dead row versions cannot be removed yet.
     :（後略）
```

上記のようにたとえスタンバイと通信ができない状態でも、参照されている行の物理削除が保留される動作が確認できます。本手順では 4.6. 節で構成したレプリケーション設定をそのまま使っていますが、このうちスタンバイにおける hot_standby_feedback = on の設定は本動作に必須でとなります。

## 4.8. ロック競合しないマテリアライズドビューリフレッシュ

PostgreSQL 9.3 バージョンから、マテリアライズドビュー機能が追加されました。




マテリアライズドビューは、通常のビューと同様、SELECT コマンドに名前を付けたもので、テーブルと同様に他の SELECT コマンドの FROM 節に記述することができます。通常のビューと違うのは、定義された SELECT コマンドによるクエリ結果を記憶しておき、自身に対するクエリ応答に使用する点です。リフレッシュ（REFRESH MATERIALIZED VIEW コマンド）を行うと、改めて大元のテーブルに問い合わせを実行して、保持しているクエリ結果を更新します。

PostgreSQL 9.4 バージョンでは、ロック競合を軽減して REFRESH を実行する CONCURRENTLY オプションが追加されました。ただし、使用できるのは特定条件下に限定されます。

### ◆ リフレッシュによるロック競合

マテリアライズドビューのリフレッシュでは、対象のマテリアライズドビューに対し、ACCESS EXCLUSIVE モードのテーブルレベルロックが発生します。これはリフレッシュ中に当該マテリアライズドビューへの SELECT コマンドは待たされてしまうことを意味します。

もともと実行に時間のかかるクエリーであればこそ、キャッシュとしての役割を期待してマテリアライズドビューにしているはずです。したがって、リフレッシュに時間を要することは珍しくありません。定期的な「アクセスの無い時間帯」を持たないシステムでは、CONCURRENTLY オプション無しのマテリアライズドビューでは適用が難しいということになります。

### ◆ *CONCURRENTLY* の動作検証

リフレッシュに CONCURRENTLY オプションを使用できるのは、マテリアライズドビューが以下条件を満たす場合のみです。

a. 全行を対象としたカラム名に対するユニークインデックスを持つこと
b. 既にデータが格納されていること

a. は元テーブルに主キーやユニークインデックスが在ってもだめで、マテリアライズドビューに対するインデックスがあることを要求しています。b. は「WITH NO DATA」でマテリアライズドビューを定義した場合の初回リフレッシュには使えないという意味です。

以下のように使用してみます。

```
        ↓  t_sale、t_item、t_customer テーブルに対するマテリアライズドビューです。
           結合して、直近１か月のデータを表示しています。
 db1=# CREATE MATERIALIZED VIEW mv_sales_month AS
  SELECT s.id, c.name AS cname, i.name AS iname, s.amount, s.ts
   FROM t_sale s
    JOIN t_item i ON (s.iid = i.id) JOIN t_customer c ON (s.cid = c.id)
   WHERE s.ts BETWEEN now() - '1 month'::interval AND now()
   ORDER BY s.ts DESC;

 db1=# BEGIN;
 db1=# REFRESH MATERIALIZED VIEW mv_sales_month;
             ← ここで他の接続から SELECT * FROM mv_sales_month; を行うと、
               ロック待ちで応答が止まってしまうことが確認できます。
 db1=# COMMIT;

 db1=# REFRESH MATERIALIZED VIEW CONCURRENTLY mv_sales_month;
 ERROR:  cannot refresh materialized view "public.mv_sales_month"
concurrently
 HINT:  Create a UNIQUE index with no WHERE clause on one or more columns
of the materialized view.
             ↑↓  ユニークインデックスが必要とメッセージが出ます。作成します。
 db1=# CREATE UNIQUE INDEX ON mv_sales_month (id);

 db1=# BEGIN;
 db1=# REFRESH MATERIALIZED VIEW CONCURRENTLY mv_sales_month;
             ← 今度は他の接続から SELECT * FROM mv_sales_month; を実行できます。
 db1=# COMMIT;
```

## 4.9. 更新ビューの *CHECK* オプション

PostgreSQL 9.3 から、単純な SELECT コマンドに基づくビューは、別途にルールやトリガーを作成することなく UPDATE、DELETE、INSERT が可能となりました。これに加えて PostgreSQL 9.4 では CREATE VIEW コマンドに WITH CHECK OPTION 構文が加わりました。下記のように CREATE VIEW コマンドの末尾に指定を加えることができます（「...」はその他オプションをあらわします）。

```
CREATE ... VIEW view_name ... AS query
  [ WITH [ CASCADED | LOCAL ] CHECK OPTION ]
```

これにより、ビュー定義の範囲外となる値を挿入したり、範囲外となる値に更新したりできないようになります。CASCADED を指定すると当該ビューだけでなく、入れ子に参照しているビューについても値の検査を行います。LOCAL であれば当該ビューだけ検査します。デフォルトは CASCADED です。

以下に実行例を示します。

```
        ↓ サンプルテーブルとビューの作成。
           v_natural は自然数（0 より大きい整数）に限定するビュー、v_even で更に偶数
           に限定する。v_even に CASCADED CHECK OPTION を指定する。
db1=# CREATE TABLE t_num (num int);
db1=# CREATE VIEW v_natural AS SELECT num FROM t_num WHERE num > 0;
db1=# CREATE VIEW v_even AS
  SELECT num FROM v_natural WHERE num % 2 = 0 WITH CASCADED CHECK OPTION;

db1=# INSERT INTO v_even VALUES (2);      ← 偶数は格納できる。
db1=# INSERT INTO v_even VALUES (3);      ← 奇数は v_even の範囲外値。
ERROR:  new row violates WITH CHECK OPTION for view "v_even"
DETAIL:  Failing row contains (3).
db1=# INSERT INTO v_even VALUES (0);      ← ゼロは v_natural の範囲外値。
                                            v_natural に CHECK OPTION は
                                            無いが、連鎖するビューも検査される。
ERROR:  new row violates WITH CHECK OPTION for view "v_natural"
DETAIL:  Failing row contains (0).

db1=# INSERT INTO v_natural VALUES (-1); ← v_natural に CHECK OPTION が
                                            無いので直接には格納できてしまう。
```

## 4.10. 集約クエリーの FILTER オプション

SELECT コマンドの集約を行う選択リスト項目に FILTER 節を指定できるようになりました。

以下のように使用することができます。本例は item カラムを集約し、また、条件付けしていますが、もちろん異なるカラムを指定することもできます。

```
$ psql -q -x db1
db1=# SELECT min(amount) FILTER (WHERE amount > 10) FROM t_sale;
-[ RECORD 1 ]
min | 11
```

FILTER は単なる表現方法の追加ではなく、これまで不可能であった実行プランを実現する効果もあります。異なる条件を SELEC の選択リストに記述することができ、選択リスト項目ごとに条件が異なっていても、実行プランにおけるデータのスキャンは一つにまとめられます。

### ◆ スキャンをまとめる例

例として以下に、ある時点（ts）である商品（iid）をある顧客（cid）むけに売上したことを記録するテーブル t_sale から、月次で、アイテム 1 の売上合計、顧客 1 への売上合計、全体の売上合計を出力する集計クエリを示します。

EXPLAIN 結果から 1 回のスキャンで処理されていることがわかります。FILTER 節は、GROUP BY のみならず、OVER ～ WINDOW 構文を使ったウィンドウ関数機能による集計にも適用可能です。

```
db1=# SELECT to_char(date_trunc('month', ts), 'YYYY-MM') as month,
             to_char(coalesce(sum(amount) FILTER
               (WHERE iid = 1), 0), '99,999,999') as sales_item_1,
             to_char(coalesce(sum(amount) FILTER
               (WHERE cid = 1), 0), '99,999,999') as sales_customer_1,
             to_char(coalesce(sum(amount), 0), '99,999,999') as sales_all
        FROM t_sale GROUP BY date_trunc('month', ts);
  month   | sales_item_1 | sales_customer_1 |  sales_all
---------+--------------+------------------+-------------
 2011-10 |       26,233 |        3,428     |   6,496,859
 2011-11 |      184,275 |       30,629     |  14,834,992
 2011-12 |      139,770 |        9,032     |  16,002,876
 2012-01 |      171,411 |            0     |  15,411,082
                        :  （後略）

db1=# EXPLAIN SELECT to_char(date_trunc('mont ... 後略／上記と同じ SQL ...
                                QUERY PLAN
-----------------------------------------------------------------------------
```

```
 GroupAggregate  (cost=10828.82..14208.72 rows=94662 width=16)
   Group Key: (date_trunc('month'::text, ts))
   ->  Sort  (cost=10828.82..11078.82 rows=100000 width=16)
         Sort Key: (date_trunc('month'::text, ts))
         ->  Seq Scan on t_sale  (cost=0.00..2524.00 rows=100000 width=16)
 Planning time: 0.333 ms
```

## 4.11. 行ロックエラーのメッセージ拡張

PostgreSQL 9.4 から行ロックに関する各種のエラーメッセージにおいて、対象となった行がどの行であるかをメッセージに含めるようになりました。ただし、どの行であるかは CTID 形式、すなわち、何ページ目の何番目の項目であるかという物理位置による表現となります。以下に例を示します。

```
$ psql -q db1
db1=# BEGIN;
db1=# SELECT * FROM t_item WHERE id = 20 FOR UPDATE;
 id |  name                                  ↑  行ロックをかけます。
----+--------
 20 | ITEM20
(1 row)
db1=# ^Z                                    ←  Ctrl-Z で抜けてロックをかけた
[1]+  停止          psql db1                    ままにします。

$ psql -q db1                               ←  別の接続を開始
db1=# BEGIN;
db1=# SET LOCAL lock_timeout TO '10s';      ←  10 秒ロックタイムアウトを設定

            ↓ id=10～30 の範囲で行ロックを試みます
db1=# SELECT * FROM t_item WHERE id BETWEEN 10 AND 30 FOR UPDATE;
ERROR:  canceling statement due to lock timeout
CONTEXT:  while locking tuple (0,21) in relation "t_item"
            ↑↓ メッセージで対象行の物理位置が報告され、以下のように特定できます。
db1=# ABORT;
db1=# SELECT * FROM t_item WHERE ctid = '(0,21)';
```

```
 id |  name
----+--------
 20 | ITEM20
(1 row)
```

## 4.12. pg_stat_archive ビュー

archive_mode = on としてアーカイブログを採取している場合に、WAL ファイルのアーカイブ状況を確認できるシステムビューが追加されました。以下のように情報を取ることができます。

以下に出力例を示します。

```
db1=# SELECT * FROM pg_stat_archiver ;
-[ RECORD 1 ]------+------------------------------
archived_count     | 5
last_archived_wal  | 000000010000000000000024
last_archived_time | 2014-08-02 15:50:54.354365+09
failed_count       | 3                                ← 正常時はゼロ
last_failed_wal    | 00000001000000000000002B         ← 正常時は空欄
last_failed_time   | 2014-08-02 17:22:55.901712+09    ← 正常時は空欄
stats_reset        | 2014-06-14 19:39:30.359029+09    ← カウント開始時点
```

アーカイブ数（archive_count）と アーカイブ失敗数（failed_count） は、PostgreSQL 再起動をしても値が保持され、引き続き累積的にカウントされます。以下の関数でリセットできます。

```
db1=# SELECT pg_stat_reset_shared('archiver');
```

# 5. 追加された拡張枠組みの検証

本章では PostgreSQL 9.4 の追加仕様のうち、直接機能を提供するのではなく、拡張枠組みを提供するものについて検証結果を報告します。

## 5.1. 論理変更内容出力

PostgreSQL 9.4 からテーブルの行レベルの変更内容を出力する API が用意されました。

本機能を使用するには、wal_level 設定を logical に指定します。これは WAL に hot_standby を指定した時の内容に加えて、論理レベル変更内容を出力させるというものです。さらにレプリケーションスロットの一つとして実現されますので、max_replication_slots に 1 以上の値を指定する必要があります。

### ◆ SQL インタフェースによる機能動作確認

以下の手順で本機能を使用してみます。

```
$ vi $PGDATA/postgresql.conf       ← wal_level = logical 、
                                     max_replication_slots = 2 を指定
$ pg_ctl restart
$ psql -q db1
              ↓ スロット名とプラグイン名を指定する
db1=# SELECT * FROM
       pg_create_logical_replication_slot('logi_slot', 'test_decoding');
 slot_name | xlog_position
-----------+---------------
 logi_slot | 0/D8DDB88
(1 row)
              ↓ 4.7. 節で扱った phusical レプリケーションスロットと同列に管理される
db1=# SELECT * FROM pg_replication_slots;
 slot_name |    plugin     | slot_type | datoid | database | active | xmin
 | catalog_xmin | restart_lsn
-----------+---------------+-----------+--------+----------+--------
+------+--------------+-------------
 a_slot    |               | physical  |        |          | f      |
 |              | 0/D8DD090
 logi_slot | test_decoding | logical   |  16384 | db1      | f      |
 |         2012 | 0/D8DDB50
```

```
(2 rows)

                ↓ logical レプリケーションスロット作成後に変更操作を行ってみる。
db1=# CREATE TABLE t_test (id int primary key, v text);
db1=# INSERT INTO t_test
        SELECT g, left(md5(g::text),5) FROM generate_series(1,5) as g;
db1=# UPDATE t_test SET v = 'XXX' WHERE id = 3;

                ↓ 以下のように変更内容を取得できる。DDL は採取されていないことに注意。
db1=# SELECT * FROM pg_logical_slot_get_changes('logi_slot', NULL, NULL);
 location  | xid  |                              data
-----------+------+-----------------------------------------------------------
 0/D907F60 | 2015 | BEGIN 2015
 0/D91CEA8 | 2015 | COMMIT 2015
 0/D91CEE0 | 2016 | BEGIN 2016
 0/D91CEE0 | 2016 | table public.t_test: INSERT: id[integer]:1 v[text]:'c4ca4'
 0/D91CFA8 | 2016 | table public.t_test: INSERT: id[integer]:2 v[text]:'c81e7'
 0/D91D038 | 2016 | table public.t_test: INSERT: id[integer]:3 v[text]:'eccbc'
 0/D91D0C8 | 2016 | table public.t_test: INSERT: id[integer]:4 v[text]:'a87ff'
 0/D91D158 | 2016 | table public.t_test: INSERT: id[integer]:5 v[text]:'e4da3'
 0/D91D268 | 2016 | COMMIT 2016
 0/D91D490 | 2017 | BEGIN 2017
 0/D91D490 | 2017 | table public.t_test: UPDATE: id[integer]:3 v[text]:'XXX'
 0/D91D658 | 2017 | COMMIT 2017
(9 rows)
                ↓ 本関数は変更差分の「取得」であり、再度実行すると空出力となる。
                  pg_logical_slot_peek_changes 関数を使えば、見るだけも可能。
db1=# SELECT * FROM pg_logical_slot_get_changes('logi_slot', NULL, NULL);
 location | xid | data
----------+-----+------
(0 rows)
```

logical なレプリケーションスロットを text_decoding というプラグインで作成すると、以降の変更内容を pg_logical_slot_get_change 関数で取得できます。テーブル定義などの DDL 命令は記録されず、取得できません。

pg_logical_slot_get_change または pg_logical_slot_peek_change 関数の第二引数には、トランザクションログ位置（LSN）を指定して、どこまでのトランザクションを取得・参照するか指定できます。また、第二引数には、取得・参照するトランザクション件数を指定できます。いずれも NULL であれば、上限なくあるだけ全てを取得・参照します。

### ◆ *pg_recvlogical* による動作確認

次に追加クライアントアプリケーションとして用意されている pg_recvlogical を使って、論理変更出力機能を確認します。

```
        ↓ 引き続きテーブルに変更を加えていきます。
db1=# TRUNCATE t_test;
db1=# INSERT INTO t_test
         SELECT g, left(md5(g::text),5) FROM generate_series(1,5) as g;
db1=# DELETE FROM t_test WHERE id = 5;
db1=# INSERT INTO t_test VALUES (6, '666');

（並行して以下を実行します）
        ↓ 以下を実行すると -f で指定したファイルに変更内容を書き出し続けます。
          これは PostgreSQL に接続するプログラムですので、必要に応じて -h ホスト名、
          -p ポート番号 -U ユーザ名 などを指定することとなります。
$ pg_recvlogical -S logi_slot -d db1 --start -f out.txt
^C                      ← 適当なところで Ctrl-C で止めます。
pg_recvlogical: unexpected termination of replication stream:
$ cat out.txt           ← 変更内容が継続的に書き出されていることが確認できます。
BEGIN 2018
COMMIT 2018
BEGIN 2019
table public.t_test: INSERT: id[integer]:1 v[text]:'c4ca4'
table public.t_test: INSERT: id[integer]:2 v[text]:'c81e7'
table public.t_test: INSERT: id[integer]:3 v[text]:'eccbc'
table public.t_test: INSERT: id[integer]:4 v[text]:'a87ff'
table public.t_test: INSERT: id[integer]:5 v[text]:'e4da3'
COMMIT 2019
BEGIN 2020
table public.t_test: UPDATE: id[integer]:3 v[text]:'XXX'
```

```
COMMIT 2020
BEGIN 2021
table public.t_test: DELETE: id[integer]:5
COMMIT 2021
BEGIN 2022
table public.t_test: INSERT: id[integer]:6 v[text]:'666'
COMMIT 2022
```

SQL インタフェースを使った場合と同様にテーブルの行の変更内容が取得できることが分かります。

上記機能の直接の利用価値は高くありません。この拡張枠組みを使って、将来にロジカルレプリケーションやその他拡張機能を実装するとなったときに役立つものとなります。

# 6. その他の拡張項目

４章では PostgreSQL 開発コミュニティーによる PostgreSQL 9.3 beta1 のアナウンスで列挙された機能について取り上げました。このほかにも多数の拡張が実装されています。本節ではそのうちのいくつかを紹介します。

なお、拡張箇所の一覧は開発バージョンのドキュメントのリリースノートのページ（以下 URL）から確認することができます。

| http://www.postgresql.org/docs/devel/static/release-9-4.html |
|---|

# 7. 免責事項

本ドキュメントは SRA OSS, Inc. 日本支社が作成したものですが、SRA OSS, Inc. 日本支社は、本ドキュメントに対して、正確性、有用性、その他いかなる保証をするものではありません。本ドキュメントの内容を利用する場合、利用者の責任において行っていただくものとなります。