PostgreSQL 9.3 beta1

検証レポート

1.0.6 版
2013 年 6 月 7 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

2013 年 5 月 9 日、PostgreSQL 9.3 beta1 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。9.3 beta1 バージョンには、9.3 バージョンむけ機能のうち現時点で完成してるものが含まれています。

本ドキュメントは、9.3 beta1 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 9.3 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

PostgreSQL 9.3 beta1 の主要な新機能は以下の通りです。

- 外部データラッパ（Foreign Data Wrapper）がテーブル書き込みに対応
- postgres_fdw （PostgreSQL に対する外部データラッパのドライバ）が同梱
- マテリアライズドビュー
- LATERAL 結合
- JSON 型データに対する機能拡充
- 正規表現にて pg_trgm のインデックス利用
- ページチェックサム
- ストリーミングレプリケーションにおけるプライマリ切り替え対応
- 外部キー制約に対する行ロックの競合軽減
- 並列 pg_dump
- 設定ファイルのディレクトリ分割配置
- 死活監視用ツール pg_isready 追加
- バルクロード性能改善むけ COPY FREEZE コマンド追加
- ユーザ定義バックグラウンドワーカープロセスの枠組み
- 再帰 VIEW 構文
- 更新可能 VIEW

これまでのメジャーバージョンアップに比べ、新機能の数は多い部類であるといえます。しかも、これらは 9.3 beta1 の段階での機能実装であって、9.3 リリースまでに更に追加される余地があります。これらの過半はもともと 9.2 バージョンでの追加が検討されていたもので、繰り越されて 9.3 に追加されたといえます。

各機能の説明は、評価検証の各項に記載します。

3. 検証のためのセットアップ

3.1. ソフトウェア入手

PostgreSQL 9.3 beta1 は以下ＵＲＬからダウンロード可能です。ただし、本検証作業においては、開発リポジトリから取り出したもの（2013/5/6 時点の最新版）を使用しています。

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

3.2. 検証環境

検証環境として、仮想マシン（IA 64bit）上の CentOS 6.2 (Linux 2.6.32 x86_64) を使用しました。

/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のビルドを行いました。今回の検証では取り扱わないため、PL/Tcl、SE-PostgreSQL、NLS（日本語メッセージに必要）を有効にするオプション、他、いくつかのオプションは与えていません。

```
$ cd postgresql-9.3beta1
$ ./configure --prefix=/home/postgres/pgsql/9.3beta1 --with-libxml --with-perl --with-python --enable-debug
$ make install-world
```

環境変数を設定するファイルを書き出して、適用します。

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

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、新機能のデータチェックサムを有効にします。

```
$ initdb --no-locale --encoding=UTF8 --data-checksums
The files belonging to this database system will be owned by user
"postgres".
 （中略）
Success. You can now start the database server using:
  postgres -D /home/postgres/data/9.3beta1p
or
  pg_ctl -D /home/postgres/data/9.3beta1p -l logfile start
```

設定ファイルに最小限の設定を与えます。

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

PostgreSQL を起動します。

```
$ pg_ctl start
pg_ctl start
server starting
```

4. 主要な追加機能の検証

主要な追加機能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/9.3beta1/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/devel/static/
```

4.1. *postgres_fdw* と外部データラッパでの書き込み

外部データラッパ機能は様々な外部データをテーブルとして扱うための枠組みです。あくまで枠組みですので、書き込み対応を確認するには、それを利用した実装を動作させなければいけません。ここでは書き込み対応の具体的な実装であり、もう一つの追加機能である postgres_fdw の動作確認をします。

外部データラッパの枠組みを利用した postgres_fdw は、複数の PostgreSQL インスタンス同士のデータ連携を実現するものですので、まずは、先程作成したデータベースクラスタとは別の、5433 ポートで動作する PostgreSQL を新たに作成し、起動します。

```
$ initdb --no-locale --encoding=UTF8 -D ${PGDATA}5433
$ cp $PGDATA/postgresql.conf ${PGDATA}5433/postgresql.conf
$ pg_ctl -D ${PGDATA}5433 -o '-p 5433' start
```

次に postgres_fdw 拡張モジュールをデフォルトの postgres データベースにインストールします。

```
$ psql -q
postgres=# CREATE EXTENSION postgres_fdw;
```

また、参照される側のテーブルを 5433 ポートの PostgreSQL 上に作っておきます。

```
$ psql -q -p 5433
postgres=# CREATE TABLE t5433 (id int primary key, v text);
postgres=# INSERT INTO t5433 VALUES (1, 'aa'),(2,'bb');
```

それでは最初に起動した 5432 ポート（デフォルトのポート）の PostgreSQL から、5433 ポートの PostgreSQL の postgres データベース上にあるテーブルにアクセスしてみます。

```
$ psql -q
postgres=# CREATE SERVER pg5433srv FOREIGN DATA WRAPPER postgres_fdw
           OPTIONS (port '5433', host 'localhost', dbname 'postgres');
postgres=# CREATE USER MAPPING FOR postgres SERVER pg5433srv
           OPTIONS (user 'postgres', password 'secret');
postgres=# CREATE FOREIGN TABLE rt5433 (id int, v text) SERVER pg5433srv
           OPTIONS (schema_name 'public', table_name 't5433');
```

以上の手順で、外部データラッパのサーバを作り、ユーザマッピングを作り、外部テーブルを定義します。

なお、本例において各コマンドの OPTIONS にリモート側のホスト名、データベース名、ユーザ名、パスワードを指定しているのは参考のためであって、これらは省略してデフォルト値に依存するようにしても動作します。

```
postgres=# SELECT * FROM rt5433;
 id | v
----+----
  1 | aa
  2 | bb
(2 rows)
postgres=# DELETE FROM rt5433 WHERE id = 2;
postgres=# SELECT * FROM rt5433;
 id | v
----+---
  1 | aa
(1 rows)
```

外部テーブルに対するデータ書き込み（本例では DELETE）も動作していることがわかります。

4.2. マテリアライズドビュー

マテリアライズドビューを検証するためにまずは pgbench ツールでデータを作ります。最初のコマンドで銀行口座をあらわす 10 万行の pgbench_accounts テーブルを含む 4 テーブルが作られます。次のコマンドでランダムに口座振り込みが行われて、口座の預金額（abalance カラム）が変動します。

```
$ pgbench -i -q
creating tables...
100000 of 100000 tuples (100%) done (elapsed 0.25 s, remaining 0.00 s).
vacuum...
set primary keys...
done.
$ pgbench -t 10 -c 10
（出力省略）
```

口座預金額トップ 3 を出力するクエリをマテリアライズドビューとして定義してみます。

```
postgres=# CREATE MATERIALIZED VIEW mv_abalance
 AS SELECT aid, abalance FROM pgbench_accounts
  ORDER BY abalance DESC LIMIT 3;
postgres=# SELECT * FROM mv_abalance;
 aid   | abalance
-------+----------
 34057 |     4976
 88671 |     4938
 87688 |     4927
(3 rows)
```

通常のビューと同様に参照することができました。マテリアライズドビュー作成時に SELECT 文が実行されてデータが作られ、以降のマテリアライズドビュー参照ではその内容が使われます。

次に、pgbench を再度実行して、pgbench_accounts テーブルのデータを変化させてみます。そうすると、直接クエリを実行した結果は変化しているにもかかわらず mv_abalance のデータは前回と同じになります。

```
$ pgbench -t 10 -c 10
（出力省略）
$ psql -q
postgres=# SELECT aid, abalance FROM pgbench_accounts ORDER BY abalance
DESC LIMIT 3;
 aid   | abalance
-------+----------
 56721 |     4981
 77639 |     4979
 34057 |     4976
(3 rows)

postgres=# SELECT * FROM mv_abalance;
 aid   | abalance
-------+----------
 34057 |     4976
 88671 |     4938
 87688 |     4927
```

```
(3 rows)
```

マテリアライズドビューのデータを元テーブルに追随させるには リフレッシュ（REFRESH MATERIALIZED VIEW）を実行します。

```
postgres=# REFRESH MATERIALIZED VIEW mv_abalance;
postgres=# SELECT * FROM mv_abalance;
 aid   | abalance
-------+----------
 56721 |     4981
 77639 |     4979
 34057 |     4976
(3 rows)
```

◆ マテリアライズドビュー制限事項

現在実装されているマテリアライズドビューは、他のＤＢＭＳソフトウェアにある以下のような機能は持っていません。

- 定期的に行われる自動リフレッシュ機能
- 参照元のテーブルの変更に連動した自動リフレッシュ機能
- 更新差分だけを反映する高速リフレッシュ機能
- 参照元テーブルに対するクエリにおける暗黙的なマテリアライズドビューの利用

自動リフレッシュについては、トリガーを定義したり、 PostgreSQL 外部の定期実行スクリプトから REFRESH MATERIALIZED VIEW を実行することで、実現するものとなります。

4.3. LATERAL 結合

LATERAL 結合は ISO SQL:1999 で追加された機能で、SELECT 文の FROM 句の並びにあるサブクエリや表を返す関数において、左側の実行結果データを参照することができるというものです。

```
postgres=# CREATE TABLE t_sales
           (id int primary key, uid int, iid int, ts timestamp);
postgres=# CREATE TABLE t_user (uid int primary key, name text);
postgres=# CREATE TABLE t_item (iid int primary key, name text);
```

上記のような定義のテーブル（データは適宜追加してあるものとします）に対して、以下のような書き方

で結合するクエリを書くことができます。

```
postgres=# SELECT id, i.*, u.*, ts FROM t_sales s,
             LATERAL (SELECT * FROM t_item WHERE iid = s.iid) i,
             LATERAL (SELECT * FROM t_user WHERE uid = s.uid) u;
 id   | iid | name   | uid | name   |             ts
------+-----+--------+-----+--------+----------------------------
 1001 |   1 | 商品A | 101 | 顧客a | 2013-05-01 13:06:41.910403
 1002 |   1 | 商品A | 102 | 顧客b | 2013-05-02 13:26:53.388375
 1003 |   2 | 商品B | 101 | 顧客a | 2013-05-03 13:47:02.461441
 1004 |   1 | 商品A | 102 | 顧客b | 2013-05-04 13:07:09.440439
(4 rows)
```

4.4. JSON 型データに対する機能拡充

PostgreSQL 9.2 まで JSON 型用の関数、演算子（「=」など自明なものを除くとして）は、以下の 2 つしかありませんでした。

- `array_to_json(anyarray [, pretty_bool])` 配列を JSON 型として返す
- `row_to_json(record [, pretty_bool])` 行を JSON 型として返す

PostgreSQL 9.3 では、十数個の JSON 型用の関数、演算子が追加されました。

◆ 要素の取り出し

矢印演算子で JSON の配列、オブジェクトから要素を取り出すことができます。また、「#>」演算子で複雑な JSON データに対してパス指定をして奥深くにある要素を取り出すことができます。なお、「->」の替わりに「->>」、「#>」の替わりに「#>>」を使うと戻り値が json 型でなく text 型になります。

```
postgres=# SELECT '[100, 101, 102]'::json -> 2;
 ?column?
----------
 102
(1 row)

postgres=# SELECT '{"a":1, "b":2}'::json -> 'b';
 ?column?
```

```
----------
 2
(1 row)

postgres=# SELECT '{"a":[1,2,{"k":[10,20,30]}],"b":[4,5,6]}'::json
                  #> '{a,2,k,1}';
 ?column?
----------
 20
(1 row)
```

他に、これらと同機能をもつ関数 json_extract_path 、json_extract_path_text もあります。

◆ *JSON* 配列の長さ

JSON 配列データの長さ（要素数）を問い合わせできます。

```
postgres=# SELECT json_array_length('["a","b","c","d"]');
 json_array_length
-------------------
                 4
(1 row)
```

◆ *JSON* データから表を構成

オブジェクトである JSON データから表データを構成するいくつかの関数があります。

json_each は key と value のペアを表データで返します。json_each_text は同じ動作ですが json 型でなく text 型の表データを返します。

```
postgres=# SELECT * FROM json_each('{"a":"foo", "b":"bar"}');
 key | value
-----+-------
 a   | "foo"
 b   | "bar"
(2 rows)
```

json_array_elements は、配列である JSON データを複数行に展開します。

```
postgres=# SELECT json_array_elements('[1,true, [2,false]]');
 json_array_elements
---------------------
 1
 true
 [2,false]
(3 rows)
```

json_object_keys は、オブジェクトである JSON データのキーだけを複数行データで返します。

```
postgres=# SELECT json_object_keys('{"f1":"abc","f2":["xx","yy"]}');
 json_object_keys
------------------
 f1
 f2
(2 rows)
```

json_populate_recordset は、オブジェクトの配列である JSON データから、指定レコード型の表データを構成します。第一引数には NULL をレコード型（複合型やテーブル名）でキャストしたものを与えます 。json_populate_record という 1 行だけを生成する関数もあります。

```
postgres=# SELECT * FROM json_populate_recordset(null::t_item,
            '[{"iid":105,"name":"商品E"},{"iid":106,"name":"商品F"}]');
 iid | name
-----+-------
 105 | 商品E
 106 | 商品F
(2 rows)
```

4.5. pg_trgm の拡張

pg_trgm モジュールは類似文字列の高速検索をサポートするインデックス演算子クラスを提供します 。LIKE、ILIKE 問い合わせにてインデックスを利用することができましたが、これに加えて、正規表現にも対応するようになりました。

ただし、マルチバイト対応は依然として、デフォルトではソースコードレベルで除外されています。マルチ

バイト対応をさせるには contrib/pg_trgm/trgm.h ファイルの以下定義をコメントアウトする必要があります。本検証ではマルチバイト対応をさせて行いました。

```
#define KEEPONLYALNUM
```

以下のようにモジュールをインストールし、テスト用のテーブル t_trgm とインデックスを作り、テストデータを投入します。ここでは本ドキュメント文字列を元にテストデータを作成しました。

```
postgres=# CREATE EXTENSION pg_trgm;
postgres=# CREATE TABLE t_trgm (id serial primary key, t text);
postgres=# CREATE INDEX ON t_trgm USING gist (t gist_trgm_ops);
postgres=# COPY t_trgm (t) FROM stdin;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> PostgreSQL 9.3 beta1
>> 検証レポート検証レポート
>> 1.0 版
>> 2013 年 5 月 10 日
（中略）
>> \.
```

以下のように正規表現の演算子「~」を検索条件とした場合に実行プランを確認すると、インデックスが使われていることが分かります。また、マルチバイト文字でも動作しているといえます。

```
postgres=# SELECT * FROM t_trgm WHERE t ~ 'ビュー';
 id  |                                  t
-----+-------------------------------------------------------------------
 117 | 4.2. マテリアライズドビュー 6
 118 | ◆ マテリアライズドビュー制限事項 8
（中略）
 564 |  29 | マテリアライズドビュー制限事項
(15 rows)

postgres=# explain SELECT * FROM t_trgm WHERE t ~ 'ビュー';
                    QUERY PLAN
```

```
------------------------------------------------------------------------
 Bitmap Heap Scan on t_trgm  (cost=4.25..9.43 rows=14 width=39)
  Recheck Cond: (t ~ 'ビュー'::text)
  ->  Bitmap Index Scan on t_trgm_t_idx  (cost=0.00..4.25 rows=14 width=0)
        Index Cond: (t ~ 'ビュー'::text)
(4 rows)
```

なお、pg_trgm のインデックスはトリグラムによる実装であるため、1 文字、2 文字キーワードの検索については効果的ではありません。マルチバイトに対応させても、この性質については変りありません。

4.6. ページチェックサム

PostgreSQL は、WAL ファイル（pg_xlog 下に出力されるトランザクションログ）についてはデータにチェックサムが付加されていますが、テーブルやインデックスのデータ本体においてはチェックサムがありませんでした。PostgreSQL 9.3 では、オプション機能としてページチェックサムが実装されました。

本検証手順では、initdb コマンドのよるデータベースクラスタ作成時に --data-checksums オプションを付けて、チェックサムを有効にしています。

機能動作を確認するためデータを壊してみます。

pgbench コマンドで生成した postgres データベース内の pgbench_accounts テーブルを壊すものとして、まずは、テーブルに対応したファイルがどれであるかを oid2name コマンドを使って確認します。

```
$ oid2name | grep postgres
 12896        postgres  pg_default
$ oid2name -d postgres | grep pgbench_accounts
   16466  pgbench_accounts
```

上記の結果から「$PGDATA/base/12896/16466」が pgbench_accounts テーブルに対応するファイルであるとわかります。いったん PostgreSQL を停止したうえでファイルのデータを壊します。本検証では bvi というバイナリエディタを使用しました。

```
$ pg_ctl stop
waiting for server to shut down.... done
server stopped
$ bvi $PGDATA/base/12896/16466
（ファイル中の適当な箇所をデタラメなバイト列で上書き）
```

```
$ pg_ctl start
server starting
$ psql -q
postgres=# SELECT count(*) FROM pgbench_accounts;
 count
--------
 100000
(1 row)

postgres=# SELECT * FROM pgbench_accounts WHERE filler ~ 'XXX';
WARNING: page verification failed, calculated checksum 61554 but expected
3960
ERROR:  invalid page in block 23 of relation base/12896/16466
```

pgbench_accounts テーブルを検索する SQL を実行すると上記のようにチェックサムが一致しないことを示すエラーが発生します。最初の「SELECT count(*) FROM pgbench_accounts」ではエラーにならなかったのは、Index Only Scan 機能が働き、障害のあるページへのアクセスが発生しなかったからといえます。

◆ *zero_damaged_pages* 設定による修復

続いて修復（障害箇所の除去）をさせるテストも行います。この機能は、以前のバージョンから存在していたものですが関連する技術情報として記載いたします。

```
postgres=# SET zero_damaged_pages TO on;
postgres=# SELECT * FROM pgbench_accounts WHERE filler ~ 'XXX';
WARNING:  page verification failed, calculated checksum 61554 but expected
3960
WARNING:  invalid page in block 23 of relation base/12896/16466; zeroing
out page
 aid | bid | abalance | filler
-----+-----+----------+--------
(0 rows)
```

上記のように zero_dameged_pages を on にすることで、データ破損を検知した際に、その場で破損を含むページをゼロ埋めして除去させることができます。

除去すると、同ページ（8KB）に含まれるデータ欠損が発生するはずです。件数カウントで確認してみます。

```
postgres=# SELECT count(*) FROM pgbench_accounts ;
 count
--------
 100000
(1 row)
```

変っていません。そこで Index Only Scan を無効に設定して再実行してみます。

```
postgres=# SET enable_indexonlyscan TO false;
postgres=# SELECT count(*) FROM pgbench_accounts;
 count
-------
 99939
(1 row)
```

今度は、61 件減っていて、テーブル上の正しいデータ件数を返しているといえます。
これは誤った結果を返す可能性があるため望ましい挙動とはいえません。

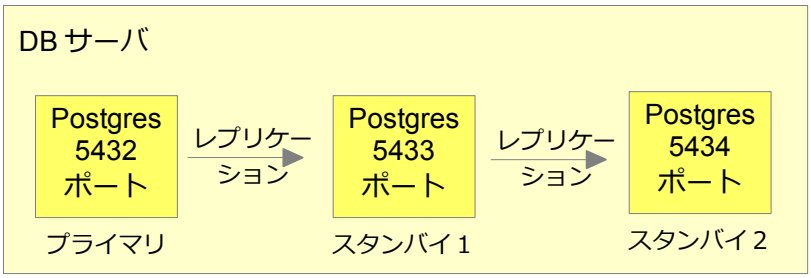
```
postgres=# SET enable_indexonlyscan TO on;
postgres=# VACUUM pgbench_accounts;
WARNING:  relation "pgbench_accounts" page 23 is uninitialized --- fixing
postgres=# SELECT count(*) FROM pgbench_accounts;
 count
--------
 100000
(1 row)
postgres=# REINDEX TABLE pgbench_accounts;
postgres=# SELECT count(*) FROM pgbench_accounts;
 count
-------
 99939
(1 row)
```

VACUUM を行っても破損ページ除去がインデックスに反映されません。したがって自動 VACUUM による自動的な反映も期待できません。「SET zero_damaged_pages TO on;」による破損ページ除去を行ったなら 、

REINDEX も行う、という運用が必要であるといえます。

4.7. ストリーミングレプリケーションの拡張

ストリーミングレプリケーションについて、いくつか機能拡張されています。



◆ 検証用レプリケーションクラスタの準備

検証のため、まずは上図のようなストリーミングレプリケーションクラスタを構成します。同サーバマシン内に異なる待ち受けポートで 3 つの PostgreSQL インスタンスを動作させます。

ストリーミングレプリケーションに必要な設定を与えて、プライマリとする現在稼働中の PostgreSQL を再起動します。

```
$ cat >> $PGDATA/postgresql.conf <<EOF
wal_level = hot_standby
max_wal_senders = 2
hot_standby = on
EOF
$ cat >> $PGDATA/pg_hba.conf << EOF
local   replication     postgres                                trust
host    replication     postgres        127.0.0.1/32            trust
host    replication     postgres        ::1/128                 trust
EOF
$ pg_ctl restart
waiting for server to shut down.... done
server stopped
server starting
```

次に pg_basebackup コマンドでスタンバイ 1 の データベースクラスタディレクトリを作ります。--write-recovery-conf は、PostgreSQL 9.3 からの新しいオプションで、recovery.conf を自動的に作ってくれます。pg_basebackup 実行時に参照したプライマリ PostgreSQL を参照してストリーミングレプリケーションを行

う設定で生成されます。最新のタイムラインに追随するように recovery_target_timeline の設定を加えます。

```
$ pg_basebackup -D ${PGDATA}S1 --xlog --write-recovery-conf
$ cat >> ${PGDATA}S1/recovery.conf << EOF
recovery_target_timeline = 'latest'
EOF
```

作ったデータベースクラスタディレクトリを使って 5433 ポートで PostgreSQL を立ち上げます。

```
$ pg_ctl start -D ${PGDATA}S1 -o '-p 5433'
server starting
```

同じ要領でスタンバイ 2 について、データベースクラスタディレクトリを作って、5434 ポートで立ち上げます。

```
$ pg_basebackup -p 5433 -D ${PGDATA}S2 --xlog --write-recovery-conf
$ cat >> ${PGDATA}S2/recovery.conf << EOF
recovery_target_timeline = 'latest'
EOF
$ pg_ctl start -D ${PGDATA}S2 -o '-p 5434'
server starting
```

◆ 昇格時のレプリケーション継続

プライマリを停止して、スタンバイ 1 を昇格させてみます。

```
$ pg_ctl stop -m fast
waiting for server to shut down.... done
server stopped
$ pg_ctl promote -D ${PGDATA}S1
server promoting
```

このとき、PostgreSQL 9.2.x までは、スタンバイ 2 がスタンバイ 1 からのレプリケーションを継続できなくなり、改めて pg_basebackup を行う必要がありました。PostgreSQL 9.3 beta 1 を使った本検証では、問題なくスタンバイ 2 がレプリケーションを継続しました。以下のように確認を行っています。

```
$ psql -q -p 5433 -c 'CREATE TABLE t (id int)'
$ psql -q -p 5434
postgres=# \d t
    Table "public.t"
 Column |  Type   | Modifiers
--------+---------+-----------
 id     | integer |
```

本例はカスケードレプリケーション構成ですが、一つのプライマリサーバに複数のスタンバイサーバがつながっている構成で、故障したプライマリサーバに替えて、スタンバイサーバの一つを新たなプライマリにする際にも、昇格しなかったスタンバイにて recovery.conf でデータ取得元を変えるだけでレプリケーション継続ができます。

◆ プライマリとスタンバイの入れ替え

ストリーミングレプリケーション構成で運用していると、pg_basebackup によるベースバックアップの再取得を行うことなく、プライマリとスタンバイを入れ替えたい場合がよくあります。

これまでは、うまくいかない場合がありましたが、本バージョンで修正されています。

以下の手順で、停止した 5432 ポートの PostgreSQL を、昇格させた 5433 ポートの PostgreSQL のスタンバイとして起動します。

```
$ cp ${PGDATA}S1/recovery.done $PGDATA/recovery.conf
$ vi $PGDATA/recovery.conf
（5432 ポートでなく、5433 ポートに接続するように書き換え）
$ pg_ctl start
server starting
```

問題なく、逆向きのレプリケーションを構成することができました。

スイッチを繰り返すように、以下のシェルスクリプトを 10 回以上繰り返し実行しても、最終的にレプリケーションが正常に構成できている状態となることが確認できました。

```
#!/bin/bash
# "5433 PGDATA-S1" to "5432 PGDATA"
pg_ctl stop -D ${PGDATA}S1 -m fast -w
pg_ctl promote
```

```
mv ${PGDATA}S1/recovery.done ${PGDATA}S1/recovery.conf
pg_ctl start -D ${PGDATA}S1 -o '-p 5433' -w
# "5432 PGDATA" to "5433 PGDATA-S1"
pg_ctl stop -m fast -w
pg_ctl promote -D  ${PGDATA}S1
mv $PGDATA/recovery.done $PGDATA/recovery.conf
pg_ctl start -w
```

4.8. ロックの拡張

いくつかロック関連の機能拡張が行われています。

◆ ロックタイムアウト

ロックタイムアウトが導入されました。

以下のように t テーブルに強い排他ロック（何も指定しないと最も強い ACCESS EXCLUSIVE モードになります）を取得したままにします。

```
$ psql -q
postgres=# BEGIN; LOCK t;
postgres=#
```

そこで、別の接続で、lock_timeout に 10 秒を設定して、当該テーブルにアクセスすると、10 秒間ロック待ちした後、エラーとなります。これまでは、ロック待ちに対して、直ちにエラーにするか、待ち続けるかのいずれかしか指定できませんでした。

```
$ psql -q
postgres=# SET lock_timeout TO '10s';
postgres=# BEGIN;
postgres=# SELECT * FROM t;
（１０秒後に）
ERROR:  canceling statement due to lock timeout
```

◆ 外部キー制約に対する行ロックの競合軽減

行ロックの種類が追加されました。

これまで SELECT ... FOR UPDATE（排他ロック）と SELECT ... FOR SHARE（共有ロック）の 2 種類

がありましたが、より競合度の低い以下 2 種類が追加されました。

- SELECT ... FOR NO KEY UPDATE

  SELECT ... FOR UPDATE とほぼ同じですが、SELECT ... FOR KEY SHARE と競合しません。

- SELECT ... FOR KEY SHARE

  SELECT ... FOR SHARE とほぼ同じですが、主キー以外のカラムを変更する UPDATE 文とは競合しません。また、SELECT ... FOR NO KEY UPDATE とも競合しません。

同じ行に対する 4 種類の行ロック取得、UPDATE 文、DELETE 文実行における競合関係は、以下のようになります。「×」マークの組み合わせで、後から取得を試みた処理がブロックされます。

	FOR UPDATE	FOR SHARE	FOR NO KEY UPDATE	FOR KEY SHARE	主キー UPDATE、DELETE	主キー以外 UPDATE
FOR UPDATE	×	×	×	×	×	×
FOR SHARE	×	.	×	.	×	×
FOR NO KEY UPDATE	×	×	×	.	.	×
FOR KEY SHARE	×	.	.	.	×	.
主キー UPDATE、DELETE	×	×	×	×	×	×
主キー以外 UPDATE	×	×	×	.	×	×

追加された「FOR KEY SHARE」行ロックは、外部キー制約のトリガー動作において使われます。これまでは「FOR SHARE」が使われていましたので、外部キー制約においてロック競合が減ることになります。

なお、以下のような SQL スクリプトを用意して、各行ロック状態に対して実行することで検証しました。ロック待ちになる組み合わせであれば、タイムアウトにより ERROR となります。

```
SET lock_timeout TO '1s';
\echo FOR UPDATE
BEGIN; SELECT * FROM t_sales WHERE id = 1003 FOR UPDATE; ABORT;
\echo FOR SHARE
BEGIN; SELECT * FROM t_sales WHERE id = 1003 FOR SHARE; ABORT;
\echo FOR NO KEY UPDATE
BEGIN; SELECT * FROM t_sales WHERE id = 1003 FOR NO KEY UPDATE; ABORT;
```

```
\echo FOR KEY SHARE
BEGIN; SELECT * FROM t_sales WHERE id = 1003 FOR KEY SHARE; ABORT;
\echo UPDATE key
BEGIN; UPDATE t_sales SET id = 0 WHERE id = 1003; ABORT;
\echo UPDATE non key
BEGIN; UPDATE t_sales SET ts = now() WHERE id = 1003; ABORT;
```

4.9. 並列 pg_dump

pg_dump に並列処理を行わせる -j オプションが追加されました。-j 2 なら 2 並列で処理が行われます。並列処理を行う場合、出力形式は、PostgreSQL 9.1 で追加された「ディレクトリ形式」でなければいけません。以下のように指定した名前のディレクトリが作られ、その中にテーブル毎のファイルにて格納されます。

```
$ pg_dump -j 2 -Fd -f out.d
$ ls out.d/
2893.dat.gz  2895.dat.gz  2898.dat.gz  2900.dat.gz  2902.dat.gz
2894.dat.gz  2897.dat.gz  2899.dat.gz  2901.dat.gz  toc.dat
```

CPU コア数が -j で指定した以上あり、ディスク I/O に余裕があったり、別々のストレージごとテーブルスペースが定義されている場合などにおいて、並列にダンプを行うことで高速化することができます。

2 つの仮想 CPU を割り当てた仮想マシンにて、以下のように 2 つの 400 万件の整数データを持つテーブルをもった test データベースを用意し、2 並列にした場合と直列（-j 1）にした場合とで所要時間を比較しました。

```
test=# CREATE TABLE t_copytest1 (d int);
test=# INSERT INTO t_copytest1 SELECT generate_series(1,40000000);
test=# CREATE TABLE t_copytest2 (d int);
test=# INSERT INTO t_copytest2 SELECT generate_series(1,40000000);

$ time pg_dump -j 1 -Fd -f out.d test
  → 3 回平均で 135.6 秒
$ time pg_dump -j 2 -Fd -f out.d test
  → 3 回平均で 73.2 秒
```

その結果、 2 並列にすると約半分の時間でダンプが完了するという結果が得られました。

4.10. 設定ファイルのディレクトリ分割配置

以下のように postgresql.conf に include_dir 設定を指定し、指定したディレクトリを作成し、その中に「.conf」が付いたファイルを置いて、設定ファイルを複数ファイルに分けて配置することができます。

```
$ cat >> $PGDATA/postgresql.conf << EOF
include_dir = 'conf.d'
EOF
$ mkdir $PGDATA/conf.d
$ cat > $PGDATA/conf.d/debug.conf << EOF
log_min_error_statement = debug5
log_connections = on
log_disconnections = on
EOF
$ pg_ctl reload
server signaled
```

上記のように debug.conf を作成して設定内容を適用できます。これを無効にする際にはファイル名を「debug.conf.inv」などと変えて「pg_ctl reload」を行う、といった使い方が考えられます。

なお、初期状態の設定ファイルには以下のように記述されますが、デフォルトで「conf.d」というディレクトリが指定されているわけではありません。include_dir に明示的な指定がなければディレクトリからの設定参照は行われません。

```
#include_dir = 'conf.d'                # include files ending in '.conf' from
                                       # directory 'conf.d'
```

4.11. 死活監視用ツール *pg_isready* 追加

PostgreSQL が稼働しているかをチェックする pg_isready コマンドが追加されました。

以下のように動作します。「-q」オプションを付けると何も表示せず、戻り値だけを返します。タイムアウトを指定することもできます。

```
$ pg_isready -h localhost -q --timeout=5
$ echo $?
```

```
0
$ pg_isready -p 5435 -h 127.0.0.1
127.0.0.1:5445 - no response
$ echo $?
2
```

4.12. COPY FREEZE コマンド

COPY コマンドに FREEZE オプションが追加されました。これは、高速データローディングを実現するものです。以下のように、実行するトランザクションにて、テーブルを作ったり、TRUNCATE した後に実行します。

```
postgres=# BEGIN;
postgres=# TRUNCATE pgbench_accounts ;
postgres=# COPY pgbench_accounts FROM '/tmp/accounts.copy' FREEZE;
postgres=# COMMIT;
```

読み込んだデータは、ちょうど VACUUM FREEZE を実行した後と同じ状態となります。このことは、以下の利点と欠点を意味します。

- （利点として）次にロードしたデータ参照する際にデータ上のフラグ変更が発生しないため、初回参照が遅くなる現象を回避できます。
- （欠点として）COMMIT を待たず、直ちに他のトランザクションから参照可能な状態となり、通常のトランザクション隔離の規則に反する振る舞いになります。

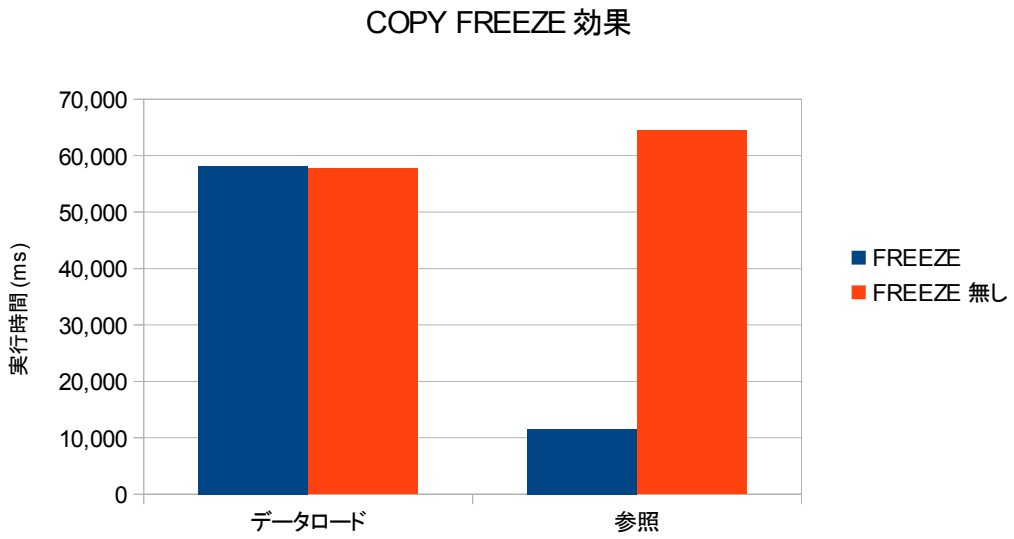
性能を検証してみます。ロード自体の速度、ロード後のデータ参照の速度を比較します。
以下のようにテストテーブル t_copytest と 4000 万行データの（333MB）を生成します。

```
postgres=# CREATE TABLE t_copytest (d int);
postgres=# INSERT INTO t_copytest SELECT generate_series(1,40000000);
postgres=# \copy t_copytest TO copytest.dat
```

このデータに対して、psql で以下のスクリプトを COPY 文に FREEZE 指定を付けた場合、付けない場合とで 3 回ずつ繰り返して、所要時間を計測しました。本テストは、仮想マシンで行い、postgresql.conf 設定はデフォルトのままとしています。

```
BEGIN; TRUNCATE t_copytest;
\timing
COPY t_copytest FROM '/home/postgres/copytest.dat' FREEZE;
COMMIT;
SELECT count(*) FROM t_copytest;
```

結果を以下グラフに示します。



「データロード」が COPY と COMMIT の所要時間の合計で、「参照」が SELECT の所要時間となります。データロードにはほとんど差がありませんが、データロード後の参照の所要時間が COPY FREEZE を使ったほうが短くなっていることがわかります。

4.13. 再帰 *VIEW* 構文

再帰的な定義の VIEW が作れるようになりました。ビュー定義の中でビュー自体を参照させることができます。記述可能なことがらは WITH RECURSIVE 構文を使った再帰的な SELECT 文と同じですが、VIEW として定義するにあたっては記述がシンプルになるといえます。

幾つか使用例を示します。

```
postgres=# CREATE RECURSIVE VIEW nums (n) AS
            VALUES (1)   UNION ALL  SELECT n+1 FROM nums WHERE n < 3;
postgres=# SELECT * FROM nums ;
 n
```

```
---
 1
 2
 3
(3 rows)

postgres=# CREATE ROLE role1;
postgres=# CREATE ROLE role2 IN ROLE role1;
postgres=# CREATE ROLE role3 IN ROLE role2;
postgres=# CREATE ROLE role4 IN ROLE role2;
postgres=# SELECT * FROM pg_group;
 groname | grosysid |    grolist
---------+----------+---------------
 role1   |    16808 | {16809}
 role2   |    16809 | {16810,16811}
 role3   |    16810 | {}
 role4   |    16811 | {}
(4 rows)
postgres=# CREATE OR REPLACE RECURSIVE VIEW v_group (groname, line) AS
  SELECT groname, (groname)::text AS line, grosysid, grolist
     FROM pg_group WHERE groname ='role1'
  UNION ALL
  SELECT g.groname, (v.line || ' -> ' || g.groname)::text AS line,
     g.grosysid, g.grolist
     FROM pg_group g , v_group v WHERE v.grolist && ARRAY[g.grosysid];

postgres=# SELECT * FROM v_group;
 groname |          line
---------+------------------------
 role1   | role1
 role2   | role1 -> role2
 role3   | role1 -> role2 -> role3
 role4   | role1 -> role2 -> role4
(4 rows)
```

4.14. 更新可能 VIEW

PostgreSQL 9.3 からシンプルなビューに対する更新処理が追加の定義や設定なしに可能になりました。これまでは、ビューは作成した時点では参照（SELECT）専用であって、INSERT、DELETE、UPDATE の各操作に対してトリガーやルールを作成することで更新処理に対応していました。

以下のようにサンプルのビューを作ってみます。

```
postgres=# CREATE TABLE t_message (id serial, mes text, ts timestamp);
postgres=# INSERT INTO t_message (mes, ts) VALUES ('Hi', '2012-11-01'),
           ('bye', '2013-01-03'), ('Its me', '2012-12-02');
postgres=# CREATE VIEW v_mes_2012 AS SELECT mes, ts FROM t_message
             WHERE ts < '2013-01-01' AND ts >= '2012-01-01' ORDER BY ts;

postgres=# \d+ v_mes_2012
（省略 - PostgreSQL 9.2.x バージョンと同様の表示となる）

postgres=# SELECT table_name, is_updatable, is_insertable_into
           FROM information_schema.views WHERE table_name = 'v_mes_2012';
 table_name | is_updatable | is_insertable_into
------------+--------------+--------------------
 v_mes_2012 | YES          | YES
(1 row)
```

作った VIEW が更新可能であるかは「\d コマンド」では表示されません。本機能は、トリガーやルールが自動的に作られるという形では作られていません。更新可能であるかを調べるには information_schema.views を参照します。is_updatable が UPDATE、DELETE 可能であること、is_insertable_into が INSERT 可能であることを示しています。

自動で更新可能であるような「シンプルなビュー」とは以下を満たした SELECT 文に対するビューです。

- SELECT リストにはテーブルカラムの単純参照だけ、同カラム複数回参照も不可
- FROM には 1 テーブルだけ（更新可能ビューでも可）、結合は不可
- WITH 句、集約、LIMIT／OFFSET 、集合操作　の利用不可
- SECURITY_BARRIER 属性なし

逆に以下が可能であると考えれば、わかりやすいといえます。

- 表示カラムの限定
- WHERE 句条件
- ORDER BY

サンプルのビューにいくつか更新操作を行ってみます。

```
postgres=# INSERT INTO v_mes_2012 VALUES
           ('from view', '2012-05-10'), ('XX', '2012-08-15');
postgres=# UPDATE v_mes_2012 SET ts = '2010-08-15' WHERE mes = 'XX';
（ビューの条件範囲外の値に更新できる）

postgres=# SELECT * FROM v_mes_2012 ;
   mes     |          ts
-----------+---------------------
 from view | 2012-05-10 00:00:00
 Hi        | 2012-11-01 00:00:00
 Its me    | 2012-12-02 00:00:00
(3 rows)

postgres=# DELETE FROM v_mes_2012;
（ビューの条件に合う行が元テーブル t_message から削除される）

postgres=# SELECT * FROM t_message;
 id | mes |          ts
----+-----+---------------------
  2 | bye | 2013-01-03 00:00:00
  5 | XX  | 2010-08-15 00:00:00
(2 rows)
```

ビュー定義で値の範囲を決めても、範囲外の値をビューに INSERT、UPDATE することは可能です。また、ビューに対する検索条件は、もともとビュー定義にあった WHERE 句条件と、ビューに対して与えた WHERE 句条件の両方（AND 結合）となります。

4.15. ユーザ定義バックグラウンドワーカープロセスの枠組み

PostgreSQL を起動するとマスタープロセス（postmaster と呼ばれることもあります）、各クライアント接続に対応したバックエンドプロセスの他に「logger process」「autovacuum launcher process」などの子プロセスが起動します。これらは各々の担当する動作を行います。ダウンするとマスタープロセスによって自動的に再起動されます。

ソースコードには contrib/worker_spi というバックグラウンドワーカーのサンプル実装が含まれています。

これを動かしてみます。以下の設定を加えます。

```
$ cat >> $PGDATA/postgresql.conf << EOF
shared_preload_libraries = 'worker_spi'
worker_spi.naptime = 3
worker_spi.total_workers = 3
EOF
$ pg_ctl restart
$ ps x
  PID TTY      STAT   TIME COMMAND
17391 pts/1    S      0:00 -bash
17600 pts/1    S      0:00 /home/postgres/pgsql/9.3beta1p/bin/postgres
17601 ?        Ss     0:00 postgres: logger process
17603 ?        Ss     0:00 postgres: checkpointer process
17604 ?        Ss     0:00 postgres: writer process
17605 ?        Ss     0:00 postgres: wal writer process
17606 ?        Ss     0:00 postgres: autovacuum launcher process
17607 ?        Ss     0:00 postgres: stats collector process
17608 ?        Ss     0:00 postgres: bgworker: worker 3
17609 ?        Ss     0:00 postgres: bgworker: worker 2
17610 ?        Ss     0:00 postgres: bgworker: worker 1
17611 pts/1    R+     0:00 ps x
```

contrib/worker-spi をビルドしてインストールされる worker-spi.so を shared_preload_libraries で指定してロードしておく必要があります。起動するとバックグラウンドワーカープロセスが 3 つできているのが確認できます。ログにもこれらプロセスが起動していることを示すメッセージが表れます。また、これらのバックグラウンドワーカーは postgres データベースに接続して繰り返し SQL を実行します。「SELECT * FROM pg_stat_activity」を実行すると、 3 つの接続が 3 秒ごとに SQL 実行していることが確認できます。

```
postgres=# SELECT query_start,state,substr(query,1,30)
             FROM pg_stat_activity ;
      query_start               | state  |              substr
--------------------------------+--------+--------------------------------
 2013-05-27 13:32:09.71757+09   | idle   | WITH deleted AS (DELETE FROM s
 2013-05-27 13:32:09.731102+09  | idle   | WITH deleted AS (DELETE FROM s
```

```
 2013-05-27 13:32:09.724484+09 | idle   | WITH deleted AS (DELETE FROM s
 2013-05-27 13:32:11.420137+09 | active | select query_start,state,subst
(4 rows)
```

5. 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-9-3.html
```

5.1. ラージオブジェクトの拡張

PostgreSQL には巨大なバイナリデータを格納する機能として、ラージオブジェクト機能がありました。このサイズ上限が 2GB であったものが、4TB に拡張されました。本拡張は弊社技術者が開発したものです。

以下のように /dev/sr0 にあるを丸々登録して、取り出しするテストを行い、問題なく動作することが確認できました。

```
postgres=# \lo_import /dev/sr0
lo_import 16387
postgres=# \lo_export 16387 dvd.iso
postgres=# \q
$ cmp /dev/sr0 dvd.iso ; echo $?
0
 （cmp コマンドでデータ一致を確認）
```

旧来バージョンでは、（圧縮後サイズで）2GB を超えるデータを格納すると、読み出しや削除の操作が正しく動作しませんでした。特に上限を超過したというエラーにならず、無限ループ動作に陥ってしまうケースなどがあり危険でした。

5.2. pg_xlogdump

トランザクションログ（WAL ファイル）を解析するツールが標準で本体に付属するようになりました。こ

れまで同様のものがサードパーティのオープンソースソフトウェアで提供されていました。

以下のように指定した WAL ファイルの中身を参照することができます。以下の末尾部分は、BEGIN 、インデックスのあるテーブルへの INSERT、COMMIT を実行して、その後、「pg_ctl stop」を実行した際の記録となります。「rel 1663/12896/16573」が操作対象のテーブルとインデックスのテーブルスペース番号、データベース番号、ファイルノード番号をあらわしています。

```
  $ pg_xlogdump $PGDATA/pg_xlog/000000130000000200000F5
  rmgr: Gist        len (rec/tot):     72/   104, tx:      13236, lsn:
2/F5000038, prev 2/F4FFFFC0, bkp: 0000, desc: page_update: rel
1663/12896/33568; block number 0
   （中略）
  rmgr: Heap        len (rec/tot):     21/   237, tx:      13238, lsn:
2/F502A4A8, prev 2/F502A440, bkp: 1000, desc: insert: rel 1663/12896/16573;
tid 0/1
  rmgr: Btree       len (rec/tot):     18/   174, tx:      13238, lsn:
2/F502A598, prev 2/F502A4A8, bkp: 1000, desc: insert: rel 1663/12896/16579;
tid 1/3
  rmgr: Transaction len (rec/tot):     12/    44, tx:      13238, lsn:
2/F502A648, prev 2/F502A598, bkp: 0000, desc: commit: 2013-06-06
17:46:32.281513 JST
  rmgr: XLOG        len (rec/tot):     72/   104, tx:          0, lsn:
2/F502A678, prev 2/F502A648, bkp: 0000, desc: checkpoint: redo 2/F502A678;
tli 19; prev tli 19; fpw true; xid 0/13239; oid 33569; multi 19; offset 37;
oldest xid 1795 in DB 1; oldest multi 1 in DB 1; oldest running xid 0;
shutdown
```

pg_xlogdump には以下のような機能はありません。これらを確認するには、oid2name ツールでテーブル名を特定したり、WAL ファイル内のデータを lsn 値を参考に位置を特定して解析したり、といった手作業が必要となります。

- 番号でなくテーブル名やインデックス名にして表示する
- 書き込みしたデータ内容を人間に読める形で表示する

pg_xlogdump はシンプルな最小限の機能しかありませんが、以下のような用途に有用と考えられます。

- WAL ファイルが壊れていないか確認する
- 新たな更新処理が発生したか否かをデータベース接続なしに確認する
- トランザクション ID を使って、PITR（ポイント・イン・タイム・リカバリ）の位置を精密に指

定する際の位置決めをする

6. 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。