PostgreSQL 9.2 検証報告

2012-06-18

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. 本ドキュメントの目的

本ドキュメントでは2012年リリース予定のPostgreSQLの新機能をSRA OSS, Inc. 日本支社にて動作検証した結果を報告します。2012年6月 にリリースされた PostgreSQL 9.2 beta2 を使用して検証を行っています。PostgreSQL 9.2 では機能の追加、性能の向上の両方が行われていますが、本検証では主として機能の追加に関する検証を実施しています。一部については性能向上の検証も行っています。

1.1. PostgreSQL 9.2 の主な改良点

PostgreSQL 9.2 の主要な改良点は以下の通りです。

1.1.1. 性能改善

◆ *Index-only scan*

テーブル本体のデータをスキャンせず、インデックスだけをスキャンして応答を返すことができます。必ずしもいつでも使えるわけではなく、対象テーブルの VACUUM 状況、データ更新状況によって、できる場合とできない場合があります。

◆ データ並行処理性能向上

トランザクションログ書き込み周辺の改善が行われ、多数の同時更新処理における性能が向上しています。

また、ロックメカニズムの改良により多数の CPU コアを持つマシンでより性能向上が実現できるようになりました。64 コアマシンにて参照ＳＱＬ同時実行で顕著な性能向上が確認されています。

◆ カスケードレプリケーション

レプリケーションを構成する際に、スタンバイサーバからレプリケーションを受けるスタンバイを設置することができます。

◆ リクエストがないときにおける *CPU* 電力消費の軽減

アクセスの少ないサーバにおけるチェックポイントと WAL の処理量が減りました。これにより CPU 電力消費を抑えることができます。

1.1.2. 機能追加

◆ *JSON* データ型

JSON データとしての書式チェックや、 データベース上の配列データとの相互変換などができます。

◆ 範囲データ型

日付時刻や数値の範囲が 格納され、重なりがあるか、ある値が範囲に含まれるかの演算ができます。

◆ *DDL* 命令改善

ALTER 文など、いくつかの DDL 命令が改善され、 稼働中の変更が行いやすくなります。

1.2. 検証環境

今回検証に使用した環境は次の通りです。
主に機能面のテストであるため、高性能サーバ機ではなく、廉価マシンを使用しています。
また、特筆していない限り、PostgreSQL 設定はデフォルトのままとしています。

マシン	HP ProLiant MicroSerer CPU：AMD Turion II Neo N40L Dual-Core Processor メモリ：6GB　HDD：SATA 500GB
OS	CentOS release 6.2 x86_64
PostgreSQL	PostgreSQL 9.2 beta2

2. PostgreSQL9.2 検証

2.1. 性能改善

2.1.1. Index-only scan

◆ Index-only scan の動作をするかを検証

まずは、動作をするかを検証しました。インデックスを作成したカラムだけを参照するような SELECT 文であれば、インデックスだけを参照し、テーブル本体を参照しない、という実行プランが選択されることが期待できます。

```
$ pgbench -i -s 5
        ≪テスト用の インデックス付きの 50 万行のテーブル pgbench_accounts を作成≫

=# EXPLAIN SELECT aid FROM pgbench_accounts;
                                  QUERY PLAN
------------------------------------------------------------------------------
 Index Only Scan using pgbench_accounts_pkey on pgbench_accounts
  (cost=0.00..12996.80 rows=500000 width=4)
        ≪aid カラムはインデックス対象なので Index Only Scan という計画タイプが使われている≫

=# EXPLAIN SELECT aid, filler FROM pgbench_accounts;
                   QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on pgbench_accounts  (cost=0.00..13197.00 rows=500000 width=89)
(1 row)
        ≪今度はインデックス対象外の filler カラムも選択しているので通常のスキャンになる≫
```

◆ 性能を検証する

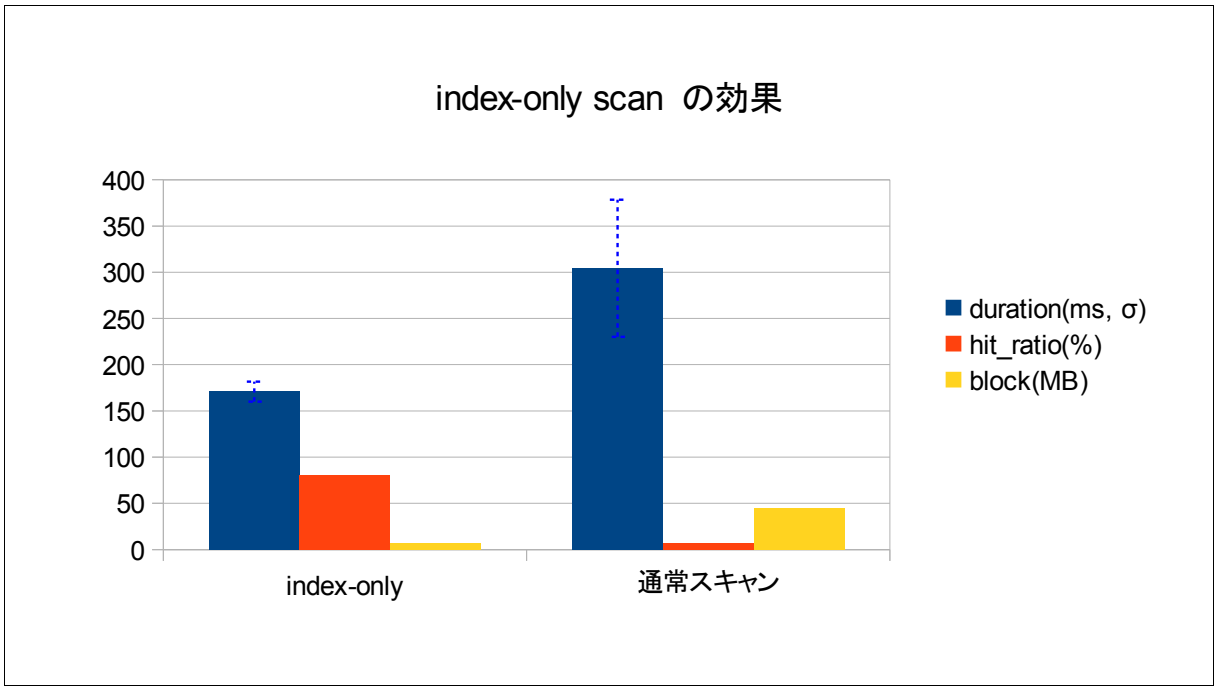
EXPLAIN コマンドに ANALYZE オプションを付けることで実際に実行した所要時間が得られます。また、設定 enable_indexonlyscan を off に切り替えることで index-only scan を使わないように指示できます。これらを使って、index-only scan を使った場合の性能上の効果を確認しました。対象に以下の 30 万件を選択する SELECT 文を使用しました。

```
=# SET enable_indexonlyscan TO off;
```

```
=# EXPLAIN (BUFFERS, ANALYZE) SELECT aid FROM pgbench_accounts WHERE aid < 300000;
                                        QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using pgbench_accounts_pkey on pgbench_accounts  (cost=0.00..13452.7
1 rows=299538 width=4) (actual time=0.100..355.221 rows=299999 loops=1)
  Index Cond: (aid < 300000)
  Buffers: shared read=5742
 Total runtime: 445.256 ms
```

以下グラフは各 8 回の計測の平均です。PostgreSQL 起動から設定 enable_indexonlyscan を on の状態、off の状態とを取り混ぜて、実行しています。SQL 処理時間（duration 単位はミリ秒) が、index-only scan では通常スキャンの 50～60%程度で済んでいます。また、通常スキャンは値のバラつきが大きくなっています（グラフの棒線は標準偏差）。

index-only scan は通常スキャンに比べてバッファヒット率(hit_ratio)が高くなっています。バッファヒット率は EXPLAIN の BUFFERS オプションの出力から取得しています。block は SQL 処理上、読まなければいけないブロック数（単位は MByte）で、約 7 倍の差があります。



本テスト環境では OS のメモリに余裕があり、OS レベルのバッファヒット率が高いため、メモリが不足している環境下では所要時間（duration）の差がもっと大きくなることが想定できます。また、本ケースと違って参照するデータが散らばって存在する場合にはディスクシークのコストと読まなければならないブロック数が増えて、より通常スキャンが遅くなるといえます。

◆ *index-only scan* が使えないケース

index-only scan は参照するページに更新されたデータがあって、未だ、VACUUM が済んでいない状態の場合は、利用できません。

postgresql.conf で autovacuum = off に設定したうえで、pgbench を使って、pgbench_accounts テーブルに多数の UPDATE をかけます。その後に、EXPLAIN を実行すると、プランとしては index-only scan をすると表示されますが、EXPLAIN ANALYZE で実際に実行させて結果を採取すると、「Heap Fetches: 68821」のような出力がされます。これは、68821 行についてはインデックスのみでなく、ヒープ（テーブル本体データ）からデータを参照しましたということを表しています。

```
        《予め autovacuum = off に設定》
$ pgbench -n -c 10 -t 100
        《pgbench を VACUUM なしで実行》
$ psql
=# EXPLAIN SELECT aid FROM pgbench_accounts WHERE aid < 300000;
                            QUERY PLAN
-------------------------------------------------------------------------------
 Index Only Scan using pgbench_accounts_pkey on pgbench_accounts  (cost=0.00..86
03.69 rows=301365 width=4)
  Index Cond: (aid < 300000)
(2 rows)
        《プランとしては index-only scan 出る》
=# EXPLAIN (BUFFERS, ANALYZE) SELECT aid FROM pgbench_accounts WHERE aid < 300000;
                                          QUERY PLAN
-------------------------------------------------------------------------------
 Index Only Scan using pgbench_accounts_pkey on pgbench_accounts  (cost=0.00..86
03.69 rows=301365 width=4) (actual time=0.209..188.913 rows=299999 loops=1)
  Index Cond: (aid < 300000)
  Heap Fetches: 68821
  Buffers: shared hit=3184 read=961
 Total runtime: 244.272 ms
(5 rows)
        《Heap Fetches が出力される》
```

2.1.2. データ並行更新性能向上

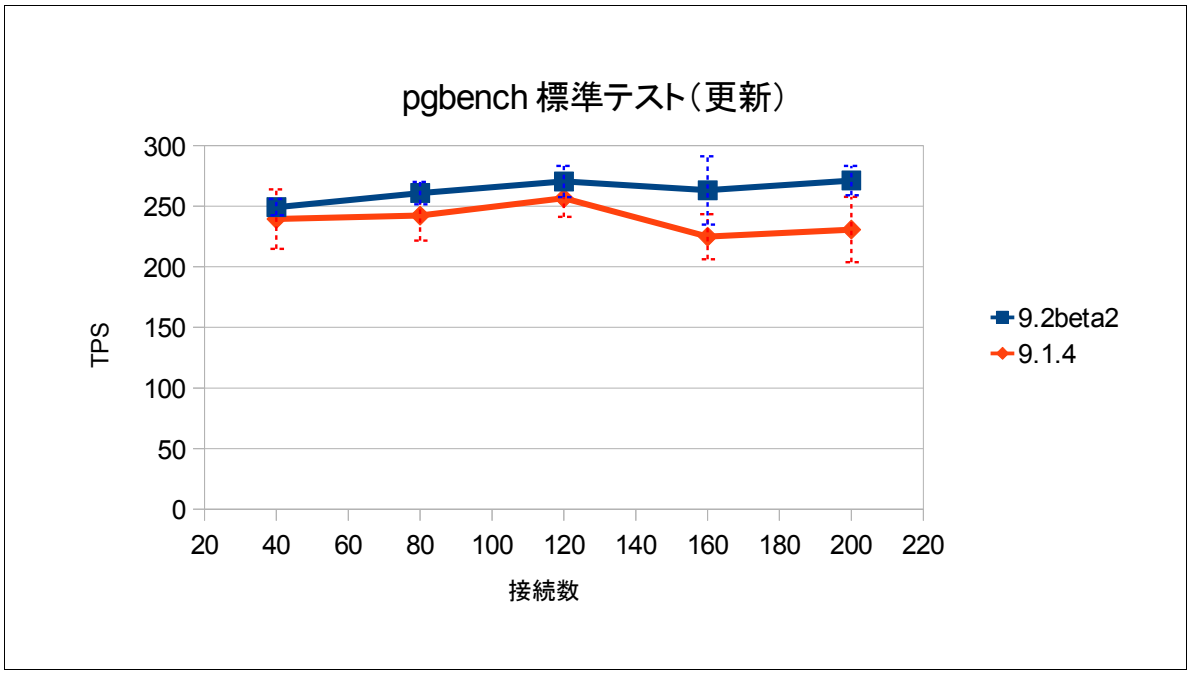
PostgreSQL 9.1.3 と PostgreSQL 9.2beta1 で同等データ量の更新処理を行い、更新性能差を確認します。両バージョンの DB はデフォルト設定で検証します。本検証では多 CPU コアマシンを使っていませんので 、WAL 書き込みの競合部分の改善だけが反映されるものと予想されます。

◆ 通常の *pgbench* による性能比較

まずは、通常の pgbench による比較を実施しました。

対象	9.1.4 バージョン と 9.2 beta2 バージョン
初期化パラメータ	pgbench -i -s 10 testdb
負荷パラメータ	pgbench -T 180 -c N -j 2 testdb　（N=40,80,120,160,200)
その他	initdb 1 回、再起動なし、pgbench 自体により毎回 VACUUM、 試験順ランダム各水準 5 回

以下の結果が得られました。同時接続数ごとの TPS 値（pgbench が報告する 1 秒辺りトランザクション処理数、including connections establishing の値を採用）のグラフで、縦棒は標準偏差となります。



CPU コア数の少ないマシン（本試験マシンは 2 Core）では pgbench の同時接続数の増加と共に TPS が増えていくものが、数十接続～200 接続くらいで頭打ちになります。この同時接続数を増やしても処理能力が頭打ちになるあたりで、5 %～20%程度の性能向上が確認できます。9.2 バージョンの方が、多接続に対して性能を維持する振る舞いをします。

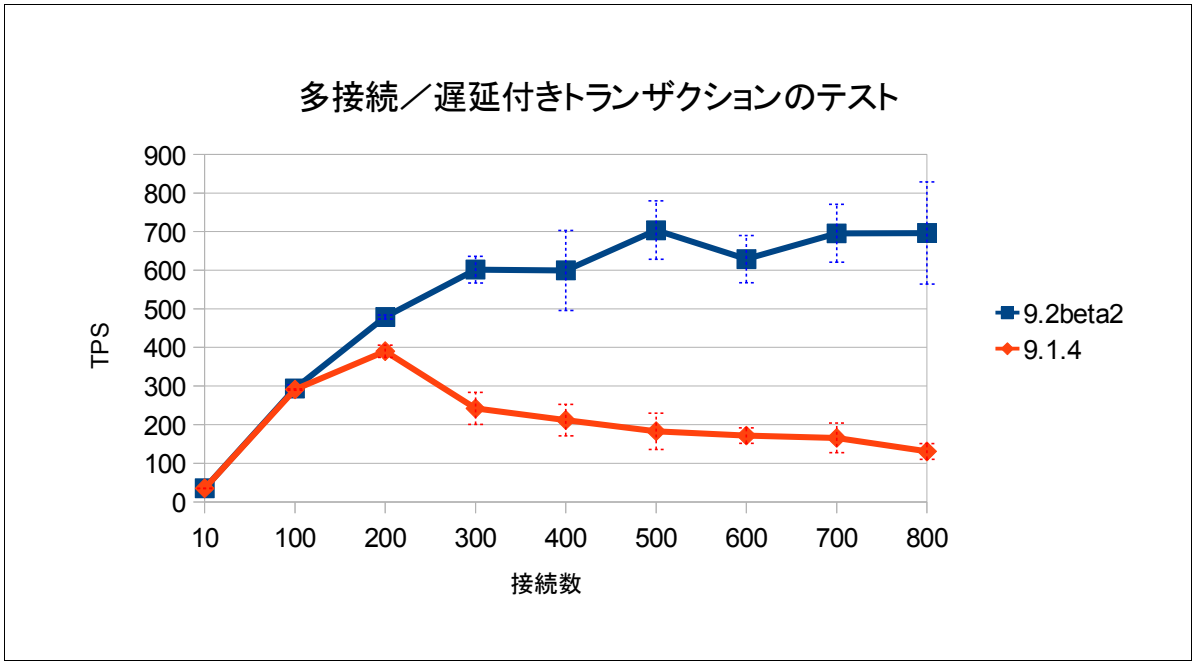
◆ 本改修の効果が特に出るパターン

PostgreSQL 9.2 の性能改善ポイントが WAL 書き込み周辺の競合の改善であるとわかっていますので、特にその効果が表れるテストを実施しました。以下の条件で pgbench を実行します。

対象	9.1.4 バージョン と 9.2 beta2 バージョン
初期化パラメータ	pgbench -i -s 10 -F 95 testdb
負荷パラメータ	pgbench -T 180 -c N -j 2 -f tx.sql testdb （N=10,100,200,…,700,800)
トランザクション (tx.sql)	\setrandom rs 1 1000000 \setrandom ru 1 1000000 \setrandom s 80 100 BEGIN \sleep :s ms SELECT * FROM pgbench_accounts WHERE aid = :rs \sleep :s ms UPDATE pgbench_accounts SET filler = 'xxxxxxxxxx' WHERE aid = :ru \sleep :s ms COMMIT
その他	initdb 1 回、再起動なし、pgbench 自体により毎回 VACUUM、 試験順ランダム各水準 5 回

ランダムな一行を SELECT して、ランダムな一行を UPDATE をします。pgbench デフォルトの更新トランザクションは 、3 つの UPDATE、2 つの SELECT、1 つの INSERT ですので、それよりも簡易な内容です。トランザクション中の各 SQL 文の合間に 80～100 ms の休止時間を挿入しています。データベースクライアントが、アプリケーション側の処理に時間を要しながらトランザクションを実行することを模しています。また、大きめの同時接続数で試験します。

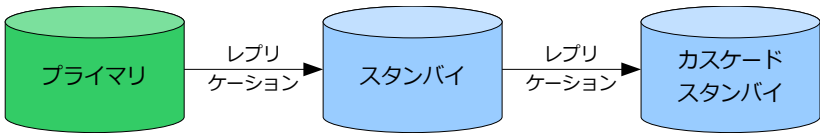
以下の結果が得られました。縦棒は TPS 値の標準偏差です。



9.2 バージョンが 9.1 バージョンの実に 3 倍～ 5 倍の性能を記録しています。同時接続数が多いほど、差が広がっています。本試験は、ディスク I/O や CPU の負荷はそれほどかけない一方で、更新トランザクションの同時 WAL 処理による競合遅延が目立つように組み立てたものです。

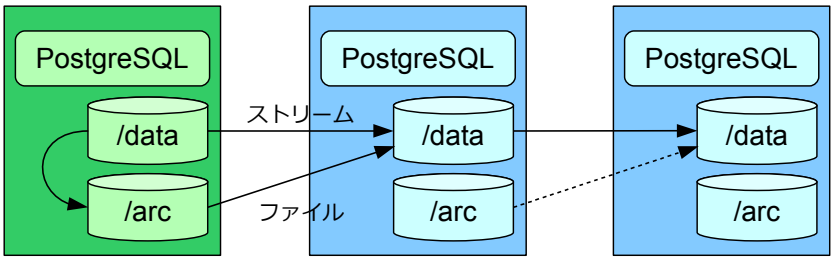
本結果は、多 CPU コアマシンを使っていなくとも、処理パターンによっては 旧バージョンを 9.2 バージョンに入れ替えることで大幅に性能アップできる場合もあることを示しています。

2.1.3. カスケードレプリケーション

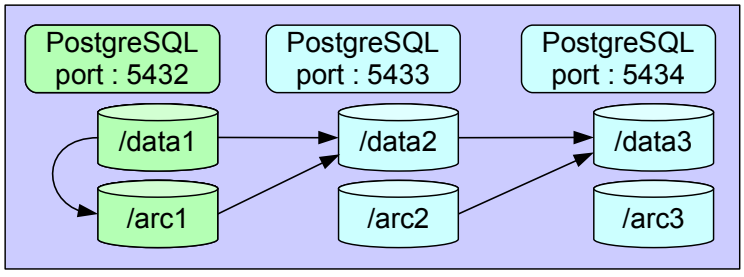


上記のようなクラスタ環境を構築し、カスケードレプリケーション機能を検証します。

レプリケーションにあたっては、ストリーミングレプリケーションとアーカイブログファイルの転送を併用します。ストリーミングレプリケーションが途絶した場合でもアーカイブログファイルの転送で復帰が可能になるため、より頑健な構成となります。一方で、アーカイブログファイルの転送経路は scp や NFS で別途に用意する必要があります。各サーバが PostgreSQL のデータベースクラスタとアーカイブログの格納領域を持つことになります。ただし、アーカイブログに書き込みされるのはプライマリサーバだけです。



本手順では、下図のように同一サーバマシン上でポートを変えたいくつかの PostgreSQL サーバを起動し、それらの間でレプリケーションを構築します。



また、スタンバイサーバは非同期レプリケーションで構築するものとします。

◆ 構築手順

スタンバイからレプリケーション用の接続を受け付けられるように pg_hba.conf を設定します。予めコメントアウトされた形でレプリケーション用の記述が用意されていますので、そのコメントを外せばよいです。リモートサーバにレプリケーションするには、ここで、そのサーバからの接続を許可するように記述します。

```
《pg_hba.conf の以下をコメントを解除して有効にする》
local   replication     postgres                                trust
host    replication     postgres        127.0.0.1/32            trust
host    replication     postgres        ::1/128                 trust
```

postgresql.conf を設定します。このとき、プライマリ側用の設定と共に、スタンバイ側で使う設定もしておきます。

```
《postgresql.conf の以下を設定》
port = 5432
wal_level = hot_standby
archive_mode = on
archive_command = 'cp "%p" "/var/lib/pgsql/arc1/%f"'
max_wal_senders = 3
logging_collector = on
```

これで起動します。

```
$ pg_ctl -D /var/lib/pgsql/data1 start
```

次にスタンバイサーバを作ります。postgres ユーザで以下の手順を実行します。

```
《プライマリ DB からベースバックアップを取る》
$ pg_basebackup -p 5432 --xlog=stream -D /var/lib/pgsql/data2 -v -P


《postgresql.conf 設定／以下２行を変更する》
$ vi /var/lib/pgsql/data2/postgresql.conf
port = 5433
archive_command = 'cp "%p" "/var/lib/pgsql/arc2/%f"'
```

```
《recovery.conf 作成／以下 4 行からなるファイルを作成する》
$ vi /var/lib/pgsql/data2/recovery.conf
standby_mode = 'on'
primary_conninfo = 'port=5432 user=postgres'
restore_command = 'cp "/var/lib/pgsql/arc1/%f" "%p"'
recovery_target_timeline = 'latest'
```

別マシン間でレプリケーションを行う場合、この restore_commnad が、scp になったり、NFS でリモートマウントしてあるディレクトリ間のコピーであったりする必要があります。

```
《 スタンバイ サーバを起動する》
$ pg_ctl -D /var/lib/pgsql/data2 start
```

さらにスタンバイから、カスケード接続されたスタンバイサーバを作ります。

```
《スタンバイ サーバからベースバックアップを取る》
$ pg_basebackup -p 5433 --xlog=stream -D /var/lib/pgsql/data3 -v -P

《 postgresql.conf 設定 ／以下 2 行を変更する》
$ vi postgresql.conf
port = 5434
archive_command = 'cp "%p" "/var/lib/pgsql/arc3/%f"'

《recovery.conf 作成／以下 4 行からなるファイルを作成する》
$ vi /var/lib/pgsql/data2/recovery.conf
standby_mode = 'on'
primary_conninfo = 'port=5433 user=postgres'
restore_command = 'cp "/var/lib/pgsql/arc2/%f" "%p"'
recovery_target_timeline = 'latest'

《 カスケードスタンバイ サーバを起動する》
$ pg_ctl -D /var/lib/pgsql/data3 start
```

◆ 動作確認

プライマリデータベースでテーブルを作ってデータを投入し、カスケードスタンバイサーバで、それが反

映されていることが確認できます。

```
$ pgbench -p 5432 -i testdb
$ psql -p 5434 -c "SELECT aid,bid,abalance FROM pgbench_ccounts LIMIT 1"
 aid | bid | abalance
-----+-----+----------
   1 |   1 |        0
(1 row)
```

《1秒 sleep を入れて更新と反映を確認》

```
$ psql -p 5432 -c "UPDATE pgbench_accounts SET abalance = 100 WHERE aid = 1" ; \
 sleep 1 ; \
 psql -p 5434 -c "SELECT aid, abalance FROM pgbench_accounts WHERE aid = 1"
UPDATE 1
 aid | abalance
-----+----------
   1 |      100
(1 row)
```

上記の「sleep 1 」を外すと、未だ更新が反映されていない結果が返ります。同一マシン内の本構成ではマシン負荷にならない小規模な更新はおおむね 1 秒以内にプライマリから 3 台目まで伝搬します。複数マシンで行う場合には、転送経路の遅延を見込む必要があります。

◆ レプリケーションの運用について

プライマリサーバを停止して、スタンバイサーバをプライマリに昇格してみます。そうしますと、スタンバイサーバ（サーバ 2）とカスケードスタンバイサーバ（サーバ 3）の間でレプリケーションが維持されていることが分かります。

《プライマリサーバ（サーバ 1）をダウンする》

```
$ pg_ctl stop -D /var/lib/pgsql/data1 -mi
```

《 スタンバイ サーバ（サーバ 2）を昇格する 》

```
$ pg_ctl promote -D /var/lib/pgsql/data2
```

《サーバ 2 とサーバ 3 でレプリケーション状態を確認 》

```
$ psql -p 5433 -c "UPDATE pgbench_accounts SET abalance = 120 WHERE aid = 1"
```

```
UPDATE 1
$ psql -p 5434 -c "SELECT aid, abalance FROM pgbench_accounts WHERE aid = 1"
 aid | abalance
-----+----------
   1 |      120
(1 row)
```

レプリケーションの同期状態を pg_stat_replication で確認することができます。ただし確認できるのは、レプリケーション元とその直接のスタンバイサーバの情報だけです。

```
《プライマリサーバ→スタンバイサーバの同期状態を確認する》
=# SELECT * FROM pg_stat_replication ;
-[ RECORD 1 ]----+------------------------------
pid              | 27528
usesysid         | 10
usename          | repuser
application_name | walreceiver
client_addr      |
client_hostname  |
client_port      | -1
backend_start    | 2012-05-28 11:01:46.857726+09
state            | streaming
sent_location    | 0/70735968
write_location   | 0/70735968
flush_location   | 0/707337C0
replay_location  | 0/707337C0
sync_priority    | 0
sync_state       | async
```

また、PostgreSQL 9.2 から pg_xlog_location_diff 関数が導入され、トランザクションログの処理位置の差をバイト単位で簡単に参照できるようになりました。

```
《プライマリサーバで、プライマリ→スタンバイの同期遅延を確認する》
=# SELECT pg_size_pretty(pg_xlog_location_diff(write_location, replay_location))
```

```
-#   FROM pg_stat_replication ;
 pg_size_pretty
----------------
 528 bytes
```

2.1.4. JSONデータ型

新たに追加された JSON データ型について動作確認します。以下のように利用することができます。

i. array_to_json 関数 (PostgreSQL 配列型 → JSON 配列型)

```
■ 一次元配列
=# SELECT array_to_json('{"a", "あ"}'::TEXT[]);
array_to_json
---------------
["a","あ"]

■ 多次元配列
=# SELECT array_to_json('{{"a","b"},{"あ","い"}}'::TEXT[]);
 array_to_json
------------------------
[["a","b"],["あ","い"]]
```

ii. row_to_json 関数 (PostgreSQL Row 型 → JSON オブジェクト型)

```
■ JSON オブジェクトのキーを自動付与する
=# SELECT row_to_json(row(1,'あ'));
 row_to_json
--------------------
{"f1":1,"f2":"あ"}

■ JSON オブジェクトのキーをカスタマイズする
=# SELECT row_to_json(s) FROM (SELECT 1 as id, 'あ' as name) s;
 row_to_json
```

```
----------------------
{"id":1,"name":"あ"}
```

iii. 関数を混在させる

```
■ JSON オブジェクトと一次元配列の混在形式
=# SELECT row_to_json(row(1,'{"あ","い"}'::TEXT[]));
  row_to_json
---------------------------
{"f1":1,"f2":["あ","い"]}

■ JSON オブジェクトと多次元配列の混在形式
=# SELECT row_to_json(row(1,'{{"あ","い"},{"う","え"}}'::TEXT[]));
   row_to_json
----------------------------------------
{"f1":1,"f2":[["あ","い"],["う","え"]]}

■ JSON オブジェクトの混在形式
=# SELECT row_to_json(row(1,row('2','あ')));
    row_to_json
-----------------------------------
{"f1":1,"f2":{"f1":"2","f2":"あ"}}
```

2.1.5. 範囲データ型

新たに追加された範囲データ型について動作確認します。

◆ 概念

以下の範囲データ型がビルドインで用意されています。元となる型の値のペアから構成されます。

```
INT4RANGE — Range of INTEGER
INT8RANGE — Range of BIGINT
NUMRANGE — Range of NUMERIC
TSRANGE — Range of TIMESTAMP WITHOUT TIME ZONE
```

```
TSTZRANGE — Range of TIMESTAMP WITH TIME ZONE
DATERANGE — Range of DATE
```

また、以下のようにしてユーザ定義の範囲データ型を作ることができます。

```
=# CREATE TYPE floatrange AS RANGE (subtype = float8, subtype_diff = float8mi);
《1.234 以上、5.678 未満の範囲》
=# SELECT '[1.234, 5.678)'::floatrange;
  floatrange
---------------
 [1.234,5.678)
```

範囲データ型は、境界値を範囲に含む、含まないを共通の記述方法で指定することができます。「 [ ] 」は境界値を含む（以上、以下）、「 ( ) 」は境界値を含まない（より大きい、未満）という意味になります。

また、空文字列を範囲データ型の値として与えると、範囲内に値を含まない「空範囲」として認識されます。

```
《空範囲》
=# select '(,)'::int4range;
 int4range
-----------
 (,)
(1 row)

《NULL は無限を意味する／100 以上という意味》
=$ select '[100,]'::numrange;
 numrange
----------
 [100,]
(1 row)
```

◆ 動作確認

範囲データ型の値を作るには、文字列リテラルからキャストするほかに、コンストラクタを使う方法も用意されています。

```
■ コンストラクタ（第三引数は省略された場合、'[)' に相当する）
=# SELECT INT4RANGE(3, 7, '[)');
 int4range
-----------
 [3,7)
```

範囲データ型にインデックスを作ることができます。有用なのは GiST インデックスで、範囲に値があるか、重なっているか、などの判定にインデックスが使われます。B-tree インデックスや hash インデックスも作れますが、実用性はありません。

```
=# CREATE INDEX t_idx ON t USING gist (during);
```

範囲データ型のカラムに EXLUDE 制約を適用できます。以下のようにして、値の範囲に重なりがあるものを除外することができます。対象の範囲データ型に GiST インデックスが使える必要があります。

```
《検証用テーブルを作成する》
=# CREATE TABLE range (id INT, during TSRANGE, EXCLUDE USING gist (during WITH &&));

《検証用データを登録する》
=# INSERT INTO RANGE VALUES (1108, '[2010-01-01 11:30, 2010-01-01 13:00)');

《範囲が重複するデータを登録し、動作を確認する／範囲が重複しているので登録できない》
=# INSERT INTO RANGE VALUES (1108, '[2010-01-01 11:31, 2010-01-01 11:32)');
ERROR:  conflicting key value violates exclusion constraint "range_during_excl"
DETAIL:  Key (during)=(["2010-01-01 11:31:00","2010-01-01 11:32:00")) conflicts
with existing key (during)=(["2010-01-01 11:30:00","2010-01-01 13:00:00")).

《 btree_gist EXTENSION 制約》
# btree_gist モジュールを事前にインストール必要
=# CREATE EXTENSION btree_gist;

《検証用テーブルを作成する》
=# CREATE TABLE range(room TEXT, during TSRANGE, EXCLUDE USING GIST (room WITH =,
during WITH &&));
```

```
《検証用データを登録する》
=# INSERT INTO RANGE VALUES ('123A', '[2010-01-01 14:00, 2010-01-01 15:00)');

《範囲が重複するデータを登録し、動作を確認／room = '123A' のレコードに重複するので登録できない》
=# INSERT INTO RANGE VALUES ('123A', '[2010-01-01 14:30, 2010-01-01 16:00)');
ERROR:  conflicting key value violates exclusion constraint "range_room_during_excl"
DETAIL:  Key (room, during)=(123A, ["2010-01-01 14:30:00","2010-01-01 16:00:00"))
conflicts with existing key (room, during)=(123A, ["2010-01-01 14:00:00","2010-01-01
15:00:00")).

《範囲が重複するデータを登録し、動作を確認／room = '123B' のレコードは存在しないので登録できる》
=# INSERT INTO RANGE VALUES ('123B', '[2010-01-01 14:30, 2010-01-01 16:00)');
=# SELECT * FROM RANGE ;
 room |                    during
------+--------------------------------------------
 123A | ["2010-01-01 14:00:00","2010-01-01 15:00:00")
 123B | ["2010-01-01 14:30:00","2010-01-01 16:00:00")
```

2.1.6. DDL 命令の改善

いくつかの DDL 命令が改善されています。このうちの一つを確認します。

◆ ***DROP INDEX CONCURRENTLY***

CONCURRENTLY オプションを指定することで、インデックス削除処理中に他のセッションをブロックしないで削除できるようになります。

i. これまでの失敗例

```
《接続その１で値を挿入する／foo テーブルのインデックスに書き込みが発生----- ※１》
=# BEGIN;
=# INSERT INTO foo VALUES (100, 'abc');
=#

《接続その２でインデックス削除／接続その１のトランザクションが終わるまでブロック ----- ※２》
```

```
=# DROP INDEX foo_idx;
DROP INDEX

《接続その３で foo テーブルを SELECT／参照処理がブロックされてしまう ----- ※３》
=# SELECT * FROM foo;
```

従来の対策としては、DROP INDEX を行うときに以下のように NOWAIT ロック取得を試みて、失敗すればトランザクション再実行、成功したら DROP INDEX を行うという方法でした。

```
=# BEGIN;
=# LOCK TABLE foo NOWAIT;
=# DROP INDEX foo_idx;
=# COMMIT;
```

新たに導入された DROP INDEX CONCURRENTLY を DROP INDEX の代わりに使うと、※１の処理があったとして、※２のところで自身はロック待ちしますが、※３のように他のトランザクションをブロックすることはありません。※３の SELECT 文はブロックされずに実行されます。

なお、 DROP INDEX CONCURRENTLY には以下の制限があります。

i. インデックス名を複数で指定できない
ii. BEGIN ...COMMIT トランザクションブロック内で実行できない
iii. CASCADE オプションはサポートされない

3．まとめ

PostgreSQL 9.2 の主要な拡張点の動作確認を行いました。PostgreSQL 9.2 では性能改善と、実用度の高い拡張がなされているといえます。

本検証で実施していない重要な拡張点としては「消費電力の軽減」「多ＣＰＵ性能アップ」などがあります。

他にも多数の拡張、改善がなされており、それらは PostgreSQL 9.2 ドキュメントのリリースノートに記載されています。また、SRA OSS, Inc. では、これまでの PostgreSQL 新バージョンリリース時と同様に、リリースノートに実行例や追加説明を付加して利用者むけの内容としたものを公開していく予定です。