PostgreSQL 10 検証レポート

1.1 版
2017 年 11 月 20 日　(修正 2019 年 5 月 31 日)

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

目次

Copyright © 2017 SRA OSS, Inc. Japan All rights reserved.

Copyright © 2017 SRA OSS, Inc. Japan All rights reserved.

1. はじめに

本文書は PostgreSQL 10 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 10 について検証しようとしているユーザの助けになることを目的としています。
2017 年 10 月 5 日リリースされた PostgreSQL 10.0 を使用して検証を行って、本文書を作成しています。

2. 概要

PostgreSQL 10 の主要な新機能は以下の通りです。本ドキュメントではこれらの項目を取り上げます。

- ロジカルレプリケーション
- 宣言的パーティショニング
- パラレルクエリの強化
- ハッシュインデックスの WAL 対応
- 同期レプリケーションの設定構文の拡張
- pg_stat_activity の拡張
- 拡張統計（CREATE STATISTICS）
- Transition Table
- postgres_fdw の改善
- libpq の拡張

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 10 ドキュメント内のリリースノート（以下 URL）に記載されています。

https://www.postgresql.org/docs/devel/static/release-10.html

3. 検証のためのセットアップ

3.1. ソフトウェア入手

PostgreSQL 10 のベータ版は以下 URL のページからダウンロード可能です。ソースコード、Windows むけバイナリのインストーラ、RPM yum リポジトリが用意されています。

https://www.postgresql.org/download/snapshots/

Copyright © 2017 SRA OSS, Inc. Japan All rights reserved.

3.2. 検証環境

検証環境として、HP ProLient MicroServer 上の CentOS 6.9 / x86_64 を使用しました。これは小型サーバマシンであり、本検証は、具体的な特定マシン上の性能の提示や大規模サーバにおける性能の検証は意図していません。

3.3. インストール

zlib、zlib-devel、readline、readline-devel の各パッケージがあらかじめインストールされている状態で、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。

```
$ cd postgresql-10
$ ./configure --prefix=/usr/local/pgsql/10.0 --enable-debug
$ make world
# su -c '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 > 10.0.env <<'EOF'
PGHOME=/usr/local/pgsql/10.0
export PATH=$PGHOME/bin:$PATH
export LD_LIBRARY_PATH=$PGHOME/lib:$LD_LIBRARY_PATH
export PGDATA=/usr/local/pgsql/data10.0
export PGPORT=15432
EOF
$ . 10.0.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 をデフォルトとします。

```
$ initdb --no-locale --encoding=UTF8
```

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
EOF
```

PostgreSQL を起動します。

```
$ pg_ctl -w start
```


Copyright © 2017 SRA OSS, Inc. Japan All rights reserved.

さらに、ロジカルレプリケーションの検証用に、環境変数（PGDATA, PGPORT）の値を変えた以下の設定でデータベースクラスタを作成しました。データベースクラスタの作成、設定ファイルの変更について上述のものと同様です。

```
$ cat > 10.0_2.env <<'EOF'
PGHOME=/usr/local/pgsql/10.b1
export PATH=$PGHOME/bin:$PATH
export LD_LIBRARY_PATH=$PGHOME/lib:$LD_LIBRARY_PATH
export PGDATA=/usr/local/pgsql/data10.0_2
export PGPORT=15433
EOF
$ . 10.0_2.env
```

◆ *--with-icu オプションでのビルド*

PostgreSQL 10 から ICU ライブラリを使った照合順序（COLLATE）の処理がサポートされました。デフォルトで使われる OS 付属ライブラリ（Linux なら glibc）の代わりに、ICU ライブラリを利用するにはビルド時 configure オプションで--with-icu を指定します。このとき、いくつか注意が必要です。

まず、ICU ライブラリのパッケージのインストールが必要です。RHEL/CentOS 6.x、7.x においては、icu、libicu、libicu-devel をインストールしておきます。加えて、ICU_LIBS、ICU_CFLAGS 環境変数にオプションを設定します。以下のように icu-config コマンドを使って設定すべき値を与えることができます。

```
$ export ICU_LIBS=$(icu-config --ldflags-libsonly)
$ export ICU_CFLAGS=$(icu-config --cppflags)
$ ./configure 《その他のオプション》 --with-icu
$ make world
```

なお、ICU ライブラリを使ったたけでは、日本語の照合順序は平仮名・片仮名の混在に対応した辞書順になってくれません。「は」→「ハ」→「ひ」→「ヒ」→「ふ」→「フ」といった順にはならず、「は」→「ひ」→「ふ」→「ハ」→「ヒ」→「フ」の順になります。照合順序（COLLATE）に 'ja-x-icu' などの指定が必要です。利用可能な照合順序の一覧は psql 内での \dOS コマンドで参照できます。

Copyright © 2017 SRA OSS, Inc. Japan All rights reserved.

4. 主要な追加機能

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。

```
/usr/local/pgsql/10.0/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/10/static/
```

4.1. ロジカルレプリケーション

PostgreSQL 10 からロジカルレプリケーションがサポートされました。PostgreSQL には既に 9.0 からサポートされているストリーミングレプリケーションがありますが、この方法ではデータベースを構成するファイルがビット単位で複製されることから「物理的な」レプリケーションと呼ばれます。一方、ロジカルレプリケーションでは、データの変更内容が「どのテーブルのどの行がどう変更されたか」という論理的に記述された形式で転送されることから、「論理的な（ロジカルな）」レプリケーションと呼ばれます。

4.1.1. ロジカルレプリケーションの概要

PostgreSQL のロジカルレプリケーションは publisher / subscriber（出版者 / 購読者）モデルが採用されており、レプリケーション元は publisher、レプリケーション先は subscriber と呼ばれます。subscriber は publisher 上に定義された publication というオブジェクトを購読することにより、テーブルに対する変更内容を取得し、データの複製を行います。

Copyright © 2017 SRA OSS, Inc. Japan All rights reserved.

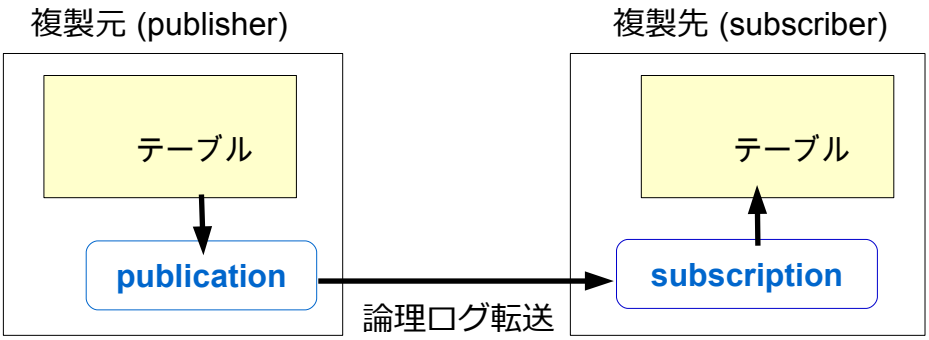


図 *1:* ロジカルレプリケーション概念図

ロジカルレプリケーションはテーブル単位で行われます。1 つの publication には複数のテーブルを登録することが可能です。また、1 つの publication を複数の subscriber から購読したり、1 つの subscriber が複数の publication を購読することができます。さらに、subscriber の中に publication を作成し、それを別の subscriber から購読するといった、カスケーディングレプリケーションを行うことも可能です。

4.1.2. 必要な設定

Publisher 側の PostgreSQL では、wal_level を logical に変更しておく必要があります。また max_replication_slots、max_wal_senders の設定値を subscriber の数以上にしておく必要がありますが、こららはデフォルトで 10 となっているので今回の検証では変更の必要はありません。

```
wal_level = logical
max_replication_slots = 10 # subscriber の数以上
max_wal_senders = 10 # subscriber の数以上
```

また、subscriber 側では max_replication_slots、max_logical_replication_workers、max_worker_processes の設定値が、作成する subscription オブジェクト（後述）の数以上である必要があります。これらについても、今回の検証ではデフォルト値で問題ありません。

```
max_replication_slots = 10
max_logical_replication_workers = 4
max_worker_processes = 8
```


Copyright © 2017 SRA OSS, Inc. Japan All rights reserved.

4.1.3. データベースおよびテーブルの作成

検証用のテーブルを publisher、subscriber の両方の PostgreSQL で作成します。今回はデータベース test にテーブル tbl を以下のように作成しました。プライマリキーは、デフォルトで UPDATE, DELETE の結果をレプリケートするために必要です（詳しくは後述、途中）。subscriber 側のテーブルでもプライマリキーを作成しておく必要があります。

```
（両方の PostgreSQL に作成）
test=# CREATE TABLE tbl(id int primary key, data text);
CREATE TABLE
```

publisher 側のテーブルに初期データとして以下を挿入しておきます。

```
（publisher 側で）
test=# INSERT INTO tbl VALUES (1,'one'), (2,'two'), (3,'three');
INSERT 0 3
```

4.1.4. publication の作成

publisher 側に publication オブジェクトを作成します。このオブジェクトは「どのテーブルのどの変更を転送するか」といった、レプリケーション内容をフィルタリングするための情報を定義します。ここでは、pub という名前の publication を作成し、テーブル tbl を登録します。この時に転送する変更内容（INSERT, UPDATE, DELETE の組み合わせ）を指定することができますが、以下では特に指定していないので、 INSERT, UPDATE, DELETE の全ての変更が転送されます。

```
（以下、publisher 側で）
test=# CREATE PUBLICATION pub FOR TABLE tbl;
CREATE PUBLICATION
```

作成済みの publication オブジェクトは psql の\dRp コマンド、および、システムテーブル pg_publication で確認できます。

```
test=# \dRp
           List of publications
 Name |  Owner   | Inserts | Updates | Deletes
```


Copyright © 2017 SRA OSS, Inc. Japan All rights reserved.

```
------+----------+---------+---------+---------
 pub  | postgres | t       | t       | t
(1 row)
```

```
test=# SELECT * FROM pg_publication;
 pubname | pubowner | puballtables | pubinsert | pubupdate | pubdelete
---------+----------+--------------+-----------+-----------+-----------
 pub     |       10 | f            | t         | t         | t
(1 row)
```

publication に登録されているテーブルは、pg_publication_tables または pg_publication_rel から確認できます。pg_publication_rel ではオブジェクト ID の番号で出力されます。

```
test=# SELECT * FROM pg_publication_tables ;
 pubname | schemaname | tablename
---------+------------+-----------
 pub     | public     | tbl
(1 row)
```

```
test=# SELECT * FROM pg_publication_rel ;
 prpubid | prrelid
---------+---------
   16406 |   16398
(1 row)
```

4.1.5. subscription の作成

subscriber 側で subscription オブジェクトを作成します。これはどの publication を購読するかを定義するものです。ここでは publication pub を購読する sub という名前の subscription を作成します。CONNECTION 句には publisher サーバへの接続文字列を指定します。接続ユーザは publisher 側に存在する replication 権限を持つユーザである必要があります。以下ではスーパーユーザの postgres を指定しています。

```
（以下、subscriber 側で）
test=# CREATE SUBSCRIPTION sub
```


Copyright © 2017 SRA OSS, Inc. Japan All rights reserved.

```
         CONNECTION 'host=localhost port=15432 user=postgres dbname=test'
         PUBLICATION pub;
NOTICE:  created replication slot "sub" on publisher
CREATE SUBSCRIPTION
```

NOTICE メッセージで出力されている通り、CREATE SUBSCRIPTION 文の実行時に、publisher 側のサーバでレプリケーションスロットが自動的に作成されます。この文がコミットされると、テーブルの初期コピーが行われ、レプリケーションが開始されます。初期コピーにより、事前に publisher 側のテーブルに挿入しておいた行が、subscriber にコピーされたことが確認できます。

```
test=# SELECT * FROM tbl ;
 id | data
----+-------
  1 | one
  2 | two
  3 | three
(3 rows)
```

作成済みの subscription は psql の\dRs コマンドか、pg_subscription カタログで確認できます。

```
test=# \dRs
         List of subscriptions
 Name |  Owner   | Enabled | Publication
------+----------+---------+-------------
 sub  | postgres | t       | {pub}
(1 row)
```

```
test=# \x on
test=# SELECT * FROM pg_subscription;
-[ RECORD 1 ]---+-----------------------------------------------------
subdbid         | 16395
subname         | sub
subowner        | 10
subenabled      | t
subconninfo     | host=localhost port=15432 user=postgres dbname=test
```


Copyright © 2017 SRA OSS, Inc. Japan All rights reserved.

```
subslotname      | sub
subsynccommit    | off
subpublications | {pub}
```

subscription に登録されているテーブルは、pg_subscription_rel から確認できます。

```
test=# SELECT * FROM pg_subscription_rel;
-[ RECORD 1 ]---------
srsubid     | 16420
srrelid     | 16396
srsubstate | r
srsublsn    | 0/16A1520
```

4.1.6. レプリケーションの確認

レプリケーションを確認するため、publisher 側のテーブルに行を挿入してみます。

```
（publisher 側で）
test=# INSERT INTO tbl VALUES (4, 'four');
INSERT 0 1
```

しばらくすると subscriber 側でその行が参照できるようになります。

```
（subscriber 側で）
test=# SELECT * FROM tbl;
 id | data
----+-------
  1 | one
  2 | two
  3 | three
  4 | four
(4 rows)
```

INSERT 文だけではなく、COPY 文を使った挿入もレプリケーション対象となります。
以下例では任意のプログラム（ここでは echo）からの出力を COPY に読み込ませる構文を使っています。

```
（publisher 側で）
```


Copyright © 2017 SRA OSS, Inc. Japan All rights reserved.

```
test=# COPY tbl FROM PROGRAM 'echo 5,five' WITH CSV;
COPY 1
```

subscriber 側で COPY 文の結果を確認します。

```
（subscriber 側で）
test=# SELECT * FROM tbl;
 id | data
----+-------
  1 | one
  2 | two
  3 | three
  4 | four
  5 | five
(5 rows)
```

次に publisher 側で行の更新を行います。

```
（publisher 側で）
test=# UPDATE tbl SET data = 'xxx' WHERE id >= 4;
UPDATE 2
```

更新内容が subscriber 側に反映されていることが確認できます。

```
（subscriber 側で）
test=# SELECT * FROM tbl;
 id | data
----+-------
  1 | one
  2 | two
  3 | three
  4 | xxx
  5 | xxx
(5 rows)
```

最後に publisher 側で行の削除を行います。

```
（publisher 側で）
```


Copyright © 2017 SRA OSS, Inc. Japan All rights reserved.

```
test=# DELETE FROM tbl WHERE data = 'xxx';
DELETE 2
```

subscriber 側で削除が反映されていることが確認できます。

```
（subscriber 側で）
test=# SELECT * FROM tbl;
 id | data
----+-------
  1 | one
  2 | two
  3 | three
(3 rows)
```

4.1.7. 衝突の解消

ロジカルレプリケーションではストリーミングレプリケーションと違い、レプリケーション先である subscriber 側で更新を実行することができます。そのため、プライマリキーや一意制約があるテーブルでは、レプリケーション時に制約の競合が発生する場合があります。

実際に衝突を発生させるため、まず subscriber 側で以下の行を挿入します。

```
（subscriber 側で）
test=# INSERT INTO tbl VALUES (10, 'Ten');
INSERT 0 1
test=# SELECT * FROM tbl;
 id | data
----+-------
  1 | one
  2 | two
  3 | three
 10 | Ten
(4 rows)
```

次に publisher 側で以下の 2 行を挿入します。

```
（publisher 側で）
```


Copyright © 2017 SRA OSS, Inc. Japan All rights reserved.

```
test=# INSERT INTO tbl VALUES (10, 'ten'), (11,'eleven');
INSERT 0 2
test=# SELECT * FROM tbl;
 id |  data
----+--------
  1 | one
  2 | two
  3 | three
 10 | ten
 11 | eleven
(5 rows)
```

これらの publisher 側で挿入された行はいつまで待っても subscriber 側には現れません。subscriber 側の PostgreSQL のログを見ると、以下のメッセージが出力されており、id カラムにおいて制約の競合が発生していることがわかります。競合が発生すると、それが解消されるまでレプリケーションは停止してしまい 、publisher で発生した変更は subscriber に反映されなくなります。

```
LOG:  starting logical replication worker for subscription "sub"
LOG:  logical replication apply for subscription sub started
ERROR:  duplicate key value violates unique constraint "tbl_pkey"
DETAIL:  Key (id)=(10) already exists.
LOG:  worker process: logical replication worker for subscription 16420 (PID
14527) exited with exit code 1
```

競合を解消する方法は 2 つあります。1 つは subscriber 側で競合が発生している行を手動で削除あるいは更新することです。ここでは、(10,'Ten')の行を subscriber 側のテーブルから削除します。

```
（subscriber 側で）
test=# DELETE FROM tbl WHERE id = 10;
DELETE 1
```

その後しばらくすると、publisher 側で挿入した行が subscriber 側に現れることが確認できます。

```
（subscriber 側で）
test=# SELECT * FROM tbl;
 id |  data
----+--------
```


Copyright © 2017 SRA OSS, Inc. Japan All rights reserved.

```
 1 | one
 2 | two
 3 | three
10 | ten
11 | eleven
(5 rows)
```

もう 1 つの方法は、publisher から転送された変更の適用を subscriber 側でスキップしてしまうことです。検証のため以下の行を subscriber 側に挿入します。

```
（subscriber 側で）
test=# INSERT INTO tbl VALUES (100, 'Hundred');
INSERT 0 1
```

これに競合する行を publisher 側のテーブルに挿入します。

```
（publisher 側で）
test=# INSERT INTO tbl VALUES (100, 'hundred');
INSERT 0 1
```

先ほどと同様に、subscriber のログから競合が発生したことが確認できました。

```
ERROR:  duplicate key value violates unique constraint "tbl_pkey"
DETAIL:  Key (id)=(100) already exists.
```

この競合を解消するため、publisher 側から転送された(100,'hundred')の行の挿入を subscriber 側でキャンセルします。具体的には、pg_replication_origin_advance 関数を使用してレプリケーションの進捗指定を強制的に進めます。現時点で subscriber に反映されている変更のトランザクションログの位置（LSN)と、その変更の publisher における LSN の対応関係は、pg_replication_origin_status から確認できます。subscriber 側で以下を実行します。

```
（subscriber 側で）
test=# SELECT * FROM pg_replication_origin_status;
 local_id | external_id | remote_lsn | local_lsn
----------+-------------+------------+-----------
        1 | pg_16420    | 0/16DFBB8  | 0/209D3F8
```

この結果から、反映済の変更の publisher 側の LSN（remote_lsn）は 0/16DFBB8、subscriber 側の

Copyright © 2017 SRA OSS, Inc. Japan All rights reserved.

LSN（local_lsn）は 0/209D3F8 であることがわかります。なお、external_id カラムの値の pg_16420 はレプリケーション進捗を追跡するための識別子で、subscription オブジェクトの OID(16420)を元に命名されています。

一方で publisher 側の現在の LSN は pg_current_wal_lsn 関数で確認できます。publisher で以下を実行すると、LSN は 0/16E0948 であることがわかります。

```
（publisher 側で）
test=# SELECT pg_current_wal_lsn();
 pg_current_wal_lsn
--------------------
  0/16E0948
(1 row)
```

以上から、subscriber 側のレプリケーション位置を 0/16E0948 まで進めれば、衝突の原因となっているトランザクションをスキップできることがわかります。subscriber 側で以下のように、レプリケーション進捗追跡の識別子と、publisher 側の LSN を指定して pg_replication_origin_advance 関数を実行します。

```
（subscriber 側で）
test=# SELECT pg_replication_origin_advance('pg_16420', '0/16E0948');
 pg_replication_origin_advance
-------------------------------


(1 row)
```

こうすることにより、subscriber 側のレプリケーション位置が強制的に変更され、競合が解消されます。

```
test=# select * from pg_replication_origin_status ;
 local_id | external_id | remote_lsn | local_lsn
----------+-------------+------------+-----------
       1 | pg_16420    | 0/16E0948  | 0/20D2D60
(1 row)
```

これ以後はレプリケーションが再開され publisher 側の変更が subscriber に反映されるようになりますが、subscriber 側には(100,'Hundred')の行が残ったままである一方、(100,'hundred')の行はレプリケートされていないので、両者のテーブルに不一致が生じた状態であることに注意して下さい。

4.1.8. REPLICA IDENTITY

レプリケーションされるテーブルで UPDATE または DELETE を実行するためには、行を識別するための

Copyright © 2017 SRA OSS, Inc. Japan All rights reserved.

情報（REPLICA IDENTITY）が必要となります。デフォルトではプライマリキーが使用されます。publisher 側で REPLICA IDENTITY を持っていないテーブルに対し UPDATE または DELETE を実行すると以下のようなエラーとなります。

```
（以下、publisher 側で）
test=# ALTER TABLE tbl DROP CONSTRAINT tbl_pkey;  -- 主キー制約を外す
ALTER TABLE
test=# DELETE FROM tbl;
ERROR:  cannot delete from table "tbl" because it does not have a replica
identity and publishes deletes
HINT:  To enable deleting from the table, set REPLICA IDENTITY using ALTER
TABLE
```

REPLICA IDENTITY にはプライマリキー以外に「NOT NULL である一意インデックス」、または「行の全内容(FULL)」を使用可能です。REPLICA IDENTITY をプライマリキー以外に変更するには ALTER TABLE 文を使用します。

```
（行の全内容を使用する場合）
test=# ALTER TABLE tbl REPLICA IDENTITY FULL ;
ALTER TABLE

（インデックスを使用する場合）
test=# CREATE UNIQUE INDEX idx_tbl ON tbl (id);
CREATE INDEX
test=# ALTER TABLE tbl ALTER COLUMN id SET NOT NULL;
ALTER TABLE
test=# ALTER TABLE tbl REPLICA IDENTITY USING INDEX idx_tbl ;
ALTER TABLE
```

4.1.9. 制限事項

ロジカルレプリケーションには以下のような制限事項があります。

- ***レプリケーションされる内容***

publisher から subscriber に転送できる変更内容は INSERT(COPY TO を含む),DELETE,UPDATE のいずれかです。TRUNCATE や DDL はレプリケートされません。

Copyright © 2017 SRA OSS, Inc. Japan All rights reserved.

◆ ***レプリケーション対象***

レプリケーションの対象となるのはテーブルのみです。シーケンスおよびラージオブジェクトはレプリケートされません。

また、レプリケーションの対象となるのは通常のテーブルのみです。ビュー、マテリアライズドビュー、パーティショニングの親テーブル、外部テーブルはレプリケートされません。パーティショニングされたテーブルの子テーブルに関してはレプリケート可能です。

4.2. 宣言的パーティショニング

PostgreSQL10 からパーティショニングの機能が正式にサポートされました。従来の PostgreSQL でもパーティショニングは可能でしたが、そのためには複数の既存機能を組み合わせる必要があり構築に手間がかかっていました。これに対し、PostgreSQL 10 ではパーティショニングが CREATE TABLE 構文を使って構築できるようになりました。この機能は宣言的パーティショニング（Declarative Partitioning）と呼ばれています。

4.2.1. 宣言的パーティショニングの概要

パーティショニングは巨大な親テーブルのレコードを複数の子テーブルに分割して格納する技術です。例えば、過去の全ての売上データを格納する親テーブルを、各年月の売上のみを格納する子テーブルに分割することができます。親テーブルへ検索を行った際には、必要な子テーブルのみを検索することで検索性能を高めることができます。また、レコードの挿入時には自動的に適切な子テーブルにレコードが振り分けられます。

従来の PostgreSQL では、パーティショニングはテーブル継承、CHECK 制約、トリガー関数といった複数の既存機能を組み合わせて実現していました。この方法は構築に手間がかかるだけではなく、トリガー関数を使用するため挿入が遅いという問題もありました。

PostgreSQL 10 ではパーティショニング機能が正式にサポートされ、実装に組み込まれました。これにより CREATE TABLE 文によるパーティショニングの構築が可能になり、パーティショニングに関する情報がシステムカタログにより管理されるようになりました。また、トリガー関数を用いないため、挿入の性能が良くなっています。なお、宣言的パーティショニングは、内部的には従来と同様にテーブル継承の仕組みを利用して実装されています。

宣言的パーティショニングでは、分割される親テーブルは partitioned table、分割後の子テーブルは partition と呼ばれます。partitioned table に挿入されたテーブルは、いずれかの partition に振り分けられます。振り分けは partition key の値に基づいて行われます。現在のところ、振り分けの方法はレンジパーティショニングとリストパーティショニングの 2 種類がサポートされています。

Copyright © 2017 SRA OSS, Inc. Japan All rights reserved.

4.2.2. パーティショニングテーブルの作成

以下ではパーティショニングを構築する方法を例とともに示します。

◆ ***レンジパーティショニング***

Partitioned table を作成するには、通常のテーブル定義を行う CREATE TABLE 文に PARTITION BY 句を付与し、パーティショニングの方法（RANGE または LIST)とパーティションキーを指定します。

以下では date 型のカラム logdata をパーティションキーとしてレンジパーティショニングを行う partitioned table として measurement を作成しています。

```
test=# CREATE TABLE measurement (
         logdate date NOT NULL, peaktemp int, unitsales int)
         PARTITION BY RANGE (logdate);
CREATE TABLE
```

レンジパーティショニングでは、パーティションキーに複数のカラムをカンマ区切りで指定することも可能です。また、パーティションキーに使用するカラムに NOT NULL 制約がついている必要はありませんが、実際にはパーティションキーアが NULL のレコードはレンジパーティショニングでは格納することはできないことに注意してください。

次に partition を追加します。CREATE TABLE 文の PARTITION OF 句で partitioned table を指定します。パーティションキーの値の範囲は FOR VALUES 句で指定します。以下では measurement に measurement_y2016m07 という名前の partition を追加し、logdate の値が 2016-07-01 から 2016-08-01 の間のレコードが格納されるよう指示しています。このように指定した範囲では 2016-07-01 は含まれますが 、2016-08-01 は含まれないことに注意して下さい。また、無限小・無限大を表すキーワードとして MINVALUE・MAXVALUE が使用可能です。

```
test=# CREATE TABLE measurement_y2016m07
         PARTITION OF measurement
         FOR VALUES FROM ('2016-07-01') TO ('2016-08-01');
CREATE TABLE
```

同様に partition をいくつか作成します。

```
test=# CREATE TABLE measurement_y2016m06
         PARTITION OF measurement
         FOR VALUES FROM ('2016-06-01') TO ('2016-07-01');
CREATE TABLE
```


Copyright © 2017 SRA OSS, Inc. Japan All rights reserved.

```
test=# CREATE TABLE measurement_y2016m08
         PARTITION OF measurement
         FOR VALUES FROM ('2016-08-01') TO ('2016-09-01');
CREATE TABLE
```

パーティショニングの種類とパーティションキー、partition の一覧は psql の\d+コマンドで確認できます。

```
test=# \d+ measurement
                                Table "public.measurement"
  Column   |  Type   | Collation | Nullable | Default | Storage | Stats target |
Description
-----------+---------+-----------+----------+---------+---------+--------------
+-------------
 logdate   | date    |           | not null |         | plain   |              |
 peaktemp  | integer |           |          |         | plain   |              |
 unitsales | integer |           |          |         | plain   |              |
Partition key: RANGE (logdate)
Partitions: measurement_y2016m06 FOR VALUES FROM ('2016-06-01') TO ('2016-07-01'),
            measurement_y2016m07 FOR VALUES FROM ('2016-07-01') TO ('2016-08-01'),
            measurement_y2016m08 FOR VALUES FROM ('2016-08-01') TO ('2016-09-01')
```

範囲が重なる partition は作成することができません。作成しようとすると以下のようなエラーとなります。

```
test=# CREATE TABLE measurement_y2016m07_08
         PARTITION OF measurement
         FOR VALUES FROM ('2016-07-15') TO ('2016-08-15');
ERROR:  partition "measurement_y2016m07_08" would overlap partition
"measurement_y2016m07"
```

いくつかの行を measurement テーブルに挿入してみます。

```
test=# INSERT INTO measurement VALUES ('2016-06-01',24,50),
         ('2016-07-15',32,100), ('2016-08-25',30,70);
INSERT 0 3
```

格納されたテーブル名と共に measurement テーブルの内容を表示してみます。各行のテーブル名を参照するにはシステム列 tableoid を使います。想定通りに logdate の値によって partition に分割されて格納されていることが確認できました。

Copyright © 2017 SRA OSS, Inc. Japan All rights reserved.

```
test=# SELECT tableoid::regclass, * FROM measurement;
      tableoid       |  logdate   | peaktemp | unitsales
---------------------+------------+----------+-----------
 measurement_y2016m07 | 2016-07-15 |       32 |       100
 measurement_y2016m06 | 2016-06-01 |       24 |        50
 measurement_y2016m08 | 2016-08-25 |       30 |        70
(3 rows)
```

なお、格納すべき partition が存在しないようなレコードを挿入した場合には、以下のようなエラーとなります。

```
test=# INSERT INTO measurement VALUES ('2016-12-25',13,5);
ERROR:  no partition of relation "measurement" found for row
DETAIL:  Partition key of the failing row contains (logdate) = (2016-12-25).
```

◆ *リストパーティショニング*

次にリストパーティショニングの場合について例を示します。以下では、text 型の name カラムの最初の一文字（小文字）をパーティションキーとしたリストパーティショニングを行う partitioned table を cities という名前で作成しています。

```
test=# CREATE TABLE cities (
         city_id bigserial not null, name text not null, population bigint)
         PARTITION BY LIST (left(lower(name), 1));
CREATE TABLE
```

このように、パーティションキーにはカラムを用いた式を指定することもできます。また、リストパーティショニングではパーティションキーの値が NULL であるレコードの格納が許されています。ただし、パーティションキーが NULL となるレコードを格納できる partition は 1 つだけです。

次に partition を追加します。CREATE TABLE 文の PARTITION OF 句で partitioned table を指定し、パーティションキーの値のリストを FOR VALUES 句で指定します。以下では cities に cities_ab という名前の partition を追加し、name の最初の文字が'a'または'b'であるレコードが格納されるよう指示しています。

```
test=# CREATE TABLE cities_ab
         PARTITION OF cities FOR VALUES IN ('a', 'b');
CREATE TABLE
```


Copyright © 2017 SRA OSS, Inc. Japan All rights reserved.

このリストに NULL を含む partition を作成することもできますが、そのような partition は 1 つしか存在できません。

```
test=# CREATE TABLE cities_null
         PARTITION OF cities FOR VALUES IN (NULL);
CREATE TABLE
```

同様にいくつか partition を作成します。

```
test=# CREATE TABLE cities_cd
         PARTITION OF cities FOR VALUES IN ('c', 'd');
CREATE TABLE
test=# CREATE TABLE cities_ef
         PARTITION OF cities FOR VALUES IN ('e', 'f');
CREATE TABLE
```

◆ *階層パーティション*

階層的にパーティションを構成することもできます。

以下の例では、リストで区分けされた cities_gh の中に、更に人口（population 列）のレンジで partition を作っています。

```
test=# CREATE TABLE cities_gh
         PARTITION OF cities FOR VALUES IN ('g', 'h')
         PARTITION BY RANGE (population);
CREATE TABLE
test=# CREATE TABLE cities_gh_10000_to_100000
         PARTITION OF cities_gh FOR VALUES FROM (10000) TO (100000);
CREATE TABLE
test=# CREATE TABLE cities_gh_100000_to_1000000
         PARTITION OF cities_gh FOR VALUES FROM (100000) TO (1000000);
CREATE TABLE
test=# CREATE TABLE cities_gh_1000000_to
         PARTITION OF cities_gh FOR VALUES FROM (1000000) TO (MAXVALUE);
CREATE TABLE
```

データを投入して、SELECT で確認すると、意図通りの partition に振り分けされていることが確認できました。

Copyright © 2017 SRA OSS, Inc. Japan All rights reserved.

```
test =# INSERT INTO cities VALUES (123, 'Budapest', 1740000),
          (124, 'Helsinki', 1177000), (125, 'Hakodate', 256000),
          (126, 'Chiba', 975000);
INSERT 0 4
test=# SELECT tableoid::regclass, * FROM cities;
          tableoid           | city_id |   name    | population
-----------------------------+---------+-----------+------------
 cities_ab                   |     123 | Budapest  |    1740000
 cities_cd                   |     126 | Chiba     |     975000
 cities_gh_100000_to_1000000 |     125 | Hakodate  |     256000
 cities_gh_1000000_to        |     124 | Helsinki  |    1177000
(4 rows)
```

4.2.3. パーティションのデタッチ・アタッチ

パーティションテーブルを削除することなく、パーティションを切り離すことができます。
以下例では cities から cities_cd を切り離しています。

```
test=# ALTER TABLE cities DETACH PARTITION cities_cd;
ALTER TABLE
test=# SELECT tableoid::regclass, * FROM cities;
          tableoid           | city_id |   name    | population
-----------------------------+---------+-----------+------------
 cities_ab                   |     123 | Budapest  |    1740000
 cities_gh_100000_to_1000000 |     125 | Hakodate  |     256000
 cities_gh_1000000_to        |     124 | Helsinki  |    1177000
(3 rows)
```

切り離されても cities_cd は単体のテーブルとして残ります。

```
test=# SELECT * FROM cities_cd ;
 city_id | name  | population
---------+-------+------------
     126 | Chiba |     975000
(1 row)
```

以下のようにして再びパーティションに付属させることができます。

Copyright © 2017 SRA OSS, Inc. Japan All rights reserved.

```
test=# ALTER TABLE cities ATTACH PARTITION cities_cd
         FOR VALUES IN ('c', 'd');
ALTER TABLE
test=# SELECT * FROM cities;
 city_id |   name    | population
---------+-----------+------------
     123 | Budapest  |    1740000
     126 | Chiba     |     975000
     125 | Hakodate  |     256000
     124 | Helsinki  |    1177000
(4 rows)
```

4.2.4. パーティションへのデータ投入速度の向上

宣言的パーティショニングは行データの投入にあたって、PostgreSQL 本体のネイティブコードが動作しますので、従来の PL/pgSQL で書かれたトリガを使った方式よりもずっと高速です。

投入速度を従来型のパーティションを作って比較してみます。

（従来型パーティションを作る）

```
test=# CREATE TABLE measurement2 (
          logdate date NOT NULL, peaktemp int, unitsales int);
CREATE TABLE
test=# CREATE TABLE measurement2_y2016m06
         (CHECK(logdate >= '2016-06-01' AND logdate < '2016-07-01'))
         INHERITS (measurement2);
CREATE TABLE
test=# CREATE TABLE measurement2_y2016m07
         (CHECK(logdate >= '2016-07-01' AND logdate < '2016-08-01'))
         INHERITS (measurement2);
CREATE TABLE
test=# CREATE TABLE measurement2_y2016m08
        (CHECK(logdate >= '2016-08-01' AND logdate < '2016-09-01'))
         INHERITS (measurement2);
CREATE TABLE
test=# CREATE OR REPLACE FUNCTION tf_ins_measurement2() RETURNS TRIGGER
         LANGUAGE plpgsql AS $$
```


Copyright © 2017 SRA OSS, Inc. Japan All rights reserved.

```
          BEGIN EXECUTE 'INSERT INTO measurement2_y' ||
            to_char(NEW.logdate, 'YYYY"m"MM') || ' VALUES (($1).*)'
            USING NEW ; RETURN NULL; END; $$;
CREATE FUNCTION
test=# CREATE TRIGGER trg_ins_measurement2 BEFORE INSERT ON measurement2
          FOR EACH ROW EXECUTE PROCEDURE tf_ins_measurement2();
CREATE TRIGGER

（10 万件データ投入の性能測定を行う）
test=# \timing on
Timing is on.
test=# INSERT INTO measurement SELECT
          '2016-06-01'::date + ('1 day'::interval * (g % 90)), 10, 10
          FROM generate_series(1, 100000) as g;
INSERT 0 100000
Time: 519.300 ms
test=# INSERT INTO measurement2 SELECT
          '2016-06-01'::date + ('1 day'::interval * (g % 90)), 10, 10
          FROM generate_series(1, 100000) as g;
INSERT 0 0
Time: 9170.039 ms (00:09.170)
```

本結果では 10 倍以上の違いが出ています。従来型パーティションの構成方法には未だいくらか工夫の余地はありますが、それでも宣言型パーティショニングよりも速くなることはないでしょう。

4.2.5. 制約事項

宣言的パーティショニングには以下の制限事項があります。

- ***インデックス***

Partitioned table はインデックスをサポートしていません。インデックスは個々の partition に作成する必要があります。同様に、partitioned table に関してはプライマリキー、一意制約、外部参照制約、ON CONFLICT 句もサポートされていません。

- ***パーティションキーの変更***

パーティションキーの値を現在の partition の条件に合わないように変更するような UPDATE は制約違反

Copyright © 2017 SRA OSS, Inc. Japan All rights reserved.

となり失敗します。自動的にレコードが他の partition に移動することはありません。

- ***外部テーブルへの振り分け***

  partitioned table に挿入されたレコードは外部テーブルの partition には振り分けられません。

- ***行レベルトリガー***

  行レベルトリガーは partitioned table には作成できません。各 partition に作成する必要があります。

4.3. パラレルクエリの強化

PostgreSQL9.6 からパラレルクエリが導入されて、一つの問い合わせに複数プロセス、すなわち複数 CPU コアを並列に使用することができるようになりました。PostgreSQL10 では、並列処理ができる範囲が大幅に拡大されています。

以下表に PostgreSQL9.6 と 10 のパラレル対応している処理内容を示します。

パラレル対応範囲	9.6	10
シーケンシャルスキャン	○	○
インデックススキャン （Index Only Scan を含む）	×	○
ビットマップヒープスキャン	×	○
ビットマップインデックススキャン	×	×
ソート	○	○
入れ子ループ（NestedLoop）結合	○	○
マージ結合	×	○
ハッシュ結合	×	○
サブクエリ	×	○

本節では以下で実際の動作を見ていきます。

4.3.1. 追加されたプラン

パラレル動作の確認用にパーティション検証で使ったテーブルに 100 万件のデータを追加し、インデックスも付加します。

Copyright © 2017 SRA OSS, Inc. Japan All rights reserved.

```
test=# INSERT INTO measurement SELECT '2016-06-01'::date +
         ('1 day'::interval * (g % 90)), ceil(random()*1000) , 10
         FROM generate_series(1, 1000000) as g;
INSERT 0 1000000
test=# CREATE INDEX ON measurement_y2016m06 (peaktemp);
CREATE INDEX
test=# CREATE INDEX ON measurement_y2016m07 (peaktemp);
CREATE INDEX
test=# CREATE INDEX ON measurement_y2016m08 (peaktemp);
CREATE INDEX
```

本節の例では、パラレル実行時の例を示すため、SET force_parallel_mode TO on 設定を加えています。実際にはプランナが並列処理しないことを選ぶ場合もあります。

- ***Parallel Bitmap Heap Scan / Parallel Index Scan / Parallel Index Only Scan***

ビットマップヒープスキャン、インデックススキャン、インデックスオンリースキャンが並列処理に対応しました。以下にプランが現れる例を示します。

実のところ本サンプルデータですと「Parallel Seq Scan」が使われます。ここではシーケンシャルスキャンを嫌うようにする「enable_seqscan TO off」を使って、これらの新プランを出現させています。件数が更に多かったり、設定パラメータ random_page_cost や cpu_index_tuple_cost を小さく調整しても同様の結果になります。

```
db1=# SET enable_seqscan TO off;
 ★ Parallel Index Only Scan
db1=# explain SELECT count(peaktemp) FROM measurement WHERE peaktemp < 1234;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Finalize Aggregate  (cost=23541.84..23541.85 rows=1 width=8)
   ->  Gather  (cost=23541.53..23541.84 rows=3 width=8)
         Workers Planned: 3
         ->  Partial Aggregate  (cost=22541.53..22541.54 rows=1 width=8)
               ->  Append  (cost=0.42..21735.07 rows=322581 width=4)
                     ->  Parallel Index Only Scan using
                           measurement_y2016m07_peaktemp_idx on
                           measurement_y2016m07
                           (cost=0.42..7486.83 rows=111110 width=4)
                           Index Cond: (peaktemp < 1234)
```


Copyright © 2017 SRA OSS, Inc. Japan All rights reserved.

```
                      ->  Parallel Index Only Scan using
                            measurement_y2016m06_peaktemp_idx on
                            measurement_y2016m06
                            (cost=0.42..7243.76 rows=107529 width=4)
                            Index Cond: (peaktemp < 1234)
                      ->  Parallel Index Only Scan using
                            measurement_y2016m08_peaktemp_idx on
                            measurement_y2016m08
                            (cost=0.42..7004.48 rows=103942 width=4)
                            Index Cond: (peaktemp < 1234)
(11 rows)

db1=# SET enable_indexonlyscan TO off;
```

★ **Parallel Bitmap Heap Scan**

```
db1=# explain SELECT count(peaktemp) FROM measurement WHERE peaktemp < 1234;
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Finalize Aggregate  (cost=33988.91..33988.92 rows=1 width=8)
   ->  Gather  (cost=33988.80..33988.91 rows=1 width=8)
         Workers Planned: 1
         ->  Partial Aggregate  (cost=32988.80..32988.81 rows=1 width=8)
               ->  Append  (cost=6461.84..31518.21 rows=588235 width=4)
                     ->  Parallel Bitmap Heap Scan on measurement_y2016m07
                          (cost=6461.84..10856.49 rows=202612 width=4)
                           Recheck Cond: (peaktemp < 1234)
                           ->  Bitmap Index Scan on
                                 measurement_y2016m07_peaktemp_idx
                                 (cost=0.00..6375.73 rows=344441 width=0)
                                 Index Cond: (peaktemp < 1234)
                     ->  Parallel Bitmap Heap Scan on measurement_y2016m06
                          (cost=6251.81..10504.84 rows=196082 width=4)
                           Recheck Cond: (peaktemp < 1234)
                           ->  Bitmap Index Scan on
                                 measurement_y2016m06_peaktemp_idx
                                (cost=0.00..6168.47 rows=333340 width=0)
```


Copyright © 2017 SRA OSS, Inc. Japan All rights reserved.

```
                                  Index Cond: (peaktemp < 1234)
                        ->  Parallel Bitmap Heap Scan on measurement_y2016m08
                              (cost=6045.62..10156.88 rows=189541 width=4)
                              Recheck Cond: (peaktemp < 1234)
                              ->  Bitmap Index Scan on
                                    measurement_y2016m08_peaktemp_idx
                                    (cost=0.00..5965.06 rows=322219 width=0)
                                    Index Cond: (peaktemp < 1234)
(17 rows)


db1=# SET enable_bitmapscan TO off;
```

★ **Parallel Index Scan**

```
db1=# explain SELECT count(peaktemp) FROM measurement WHERE peaktemp < 1234;
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Finalize Aggregate  (cost=48485.90..48485.91 rows=1 width=8)
   ->  Gather  (cost=48485.79..48485.90 rows=1 width=8)
         Workers Planned: 1
         ->  Partial Aggregate  (cost=47485.79..47485.80 rows=1 width=8)
               ->  Append  (cost=0.42..46015.20 rows=588235 width=4)
                     ->  Parallel Index Scan using
                          measurement_y2016m07_peaktemp_idx on
                          measurement_y2016m07
                          (cost=0.42..15849.66 rows=202612 width=4)
                           Index Cond: (peaktemp < 1234)
                     ->  Parallel Index Scan using
                          measurement_y2016m06_peaktemp_idx on
                          measurement_y2016m06
                          (cost=0.42..15337.06 rows=196082 width=4)
                           Index Cond: (peaktemp < 1234)
                     ->  Parallel Index Scan using
                          measurement_y2016m08_peaktemp_idx on
                          measurement_y2016m08
```


Copyright © 2017 SRA OSS, Inc. Japan All rights reserved.

```
                          (cost=0.42..14828.47 rows=189541 width=4)
                           Index Cond: (peaktemp < 1234)
(11 rows)
```

- ***Parallel Merge Join***

  パラレル処理を含んだマージ結合ができるようになりました。

  以下に例を示します。

```
★マージ結合が選ばれやすいデータを作ります
db1=# CREATE TABLE t_info (id int primary key, c1 text, c2 text);
db1=# CREATE TABLE t_info_detail (id int primary key, c3 text, c4 text);
db1=# INSERT INTO t_info SELECT g, md5(g::text), md5(g::text)
        FROM generate_series(1, 100000) as g;
db1=# INSERT INTO t_info_detail SELECT g, md5(g::text), md5(g::text)
        FROM generate_series(1, 100000) as g;

★「Parallel Merge Join」というプランがあるわけではありません。
  以下の形でパラレル処理ができることをMerge Joinのパラレル対応と呼んでいます。
db1=# explain SELECT count(*) FROM t_info JOIN t_info_detail USING (id);
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Finalize Aggregate  (cost=6500.39..6500.40 rows=1 width=8)
   ->  Gather  (cost=6500.17..6500.38 rows=2 width=8)
         Workers Planned: 2
         ->  Partial Aggregate  (cost=5500.17..5500.18 rows=1 width=8)
               ->  Merge Join  (cost=0.69..5396.00 rows=41667 width=0)
                     Merge Cond: (t_info_detail.id = t_info.id)
                     ->  Parallel Index Only Scan using t_info_detail_pkey on
                         t_info_detail  (cost=0.29..2020.96 rows=41667 width=4)
                     ->  Index Only Scan using t_info_pkey on
                         t_info  (cost=0.29..2604.29 rows=100000 width=4)
(8 rows)
```


Copyright © 2017 SRA OSS, Inc. Japan All rights reserved.

◆ *Gather Merge*

Gather Merge と Partial GroupAggregate は Group Aggregate を並列実行するプランタイプです。以下例のように GroupAggregate が使われるような問い合わせに適用されます。

```
db1=# SET enable_hashagg TO off;
SET
db1=# explain SELECT peaktemp, count(*) FROM measurement
        GROUP BY peaktemp ORDER BY peaktemp ;
                                          QUERY PLAN
------------------------------------------------------------------------------
 Finalize GroupAggregate  (cost=73164.22..76993.28 rows=1000 width=12)
   Group Key: measurement_y2016m07.peaktemp
   ->  Gather Merge  (cost=73164.22..76973.28 rows=2000 width=12)
         Workers Planned: 2
         ->  Partial GroupAggregate
             (cost=72164.19..75742.41 rows=1000 width=12)
               Group Key: measurement_y2016m07.peaktemp
               ->  Sort  (cost=72164.19..73353.60 rows=475762 width=4)
                     Sort Key: measurement_y2016m07.peaktemp
                     ->  Append  (cost=0.00..20793.62 rows=475762 width=4)
                           ->  Parallel Seq Scan on measurement_y2016m07
                               (cost=0.00..3888.12 rows=202612 width=4)
                           ->  Parallel Seq Scan on measurement_y2016m06
                               (cost=0.00..8595.92 rows=138892 width=4)
                           ->  Parallel Seq Scan on measurement_y2016m08
                               (cost=0.00..8309.58 rows=134258 width=4)
(12 rows)
```

◆ *サブクエリのパラレル実行*

以下例のようにプランタイプ SubPlan として実行されるサブクエリが並列実行可能になりました。

```
db1=# explain SELECT count(*) FROM measurement_y2016m07
  WHERE peaktemp > ALL (SELECT peaktemp FROM measurement_y2016m06);
                              QUERY PLAN
------------------------------------------------------------------------------
```


Copyright © 2017 SRA OSS, Inc. Japan All rights reserved.

```
 Finalize Aggregate  (cost=617581863.74..617581863.75 rows=1 width=8)
   ->  Gather  (cost=617581863.32..617581863.73 rows=4 width=8)
         Workers Planned: 4
         ->  Partial Aggregate
             (cost=617580863.32..617580863.33 rows=1 width=8)
               ->  Parallel Index Only Scan using
                   measurement_y2016m07_peaktemp_idx on measurement_y2016m07
                   (cost=0.42..617580755.69 rows=43055 width=0)
                     Filter: (SubPlan 1)
                     SubPlan 1
                       ->  Materialize
                           (cost=0.00..13510.10 rows=333340 width=4)
                             ->  Seq Scan on measurement_y2016m06
                                 (cost=0.00..10540.40 rows=333340 width=4)
(9 rows)
```

4.3.2. パラレルクエリのモニタリング

pg_stat_activity ビューでパラレルクエリのワーカプロセスが見えるようになりました。

「Parallel Seq Scan」が使われる以下の SQL を実行します。Worker Launched: 2 と出力されていて、2 つのワーカプロセスが起動したことがわかります。

```
db1=# explain analyze SELECT avg(peaktemp + unitsales) FROM measurement;
                                                        QUERY PLAN
-----------------------------------------------------------------------------------
 Finalize Aggregate  (cost=24172.65..24172.66 rows=1 width=32) (actual
time=523.761..523.761 rows=1 loops=1)
   ->  Gather  (cost=24172.43..24172.64 rows=2 width=32) (actual
time=523.274..523.720 rows=3 loops=1)
         Workers Planned: 2
         Workers Launched: 2
         ->  Partial Aggregate  (cost=23172.43..23172.44 rows=1 width=32) (ac
：以下略
```

その SQL 実行の最中に別の接続から pg_stat_activity ビューを参照すると、一つの問い合わせに対して複

Copyright © 2017 SRA OSS, Inc. Japan All rights reserved.

数のプロセスが働いている様子を観測することができます。パラレルクエリを実行する子プロセスは backend_type が background worker であるものとして表示されます。

```
db1=# SELECT pid, query_start, query, backend_type FROM pg_stat_activity
        WHERE query ~ '^explain';
-[ RECORD 1 ]--+-------------------------------------------------------------
datid          | 16544
query_start    | 2017-11-14 16:00:45.627334+09
query          | explain analyze SELECT avg(peaktemp + unitsales) FROM measu
backend_type   | client backend
-[ RECORD 2 ]--+-------------------------------------------------------------
pid            | 31214
query_start    | 2017-11-14 16:00:45.63595+09
query          | explain analyze SELECT avg(peaktemp + unitsales) FROM measu
backend_type   | background worker
-[ RECORD 3 ]--+-------------------------------------------------------------
pid            | 31215
query_start    | 2017-11-14 16:00:45.63965+09
query          | explain analyze SELECT avg(peaktemp + unitsales) FROM measu
backend_type   | background worker
                                                          ★出力は右端を省略
```

この例では、PID=31214、PID=31215 のプロセスが、PID=16544 から起動されたものですが、pg_stats_activity にはプロセスの親子関係を直接示す列はありません。また、query_start が親プロセスとは異なり、ワーカが起動した時刻になっている点にも注意が必要です。

4.4. その他の新機能

PostgreSQL 10 では上述のもの以外にも多くの新機能があります。以下ではその一部を紹介します。

4.4.1. Hash インデックスの拡張

長らく改修されないままであったハッシュインデックスに久しぶりに大きな拡張が加わりました。

Copyright © 2017 SRA OSS, Inc. Japan All rights reserved.

◆ *WAL書き出しに対応*

これまで Hash インデックスは WAL 書き出しに対応していませんでした。そのため、PostgreSQL が非正常に終了したときには壊れているかもしれず、次に起動したとき REINDEX を行うことが推奨されました。また、ストリーミングレプリケーションで変更がスタンバイサーバに伝播しません。そのため、トラブルの元になるのでできるだけ使わないのが最善、という残念な状況でした。

バージョン 10 から WAL 書き出しをするようになりました。上記の制限事項がなくなり、純粋に Hash インデックスの特徴に基づいて使用するかどうか判断することができます。Hash インデックスは一致条件で使われ、長い文字列などの 1 件データが大きい列に適しています。

◆ *性能改善*

バージョン 10 では Hash インデックスにいくつかの性能改善が適用されています。同時実行でデータ追加が行われた場合の排他競合が軽減されました。また、メタ情報のキャッシュを持つことでの検索を高速化、データ削除処理の効率化が適用されています。

（冒頭に述べたローカルの仮想環境とは別の）36 CPU / SSD ストレージの仮想マシンで単一および並列のデータ挿入の時間計測を行いました。

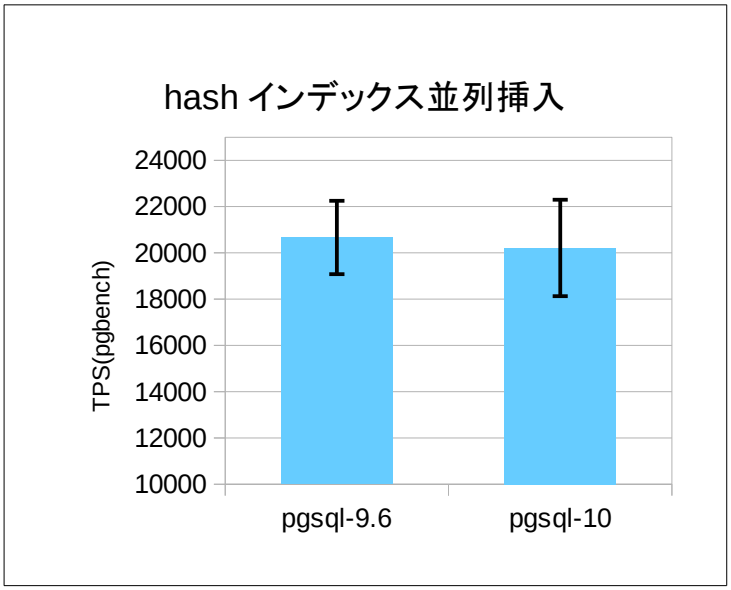
《テスト手順》

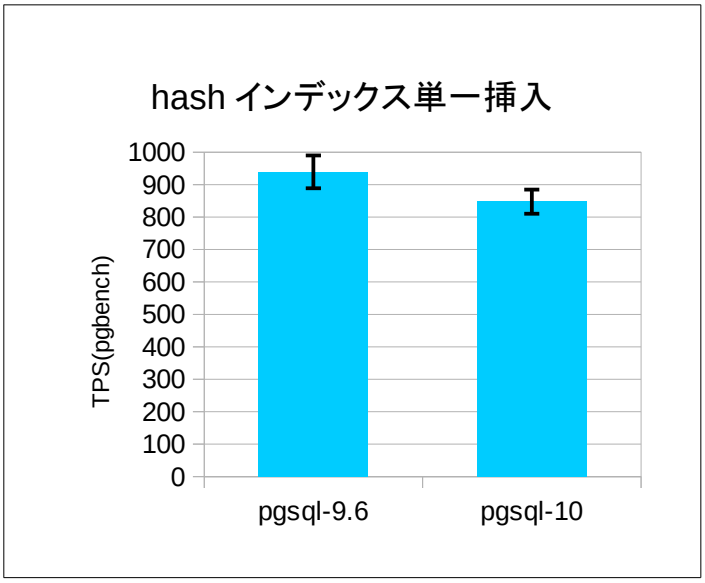
```
$ psql db1 <<'EOS'
CREATE TABLE t_hashtest (code text);
CREATE INDEX ON t_hashtest (code);
EOS
$ cat hashtest.sql <<'EOS'
\set r random(1, 100000)
INSERT INTO t_hashtest VALUES ('ABC-123-def-GHI-456-000-' || :r );
EOS
$ pgbench -f hashtest.sql -c 36 -j 36 -n -t 1000 db1    # 《繰り返し実行》
```

単一クライアントからの挿入では 9.6 バージョンが 10%ほど高速でした。これは WAL を出力しなかったものが出力するようになった結果であり、仕方のないところです。といえます。なお、ハッシュインデックスが保持するのはハッシュ値であるため、対象のデータ長は WAL 出力の負荷の大小に影響しないと考えられます。

並列挿入では同程度の性能となりました。WAL 出力が増えたマイナスを、並列実行処理の改善が補った結果と考えられます。つまり、より CPU 数の多いサーバでより並列実行数が多いであれば、むしろ WAL 書き出しをしない 9.6 以前のバージョンよりも高速になることが予想されます。

Copyright © 2017 SRA OSS, Inc. Japan All rights reserved.





4.4.2. 同期レプリケーションの拡張

同期ストリーミングレプリケーションを構成する場合に「複数あるスタンバイサーバの内、いずれかN台に伝播したなら同期完了とみなす」という設定が可能になりました。

プライマリ postgresql.conf の synchronous_standby_names に以下のように指定します。

（新しい指定方法）

```
synchronous_standby_names = 'ANY 2 (s1 s2 s3)' ## いずれか 2 つが同期スタンバイ
```

9.6 バージョンから対応していた以下の書き方との違いを見ていきます。

（9.6 で登場した指定方法）

```
synchronous_standby_names = '2 (s1 s2 s3)'      ## 3 つのうち 2 つが同期スタンバイ
```

以下の要領でスタンバイを 3 つ作ります。PostgreSQL10 ではデフォルトで postgresql.conf ストリーミングレプリケーションのプライマリとして使える設定が適用されています。

（スタンバイサーバ用のベースバックアップを作成）

```
$ pg_basebackup -D ${PGDATA}_s1 -Xs -R
$ pg_basebackup -D ${PGDATA}_s2 -Xs -R
$ pg_basebackup -D ${PGDATA}_s3 -Xs -R
```

（設定を編集して primary_conninfo に application_name=s1 指定を追加します）

```
$ vi ${PGDATA}_s*/recovery.conf
```


Copyright © 2017 SRA OSS, Inc. Japan All rights reserved.

```
（ポートを分けてスタンバイサーバを起動します）
$ pg_ctl start -D ${PGDATA}_s1 -o '-p 5401'
$ pg_ctl start -D ${PGDATA}_s2 -o '-p 5402'
$ pg_ctl start -D ${PGDATA}_s3 -o '-p 5403'
```

このように起動したあと、s1 のスタンバイサーバの startup プロセスを STOP シグナルで止めてみます。これによりスタンバイサーバ s1 は変更 WAL データを受領しないままになります。

```
$ ps -ef | grep $(head -1 ${PGDATA}_s1/postmaster.pid) | grep receiver
postgres 12655 12289  0 15:01 ?        00:00:00 postgres: wal receiver
process   streaming 0/1F045B28
$ kill -STOP 12655
```

このときプライマリの synchronous_standby_names が '2 (s1 s2 s3)' である場合には、プライマリで更新 SQL を実行すると応答がなくなります。先頭にあるスタンバイ s1 の応答を待ち続けてしまうからです。

これに対してプライマリの synchronous_standby_names が 'SNY 2 (s1 s2 s3)' であれは、スタンバイ s2 と s3 が応答するので、プライマリで更新 SQL を実行すると応答が戻ります。

4.4.3. pg_stat_activity の拡張

PostgreSQL10 では pg_stat_activity ビューで参照できる内容が増えました。

以下のように pgbench をバックグラウンドで実行します。

```
（pgbench 実行 - コンソール出力は省略しています）
$ pgbench -i test
$ pgbench -c 4 -T 120 -n test &> /dev/null &
```

その状態で pg_stat_activity ビューを参照します。

```
test=# \x on
Expanded display is on.
test=# SELECT pid, xact_start, state_change, wait_event_type, wait_event,
query, backend_type FROM pg_stat_activity ;
-[ RECORD 1 ]---+-----------------------------------------------------------
pid             | 14854
xact_start      |
```


Copyright © 2017 SRA OSS, Inc. Japan All rights reserved.

```
state_change    |
wait_event_type | Activity
wait_event      | AutoVacuumMain
query           |
backend_type    | autovacuum launcher
-[ RECORD 2 ]---+-----------------------------------------------------------
pid             | 14856
xact_start      |
state_change    |
wait_event_type | Activity
wait_event      | LogicalLauncherMain
query           |
backend_type    | background worker
-[ RECORD 3 ]---+-----------------------------------------------------------
pid             | 14978
xact_start      |
state_change    | 2017-10-31 17:59:52.589072+09
wait_event_type | Client
wait_event      | ClientRead
query           | END;
backend_type    | client backend
-[ RECORD 4 ]---+-----------------------------------------------------------
pid             | 14979
xact_start      | 2017-10-31 17:59:52.581243+09
state_change    | 2017-10-31 17:59:52.584589+09
wait_event_type | Lock
wait_event      | transactionid
query           | UPDATE pgbench_tellers SET tbalance = tbalance + -1429
WHERE tid = 10;
backend_type    | client backend
-[ RECORD 5 ]---+-----------------------------------------------------------
pid             | 14980
xact_start      | 2017-10-31 17:59:52.564402+09
state_change    | 2017-10-31 17:59:52.566337+09
wait_event_type | LWLock
```


Copyright © 2017 SRA OSS, Inc. Japan All rights reserved.

```
wait_event       | buffer_content
query            | UPDATE pgbench_branches SET bbalance = bbalance + -1082
WHERE bid = 1;
backend_type     | client backend
-[ RECORD 6 ]----+-----------------------------------------------------------
pid              | 14981
xact_start       | 2017-10-31 17:59:52.548762+09
state_change     | 2017-10-31 17:59:52.5502+09
wait_event_type  |
wait_event       |
query            | UPDATE pgbench_branches SET bbalance = bbalance + 2738
WHERE bid = 1;
backend_type     | client backend
: （後略）
```

◆ ***待機イベントの追加***

これまでロックをまっているかどうかだけを示す waiting 列がありましたが、代わりに wait_event_type と wait_event が用意されました。これらが空欄（NULL）なら待っていない状態となります。

これまで観測できなかった LWLock（ライトウェイトロック）や、クラアントの読み込み待ちを参照できるようになります。

◆ ***表示されるプロセスの追加***

クライアント接続に対応したプロセス以外の autovacuum launcher のような PostgreSQL サービスの所定の仕事をするプロセスも表示されるようになりました。プロセスの種類は backend_type 列で識別されます。

4.4.4. 拡張統計情報（*CREATE STATISTICS*）

PostgreSQL10 では拡張統計情報と呼ばれる新たな種類のデータベースオブジェクトが追加されました 。CREATE STATISTICS、DROP STATISTICS 文で作成、削除します。これは、テーブル内の複数の列にまたがるプランナむけ統計情報を保持します。

現在サポートされている拡張統計情報の種類は、値の種類数（ndistinct）と関数従属性（dependencies）です。 これらのうち、値の種類数は意味が明確ですので、ここでは関数従属性について実際の動作を確認していきます。

（サンプルテーブルとデータを用意します）

```
test=# CREATE TABLE product (id int, subid int, p_number int,
```


Copyright © 2017 SRA OSS, Inc. Japan All rights reserved.

```
         p_code text, p_name text);
CREATE TABLE
test=# INSERT INTO product SELECT g, i, 100000 + g + 10,
         'P' || ( 100000 + g + 10  ) || 'ZZ', '.....'
         FROM generate_series(1, 1000) as g, generate_series(1, 100) as i;
INSERT 0 100000
test=# SELECT * FROM product ;
 id | subid | p_number |  p_code   | p_name
----+-------+----------+-----------+--------
  1 |     1 |   100010 | P100010ZZ | .....
  1 |     2 |   100010 | P100010ZZ | .....
  1 |     3 |   100010 | P100010ZZ | .....
  1 |     4 |   100010 | P100010ZZ | .....
  1 |     5 |   100010 | P100010ZZ | .....
  :  （後略）
```

上記の product テーブルの p_number 列の値は（id 列 × 10 + 100000）式の結果に等しいですし、p_code 列は p_number 列の先頭に「P」を末尾に「ZZ」を付けた文字列にすぎません。id 列を決めれば 、p_number 、p_code 列の値は一意にきまります。このような関係を関数従属性がある、といいます。

このようなデータに対して以下のような問い合わせをすると、プランナが列の値が連動していることを知らないために、結果件数予測がしばしば大きく狂います。

```
test=# EXPLAIN ANALYZE SELECT * FROM product
         WHERE id = 1 AND p_number = 100010 AND p_code = 'P100010ZZ';
                                        QUERY PLAN
-------------------------------------------------------------------------------
---------------------------
 Seq Scan on product  (cost=0.00..2486.00 rows=1 width=28) (actual
time=0.028..37.638 rows=100 loops=1)
   Filter: ((id = 1) AND (p_number = 100010) AND (p_code =
'P100010ZZ'::text))
   Rows Removed by Filter: 99900
 Planning time: 0.207 ms
 Execution time: 37.793 ms
(5 rows)              ★列の値が独立している想定をするため、件数予測が大幅に異なる
```


Copyright © 2017 SRA OSS, Inc. Japan All rights reserved.

そこで拡張統計情報を作って対応します。

```
test=# CREATE STATISTICS s_product ON id, p_number, p_code FROM product;
CREATE STATISTICS    ★product テーブルの 3 列に拡張統計を作ります
test=# ANALYZE ;
ANALYZE              ★拡張統計情報も ANALYZE 処理で更新されます

test=# EXPLAIN ANALYZE SELECT * FROM product
         WHERE id = 1 AND p_number = 100010 AND p_code = 'P100010ZZ';
                                        QUERY PLAN
------------------------------------------------------------------------------
----------------------------
 Seq Scan on product  (cost=0.00..2486.00 rows=98 width=28) (actual
time=0.021..26.339 rows=100 loops=1)
   Filter: ((id = 1) AND (p_number = 100010) AND (p_code =
'P100010ZZ'::text))
   Rows Removed by Filter: 99900
 Planning time: 0.494 ms
 Execution time: 26.413 ms
```

拡張統計情報を設定すると、ANALYZE で列の間に関数従属性があるかどうかを検出して保持します。これにより、関数従属性のある列を使った問い合わせでも比較的正確な件数予測が実現できています。拡張統計情報の内容はシステムテーブル pg_statistic_ext に格納されます。

```
test=# SELECT * FROM pg_statistic_ext WHERE stxname = 's_product';
-[ RECORD 1 ]---
+------------------------------------------------------------------------------
------------------------------------------------------------------------------
--------------------------------------
stxrelid        | 16976
stxname         | s_product
stxnamespace    | 2200
stxowner        | 10
stxkeys         | 1 3 4
```


Copyright © 2017 SRA OSS, Inc. Japan All rights reserved.

```
stxkind         | {d,f}
stxndistinct    | {"1, 3": 1000, "1, 4": 1000, "3, 4": 1000, "1, 3, 4":
1000}
stxdependencies | {"1 => 3": 1.000000, "1 => 4": 1.000000, "3 => 1":
1.000000, "3 => 4": 1.000000, "4 => 1": 1.000000, "4 => 3": 1.000000, "1, 3
=> 4": 1.000000, "1, 4 => 3": 1.000000, "3, 4 => 1": 1.000000}
```

4.4.5. トリガの遷移表

AFTER トリガにて、UPDATE、INSERT、DELETE 等の更新コマンドの前後で対象テーブルのデータがどのように変わったかをトリガ関数内で参照できるようになりました。

以下に使用例を示します。

```
test=# CREATE TABLE score (test text, score real);
CREATE TABLE
test=# CREATE OR REPLACE FUNCTION show_transition()
         RETURNS trigger LANGUAGE plpgsql AS $$
  DECLARE
    newdata json;
    olddata json;
  BEGIN
    SELECT json_agg(newtab.*) INTO newdata FROM newtab;
    SELECT json_agg(oldtab.*) INTO olddata FROM oldtab;
    RAISE NOTICE 'old: %', olddata;
    RAISE NOTICE 'new: %', newdata;
    RETURN NEW;
  END;  $$;
CREATE FUNCTION            ★newtab、oldtab をテーブルのように参照しています
test=# CREATE TRIGGER trg_score_update AFTER UPDATE ON score
         REFERENCING NEW TABLE AS newtab OLD TABLE AS oldtab
         FOR EACH ROW EXECUTE PROCEDURE show_transition();
CREATE TRIGGER             ★上記の赤字部分が新構文です
test=# INSERT INTO score VALUES
         ('test1', 0.92), ('test2', 0.85), ('test3', 0.77);
```


Copyright © 2017 SRA OSS, Inc. Japan All rights reserved.

```
INSERT 0 3
test=# UPDATE score SET score = 1.0 WHERE score > 0.80;
NOTICE:  old: [{"test":"test1","score":0.92},
 {"test":"test2","score":0.85}]
NOTICE:  new: [{"test":"test1","score":1},
 {"test":"test2","score":1}]      ★変更の無かった test='test3' の行は含まれない
NOTICE:  old: [{"test":"test1","score":0.92},
 {"test":"test2","score":0.85}]
NOTICE:  new: [{"test":"test1","score":1},
 {"test":"test2","score":1}]
UPDATE 2                          ★繰り返し２回出力されているのは行単位トリガだから
```

トリガ関数の中で UPDATE 実行前・実行後のテーブル全体の内容が参照できていることがわかります。なお、集約関数 json_agg を使っているのは複数列×複数行のデータを 1 つの変数に詰め込んで、サンプルトリガ関数を短く済ますためだけの意図です。

この構文は文単位（FOR EACH STATEMENT）のトリガにも適用できます。

これまで、文単位トリガでは変更前後のテーブルのデータにアクセスすることができず、行単位トリガであっても該当行以外の行について情報を得ることができませんでしたが、本機能でこれらを実現する手段が提供されました。

4.4.6. libpq 複数接続先指定

PostgreSQL のクライアントライブラリ libpq で接続先に複数のホストを指定できるようになりました。

「4.4.2 同期レプリケーションの拡張」の節で作ったスタンバイサーバを起動した状態で、以下のようにホスト名とポート番号をカンマ区切りの並びで指定して、psql で接続してみます。

```
$ export PGHOST=localhost,localhost,localhost,localhost
$ export PGPORT=5401,5402,5403,15432
                          ★ここでは環境変数で指定しているがオプションでも指定可能
$ psql -d test -c 'SELECT inet_server_port()'
 inet_server_port
------------------
             5401
(1 row)                   ★利用可能な先頭サーバに接続される
```


Copyright © 2017 SRA OSS, Inc. Japan All rights reserved.

```
$ pg_ctl stop -D ${PGDATA}_s1
waiting for server to shut down.... done
server stopped
$ psql -d test -c 'SELECT inet_server_port()'
 inet_server_port
------------------
            5402
(1 row)                          ★先頭がダウンしているので 2 番目サーバに接続された

$ psql 'dbname=test target_session_attrs=read-write' \
        -c 'SELECT inet_server_port()'
 inet_server_port
------------------
           15432
(1 row)                          ★書き込み可能サーバに限定するオプション
```

複数あるサーバに先頭から順に接続を試みて、接続可能なサーバに接続している動作が確認できます。また、新たに導入された target_session_attrs=read-write オプションを指定すると書き込み可能なサーバに接続先を限定できます。この場合、ホットスタンバイサーバはスキップされます。

5. 免責事項

本ドキュメントは SRA OSS, Inc. 日本支社により作成されました。しかし、SRA OSS, Inc. 日本支社は本ドキュメントにおいて正確性、有用性、その他いかなる保証をするものではありません。本ドキュメントを利用する場合、利用者の責任において行なって頂くものとなります。

Copyright © 2017 SRA OSS, Inc. Japan All rights reserved.