PostgreSQL 11 検証レポート

1.1 版
2018 年 10 月 18 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 © 2018 SRA OSS, Inc. Japan All rights reserved.

1. はじめに

本文書は PostgreSQL 11 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 11 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。
2018 年 5 月 24 日にリリースされた PostgreSQL 11beta1 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 11 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。

- JIT コンパイルの追加
- パーティショニングの様々な改善
- 並列実行の様々な改善
- プロシージャ内でのトランザクション制御をサポート
- 認証で SCRAM チャンネルバインドをサポート
- ALTER TABLE .. ADD COLUMN の性能改善
- ウィンドウ関数の拡張

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 11 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-11.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 11 （ベータ版を含む）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/

3.2. 検証環境

検証環境として、HP ProLiant MicroServer 上の CentOS 7.x / x86_64 を使用しました。これは小型サーバ

Copyright © 2018 SRA OSS, Inc. Japan All rights reserved.

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。/usr/local/pgsql ディレクトリを postgres ユーザで読み書き可能なディレクトリとして用意したうえで、postgres ユーザにて実行しました。

```
$ cd postgresql-11beta1
$ ./configure --prefix=/usr/local/pgsql/11.0 --enable-cassert \
  --enable-debug --with-libxml --with-openssl --with-perl --with-python
$ make world
$ 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 > 11.0.env <<'EOF'
PGHOME=/usr/local/pgsql/11.0
export PATH=$PGHOME/bin:$PATH
export LD_LIBRARY_PATH=$PGHOME/lib:$LD_LIBRARY_PATH
export PGDATA=/usr/local/pgsql/data11.0
export PGPORT=5432
EOF
$ . 11.0.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 start
```


Copyright © 2018 SRA OSS, Inc. Japan All rights reserved.

3.3.1. JIT コンパイルを使うためのビルド

新たに追加された JIT コンパイル機能を使うためには以下のビルド手順が必要です。

まず、LLVM コンパイラ基盤が必要です。RHEL/CentOS 7.x の LLVM はバージョン 3.4.x とが古いため 、Software Collections から llvm 4.0.x を導入しました。

```
# yum install centos-release-scl
# yum install llvm-toolset-7 llvm-toolset-7-llvm-devel
```

configure オプションに--with-llvm を、また、LLVM_CONFIG 変数に llvm-config コマンドのパスを指定します。さらに C++や CLANG も使いますので、これらについて導入した llvm ツールセットに含まれるものを使うように、CXX、CLANG を指定します。本例では C コンパイラにも llvm ツールセットに含まれる gcc 7.x を使うように CC も指定しています。

```
$ cd postgresql-11beta1/
$ CC=/opt/rh/devtoolset-7/root/usr/bin/cc \
  CXX=/opt/rh/devtoolset-7/root/usr/bin/g++ \
  CLANG=/opt/rh/llvm-toolset-7/root/usr/bin/clang \
  LLVM_CONFIG=/opt/rh/llvm-toolset-7/root/usr/bin/llvm-config \
  ./configure --prefix=/usr/local/pgsql/11.0 --enable-debug --with-llvm
$ make world
$ make install-world
```

このコンパイルは--with-llvm オプション無しの場合とくらべて時間を要します。検証環境では 20 分ほど要しました。

このビルドを使う場合には環境変数設定で LD_LIBRARY_PATH に llvm のライブラリのパスを加えておきます。

```
$ cat >> 11.0.env <<EOF
LLVM7LIB=/opt/rh/llvm-toolset-7/root/usr/lib64
export LD_LIBRARY_PATH=$LLVM7LIB:$LD_LIBRARY_PATH
EOF
$ LLVM7LIB=/opt/rh/llvm-toolset-7/root/usr/lib64
$ export LD_LIBRARY_PATH=$LLVM7LIB:$LD_LIBRARY_PATH
```

4.1 節「JIT コンパイル機能の追加」に限り本手順でビルドした PostgreSQL を使用しています。

Copyright © 2018 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/11.0/share/doc/html/
```

また、以下 URL にて PostgreSQL 11 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/11/static/
```

4.1. JIT コンパイル機能の追加

JIT が組み入れられた PostgreSQL では、SQL 実行プランで JIT を使うという選択肢が生じます。JIT を使う場合には、SQL 実行時の行データの取り出し処理や SELECT リストの式、条件句の式の計算が最初にコンパイルされてから実行されます。最初にコンパイルする処理が増えますが 1 回あたりの実行は速くなりますので、件数が多い処理での性能向上が期待できます。

以下で JIT が使われる動作を見てみます。JIT 用にいくつか設定変数が追加されていて、JIT を使うか、JIT を使う場合のコスト値の閾値を設定できます。件数が多くて十分にコストが大きくなる場合に使用する価値が生じますので、指定コストを下回るようなら利用されないということです。

```
（確認用に小コストでも JIT が動作するように設定する）
db1=# SET jit TO on;                        -- デフォルトも on
db1=# SET jit_above_cost TO 10;             -- デフォルト 100000、JIT コンパイル適用
db1=# SET jit_optimize_above_cost TO 10; -- デフォルト 500000、最適化も行う
db1=# SET jit_inline_above_cost TO 10;    -- デフォルト 500000、インライン化も行う

（適当なテーブルを作る）
db1=# CREATE TABLE t1 (id int PRIMARY KEY, v text);
db1=# INSERT INTO t1
        SELECT g, md5(g::text) FROM generate_series(1, 1000) as g;

（適当な問い合わせを JIT が有効になるように設定して実行する）
db1=# explain analyze SELECT id, translate(v,'abcdef','')::numeric
        FROM t1 WHERE id % 2 = 1;
```


Copyright © 2018 SRA OSS, Inc. Japan All rights reserved.

```
                                  QUERY PLAN
------------------------------------------------------------------------------
 Seq Scan on t1  (cost=0.00..24.04 rows=5 width=36)
                 (actual time=85.110..87.215 rows=500 loops=1)
   Filter: ((id % 2) = 1)
   Rows Removed by Filter: 500
 Planning Time: 0.187 ms
 JIT:
   Functions: 4                          《JIT 処理された箇所の数》
   Generation Time: 2.939 ms             《JIT コンパイル所要時間》
   Inlining: true                        《JIT でのインライン化も行ったか？》
   Inlining Time: 6.707 ms               《JIT でのインライン化の所要時間》
   Optimization: true                    《JIT での最適化も行ったか？》
   Optimization Time: 54.819 ms          《JIT での最適化の所要時間》
   Emission Time: 23.230 ms              《JIT コード出力の所要時間》
 Execution Time: 90.337 ms
(13 rows)
```

上記は小さいテーブルへの問い合わせですので JIT 実行したほうが遅くなるケースです。今度は JIT が効果的に働く例を見てみましょう。CPU 高負荷であって件数が多いケースを作ってみます。

1 億件を処理させることにします。1 億件のテーブルを作るとストレージ I/O 時間の寄与が大きくなってしまうので、1 億件の行を返す関数を作ります。ROWS オプションも指定しておけば、プランナにも件数が伝わります。また、出力はたくさんの関数や演算子を組み合わせた式にします。

```
db1=# CREATE FUNCTION f100000000() RETURNS SETOF bigint ROWS 100000000
        LANGUAGE sql AS $$
          SELECT g FROM generate_series(1::bigint, 100000000::bigint) AS g;
        $$;
db1=# SET jit TO off;

db1=# explain analyze
        SELECT g, 'X is "' || random() * pi() * substr((g *
          ln(g::float8 + g / 2))::text, 1, 5)::float8 || '"'
            FROM f100000000() AS g;
                            QUERY PLAN
```


Copyright © 2018 SRA OSS, Inc. Japan All rights reserved.

```
------------------------------------------------------------------------------
 Function Scan on f100000000 g
  (cost=0.25..5750000.25 rows=100000000 width=40)
  (actual time=34739.347..333405.234 rows=100000000 loops=1)
 Planning Time: 0.121 ms
 Execution Time: 341003.279 ms
(3 rows)

db1=# SET jit TO on;
db1=# explain analyze
        SELECT g, 'X is "' || random() * pi() * substr((g *
          ln(g::float8 + g / 2))::text, 1, 5)::float8 || '"'
            FROM f100000000() AS g;
                                QUERY PLAN
------------------------------------------------------------------------------
 Function Scan on f100000000 g
  (cost=0.25..5750000.25 rows=100000000 width=40)
  (actual time=56401.345..334782.504 rows=100000000 loops=1)
 Planning Time: 0.049 ms
 JIT:
  Functions: 2
  Generation Time: 0.446 ms
  Inlining: true
  Inlining Time: 9.058 ms
  Optimization: true
  Optimization Time: 42.945 ms
  Emission Time: 32.385 ms
 Execution Time: 280045.147 ms
(11 rows)
```

コスト値が 5750000.25 ですのでデフォルト設定でも JIT が適用されます。JIT 有りの方が所要時間が短くなっています。

下記グラフはこの SQL を JIT 有り無しで各 5 回ずつ実行した結果の平均値とばらつき（標準偏差）です。平均 20%程度の所要時間が短縮されています。

Copyright © 2018 SRA OSS, Inc. Japan All rights reserved.

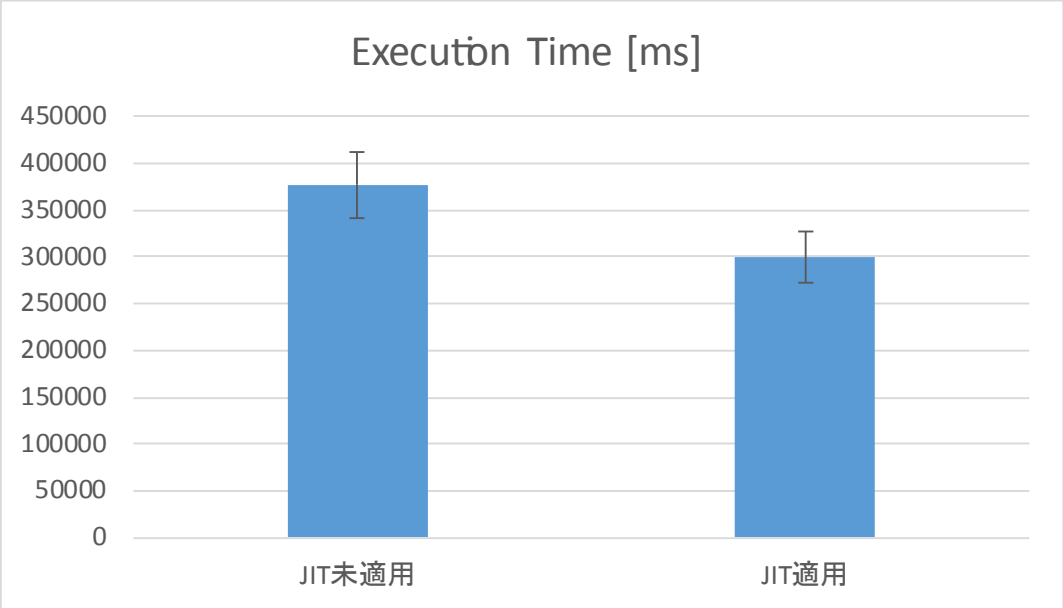


図 *4.1-1: JIT* で性能アップするケース

4.2. パーティショニングの改善

PostgreSQL 10 から CREATE TABLE 文に PARTITION オプションが追加されて、明示的な構文上の指示でテーブルのパーティショニングを実現できるようになりました。PostgreSQL 11 では、このパーティショニング機能についていくつか改善が行われています。

本機能ではいくらか紛らわしい用語が登場しますので、改めて整理しておきます。一つのテーブルを複数の子テーブルに分割するのがパーティショニングです。このとき、子テーブルを含む全体を代表するテーブルを**パーティションテーブル**と呼びます。子テーブルのことを**パーティション**と呼びます。

4.2.1. パーティション全体に対するインデックス作成

PostgreSQL 10 までは、インデックスは個々のパーティションにしか付与できず、パーティションテーブル全体にプライマリキー制約やユニーク制約を設けることができませんでした。これらが PostgreSQL 11 から条件付きながら可能となります。その条件は、インデックス対象にパーティションキーの列（すなわちパーティション分割の基準となる列）を含むことです。

例を示します。以下のように PRIMARY KEY 指定をつけてパーティションテーブルを作ることができます

（パーティション定義）

```
db1=# CREATE TABLE t_log (id int PRIMARY KEY, ts timestamp,
      mes text, typ int) PARTITION BY RANGE (id);
```


Copyright © 2018 SRA OSS, Inc. Japan All rights reserved.

```
CREATE TABLE
db1=# CREATE TABLE t_log_0 PARTITION OF t_log FOR VALUES
        FROM (0) TO (100000);
CREATE TABLE
db1=# CREATE TABLE t_log_1 PARTITION OF t_log FOR VALUES
        FROM (100000) TO (200000);
CREATE TABLE
db1=# CREATE TABLE t_log_2 PARTITION OF t_log FOR VALUES
        FROM (200000) TO (300000);
CREATE TABLE

（データ挿入）
db1=# INSERT INTO t_log SELECT g, '2018-05-24 22:00'::timestamp +
      (g || 's')::interval, md5(g::text), g % 5
      FROM generate_series(0, 250000) AS g;
INSERT 0 250001
```

また、含みさえすれば良いので、以下のように主として ts 列を対象にしているインデックスも追加できます。また、Btree インデックス以外のインデックスでも作成できます。

```
db1=# CREATE INDEX ON t_log (ts, id);
CREATE INDEX
db1=# CREATE INDEX ON t_log USING brin (ts, id);  -- brin インデックス
CREATE INDEX
```

パーティションテーブルに付与したインデックスは、実体としては個々のパーティションに対するインデックスとして実現されています。パーティションの一つを \d で調べると固有のインデックスを持っていることがわかります。

```
db1=# \d t_log_2
                    Table "public.t_log_2"
 Column |            Type             | Collation | Nullable | Default
--------+-----------------------------+-----------+----------+---------
 id     | integer                     |           | not null |
 ts     | timestamp without time zone |           |          |
 mes    | text                        |           |          |
 typ    | integer                     |           |          |
```


Copyright © 2018 SRA OSS, Inc. Japan All rights reserved.

```
Partition of: t_log FOR VALUES FROM (200000) TO (300000)
Indexes:
    "t_log_2_pkey" PRIMARY KEY, btree (id)
    "t_log_2_ts_id_idx" btree (ts, id)
    "t_log_2_ts_id_idx1" brin (ts, id)
```

また、プライマリ制約違反のデータを投入すると、エラー内容は以下のようになります。

（パーティションテーブルへの誤った挿入）

```
db1=# INSERT INTO t_log VALUES (1, CURRENT_TIMESTAMP, 'mes', 1);
ERROR:  duplicate key value violates unique constraint "t_log_0_pkey"
DETAIL:  Key (id)=(1) already exists.
```

（個別パーティションへの誤った挿入）

```
db1=# INSERT INTO t_log_2 VALUES (1, CURRENT_TIMESTAMP, 'mes', 1);
ERROR:  new row for relation "t_log_2" violates partition constraint
DETAIL:  Failing row contains (1, 2018-05-31 13:53:18.746874, mes, 1).
```

つまり、パーティションテーブルに対するプライマリキー制約やユニーク制約は、振り分け規則に基づく制約と、個々パーティション上のインデックスによる制約によって実現されていることがわかります。

4.2.2. パーティションテーブルに対する外部キー

PostgreSQL 11 からパーティションテーブルに対して外部キーが定義できるようになります。

以下のように、t_log テーブルの typ 列のマスタとなる t_log_typ_master テーブルを作って、パーティションテーブル t_log に外部キー制約を設けることができました。

```
db1=# CREATE TABLE t_log_typ_master (typ int primary key, txt text);
db1=# INSERT INTO t_log_typ_master VALUES (0, 'type 0'), (1, 'type 1'),
       (2, 'type 2'), (3, 'type 3'), (4, 'type 4');
db1=# ALTER TABLE t_log
        ADD FOREIGN KEY (typ) REFERENCES t_log_typ_master (typ);
```

以下のように、パーティションテーブル全体に対しても、個々のパーティションに対しても、制約が機能していることが確認できます。

（パーティションテーブルに対して制約違反するデータ挿入）

Copyright © 2018 SRA OSS, Inc. Japan All rights reserved.

```
db1=# INSERT INTO t_log VALUES (250001, '2018-05-29 00:00', 'message', 9);
ERROR:  insert or update on table "t_log_2" violates foreign key constraint
"t_log_typ_fkey"
DETAIL:  Key (typ)=(9) is not present in table "t_log_typ_master".
```

（パーティションテーブルに対して制約違反するデータ挿入）

```
db1=# INSERT INTO t_log_2 VALUES (250001, '2018-05-29 00:00', 'message', 9);
ERROR:  insert or update on table "t_log_2" violates foreign key constraint
"t_log_typ_fkey"
DETAIL:  Key (typ)=(9) is not present in table "t_log_typ_master".
```

ただし、被参照側としてパーティションテーブルを使うことはできません。

以下のように t_log と 1 対 1 対応するパーティションテーブルを作って、

```
db1=# CREATE TABLE t_log_detail (id int primary key, detail text)
        PARTITION BY RANGE (id);
db1=# CREATE TABLE t_log_detail_0 PARTITION OF t_log_detail
        FOR VALUES FROM (0) TO (100000);
db1=# CREATE TABLE t_log_detail_1 PARTITION OF t_log_detail
        FOR VALUES FROM (100000) TO (200000);
db1=# CREATE TABLE t_log_detail_2 PARTITION OF t_log_detail
        FOR VALUES FROM (200000) TO (300000);
db1=# INSERT INTO t_log_detail SELECT g, md5(g::text)
        FROM generate_series(0, 250000) AS g;
```

そこに外部キー制約を指定しようとすると、エラーになります。

```
db1=# ALTER TABLE t_log_detail ADD FOREIGN KEY (id) REFERENCES t_log (id);
ERROR:  cannot reference partitioned table "t_log"
```


Copyright © 2018 SRA OSS, Inc. Japan All rights reserved.

4.2.3. パーティション間の透過的データ移動

パーティションキーとなる列の値を UPDATE したときに、自動的に行が新しい値にふさわしい別パーティションに移動するようになりました。

この動作を確認してみます。

```
（テスト対象の行です。id = 0 なので t_log_0 に属しているはずです。）
db1=# SELECT * FROM t_log WHERE id = 0;
 id |         ts          |               mes                | typ
----+---------------------+----------------------------------+-----
  0 | 2018-05-24 22:00:00 | cfcd208495d565ef66e7dff9f98764da |   0
(1 row)

（id を変えます。）
db1=# UPDATE t_log SET id = 250001 WHERE id = 0;

db1=# SELECT * FROM t_log WHERE id = 250001;
   id   |         ts          |               mes                | typ
--------+---------------------+----------------------------------+-----
 250001 | 2018-05-24 22:00:00 | cfcd208495d565ef66e7dff9f98764da |   0
(1 row)

（t_log_0 から t_log_2 に移動していることがわかります。）
db1=# explain SELECT * FROM t_log WHERE id = 250001;
                              QUERY PLAN
-----------------------------------------------------------------------------
 Append  (cost=0.29..8.31 rows=1 width=49)
   ->  Index Scan using t_log_2_pkey on t_log_2
         (cost=0.29..8.31 rows=1 width=49)
         Index Cond: (id = 250001)
(3 rows)
```

上記の UPDATE について EXPLAIN ANALYZE を付けて実行すると、実行プランにおいてもパーティションを変える動作をしていることが確認できます。

```
db1=# EXPLAIN ANALYZE UPDATE t_log SET id = 250002 WHERE id = 0;
```


Copyright © 2018 SRA OSS, Inc. Japan All rights reserved.

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Update on t_log  (cost=0.29..8.31 rows=1 width=55) (actual
time=0.059..0.059 rows=0 loops=1)
   Update on t_log_0
   ->  Index Scan using t_log_0_pkey on t_log_0
         (cost=0.29..8.31 rows=1 width=55)
         (actual time=0.008..0.009 rows=1 loops=1)
        Index Cond: (id = 0)
 Planning Time: 0.138 ms
 Trigger for constraint t_log_typ_fkey on t_log_2: time=0.026 calls=1
 Execution Time: 0.114 ms
(7 rows)
```

4.2.4. ハッシュパーティショニング

PostgreSQL 10 でのテーブルパーティショニングではパーティショニング戦略として、リスト（LIST）と範囲（RANGE）が使用できました。PostgreSQL 11 ではこれに加えてハッシュ（HASH）を使用できるようになります。これを使う方式をハッシュパーティショニングと呼びます。ハッシュパーティショニングでは、パーティションキー列の値に対して正整数のハッシュ値を生成して、その値を割った余りがいくつになるかに基づいて振り分けを行います。

以下に、実際にハッシュパーティショニングを使ったパーティションテーブルを作ってみます。

```
（パーティションテーブルを作成する）
db1=# CREATE TABLE t_message (mid bigint, mes text, ts timestamp, flag int)
        PARTITION BY HASH (mes); -- メッセージ文字列(mes)をキーにする
db1=# CREATE TABLE t_message_3_0 PARTITION OF t_message
        FOR VALUES WITH (MODULUS 3, REMAINDER 0 ); -- 3で割って余り0
db1=# CREATE TABLE t_message_3_1 PARTITION OF t_message
        FOR VALUES WITH (MODULUS 3, REMAINDER 1 ); -- 3で割って余り1
db1=# CREATE TABLE t_message_3_2 PARTITION OF t_message
        FOR VALUES WITH (MODULUS 3, REMAINDER 2 ); -- 3で割って余り2

（値を投入するとほぼ均等に振り分けられる）
```


Copyright © 2018 SRA OSS, Inc. Japan All rights reserved.

```
db1=# INSERT INTO t_message
        SELECT g, md5(g::text),
         '2018-05-24 22:00'::timestamp + (g || 's')::interval, g % 5
            FROM generate_series(1, 150000) AS g;
INSERT 0 150000
db1=# SELECT count(*) FROM t_message_3_0;
 count
-------
 49865
(1 row)

db1=# SELECT count(*) FROM t_message_3_1;
 count
-------
 50243
(1 row)

db1=# SELECT count(*) FROM t_message_3_2;
 count
-------
 49892
(1 row)
```

PostgreSQL 11 のハッシュパーティショニングではパーティションの数を後から増やす手段が用意されています。

ポイントは全てのパーティションでハッシュ値を割る数（本例では 3）が同じでなくても良い、ということです。3 で割って余り 0 となる数（0、3、6、9、12、15、…）は、6 で割って余り 0 となる数（0、6、12、…）と、6 で割って余り 3 となる数（3、9、15、…）に分割することができます。

そこで「(MODULUS 3, REMAINDER 0)」と定義したパーティションを取り外して、代わりに「MODULUS 6, REMAINDER 0」のパーティション、「MODULUS 6, REMAINDER 3」のパーティションを定義することで、パーティションを増やすことができます。その後に、取り外したパーティションのデータを再投入して、データを合流させれば完了です。SQL コマンドとしては以下のようになります。

（１つのパーティションを２つのパーティションで置き換え）

```
db1=# BEGIN;
db1=# ALTER TABLE t_message DETACH PARTITION t_message_3_0;
```


Copyright © 2018 SRA OSS, Inc. Japan All rights reserved.

```
db1=# CREATE TABLE t_message_6_0 PARTITION OF t_message FOR VALUES WITH
        (MODULUS 6, REMAINDER 0);
db1=# CREATE TABLE t_message_6_3 PARTITION OF t_message FOR VALUES WITH
        (MODULUS 6, REMAINDER 3);
db1=# COMMIT;

（デタッチした 3 で割って余り 0 のパーティションのデータを再投入した後、削除する）
db1=# INSERT INTO t_message SELECT * FROM t_message_3_0;
db1=# TRUNCATE t_message_3_0;
```

この手順を少しずつ実行していけば、パーティションの再編成を長いサービス停止なしに実現できます。

パーティションを置き換えるときに 1 つのトランザクションとしてるのは、データ投入時に振り分け先となるパーティションが無い状態ですとエラーになり、また、一つのハッシュ値に対して複数のパーティションが格納先候補となる状態もエラーとなるためです。

4.2.5. 問い合わせ実行時のパーティション除外処理

PostgreSQL 10 および、それ以前のテーブル継承を使ったパーティショニングにおいても、検索条件に合わないパーティション（あるいは子テーブル）に対するスキャンを省略する機能がありました。これは制約による除外（constraint exclusion）と呼ばれ、設定 constraint_exclusion によって有効・無効が制御されます。この機能は SQL の実行プランを作成するときに発動していました。

これに対して、PostgreSQL 11 では プラン作成時だけではなく SQL を実行する段階で不要なパーティションの除外ができます。この動作を partition pruning と呼び、設定 enable_partition_pruning で有効・無効を制御できます（デフォルトは on ）。

実行時に行うことで、パラメータによって条件が変わるプリペアドステートメントにおいても確実に不要パーティション除外をすることができます。PostgreSQL 9.2 以降であればプリペアドステートメントの実行に際して多くの場合にパラメータも含めたプラン作成が行われますが、パラメータを考慮しないプランが実行される動作もありえました。

また、パーティション除外では「制約による除外」では対応できなかった形状の SQL において、プラン作成時にスキャン不要パーティションの除外が可能です。以下に例を示します。

```
db1=# SET constraint_exclusion TO on;
db1=# SET enable_partition_pruning TO on;
db1=# explain SELECT count(*) FROM t_log WHERE id < 100000;
```


Copyright © 2018 SRA OSS, Inc. Japan All rights reserved.

```
                            QUERY PLAN
------------------------------------------------------------------------------
 Aggregate  (cost=2258.00..2258.01 rows=1 width=8)
   ->  Append  (cost=0.00..2008.00 rows=100000 width=0)
         ->  Seq Scan on t_log_0  (cost=0.00..1508.00 rows=100000 width=0)
               Filter: (id < 100000)
(4 rows)

db1=# SET enable_partition_pruning TO off;
db1=# explain SELECT count(*) FROM t_log WHERE id < 100000;
                                    QUERY PLAN
------------------------------------------------------------------------------
 Aggregate  (cost=2266.63..2266.64 rows=1 width=8)
   ->  Append  (cost=0.00..2016.63 rows=100002 width=0)
         ->  Seq Scan on t_log_0  (cost=0.00..1508.00 rows=100000 width=0)
               Filter: (id < 100000)
         ->  Index Only Scan using t_log_1_pkey on t_log_1
               (cost=0.29..4.31 rows=1 width=0)
               Index Cond: (id < 100000)
         ->  Index Only Scan using t_log_2_pkey on t_log_2
               (cost=0.29..4.31 rows=1 width=0)
               Index Cond: (id < 100000)
(8 rows)
```

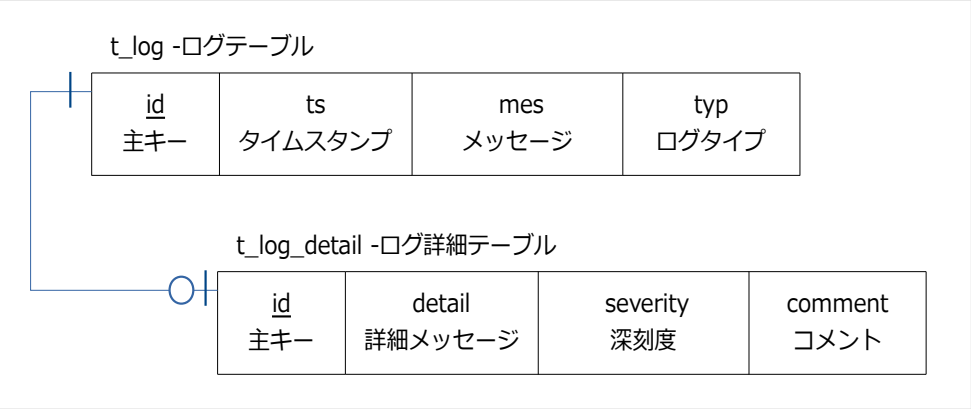
この結果から、WHERE 句で除外が可能な集約の問い合わせが、constraint_exclusion = on だけではパーティション除外ができず、 enable_partition_pruning = on でパーティション除外が可能になっていることが確認できます。

Copyright © 2018 SRA OSS, Inc. Japan All rights reserved.

4.2.6. パーティション指向の結合・集約

◆ *パーティション指向の結合*

以下のような構造の二つ組のテーブルがあって、それぞれ主キー列 id を使って同じ条件でパーティショニングをしているとします。これらテーブルを結合するときに期待したい動作は t_log と t_log_detail のパーティション同士で各々結合することです（ここではこれを「パーティション指向の結合」と呼びます）。そうすれば並列動作の効果も出やすいですし、スキャンを一部のパーティションに限定させやすくなります。



PostgreSQL 11 からこのような動作がサポートされました。実際に動作を確認してみます。ここでの t_log_detail は上図よりシンプルな定義にしてあります。

（簡易な定義で t_log_detail テーブル作成 - 4.2.2節で作ったものと同じです）

```
db1=# CREATE TABLE t_log_detail (id int primary key, detail text)
        PARTITION BY RANGE (id);
db1=# CREATE TABLE t_log_detail_0 PARTITION OF t_log_detail
        FOR VALUES FROM (0) TO (100000);
db1=# CREATE TABLE t_log_detail_1 PARTITION OF t_log_detail
        FOR VALUES FROM (100000) TO (200000);
db1=# CREATE TABLE t_log_detail_2 PARTITION OF t_log_detail
        FOR VALUES FROM (200000) TO (300000);
db1=# INSERT INTO t_log_detail
        SELECT g, md5(g::text) FROM generate_series(0, 250000) AS g;
```

パーティション指向の結合をさせるには設定パラメータ enable_partitionwise_join を on にします。デフォルトは off です。また、パラレル動作と組み合わせて効果がでる機能ですが、実行プランにあらわれる効果を明確にするため、パラレル動作を抑止する設定を与えておきます。

Copyright © 2018 SRA OSS, Inc. Japan All rights reserved.

```
db1=# SET parallel_setup_cost TO 999999999; -- パラレル無しでプランを見るため
db1=# SET enable_partitionwise_join TO on;  -- パーティション指向結合を有効に

db1=# explain SELECT lo.mes, ld.detail FROM t_log lo
        LEFT JOIN t_log_detail ld ON (lo.id = ld.id);
                            QUERY PLAN
-------------------------------------------------------------------------------
 Append  (cost=0.58..22710.76 rows=250001 width=66)
   ->  Merge Left Join  (cost=0.58..8579.55 rows=100000 width=66)
         Merge Cond: (lo.id = ld.id)
         ->  Index Scan using t_log_0_pkey on t_log_0 lo  (cost=0.29..3638.29
rows=100000 width=37)
         ->  Index Scan using t_log_detail_0_pkey on t_log_detail_0 ld
(cost=0.29..3441.29 rows=100000 width=37)
   ->  Merge Left Join  (cost=0.58..8579.58 rows=100000 width=66)
         Merge Cond: (lo_1.id = ld_1.id)
         ->  Index Scan using t_log_1_pkey on t_log_1 lo_1
(cost=0.29..3638.29 rows=100000 width=37)
         ->  Index Scan using t_log_detail_1_pkey on t_log_detail_1 ld_1
(cost=0.29..3441.29 rows=100000 width=37)
   ->  Merge Left Join  (cost=0.62..4301.63 rows=50001 width=66)
         Merge Cond: (lo_2.id = ld_2.id)
         ->  Index Scan using t_log_2_pkey on t_log_2 lo_2
(cost=0.29..1825.31 rows=50001 width=37)
         ->  Index Scan using t_log_detail_2_pkey on t_log_detail_2 ld_2
(cost=0.29..1726.31 rows=50001 width=37)
(13 rows)
```

上記のように t_log_0 と t_log_detail_0 、t_log_1 と t_log_detail_1 など、パーティション同士がまず結合するプランが生成されました。

◆ *パーティション指向の集約*

集約処理をするときにパーティション毎に集約処理を各々先に行って、最後にその結果を統合するという実行プランが可能になりました。

この動作をさせるには設定パラメータ enable_partitionwise_aggregate を on にします。デフォルトは off

Copyright © 2018 SRA OSS, Inc. Japan All rights reserved.

です。プランの違いを確認してみます。今回もプランを見やすくするためパラレル処理は選択されないようにしています。

```
db1=# SET parallel_setup_cost TO 999999999; -- パラレル無しでプランを見るため

（パーティション指向の集約を使わない場合）
db1=# SET enable_partitionwise_aggregate TO off;
db1=# explain SELECT avg(length(mes)) FROM t_log;
                            QUERY PLAN
-------------------------------------------------------------------------------
 Aggregate  (cost=7578.02..7578.03 rows=1 width=32)
   ->  Append  (cost=0.00..6328.02 rows=250001 width=33)
         ->  Seq Scan on t_log_0  (cost=0.00..2031.00 rows=100000 width=33)
         ->  Seq Scan on t_log_1  (cost=0.00..2031.00 rows=100000 width=33)
         ->  Seq Scan on t_log_2  (cost=0.00..1016.01 rows=50001 width=33)
(5 rows)

（パーティション指向の集約を使う場合）
db1=# SET enable_partitionwise_aggregate TO on;
db1=# explain SELECT avg(length(mes)) FROM t_log;
                              QUERY PLAN
-------------------------------------------------------------------------------
 Finalize Aggregate  (cost=6328.07..6328.08 rows=1 width=32)
   ->  Append  (cost=2531.00..6328.06 rows=3 width=32)
         ->  Partial Aggregate  (cost=2531.00..2531.01 rows=1 width=32)
               ->  Seq Scan on t_log_0
                     (cost=0.00..2031.00 rows=100000 width=33)
         ->  Partial Aggregate  (cost=2531.00..2531.01 rows=1 width=32)
               ->  Seq Scan on t_log_1
                     (cost=0.00..2031.00 rows=100000 width=33)
         ->  Partial Aggregate  (cost=1266.01..1266.02 rows=1 width=32)
               ->  Seq Scan on t_log_2
                     (cost=0.00..1016.01 rows=50001 width=33)
(8 rows)
```

上記２番目のプランでは各パーティションごとに Aggregate していることが確認できます。

Copyright © 2018 SRA OSS, Inc. Japan All rights reserved.

4.3. パラレル実行の改善

パラレル実行は PostgreSQL 9.6、PostgreSQL 10 でもサポートされていますが、PostgreSQL 11 では SQL 実行で並列処理できる項目が更にいくつか追加されています。

4.3.1. 並列ハッシュ結合

ハッシュ結合がパラレル実行できるようになりました。4.2.1 節で作った t_log と、4.2.2 節（あるいは 4.2.6 節）で作った t_log_detail テーブルを外部結合して集約する問い合わせを実行すると、「Parallel Hash Left Join 」「Parallel Hash」が選択されました。ハッシュ作成もパラレルで実行されるようになります。

```
db1=# SET enable_parallel_hash TO on; -- これはデフォルトでも on
db1=# explain SELECT max(length(mes || detail)) FROM t_log lo
         LEFT JOIN t_log_detail ld ON (lo.id = ld.id);
                                QUERY PLAN
-------------------------------------------------------------------------------
 Finalize Aggregate  (cost=15810.11..15810.12 rows=1 width=4)
   ->  Gather  (cost=15809.89..15810.10 rows=2 width=4)
         Workers Planned: 2
         ->  Partial Aggregate  (cost=14809.89..14809.90 rows=1 width=4)
               ->  Parallel Hash Left Join
                    (cost=6192.53..14028.64 rows=104167 width=66)
                    Hash Cond: (lo.id = ld.id)
                    ->  Parallel Append
                         (cost=0.00..4569.43 rows=104168 width=37)
                         ->  Parallel Seq Scan on t_log_0 lo
                             (cost=0.00..1619.24 rows=58824 width=37)
                         ->  Parallel Seq Scan on t_log_1 lo_1
                             (cost=0.00..1619.24 rows=58824 width=37)
                         ->  Parallel Seq Scan on t_log_2 lo_2
                             (cost=0.00..810.12 rows=29412 width=37)
                    ->  Parallel Hash
                         (cost=4076.43..4076.43 rows=104168 width=37)
                         ->  Parallel Append
                             (cost=0.00..4076.43 rows=104168 width=37)
```


Copyright © 2018 SRA OSS, Inc. Japan All rights reserved.

```
                              ->  Parallel Seq Scan on t_log_detail_0 ld
                                   (cost=0.00..1422.24 rows=58824 width=37)
                              ->  Parallel Seq Scan on t_log_detail_1 ld_1
                                   (cost=0.00..1422.24 rows=58824 width=37)
                              ->  Parallel Seq Scan on t_log_detail_2 ld_2
                                   (cost=0.00..711.12 rows=29412 width=37)
(15 rows)
```

4.3.2. 並列インデックス作成

Btree インデックスの作成で並列処理が行われるようになりました。

以下のように 4.2.1 節で作った t_log のインデックスを再作成してみますと、パラレル動作が確認できます。確認のため、DEBUG1 レベルのメッセージを出力させています。

```
（本設定が 2 以上だと並列処理が試みられます。デフォルトは 2 です。）
db1=# SHOW max_parallel_maintenance_workers;
 max_parallel_maintenance_workers
----------------------------------
 2
(1 row)

db1=# DROP INDEX t_log_ts_id_idx;            -- インデックスを削除します
db1=# SET client_min_messages TO DEBUG1;     -- 動作確認のため DEBUG1 にします

（インデックスを再作成します。パラレル動作が確認できます。）
db1=# CREATE INDEX ON t_log (ts, id);
DEBUG:  building index "t_log_0_ts_id_idx" on table "t_log_0" with request
for 1 parallel worker
DEBUG:  building index "t_log_1_ts_id_idx" on table "t_log_1" with request
for 1 parallel worker
DEBUG:  building index "t_log_2_ts_id_idx" on table "t_log_2" serially
CREATE INDEX
```


Copyright © 2018 SRA OSS, Inc. Japan All rights reserved.

4.3.3. 並列 CREATE TABLE AS

「CREATE TABLE ... AS .... 」など、SELECT した結果をテーブルに格納する SQL 命令がいくつかあります。これらの SQL で並列処理が行われるようになりました。PostgreSQL 10 までは、対応が見落とされていました。

対応が追加されたのは以下の 3 構文です。

- CREATE TABLE ... AS ...
- SELECT ... INTO ...
- CREATE MATERIALIZED VIEW

以下に例を示します。

```
db1=# SET force_parallel_mode TO on;          -- これらの設定では
db1=# SET parallel_setup_cost TO 0;           -- 確認用にプランナにパラレル指向を
db1=# SET parallel_tuple_cost TO 0;           -- 強制する設定を付与しています。

db1=# explain CREATE TABLE t_create AS SELECT * FROM t_log;
                                QUERY PLAN
-------------------------------------------------------------------------------
 Gather  (cost=1000.00..5569.43 rows=250001 width=49)
   Workers Planned: 2
   ->  Parallel Append  (cost=0.00..4569.43 rows=104168 width=49)
         ->  Parallel Seq Scan on t_log_0
               (cost=0.00..1619.24 rows=58824 width=49)
         ->  Parallel Seq Scan on t_log_1
               (cost=0.00..1619.24 rows=58824 width=49)
         ->  Parallel Seq Scan on t_log_2
               (cost=0.00..810.12 rows=29412 width=49)
(6 rows)

db1=# explain CREATE MATERIALIZED VIEW mv_log AS SELECT * FROM t_log;
                                QUERY PLAN
-------------------------------------------------------------------------------
 Gather  (cost=1000.00..5569.43 rows=250001 width=49)
   Workers Planned: 2
   ->  Parallel Append  (cost=0.00..4569.43 rows=104168 width=49)
```


Copyright © 2018 SRA OSS, Inc. Japan All rights reserved.

```
        ->  Parallel Seq Scan on t_log_0
              (cost=0.00..1619.24 rows=58824 width=49)
        ->  Parallel Seq Scan on t_log_1
              (cost=0.00..1619.24 rows=58824 width=49)
        ->  Parallel Seq Scan on t_log_2
              (cost=0.00..810.12 rows=29412 width=49)
(6 rows)
```

なお、REFRESH MATERIALIZED VIEW は依然パラレルになりません。

4.3.4. 並列 *Append*

Append とは検索結果行をたてにつなげるプラン要素です。パーティションテーブルや UNION ALL を用いた SELECT 文の実行で出現します。これの並列版 Parallel Append が追加されました。

```
db1=# SET force_parallel_mode TO on;          -- これらの設定では
db1=# SET parallel_setup_cost TO 0;           -- 確認用にプランナにパラレル指向を
db1=# SET parallel_tuple_cost TO 0;           -- 強制する設定を付与しています。

db1=# explain SELECT * FROM t_log ;
                                QUERY PLAN
-------------------------------------------------------------------------------
 Gather  (cost=0.00..4569.43 rows=250001 width=49)
   Workers Planned: 2
   ->  Parallel Append  (cost=0.00..4569.43 rows=104168 width=49)
         ->  Parallel Seq Scan on t_log_0
               (cost=0.00..1619.24 rows=58824 width=49)
         ->  Parallel Seq Scan on t_log_1
               (cost=0.00..1619.24 rows=58824 width=49)
         ->  Parallel Seq Scan on t_log_2
               (cost=0.00..810.12 rows=29412 width=49)
(6 rows)

db1=# explain SELECT * FROM (SELECT * FROM t_log_0 UNION ALL
        SELECT * FROM t_log_1 UNION ALL SELECT * FROM t_log_2) AS t_log ;
```


Copyright © 2018 SRA OSS, Inc. Japan All rights reserved.

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Gather  (cost=0.00..4569.43 rows=250001 width=49)
   Workers Planned: 2
   ->  Parallel Append  (cost=0.00..4569.43 rows=104168 width=49)
         ->  Parallel Seq Scan on t_log_0
               (cost=0.00..1619.24 rows=58824 width=49)
         ->  Parallel Seq Scan on t_log_1
               (cost=0.00..1619.24 rows=58824 width=49)
         ->  Parallel Seq Scan on t_log_2
               (cost=0.00..810.12 rows=29412 width=49)
(6 rows)
```

PostgreSQL 10 の場合、各スキャンがパラレルになっても Append 部分は直列実行でした。

4.3.5. parallel_leader_participation

PostgreSQL では通常並列処理をするときにメインのプロセスも並列処理に参加する動作をします。3 パラレルで実行しようとするとき、PostgreSQL はワーカ子プロセスを 2 つ立ち上げます。

新たな postgresql.conf 設定項目 parallel_leader_participation （デフォルト on ）を off にすると、主プロセスは並列処理に参加せず結果を待つようになります。以下はパラメータを変えた際の実行例です。

```
db1=# SET force_parallel_mode TO on;          -- これらの設定では
db1=# SET parallel_setup_cost TO 0;           -- 確認用にプランナにパラレル指向を
db1=# SET parallel_tuple_cost TO 0;           -- 強制する設定を付与しています。

db1=# SET parallel_leader_participation TO on;
db1=# explain (analyze, verbose) SELECT * FROM t_log;
                                QUERY PLAN
-------------------------------------------------------------------------------
 Gather  (cost=0.00..2636.43 rows=250001 width=49) (actual
time=1.354..196.368 rows=250001 loops=1)
   Output: t_log_0.id, t_log_0.ts, t_log_0.mes, t_log_0.typ
   Workers Planned: 2
   Workers Launched: 2
   ->  Parallel Append  (cost=0.00..2636.43 rows=104168 width=49) (actual
time=0.048..93.421 rows=83334 loops=3)
```


Copyright © 2018 SRA OSS, Inc. Japan All rights reserved.

```
        Worker 0: actual time=0.050..100.496 rows=81690 loops=1
        Worker 1: actual time=0.037..122.092 rows=77465 loops=1
        ->  Parallel Seq Scan on public.t_log_0
              (cost=0.00..846.24 rows=58824 width=49)
              (actual time=0.041..16.937 rows=33333 loops=3)
             Output: t_log_0.id, t_log_0.ts, t_log_0.mes, t_log_0.typ
             Worker 0: actual time=0.047..38.800 rows=81690 loops=1
             Worker 1: actual time=0.045..9.331 rows=9752 loops=1
        ->  Parallel Seq Scan on public.t_log_1
              (cost=0.00..846.24 rows=58824 width=49)
              (actual time=0.033..32.465 rows=50000 loops=2)
             Output: t_log_1.id, t_log_1.ts, t_log_1.mes, t_log_1.typ
             Worker 1: actual time=0.035..55.294 rows=67713 loops=1
        ->  Parallel Seq Scan on public.t_log_2
              (cost=0.00..423.12 rows=29412 width=49)
              (actual time=0.056..15.637 rows=50001 loops=1)
             Output: t_log_2.id, t_log_2.ts, t_log_2.mes, t_log_2.typ
 Planning Time: 0.433 ms
 Execution Time: 236.220 ms
(18 rows)
  -- どの Worker も担当していないパーティションがある、すなわち主プロセスが担当している

db1=# SET parallel_leader_participation TO off;
SET
db1=# explain (analyze, verbose) SELECT * FROM t_log;
                                QUERY PLAN
-------------------------------------------------------------------------------
 Gather  (cost=0.00..2512.01 rows=250001 width=49) (actual
time=16.532..277.969 rows=250001 loops=1)
   Output: t_log_0.id, t_log_0.ts, t_log_0.mes, t_log_0.typ
   Workers Planned: 2
   Workers Launched: 2
   ->  Parallel Append  (cost=0.00..2512.01 rows=125000 width=49)
                        (actual time=0.086..84.206 rows=125000 loops=2)
         Worker 0: actual time=0.079..66.568 rows=100000 loops=1
```


Copyright © 2018 SRA OSS, Inc. Japan All rights reserved.

```
        Worker 1: actual time=0.094..101.845 rows=150001 loops=1
        ->  Seq Scan on public.t_log_0
              (cost=0.00..1258.00 rows=100000 width=49)
              (actual time=0.076..34.147 rows=100000 loops=1)
             Output: t_log_0.id, t_log_0.ts, t_log_0.mes, t_log_0.typ
             Worker 0: actual time=0.076..34.147 rows=100000 loops=1
        ->  Seq Scan on public.t_log_1
              (cost=0.00..1258.00 rows=100000 width=49)
              (actual time=0.090..34.300 rows=100000 loops=1)
             Output: t_log_1.id, t_log_1.ts, t_log_1.mes, t_log_1.typ
             Worker 1: actual time=0.090..34.300 rows=100000 loops=1
        ->  Seq Scan on public.t_log_2
              (cost=0.00..629.01 rows=50001 width=49)
              (actual time=0.079..18.833 rows=50001 loops=1)
             Output: t_log_2.id, t_log_2.ts, t_log_2.mes, t_log_2.typ
             Worker 1: actual time=0.079..18.833 rows=50001 loops=1
 Planning Time: 0.417 ms
 Execution Time: 324.792 ms
(18 rows)
```

parallel_leader_participation が on か off かの違いは違いが見えにくいのですが、on の場合 Worker が担当しない処理箇所が存在することがわかります。その部分はメインプロセスが処理を行っています。

Copyright © 2018 SRA OSS, Inc. Japan All rights reserved.

4.4. プロシージャ内でのトランザクション制御

PostgerSQL11 から、データベース内の手続き言語オブジェクトとして FUNCTION（関数）の他に PROCEDURE（プロシージャ）が追加されました。プロシージャと関数の違いは以下の通りです。

- CREATE PROCEDURE 文は RETURNS オプションを持ちません。
  ただし、INOUT 引数を使って戻り値を返すことは可能です。OUT 引数は禁止です。
- プロシージャは式の中に組み入れることはできません。CALL 文で呼び出します。
- プロシージャ内でトランザクション制御が可能です。
- DO 構文はプロシージャと同様の扱いとなります。

以下は PostgreSQL 11 ドキュメント（43.8.Transaction Management）に記載のプロシージャ実行例です。

```
db1=# CREATE TABLE test1 (a int);
db1=# CREATE OR REPLACE PROCEDURE p_tx5() LANGUAGE plpgsql AS $$
        BEGIN
          FOR i IN 0..9 LOOP
            INSERT INTO test1 (a) VALUES (i);
            IF i % 2 = 0 THEN COMMIT;        -- 偶数ならコミット
            ELSE ROLLBACK;                   -- 奇数ならロールバック
            END IF;
          END LOOP;
        END; $$;

db1=# CALL p_tx5();        -- CALL 文で実行します
db1=# SELECT * FROM test1;
 a
---
 0
 2
 4
 6
 8
(5 rows)
```

プロシージャ内で COMMIT、ROLLBACK が機能していることがわかります。対応しているのは COMMIT、ROLLBACK だけで、SAVEPOINT と ROLLBACK TO には対応していません。プロシージャを実行したトラン

Copyright © 2018 SRA OSS, Inc. Japan All rights reserved.

ザクションに対して COMMIT や ROLLBACK が実行され、次のトランザクションは暗黙に開始されます。

上記の CALL 文は明示的なトランザクション内では実行できません。

以下のようにエラーになります。

```
db1=# BEGIN;
BEGIN
db1=# CALL p_tx5();
ERROR:  invalid transaction termination
CONTEXT:  PL/pgSQL function p_tx5() line 5 at COMMIT
```

プロシージャの CALL は入れ子で実行可能です。ただし、関数内からの入れ子 CALL はできません。

また、各プロシージャがサブトランザクションを構成するわけではありません。以下の例を参照ください。

```
db1=# CREATE PROCEDURE p_tx8() LANGUAGE plpgsql AS $$
        BEGIN
          INSERT INTO test1 VALUES (0);      -- この行挿入が、
          CALL p_tx7();                      -- この呼び出し先にて、
          COMMIT;
        END; $$;
db1=# CREATE PROCEDURE p_tx7() LANGUAGE plpgsql AS $$
        BEGIN
          INSERT INTO test1 VALUES (1);
          ROLLBACK;                          -- ロールバックされてしまいます。
        END; $$;

db1=# DELETE FROM test1;   -- 検証用に最初にテーブルを空にします
db1=# CALL p_tx8();
CALL
db1=# SELECT * FROM test1;   -- 入れ子 CALL 先の ROLLBACK が効いて 0 行になりました。
 a
---
(0 rows)
```

呼び出し先のトランザクション制御をサブトランザクションにしたい場合には、従来からある BEGIN...END ブロックによるサブトランザクションと併用します。以下のようにすれば、プロシージャ中で呼ばれた ROLLBACK をそのプロシージャ内（この例では p_tx7）の処理だけに限定できます。

```
db1=# CREATE OR REPLACE PROCEDURE p_tx11() LANGUAGE plpgsql AS $$
```


Copyright © 2018 SRA OSS, Inc. Japan All rights reserved.

```
        BEGIN
          INSERT INTO test1 VALUES (0);      -- 0 の行を挿入
          BEGIN
            CALL p_tx7();                    -- ROLLBACK を含むプロシージャ
          EXCEPTION WHEN others THEN NULL;
          END;
          COMMIT;
        END; $$;

db1=# DELETE FROM test1;
db1=# CALL p_tx11();
db1=# SELECT * FROM test1;                   -- 0 の行を挿入した結果が残る
 a
---
 0
(1 row)
```

しかし、以下のような直感に反する動作もします。

```
db1=# CREATE OR REPLACE PROCEDURE p_tx12() LANGUAGE plpgsql AS $$
        BEGIN
          INSERT INTO test1 VALUES (41);
          ROLLBACK;
          INSERT INTO test1 VALUES (42);
          COMMIT;
        END; $$;

db1=# CREATE OR REPLACE PROCEDURE p_tx13() LANGUAGE plpgsql AS $$
        BEGIN
          INSERT INTO test1 VALUES (0);
          BEGIN
            CALL p_tx12();
          EXCEPTION WHEN others THEN NULL;
          END;
          COMMIT;
        END; $$;
```


Copyright © 2018 SRA OSS, Inc. Japan All rights reserved.

```
db1=# DELETE FROM test1;
db1=# CALL p_tx13();
db1=# SELECT * FROM test1;       -- p_tx12 で挿入された行は含まれない
 a
---
 0
(1 row)
```

p_tx12 の中でロールバック後にコミット処理があっても、呼び出し元 p_tx13 では p_tx12 内でエラーが生じたあつかいとなって結局ロールバックされてしまいます。

見てきたようにトランザクション制御を伴うプロシージャを入れ子で呼び出すには、様々な注意が必要となります。

4.4.1. 各手続き言語でのトランザクション制御対応

PL/pgSQL のみならず、PL/Perl、PL/Python、PL/Tcl でもプロシージャ定義とプロシージャ内でのトランザクション制御に対応しています。各言語それぞれにトランザクション制御をする命令が追加されました。

例えば、PL/Perl であれば『spi_commit()』『spi_rollback()』という関数が追加されています。以下のように使用します。

（PL/Perl でのトランザクション制御の例）

```
DO LANGUAGE plperl $$
my $sth = spi_query("SELECT x FROM generate_series(1, 10) AS x");
my $row;
while (defined($row = spi_fetchrow($sth))) {
    spi_exec_query("INSERT INTO test1 (a) VALUES (" . $row->{x} . ")");
    if ($row->{x} % 2 == 0) {
        spi_commit();
    } else {
        spi_rollback();
    }
}
$$;
```


Copyright © 2018 SRA OSS, Inc. Japan All rights reserved.

4.5. ALTER TABLE .. ADD COLUMN の性能改善

テーブルに列を追加するときに追加列に値を設定している場合には従来実行に時間を要しましたが 、PostgreSQL 11 で高速化されました。

以下に実行例を示します。

```
（PostgreSQL 11 での列追加）
db1=# CREATE TABLE t_alt (id int primary key, c1 int);
db1=# INSERT INTO t_alt SELECT g, g FROM generate_series(1, 100000) AS g;
db1=# \timing
Timing is on.
db1=# ALTER TABLE t_alt ADD c2 int;
ALTER TABLE
Time: 16.918 ms            -- 値の無い列追加は高速
db1=# ALTER TABLE t_alt ADD c3 int DEFAULT 100;
ALTER TABLE
Time: 25.824 ms            -- 値のある列追加も高速

（PostgreSQL 10 で同等テーブルに列追加）
db1=# ALTER TABLE t_alt ADD c2 int DEFAULT NULL;
ALTER TABLE
Time: 14.717 ms            -- 値の無い列追加は 10.x でも高速
db1=# ALTER TABLE t_alt ADD c3 int DEFAULT 100;
ALTER TABLE
Time: 718.629 ms           -- 10.x では値のある列追加は遅い
```

なお、値の入った列の型変更については PostgreSQL 11 でも依然として時間を要します。

```
db1=# ALTER TABLE t_alt ALTER c3 TYPE bigint;
ALTER TABLE
Time: 640.206 ms           -- PostgreSQL 11 でもデータ型変換は高速になっていない
```


Copyright © 2018 SRA OSS, Inc. Japan All rights reserved.

4.6. ウィンドウ関数の拡張

PostgreSQL 11 でウィンドウ関数で指定可能なオプションが拡充されました。

まずは、機能確認のためにウィンドウ関数を使った SQL のテスト用のデータを作ります。以下のテーブル t_temperature1 は 5000 件の日時と高度ごとの気温のデータを模したものです。

```
db1=# CREATE TABLE t_temperature1
        (id int primary key, dt timestamp, high int, t float4);
db1=# INSERT INTO t_temperature1 SELECT g,
        '2018-01-01'::timestamp + g * '10min'::interval +
        (floor(random() * 600) || 'sec')::interval,
        trunc((random() + random() + random()) * 5) * 100, NULL
        FROM generate_series(1, 5000) AS g;
db1=# UPDATE t_temperature1
        SET t = 30 - abs(8 - to_char(dt, 'MM')::int) * 0.5
              - abs(14 - to_char(dt, 'HH24')::int) * 0.5 - 40 * 0.5
              - high * 0.001 + (random() - 0.5) * 2;
db1=# SELECT * FROM t_temperature1 LIMIT 5;
 id |         dt          | high |     t
----+---------------------+------+-----------
  1 | 2018-01-01 00:19:47 |  600 |  -1.78712
  2 | 2018-01-01 00:23:30 |  400 |  -1.18362
  3 | 2018-01-01 00:34:34 |  100 | -0.812903
  4 | 2018-01-01 00:43:27 |  600 | -0.749416
  5 | 2018-01-01 00:53:45 |  600 |  -1.11721
(5 rows)
```

4.6.1. *GROUPS* ウィンドウフレーム

ウィンドウフレームの指定で GROUPS モードに対応しました。これは ROWS モードと似ていますが 、ORDER BY 指定した列の値が同値の場合には一つのグループとして扱います。

以下の SQL では高度（high）が 600m より高いか低いかで PARTITION BY にて分けたうえで、何日何時台で並び替えたときの前後 1 つの範囲で平均気温を t_avg として出力しています。dt = 2018-01-01 01:24:18（2018 年 1 月 1 日 1 時台のデータ）の行の t_avg 値は、 2018 年 1 月 1 日 0 時台から 2 時台の平均気温値ということです。

Copyright © 2018 SRA OSS, Inc. Japan All rights reserved.

```
db1=# SELECT dt, (high > 600) high, t, avg(t) OVER (
        PARTITION BY (high > 600) ORDER BY to_char(dt,'YYYYMMDD-HH24')
          GROUPS BETWEEN 1 PRECEDING AND 1 FOLLOWING /**/ ) AS t_avg
            FROM t_temperature1 ORDER BY high, dt LIMIT 10;
         dt          | high |     t     |       t_avg
---------------------+------+-----------+--------------------
 2018-01-01 00:19:47 | f    |  -1.78712 | -0.966641413668791
 2018-01-01 00:23:30 | f    |  -1.18362 | -0.966641413668791
 2018-01-01 00:34:34 | f    | -0.812903 | -0.966641413668791
 2018-01-01 00:43:27 | f    | -0.749416 | -0.966641413668791
 2018-01-01 00:53:45 | f    |  -1.11721 | -0.966641413668791
 2018-01-01 01:24:18 | f    | -0.149579 | -0.537378480657935
 2018-01-01 02:11:30 | f    |  0.873217 |  0.305551156401634
 2018-01-01 02:38:01 | f    |  0.627603 |  0.305551156401634
 2018-01-01 03:03:48 | f    | -0.129037 |  0.620003839333852
 2018-01-01 04:03:42 | f    |  0.158182 |  0.963712198393685
(10 rows)
```

4.6.2. ウィンドウフレーム *RANGE* モードの距離指定

ウィンドウフレームでこれまでは value PRECEDING および value FOLLOWING という指定が RANGE モードでも使用可能になりました。ウィンドウフレーム内の ORDER BY で指定している列の当該行の値からの距離で含めるかどうかを指定できるようになります。

前項の SQL を今度は前後 1 時間以内のデータについて平均を求めるように書き換えると以下のようになります。

```
db1=# SELECT dt, (high > 600) high, t, avg(t) OVER (
        PARTITION BY (high > 600) ORDER BY dt RANGE BETWEEN
        '1 hour'::interval PRECEDING AND
        '1 hour'::interval FOLLOWING /**/ ) AS t_avg
          FROM t_temperature1 ORDER BY high, dt LIMIT 10;
         dt          | high |     t     |        t_avg
---------------------+------+-----------+---------------------
 2018-01-01 00:19:47 | f    | -0.273468 |  -0.835916876792908
 2018-01-01 00:23:30 | f    | -0.270229 |  -0.835916876792908
```


Copyright © 2018 SRA OSS, Inc. Japan All rights reserved.

```
 2018-01-01 00:34:34 | f    |  -1.23344 |  -0.962530215581258
 2018-01-01 00:43:27 | f    |  -1.94818 |  -0.962530215581258
 2018-01-01 00:53:45 | f    | -0.454267 |  -0.962530215581258
 2018-01-01 01:24:18 | f    |   -1.5956 |  -0.955757999420166
 2018-01-01 02:11:30 | f    |  0.452694 |  -0.441849559545517
 2018-01-01 02:38:01 | f    | -0.859478 | -0.0572671095530192
 2018-01-01 03:03:48 | f    |  0.234983 |   0.238512098789215
 2018-01-01 04:03:42 | f    |   1.12585 |   0.843094613403082
(10 rows)
```

今度は dt = '2018-01-01 01:24:18' の行の t_avg 値は、2018-01-01 00:24:18 から 2018-01-01 02:24:18 までの範囲にあるデータの平均気温値ということです。

4.6.3. ウィンドウフレームの *EXCLUDE* オプション

ウィンドウフレームに EXCLUDE オプションに対応しました。RANGE、ROWS、GROUPS の指定の末尾に以下の EXCLUDE のいずれかを指定して、フレームから除外するものを指定できます。

EXCLUDE CURRENT ROW	当該行を除外します。
EXCLUDE GROUP	当該行が含まれる整列時の同値グループを除外します。
EXCLUDE TIES	当該行が含まれる整列時の同値グループの当該行以外を除外します。
EXCLUDE NO OTHERS	除外をしません。EXCLUDE を指定しない場合と同じです。

4.6.1 節、4.6.2 節のサンプル SQL のコメント記号部分（/**/）にこれらオプション指定を与えて、動作を確認することができます。

4.7. *SCRAM* チャンネルバインド

PostgreSQL 10 から接続時の認証方式に scram-sha-256 が追加されました。PostgreSQL 11 では、この scram-sha-256 を使う際にチャンネルバインド動作をさせることができるようになりました。チャンネルバインドとは SSL（TLS）接続の場合の専用機能です。認証のためのメッセージ取り交わしにおいて当該の SSL 接続固有の情報を含めることで、その内容を他の接続に使いまわせないようにして、いわゆる Man-in-the-middle （中間者）攻撃を防ぐものです。SCRAM チャンネルバインドは ssl 接続で scram-sha-256 認証を使う場合にデフォルトで使われます。

以下の手順で実際に使用してみます。

Copyright © 2018 SRA OSS, Inc. Japan All rights reserved.

（postgresql.conf を編集して以下設定を付与します）

```
listen_addresses = '*'
ssl = on
password_encryption = scram-sha-256
```

（pg_hba.conf の先頭の方に以下設定を付与します）

```
# TYPE   DATABASE     USER          ADDRESS                  METHOD
hostssl db1          test          test1.example.com      scram-sha-256
```

（SSL 接続を受け付けられるように鍵と証明書のファイルを配置します）

```
# su - postgres
$ cd /usr/local/pgsql/data11.0
$ openssl req -new -x509 -days 365 -nodes -text -out server.crt \
    -keyout server.key -subj "/CN=test1.exsample.com"
Generating a 2048 bit RSA private key
.....................................................................+++
.........................+++
writing new private key to 'server.key'
-----
$ chmod 600 server.key
```

（PostgreSQL を再起動します）

```
$ pg_ctl restart
```

（テスト用の適当なユーザを作ります）

```
db1=# CREATE ROLE test LOGIN PASSWORD 'password';
CREATE ROLE
```

（テスト用ユーザで SSL 接続します）

```
$ psql "host=test1.example.com dbname=db1 user=test sslmode=require
scram_channel_binding=tls-unique"                    # 実際は１行
Password for user test:                              # パスワード入力
Pager usage is off.
psql (11beta1)
```


Copyright © 2018 SRA OSS, Inc. Japan All rights reserved.

```
SSL connection (protocol: TLSv1.2, cipher: ECDHE-RSA-AES256-GCM-SHA384,
bits: 256, compression: off)
Type "help" for help.

db1=>                                                    # 無事接続できました
                                # ~/.postgresql/以下に鍵や証明書のファイルを置いていない前提の手順です
```

上記例では psql にて接続文字列を与える形式でデータベースに接続しています。ここで scram_channel_binding というオプションを指定しています。デフォルトは `tls-unique` で、他に空欄と `tls-server-end-point` が指定できます。通常はデフォルト指定で問題ありません。空欄はチャンネルバインディングを使わないという意味です。

さて、SSL 接続を有効にした PostgreSQL 11 に対して scram-sha-256 で接続できたわけですが、このとき SCRAM チャンネルバインディングが使われているか確認することは難しいです。DEBUG レベルを含めて特にログメッセージがは出ませんし、パケットを見ても SSL 上なので解析困難です。

注記
PostgreSQL 11 beta3 バージョン以降で scram_channel_binding オプションは無くなり、 tls-server-end-point 方式のみ固定となりました。

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 © 2018 SRA OSS, Inc. Japan All rights reserved.