PostgreSQL 9.5 検証レポート

1.0.1 版
2016 年 1 月 12 日

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

目次

1. はじめに

本文書は PostgreSQL 9.5 に含まれる主要な新機能を説明し、実際に動作させた検証結果を報告するものです。PostgreSQL 9.5 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。
2015 年 8 月 6 日リリースされた PostgreSQL 9.5 alpha2 を使用して検証を行って、本文書を作成しました。

その後の 2016 年 1 月の正式リリースまでの間で仕様が大きく変わった個所があります。該当部分については、補足記載を追加しました。

2. 概要

PostgreSQL 9.5 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。

- 行単位セキュリティ
- BRIN インデックス
- pg_rewind
- INSERT ... ON CONFLICT ...
- 外部スキーマのインポート
- 外部スキーマの継承
- GROUP BY の拡張
- JSONB に関する演算子と関数

この他にも細かな機能追加や変更が多数あります。全ての変更点の一覧については PostgreSQL 9.5 ドキュメント内のリリースノートに記載されています。また、これらは alpha2 バージョン時点での機能実装です。バージョン 9.5 の本リリースまでにいくつか削除や変更される可能性があります。

3. 検証のためのセットアップ

3.1. ソフトウェア入手

PostgreSQL 9.5 のアルファ版・ベータ版は以下ＵＲＬからダウンロード可能です。

http://www.postgresql.org/download

3.2. 検証環境

検証環境として、HP Pavilion Desktop PC HPE-560jp (CPU: AMD Phenom II X6 1065T / Memory: 16GB, HDD: 500GB SATA 7200rpm) 上の CentOS 6.4 (Linux 2.6.32-431.el6.x86_64)を使用しました。PC 用途マシ

ン上の Linux 環境であり、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

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-9.5alpha2
$ ./configure --prefix=/usr/local/pgsql/9.5alpha2
$ make install-world
```

環境変数を設定するファイルを書き出して、適用します。

```
$ cat > 9.5alpha2.env <<'EOF'
PGHOME=/usr/local/pgsql/9.5alpha2
export PATH=$PGHOME/bin:$PATH
export LD_LIBRARY_PATH=$PGHOME/lib:$LD_LIBRARY_PATH
export PGDATA=/usr/local/pgsql/data9.5alpha2
EOF
$ . 9.5alpha2.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
log_line_prefix = '%t %p '
EOF
```

PostgreSQL を起動します。

```
$ pg_ctl -w start
```

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/9.5alpha2/share/doc/html/
```

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

```
http://www.postgresql.org/docs/9.5/static/
```

4.1. 行単位セキュリティ

以前は、データベースオブジェクト単位や列単位で権限指定のみが可能でしたが、9.5 からは各テーブルにポリシーを定義することで行単位の権限指定が可能になりました。

行単位セキュリティには以下の設定があります。

- テーブルのオブジェクト権限（既存の権限指定）
- ポリシー
- テーブルの ROW LEVEL SECURITY 許可
- row_security パラメータ
- ユーザの BYPASSRLS 属性

> 本節内容は PostgreSQL9.5 正式リリースでは仕様が変更されています。
> 4.1.6. 節と PostgreSQL マニュアルを確認ください。

4.1.1. ポリシーの定義

ポリシーの定義は CREATE POLICY 文で行います。

「どのテーブルの何の操作に対して、誰が行った場合に、指定の行限定やチェックを与える」ということを指定します。

◆ *CREATE POLICY* の構文

```
CREATE POLICY name ON table_name
   [ FOR { ALL | SELECT | INSERT | UPDATE | DELETE } ]
    [ TO { role_name | PUBLIC | CURRENT_USER | SESSION_USER } [, ...] ]
    [ USING ( using_expression ) ]
    [ WITH CHECK ( check_expression ) ]
```

構文要素	意味
name	定義するポリシー名
table_name	ポリシーの定義先テーブル名
FOR { ALL | SELECT | INSERT | UPDATE | DELETE }	適用するクエリの種類。ALL は全てを適用対象とする。
TO { role_name | PUBLIC | CURRENT_USER | SESSION_USER } [, ...]	適用するロール名。カンマ区切りで複数指定可能。 PUBLIC は全ロール、CURRENT_USER は current_user()、SESSION_USER は session_user() の返り値にあたるロールが設定される。
USING (using_expression)	SELECT, UPDATE, DELETE で検索される行の条件式
WITH CHECK (check_expression)	INSERT, UPDATE で作成される行の条件式

4.1.2. ROW LEVEL SECURITY

ポリシーを適用させるため、テーブルに ROW LEVEL SECURITY を許可する必要があります(デフォルトは disable)。テーブルに ROW LEVEL SECURITY を許可すると、スーパーユーザかテーブル所有者以外はポリシーが定義されない限り、検索も挿入もできない状態になります。

◆ ALTER TABLE 文による ROW LEVEL SECURITY を許可する構文

```
ALTER TABLE table_name ENABLE ROW LEVEL SECURITY;
```

4.1.3. row_security パラメータ, BYPASSRLS 属性

ポリシー適用に関係するパラメータとして row_security が、ロール属性に BYPASSRLS 属性が追加されました。row_security は force,on,off のいずれかの値となります。それぞれの値の意味は下表のとおりです。

BYPASSRLS 属性は row_security = off の際に行レベルセキュリティによって保護されている行を検索した際にエラーを回避する権限となっています。一般ユーザ（ロール）はデフォルトでは BYPASSRLS 属性を持ちません。

◆ ***row_security*の設定値**

設定値	意味
force	スーパーユーザとテーブル所有者に対してもポリシーを適用させる 。 BYPASSRLS 属性を持ったロールにもポリシーは適用される。
on (デフォルト)	スーパーユーザとテーブル所有者以外にポリシーを適用させる。 BYPASSRLS 属性を持ったロールにもポリシーは適用される。
off	スーパーユーザとテーブル所有者以外のユーザにおいて、ROW LEVEL SECURITY が許可されているテーブルに対する操作をエラーにする。ただし 、BYPASSRLS 属性を持ったロールはエラーが発生することなく、ポリシーも適用されない（スーパーユーザやテーブル所有者と同じ）。

4.1.4. 行単位セキュリティの利用手順例

本例では id カラムによって行単位セキュリティ設定を行います。

1. まずはスーパーユーザでテーブルを作成し、行単位セキュリティとは別に、テーブルの SELECT 権限をログイン権限のみを持った一般ユーザ user1 に与えます。

```
db1=# CREATE TABLE policy_test_table (id serial, content text);
db1=# INSERT INTO policy_test_table (content) VALUES (md5(generate_series(1,5)::text));
db1=# SELECT * FROM policy_test_table;  -- id = 1..5 の行を持つテーブルを作成
 id |             content
----+----------------------------------
  1 | c4ca4238a0b923820dcc509a6f75849b
  2 | c81e728d9d4c2f636f067f89cc14862c
  3 | eccbc87e4b5ce2fe28308fd9f2a7baf3
  4 | a87ff679a2f3e71d9181a67b7542122c
  5 | e4da3b7fbbce2345d7772b0674a318d5
(5 rows)
db1=# GRANT SELECT ON policy_test_table TO user1;
```

2. row_security = on (デフォルト)であることを確認します。

```
db1=# SHOW row_security;
 row_security
```

```
--------------
 on
(1 row)
```

3. ポリシー適用のため policy_test_table テーブルに ROW LEVEL SECURITY を許可します。（デフォルトは disable）

```
db1=# ALTER TABLE policy_test_table ENABLE ROW LEVEL SECURITY;
```

4. policy1 を policy_test_table テーブルに定義します。

```
db1=# CREATE POLICY policy1 ON policy_test_table FOR ALL TO PUBLIC USING (id <= 2);
CREATE POLICY
```

5. user1 で policy_test_table テーブルのデータを全件検索すると、policy1 (id <= 2) が適用された結果が返ってきます。

```
db1=> SELECT * FROM policy_test_table;
 id |             content
----+----------------------------------
  1 | c4ca4238a0b923820dcc509a6f75849b
  2 | c81e728d9d4c2f636f067f89cc14862c
(2 rows)
```

同一テーブルに対して複数のポリシー定義が存在する場合、全条件式が OR で接続されます。全てのポリシーによる制限が適用されるということです。

4.1.5. ポリシー情報の表示

ポリシーに関するシステムカタログには pg_policy テーブルとそれを基に定義されている pg_policies ビューがあります。また、psql であれば\d コマンドを定義しているテーブルに対して実行すればそのテーブルに定義されているポリシーを確認することができます。

◆ *pg_policy* テーブルによるポリシー定義確認

```
db1=> ¥x on
db1=> SELECT * FROM pg_policy WHERE polname = 'policy1';
polname     | policy1
polrelid    | 17662
polcmd      | *
```

```
  polroles     | {0}
  polqual      | {OPEXPR :opno 523 :opfuncid 149 :opresulttype 16 :opretset false
:opcollid 0 :inputcollid 0 :args ({VAR :varno 1 :varattno 1 :vartype 23 :vartypmod -1
:varcollid 0 :varlevelsup 0 :varnoold 1 :varoattno 1 :location 68} {CONST :consttype 23
:consttypmod -1 :constcollid 0 :constlen 4 :constbyval true :constisnull false :location
74 :constvalue 4 [ 2 0 0 0 0 0 0 0 ]}) :location 71}
  polwithcheck |
```

◆ *pg_policies* ビューによるポリシー定義確認

```
  db1=> ¥x off
  db1=> SELECT * FROM pg_policies;
   schemaname |     tablename      | policyname |   roles    | cmd |   qual    | with_check
  ------------+--------------------+------------+------------+-----+-----------
+------------
   public     | policy_test_table | policy1    | {public}   | ALL | (id <= 2) |
  (1 rows)
```

◆ *\d* コマンドによるポリシー定義確認

```
  db1=> ¥d policy_test_table;
                  Table "public.policy_test_table"
   Column  |  Type   |                           Modifiers
  ---------+---------+---------------------------------------------------------------
   id      | integer | not null default nextval('policy_test_table_id_seq'::regclass)
   content | text    |
  Policies:
    POLICY "policy1" FOR ALL
      USING ((id <= 2))
```

4.1.6. 仕様変更について

以下の点が PostgreSQL9.5 正式リリース（9.5beta2 以降）で変更されています。

- row_security 設定値で force 指定が廃止され、on / off のみとなりました。
- row_security = on であるとき、BYPASSRLS 属性を持つロールで ENABLE ROW LEVEL SECURITY であるテーブルのポリシーを無視するようになりました。
- テーブル単位での FORCE ROW LEVEL SECURITY の指定が加わりました（FORCE であるとテーブル所有者であっても BYPASSRLS 属性を持たない限りポリシーが効きます）。

4.2. BRIN (Block Range INdex) インデックス

PostgreSQL のインデックス方式に BRIN (Block Range INdex) が加わりました。

BRIN は下図のようにテーブルにおける pages_per_range（デフォルト 128）個のヒープブロックごとの最小値と最大値を保存するインデックスです。

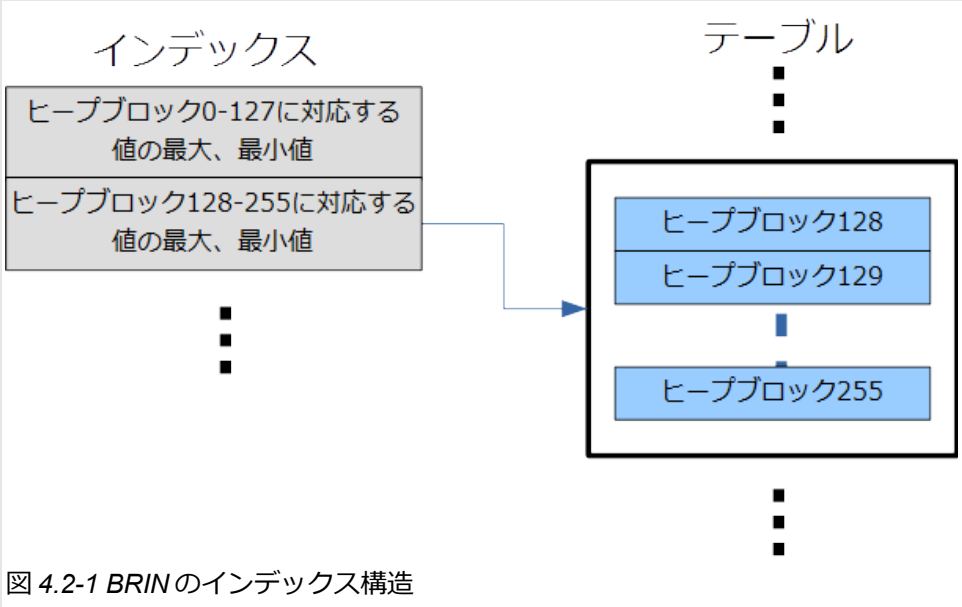


図 4.2-1 BRIN のインデックス構造

4.2.1. BRIN 作成の構文

BRIN インデックスは他方式のインデックスと同様に CREATE INDEX 文で作成します。

```
CREATE INDEX index_name ON table_name USING brin ( column [ ,... ] )
  [ WITH ( page_per_range = value ) ];
```

構文要素	説明
index_name	BRIN インデックス名
table_name	インデックスの対象とするテーブル名
USING brin	インデックス方式に brin を指定
(column [, ..])	インデックスの対象となるカラム
WITH (pages_per_range = *value*)	pages_per_range の値を指定

4.2.2. BRIN, B-Tree, インデックスなしの性能比較

ログデータのタイムスタンプや連番による識別番号のように値が投入順に並ぶカラムにインデックスを付けた場合において、BRIN 利用時と B-Tree 利用時そしてインデックスを用いない時の各性能を比較すると以下の表のようになります。B-Tree に対し BRIN はインデックスファイルサイズ、インデックス作成時間、検索速度（範囲検索）の面で優れています。

性能　＼　インデックス	BRIN	B-Tree	インデックスなし
インデックスファイルサイズ	小	大	なし
インデックス作成時間	短	長	なし
検索速度（等価検索）	やや遅	速	遅
検索速度（範囲検索）	やや速	やや速(BRIN と同程度)	遅

以下、実測例を示していきます。

◆ インデックスファイルサイズ比較

B-Tree はテーブルの 1 行をインデックスの 1 項目として保存しているのに対して、BRIN は複数ブロックの最大値、最小値を 1 項目に保存しています。そのため B-Tree に比べ BRIN のインデックスのサイズは小さいものになることが期待されます。

1 つの int カラムを対象としたインデックスで 10 万件のデータを保存した例を以下に示します。

結果はグラフに示される通り、BRIN のサイズ方が 1 / 40 程度に小さくなりました。

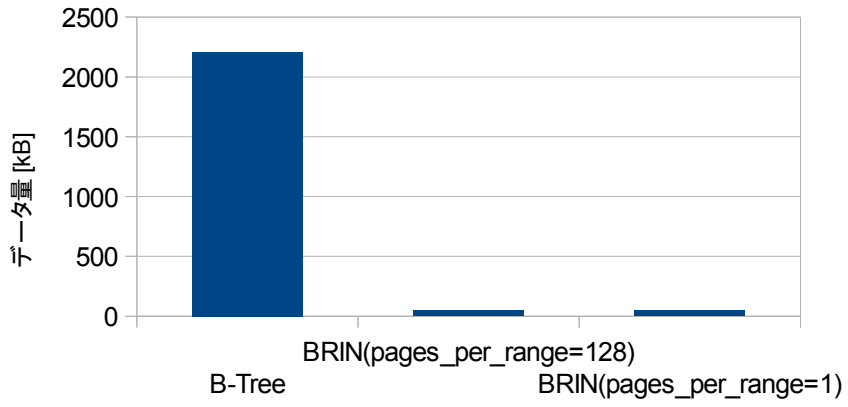


図 *4.2.2-1* サイズ比較

また、pages_per_range の違いは大きく影響しませんでした。本テーブル 10 万件ではテーブルサイズはス数百ブロックに過ぎず、pages_per_range が小さくなったことによりレンジの数が増えた（約 128 倍）としてもレギュラーページが 1 ブロック(8kB)増えただけで収まったからです。内部構造については 4.2.4. 節 、4.2.5. 節で説明します。

本例の具体的なクエリと結果は以下の通りです。

```
db1=> CREATE TABLE t1(i int);
db1=> INSERT INTO t1 SELECT * FROM generate_series(1, 100000);     -- 10万件を投入
db1=> CREATE INDEX btreeidx ON t1 (i);
db1=> CREATE INDEX brinidx ON t1 USING brin (i);
db1=> CREATE INDEX brinidx1 ON t1 USING brin (i) WITH (pages_per_range=1);
db1=> ¥di+
List of relations
Schema | Name | Type | Owner | Table | Size | Description
--------+----------+-------+----------+-------+---------+-------------
public | btreeidx | index | postgres | t1 | 2208 kB |
public | brinidx  | index | postgres | t1 | 48 kB   |
public | brinidx1 | index | postgres | t1 | 56 kB   |
(3 row)
```

◆ インデックス作成時間

10 万件の単調増加データと同じく 10 万件のランダム順データに対して B-Tree、 BRIN の作成時間を比較します。

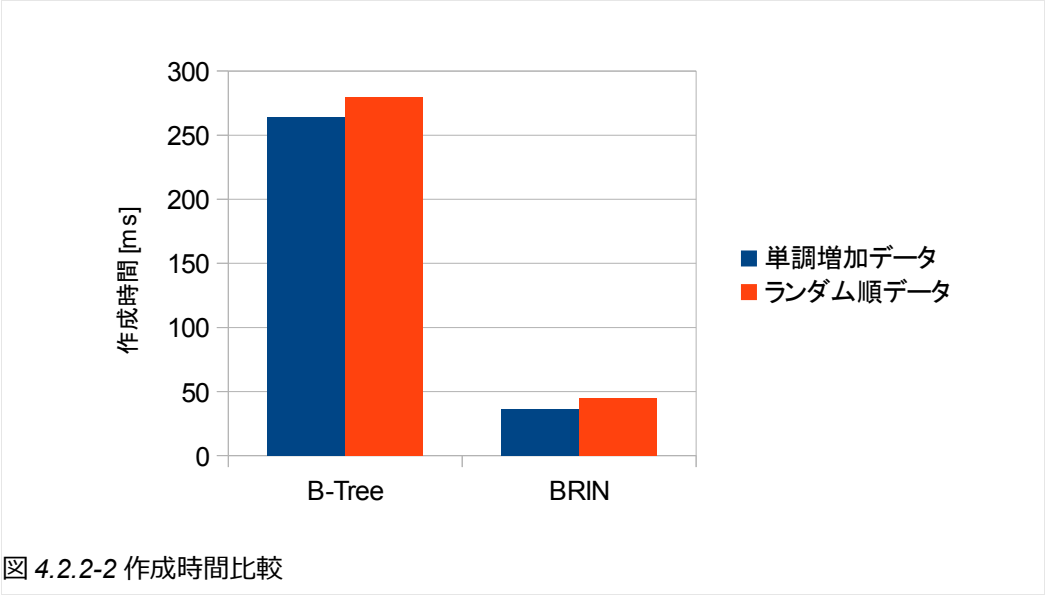


図 *4.2.2-2* 作成時間比較

単調増加データは generate_series(1,100000)::text、ランダム順データは md5(generate_series(1,100000)::text) を採用しています。

結果は、B-Tree に比べ、BRIN の作成時間は圧倒的に短くなりました。また、どちらのインデックスの作成時間も単調増加のデータかそうでないかは作成時間に大きく影響していませんでした。

本例の具体的なクエリと実行結果は以下の通りです。バッファヒット無し状態で比較しています。

```
db1=> ¥timing                                  -- 実行時間表示
db1=> CREATE TABLE t1 (a int, b text); -- a は単調増加データ、b はランダム順データ
db1=> INSERT INTO t1 VALUES
       (generate_series(1,100000)::text, md5(generate_series(1,100000)::text));

-- 単調増加データに対するインデックス作成時間比較
db1=> CREATE INDEX btreeidx1 ON t1 (a);
CREATE INDEX
Time: 263.916 ms
db1=> CREATE INDEX brinidx1 ON t1 USING BRIN (a);
CREATE INDEX
Time: 36.161 ms
-- ここで共有バッファの影響を避けるためサーバ再起動、テーブル再作成
-- ランダム順データに対するインデックス作成時間比較
db1=> CREATE INDEX btreeidx2 ON t1 (b);
CREATE INDEX
Time: 279.815 ms
db1=> CREATE INDEX brinidx2 ON t1 USING BRIN (b);
CREATE INDEX
Time: 45.017 ms
```

◆ 検索速度

単一整数カラムを持つテーブルに 1 から 1,000,000 の値をもつ 100 万行を格納して、以下の条件で検索を行い、速度を比較しました。

- = 500000 (等価検索)
- between 100000 and 200000 (範囲検索)
- between 100000 and 600000 (範囲検索)

各検索時間は以下の通りです。

- 等価検索では B-Tree が圧倒的に速い。ただし、BRIN もインデックスなしに比べれば十分速い。
- 範囲検索では B-Tree よりも BRIN のほうが僅かだが速い。

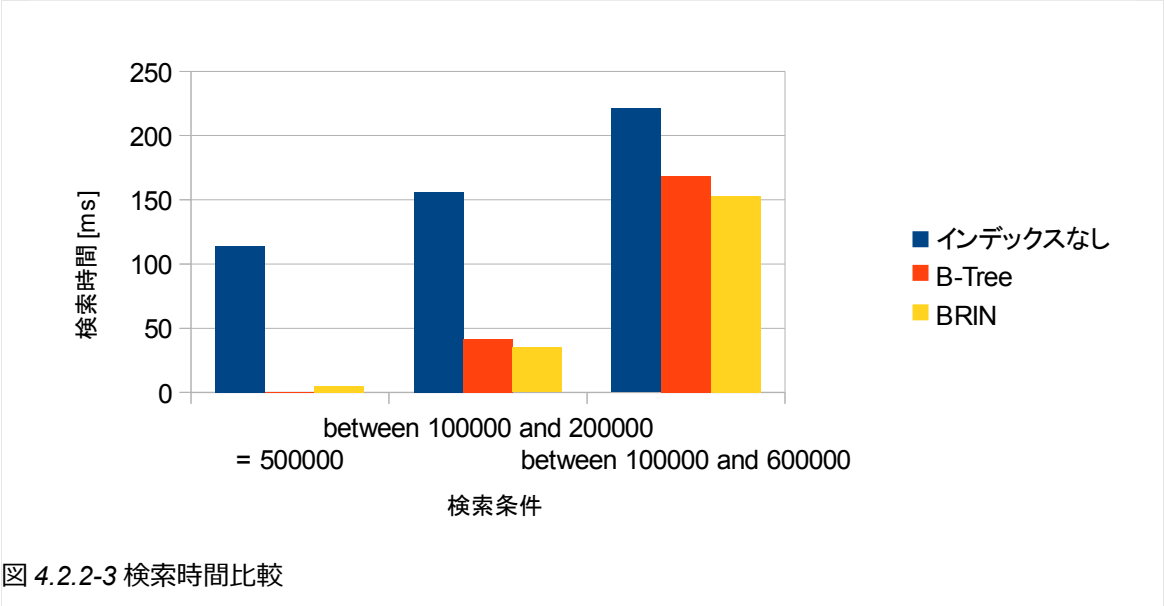


図 *4.2.2-3* 検索時間比較

本例のクエリと実行結果は以下の通りです。バッファヒットするようになり所要時間のバラつきがなくなった時点の値を採用しています。

```
db1=> CREATE TABLE t1 (i int);
CREATE TABLE
db1=> INSERT INTO t1 VALUES (generate_series(1,1000000));
INSERT 0 1000000

-- 下記はインデックスなしで検索
db1=> EXPLAIN ANALYZE SELECT * FROM t1 WHERE i = 500000;
                                QUERY PLAN
-------------------------------------------------------------------------------------------------------
--------------
 Seq Scan on t1  (cost=0.00..18529.69 rows=5642 width=4) (actual time=59.418..113.591
rows=1 loops=1)
   Filter: (i = 500000)
   Rows Removed by Filter: 999999
 Planning time: 0.726 ms
 Execution time: 113.642 ms
```

```
  (5 rows)

  db1=> EXPLAIN ANALYZE SELECT * FROM t1 WHERE i BETWEEN 100000 AND 200000;
                                    QUERY PLAN
  ------------------------------------------------------------------------------------------
-------------------
   Seq Scan on t1  (cost=0.00..21350.62 rows=5642 width=4) (actual time=10.605..148.368
rows=100001 loops=1)
     Filter: ((i >= 100000) AND (i <= 200000))
     Rows Removed by Filter: 899999
   Planning time: 0.099 ms
   Execution time: 156.352 ms
  (5 rows)

  db1=> EXPLAIN ANALYZE SELECT * FROM t1 WHERE i BETWEEN 100000 AND 600000;
                                    QUERY PLAN
  ------------------------------------------------------------------------------------------
-------------------
   Seq Scan on t1  (cost=0.00..21350.62 rows=5642 width=4) (actual time=10.976..181.408
rows=500001 loops=1)
     Filter: ((i >= 100000) AND (i <= 600000))
     Rows Removed by Filter: 499999
   Planning time: 0.094 ms
   Execution time: 221.290 ms
  (5 rows)

  -- サーバ再起動、テーブル再作成
  -- B-Tree を利用した検索
  db1=> CREATE INDEX btreeidx ON t1(i);
  CREATE INDEX

  db1=> EXPLAIN ANALYZE SELECT * FROM t1 WHERE i = 500000;
                                      QUERY PLAN
  ------------------------------------------------------------------------------------------
-----------------------------
```

```
   Bitmap Heap Scan on t1  (cost=95.17..4793.05 rows=5000 width=4) (actual
time=0.109..0.110 rows=1 loops=1)
     Recheck Cond: (i = 500000)
     Heap Blocks: exact=1
     ->  Bitmap Index Scan on btreeidx  (cost=0.00..93.92 rows=5000 width=0) (actual
time=0.094..0.094 rows=1 loops=1)
           Index Cond: (i = 500000)
   Planning time: 0.391 ms
   Execution time: 0.184 ms
   (7 rows)

   db1=> EXPLAIN ANALYZE SELECT * FROM t1 WHERE i BETWEEN 100000 AND 200000;
                                        QUERY PLAN
   ------------------------------------------------------------------------------------------
-------------------------------------
   Bitmap Heap Scan on t1  (cost=107.67..4818.05 rows=5000 width=4) (actual
time=19.079..33.577 rows=100001 loops=1)
     Recheck Cond: ((i >= 100000) AND (i <= 200000))
     Heap Blocks: exact=443
     ->  Bitmap Index Scan on btreeidx  (cost=0.00..106.42 rows=5000 width=0) (actual
time=19.014..19.014 rows=100001 loops=1)
           Index Cond: ((i >= 100000) AND (i <= 200000))
   Planning time: 0.179 ms
   Execution time: 41.494 ms
   (7 rows)

   db1=> EXPLAIN ANALYZE SELECT * FROM t1 WHERE i BETWEEN 100000 AND 600000;
                                        QUERY PLAN
   ------------------------------------------------------------------------------------------
-------------------------------------
   Bitmap Heap Scan on t1  (cost=107.67..4818.05 rows=5000 width=4) (actual
time=59.356..130.095 rows=500001 loops=1)
     Recheck Cond: ((i >= 100000) AND (i <= 600000))
     Heap Blocks: exact=2213
     ->  Bitmap Index Scan on btreeidx  (cost=0.00..106.42 rows=5000 width=0) (actual
```

```
time=58.986..58.986 rows=500001 loops=1)
        Index Cond: ((i >= 100000) AND (i <= 600000))
  Planning time: 0.167 ms
  Execution time: 168.404 ms
  (7 rows)

  -- BRIN を利用した検索
  db1=> CREATE INDEX brinidx ON t1 USING BRIN (i);
  CREATE INDEX
  db1=> EXPLAIN ANALYZE SELECT * FROM t1 WHERE i = 500000;
                                    QUERY PLAN
  ------------------------------------------------------------------------------------------
------------------------------
   Bitmap Heap Scan on t1  (cost=50.75..4748.63 rows=5000 width=4) (actual
time=2.018..4.593 rows=1 loops=1)
     Recheck Cond: (i = 500000)
     Rows Removed by Index Recheck: 28927
     Heap Blocks: lossy=128
     ->  Bitmap Index Scan on brinidx  (cost=0.00..49.50 rows=5000 width=0) (actual
time=0.173..0.173 rows=1280 loops=1)
        Index Cond: (i = 500000)
   Planning time: 0.241 ms
   Execution time: 4.642 ms
  (8 rows)

  db1=> EXPLAIN ANALYZE SELECT * FROM t1 WHERE i BETWEEN 100000 AND 200000;
                                    QUERY PLAN
  ------------------------------------------------------------------------------------------
------------------------------
   Bitmap Heap Scan on t1  (cost=63.25..4773.63 rows=5000 width=4) (actual
time=1.688..26.126 rows=100001 loops=1)
     Recheck Cond: ((i >= 100000) AND (i <= 200000))
     Rows Removed by Index Recheck: 15711
     Heap Blocks: lossy=512
     ->  Bitmap Index Scan on brinidx  (cost=0.00..62.00 rows=5000 width=0) (actual
```

```
time=0.061..0.061 rows=5120 loops=1)
        Index Cond: ((i >= 100000) AND (i <= 200000))
  Planning time: 0.215 ms
  Execution time: 34.704 ms
 (8 rows)

 db1=> EXPLAIN ANALYZE SELECT * FROM t1 WHERE i BETWEEN 100000 AND 600000;
                                    QUERY PLAN
 ------------------------------------------------------------------------------------------
--------------------------------
  Bitmap Heap Scan on t1  (cost=63.25..4773.63 rows=5000 width=4) (actual
time=1.838..114.476 rows=500001 loops=1)
    Recheck Cond: ((i >= 100000) AND (i <= 600000))
    Rows Removed by Index Recheck: 20703
    Heap Blocks: lossy=2304
    ->  Bitmap Index Scan on brinidx  (cost=0.00..62.00 rows=5000 width=0) (actual
time=0.187..0.187 rows=23040 loops=1)
        Index Cond: ((i >= 100000) AND (i <= 600000))
  Planning time: 0.047 ms
  Execution time: 153.752 ms
 (8 rows)
```

4.2.3. ランダム配置データに対する *BRIN* 適用

BRIN は、その仕組み上、データの物理的な配置がおおむねインデックス対象カラム値の順になっていないと効果が出ないはずです。

以下のようにインデックスカラム値の順にデータを投入したテーブル（整列データ）、おおむね値の順に投入したテーブル（やや乱れた整列データ）、値と位置がランダムなテーブル（ランダム順データ）を作成して、検索性能を比較しました。

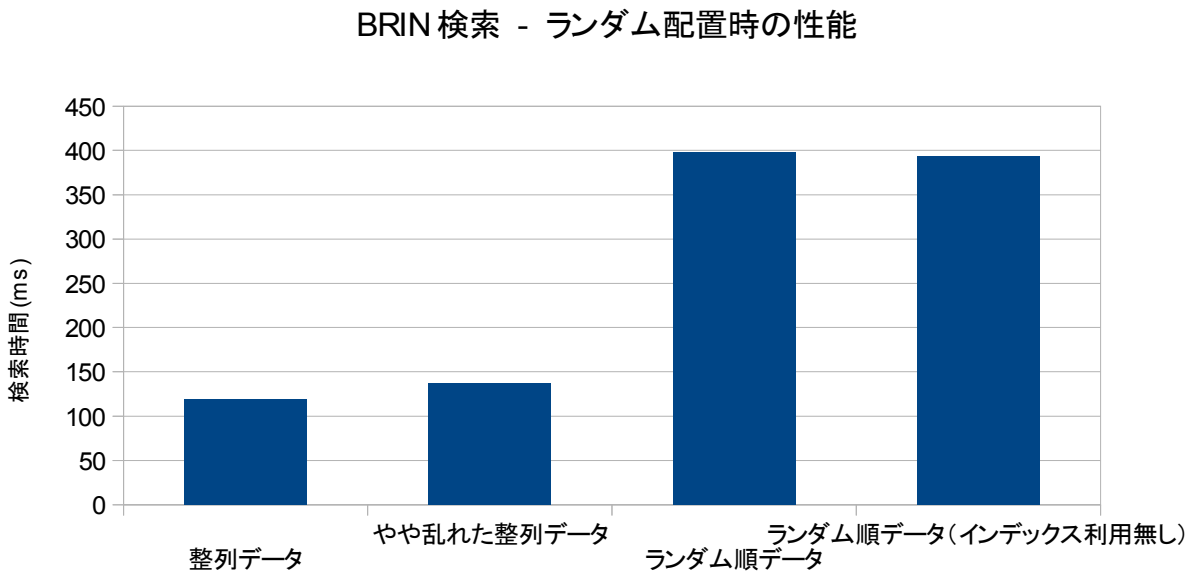
```
db1=# CREATE TABLE t_seq (id int primary key, v text, ts timestamp);
db1=# INSERT INTO t_seq SELECT g, md5(g::text),
       '2015-09-01'::timestamp + (g || 'sec')::interval
        FROM generate_series(1, 1000000) g;
db1=# CREATE INDEX ON t_seq USING brin (ts);
```

```
db1=# CREATE TABLE t_semirand (id int primary key, v text, ts timestamp);
db1=# INSERT INTO t_semirand SELECT g, md5(g::text),
       '2015-09-01'::timestamp + ((g + random() * 50000)::int || 'sec')::interval
       FROM generate_series(1, 1000000) g;
db1=# CREATE INDEX ON t_semirand USING brin (ts);

db1=# CREATE TABLE t_rand (id int primary key, v text, ts timestamp);
db1=# INSERT INTO t_rand SELECT g, md5(g::text),
       '2015-09-01'::timestamp + ((random() * 1000000)::int || 'sec')::interval
       FROM generate_series(1, 1000000) g;
db1=# CREATE INDEX ON t_rand USING brin (ts);
```

範囲検索の性能を比較すると以下グラフに示す結果となりました。

ランダム順データはインデックス無しと変らない検索時間です。また、多少整列が乱れていても急激に検索時間が増えることは無いことが分かります。



比較した SQL 問い合わせとプラン内容は以下の通りです。バッファヒットして応答時間にバラつきがなくなった時点の結果を採用しています。ヒット件数が異なりますが、その差異は 1%未満です。「Heap Blocks: lossy=....」という箇所が読み込みを必要とした対象テーブルのページ数で、各プランの応答時間差の要因といえます。

```
db1=# EXPLAIN ANALYZE SELECT * FROM t_seq WHERE ts BETWEEN '2015-09-05' AND '2015-09-07';
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on t_seq  (cost=1800.17..13763.01 rows=174456 width=45) (actual time=2.294..90.775 rows=172801 loops=1)
   Recheck Cond: ((ts >= '2015-09-05 00:00:00'::timestamp without time zone) AND (ts <= '2015-09-07 00:00:00'::timestamp without time zone))
   Rows Removed by Index Recheck: 5247
   Heap Blocks: lossy=1664
   ->  Bitmap Index Scan on t_seq_ts_idx  (cost=0.00..1756.56 rows=174456 width=0) (actual time=0.550..0.550 rows=16640 loops=1)
         Index Cond: ((ts >= '2015-09-05 00:00:00'::timestamp without time zone) AND (ts <= '2015-09-07 00:00:00'::timestamp without time zone))
 Planning time: 0.218 ms
 Execution time: 118.884 ms
(8 rows)

db1=# EXPLAIN ANALYZE SELECT * FROM t_semirand WHERE ts BETWEEN '2015-09-05' AND '2015-09-07';
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on t_semirand  (cost=1806.88..13779.53 rows=175110 width=45) (actual time=3.649..110.097 rows=172765 loops=1)
   Recheck Cond: ((ts >= '2015-09-05 00:00:00'::timestamp without time zone) AND (ts <= '2015-09-07 00:00:00'::timestamp without time zone))
   Rows Removed by Index Recheck: 60067
   Heap Blocks: lossy=2176
   ->  Bitmap Index Scan on t_semirand_ts_idx1  (cost=0.00..1763.10 rows=175110 width=0) (actual time=0.463..0.463 rows=21760 loops=1)
         Index Cond: ((ts >= '2015-09-05 00:00:00'::timestamp without time zone) AND (ts <= '2015-09-07 00:00:00'::timestamp without time zone))
 Planning time: 0.153 ms
```

```
   Execution time: 136.742 ms
  (8 rows)

  b1=# EXPLAIN ANALYZE SELECT * FROM t_rand WHERE ts BETWEEN '2015-09-05' AND '2015-09-
07';
                                                                  QUERY PLAN
  ------------------------------------------------------------------------------------------
------------------------------------------------------
   Bitmap Heap Scan on t_rand  (cost=1802.13..13767.84 rows=174647 width=45) (actual
time=2.899..371.217 rows=173157 loops=1)
     Recheck Cond: ((ts >= '2015-09-05 00:00:00'::timestamp without time zone) AND (ts <=
'2015-09-07 00:00:00'::timestamp without time zone))
     Rows Removed by Index Recheck: 826843
     Heap Blocks: lossy=9346
     ->  Bitmap Index Scan on t_rand_ts_idx  (cost=0.00..1758.47 rows=174647 width=0)
(actual time=2.798..2.798 rows=94720 loops=1)
           Index Cond: ((ts >= '2015-09-05 00:00:00'::timestamp without time zone) AND (ts <=
'2015-09-07 00:00:00'::timestamp without time zone))
   Planning time: 0.233 ms
   Execution time: 398.147 ms
  (8 rows)
  -- 下記はインデックスなしで検索
  db1=# EXPLAIN ANALYZE SELECT * FROM t_rand WHERE ts BETWEEN '2015-09-05' AND '2015-09-
07';
                                                            QUERY PLAN
  ------------------------------------------------------------------------------------------
-----------------------------------------------
   Seq Scan on t_rand  (cost=0.00..24346.00 rows=174647 width=45) (actual
time=0.086..366.294 rows=173157 loops=1)
     Filter: ((ts >= '2015-09-05 00:00:00'::timestamp without time zone) AND (ts <= '2015-
09-07 00:00:00'::timestamp without time zone))
     Rows Removed by Filter: 826843
   Planning time: 0.346 ms
   Execution time: 393.757 ms
  (5 rows)
```

4.2.4. BRIN のブロック構造

BRIN は下図のようにメタページ、レンジマップ、レギュラーページで構成されています。

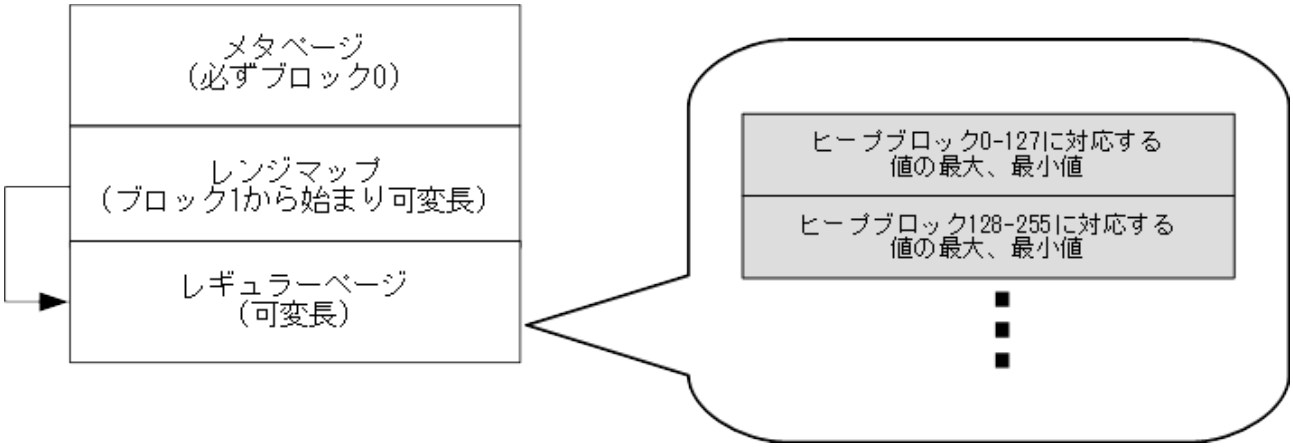


図 4.2.3-1 BRIN のブロック構造

ブロック中の要素の意味は以下の通りです。

- メタページ

  バージョン番号、pages_per_range、レンジマップのブロック番号が含まれます。
- レンジマップ

  BRIN インデックスの本体(レギュラーページ)へのインデックスが含まれます。
- レギュラーページ

  BRIN インデックスの本体が含まれます。各タプルごとに pages_per_range 個のブロック内の最大値と最小値を格納しています。

4.2.5. BRIN が持つデータの取得

BRIN を作成し、インデックスにどのようなデータが保存されているかを pageinspect モジュールを用いて取得します。9.5 では pageinspect モジュールに BRIN のデータの情報を取得する関数が追加されています。

pageinspect の関数を使う前にモジュールのインストールと BRIN の作成を行います。

```
postgres=# CREATE EXTENSION pageinspect;


db1=> CREATE TABLE t1(i int, j int);
db1=> INSERT INTO t1 SELECT * FROM generate_series(1, 100000);
db1=> CREATE INDEX brinidx ON t1 USING brin (i);          -- BRIN でインデックス作成
```

◆ *brin_metapage_info* 関数によるメタページのデータ取得

pagesperrange が pages_per_range を示します。

```
db1=> SELECT * FROM brin_metapage_info(get_raw_page('brinidx', 0));
  magic     | version | pagesperrange | lastrevmappage
------------+---------+---------------+----------------
 0xA8109CFA |       1 |           128 |              1
(1 row)
```

◆ *brin_revmap_data* 関数によるレンジマップのデータ取得

pages が存在する各レギュラーページの場所を示します。

```
db1=> SELECT * FROM brin_revmap_data(get_raw_page('brinidx', 1)) LIMIT 10;
 pages
-------
 (2,1)
 (2,2)
 (2,3)
 (2,4) -- 実際に存在するレギュラーページ（およびオフセット）はここまで
 (0,0) -- この行以降はレギュラーページは存在せず、(0,0)が続く
 (0,0)
後略
```

◆ *brin_page_items* 関数によるレギュラーページのデータ取得

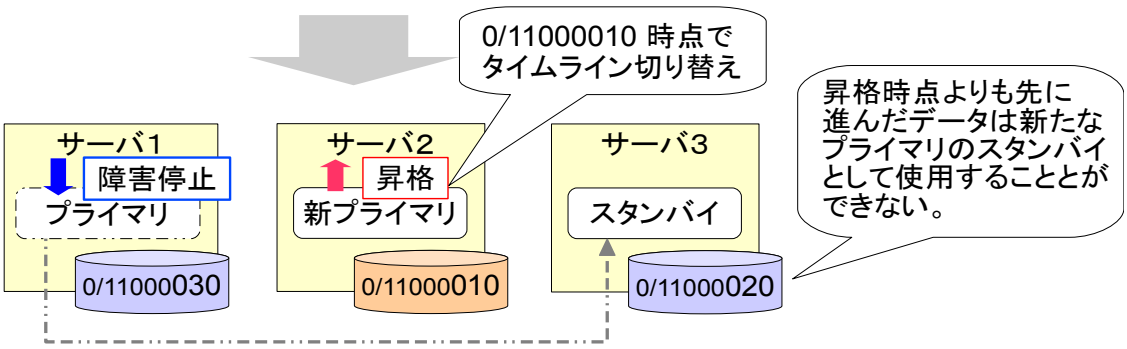
itemoffset が各ブロックグループの番号、value が各ブロックグループの最小値と最大値を示します。

```
db1=> SELECT * FROM brin_page_items(get_raw_page('brinidx', 2),'brinidx');
 itemoffset | blknum | attnum | allnulls | hasnulls | placeholder |       value
------------+--------+--------+----------+----------+-------------+-------------------
          1 |      0 |      1 | f        | f        | f           | {1 .. 28928}
          2 |    128 |      1 | f        | f        | f           | {28929 .. 57856}
          3 |    256 |      1 | f        | f        | f           | {57857 .. 86784}
          4 |    384 |      1 | f        | f        | f           | {86785 .. 100000}
(4 rows)
```

4.3. pg_rewind

ストリーミングレプリケーションでフェイルオーバした後に、旧プライマリサーバをスタンバイとして復帰させるには、多くの場合、新たなプライマリサーバからベースバックアップを取得して、サーバを再構築することが必要でした。

なぜなら、旧プライマリサーバにデータが残っていたとしても、新プライマリサーバが昇格してタイムラインが切り替わった時点よりも、トランザクションログ書き込み位置が先に進んでしまっている場合、新プライマリのトランザクション情報を受け取って追従することができないからです。これは昇格しなかった別のスタンバイサーバでトランザクションログ書き込み位置がより先行していた場合も同じで、そのスタンバイサーバは新プライマリに追従することができません。下図はこのことを模式図にあらわしたものです。



pg_rewind は、新たなプライマリサーバに追従するために必要なだけトランザクションログ書き込み位置を巻き戻して、残されたデータをスタンバイ用に利用可能にするツールです。復旧したいサーバ上で実行します。プライマリにサーバに接続して、必要に応じて巻き戻しとファイルの取得を行い、ローカル側のデータを補正します。

◆ pg_rewind 構文

```
pg_rewind [option...] {-D | --target-pgdata} directory
                      {--source-pgdata=directory | --source-server=connstr}
```

オプション	意味
-D --target-pgdata	巻き戻しを行うデータベースクラスタをディレクトリ指定。
--source-pgdata	追従したいプライマリサーバのデータベースクラスタディレクトリを指定。同ホスト内に在る場合に使用する。
--source-server	追従したいプライマリサーバを指定。サーバは libpq の接続文字列形式で指定する。

4.3.1. pg_rewind を利用するための条件

pg_rewind を利用するためには、postgresql.conf 設定で以下の設定が必要です。

```
full_page_writes = on （デフォルト）

wal_log_hints = on （デフォルトは off）
        もしくは initdb 時にチェックサムを有効化（デフォルトは無効）
```

また、新たなスタンバイにするデータベースクラスタは、最後に正常停止したものでなければいけません。最後にクラッシュ終了したのであれば、１回起動後、正常停止しておきます。

さらに、当然のことながら、新プライマリが昇格した時点以降の WAL アーカイブが必要となります。

4.3.2. コマンド実行手順

レプリケーションによりマスターとスタンバイの関係にあるサーバクラスタに対してフェイルオーバー発生後のスタンバイ復帰までのシナリオを例にコマンド実行例を示します。

1. プライマリサーバがクラッシュ
2. スタンバイサーバをプライマリに昇格
3. 旧プライマリサーバに pg_rewind を使用し、新プライマリサーバに追随できるようにする
4. 旧プライマリサーバをスタンバイとして復帰

実行例では以下のサーバを用います。

ホスト名	server1	server2
最初の役割	プライマリ	スタンバイ

◆ **実行例**

```
server1$ pg_ctl stop -m i     # プライマリのクラッシュ停止を模す
server2$ pg_ctl promote       # スタンバイをマスターに昇格
server1$ pg_ctl start -w
                # pg_rewind 利用前に正常停止させる必要があるので旧プライマリを一旦起動
server1$ pg_ctl stop -w       # 正常停止
server1$ pg_rewind -D $PGDATA --source-server='host=server2 port=5432 user=postgres'
                # 旧プライマリに pg_rewind を使用
```

```
server1$ vi $PGDATA/postgresql.conf
              # 設定ファイルは pg_rewind によって上書きされるので必要に応じて修正

server1$ vi $PGDATA/recovery.conf    # recovery.conf 作成、以下を記述
  standby_mode = 'on'
  primary_conninfo = 'user=postgres port=5432'

server1$ pg_ctl start -w             # リカバリモード起動
```

ここまでの操作で、元プライマリサーバを既存データベースクラスタディレクトリを使って、スタンバイとして復帰させることができました。

4.4. INSERT ... ON CONFLICT ...

INSERT でユニーク制約などによる衝突が起きた時に別のアクションを実行させる構文が追加されました。

現在、別のアクションには「何もしない」か「UPDATE」が選択できます。別のアクションに「UPDATE」を選択することで、いわゆる UPSERT（挿入しようとした行が既に存在すれば更新する）が実現できます。

追加された構文は以下の通りです。

```
INSERT INTO ... ON CONFLICT [ { (column, ...) | ON CONSTRAINT constraint } ]
 { DO NOTHING | DO UPDATE SET ...}
```

構文要素	説明
(column, ...)	衝突したときに別アクションを実行するカラム
ON CONSTRAINT constraint	衝突したときに別アクションを実行する制約
DO NOTHING	別アクションとして何もしない
DO UPDATE	別アクションとして UPDATE を実行する

◆ 利用例

```
db1=> CREATE TABLE t1 (id int UNIQUE, name text);        -- id カラムにユニーク制約付与
CREATE TABLE
db1=> INSERT INTO t1 VALUES (1, 'Tom');
INSERT 0 1
db1=> INSERT INTO t1 VALUES (2, 'Kate');
INSERT 0 1                          -- id=2 はユニーク制約違反ではないのでそのまま挿入される
```

```
db1=> SELECT * FROM t1;
 id | name
----+------
  1 | Tom
  2 | Kate
(2 rows)
db1=> INSERT INTO t1 VALUES (1, 'John');
psql:/tmp/upsert/test.sql:6: ERROR:  duplicate key value violates unique constraint
"t1_id_key"
DETAIL:  Key (id)=(1) already exists.              -- id=1 はユニーク制約違反なのでエラー
db1=> INSERT INTO t1 VALUES (1, 'John') ON CONFLICT (id)
       DO UPDATE SET name = EXCLUDED.name;
                    -- ON CONFLICT 文で id カラムに対する衝突が生じた際に UPDATE を実行
INSERT 0 1          -- 成功。このとき表示が INSERT であることに注意。
db1=> SELECT * FROM t1;
 id | name
----+------
  2 | Kate
  1 | John
(2 rows)
```

4.5. 外部スキーマのインポート

外部サーバのスキーマを参照側サーバのスキーマにインポートできるようになりました。以前は外部テーブルを用いるために、参照側で被参照側のテーブルを別途個別に定義する必要がありました。

以前までの方法も含め、被参照側テーブルを参照側サーバのスキーマに定義する方法を以下に示します。

まず、事前に外部テーブルを用いるために以下の準備をしておきます。foreign_db データベースには、public スキーマ上にいくつかのテーブルが定義されているものとします。

```
-- postgres_fdw 拡張モジュールのインストール
postgres=# CREATE EXTENSION postgres_fdw;
-- CREATE SERVER 文で外部サーバオブジェクトの定義
db1=> CREATE SERVER foreign_server FOREIGN DATA WRAPPER postgres_fdw
     OPTIONS (host '192.83.123.89', port '5432', dbname 'foreign_db');
```

```
-- CREATE USER MAPPING 文でユーザマッピングの定義
db1=> CREATE USER MAPPING FOR local_user SERVER foreign_server
      OPTIONS (user 'foreign_user', password 'password');
```

◆ 個別に外部テーブルを参照側のスキーマ *remote* に定義する例（以前の方法）

```
-- インポート先の remote スキーマを作成
db1=> CREATE SCHEMA remote;
-- 外部サーバの public.customers テーブルを remote スキーマに定義
db1=> CREATE FOREIGN TABLE remote.customers (
        id int NOT NULL, name text, address text)
        SERVER foreign_server OPTIONS (schema_name 'public');
-- 外部サーバの public.customers テーブルを products スキーマに定義
db1=> CREATE FOREIGN TABLE remote.products (
        id int NOT NULL, name text, price int)
        SERVER foreign_server OPTIONS (schema_name 'public');
-- 以下各テーブルごと定義する
```

◆ 参照側のスキーマ *remote* に外部サーバのスキーマ *public* をインポートする例

外部サーバのスキーマのインポートが可能になったことで参照先のテーブル定義が必要なくなりました。

```
-- インポート先の remote スキーマを作成
db1=> CREATE SCHEMA remote;
-- 外部サーバの pulic スキーマに属するオブジェクトを remote スキーマにインポート
db1=> IMPORT FOREIGN SCHEMA public FROM SERVER foreign_server INTO remote;
-- remote スキーマのオブジェクトを確認
db1=> ¥dE remote.*
               List of relations
 Schema |    Name    |      Type       | Owner
--------+------------+-----------------+------
 remote | customers  | foreign table | user1
 remote | products   | foreign table | user1
...
```

4.6. 外部テーブル継承

外部テーブルをローカルテーブルとで継承関係を設定できるようになりました。

リモートテーブルを親テーブルとしてローカルに子テーブルを作る場合は、構文は通常のテーブル継承と同様の書き方になります。

```
db1=> CREATE TABLE local_customers () INHERITS (remote.customers);
CREATE TABLE
```

以下のように複数テーブルに跨る透過的な検索ができます。

```
db1=# EXPLAIN SELECT * FROM remote.customers;
                        QUERY PLAN
-----------------------------------------------------------------------
 Append  (cost=100.00..155.20 rows=1740 width=68)
   ->  Foreign Scan on customers  (cost=100.00..136.70 rows=890 width=68)
   ->  Seq Scan on local_customers  (cost=0.00..18.50 rows=850 width=68)
(3 rows)
```

リモートテーブルを子テーブルとしてローカルの親テーブルに紐づけるには、あらかじめ同じカラムを含む親テーブルを定義したうえで、以下のように実行します。この ALTER TABLE...INHERIT 構文自体は以前からありました。

```
db1=> ALTER TABLE remote.products INHERIT products ;
ALTER TABLE
db1=> EXPLAIN SELECT * FROM products ;
                           QUERY PLAN
-----------------------------------------------------------------------------------
 Append  (cost=0.00..150.95 rows=1366 width=36)
   ->  Seq Scan on products  (cost=0.00..0.00 rows=1 width=36)
   ->  Foreign Scan on products products_1  (cost=100.00..150.95 rows=1365 width=36)
(3 rows)
```

これらのリモートテーブルを参照したテーブル継承定義は、あくまで定義したローカルデータベース上での定義に過ぎません。リモートデータベース上のテーブル定義には何ら影響を与えません。ローカルデータベース上（上記例では db1）での SQL 結果にだけ影響を与えるものです。

4.7. GROUP BY の拡張

以前はカラム列を指定した単一のグループ化のみ可能でしたが、9.5 からは複数のグループ化を同時に行うことが可能になりました。

複数グループの指定形式は３つあります。

- GROUPING SETS
- CUBE
- ROLLUP

既存のグルーピングと各形式の違いを以下に示します。また、例で用いるテーブル内容は下の通りです。

```
db1=> SELECT * FROM sales;
customer | product | num
----------+---------+-----
 Tom      | egg     |  10
 Tom      | apple   |  20
 John     | egg     |  15
 John     | egg     |   5
(4 rows)
```

◆ 既存のグルーピング

```
db1=> SELECT customer, sum(num) FROM sales GROUP BY customer;    -- 単一カラムを指定
 customer | sum
----------+-----
 Tom      |  30
 John     |  20
(2 rows)
db1=> SELECT customer, product, sum(num) FROM sales GROUP BY customer, product;
                                                                  -- 複数カラムを指定
 customer | product | sum
----------+---------+-----
 Tom      | apple   |  20
 Tom      | egg     |  10
 John     | egg     |  20
(3 rows)
```

◆ *GROUPING SETS*

GROUPING SETS 形式は複数のカラム列を指定することで同時に複数のグループ化を行います。

```
db1=> SELECT customer, product, sum(num) FROM sales
        GROUP BY GROUPING SETS ((customer), (customer, product), ());
             -- (customer), (customer, product), ()のグループ化を同時に行なっている。
             -- ()はGROUP BYしない結果を返す。
 customer | product | sum
----------+---------+-----
 John     | egg     |  20
 John     |         |  20
 Tom      | apple   |  20
 Tom      | egg     |  10
 Tom      |         |  30
          |         |  50
(6 rows)
```

◆ *CUBE*

CUBE は指定したカラムの全組み合わせをグループ化します。例えば、CUBE (customer, product) は GROUPING SETS ((), (customer), (product), (customer, product)) に等しい表現となります。

```
db1=> SELECT customer, product, sum(num) FROM sales GROUP BY CUBE (customer, product);
    -- (), (customer), (product), (customer, product) のグループ化を同時に行なっている。
 customer | product | sum
----------+---------+-----
 John     | egg     |  20
 John     |         |  20
 Tom      | apple   |  20
 Tom      | egg     |  10
 Tom      |         |  30
          |         |  50
          | apple   |  20
          | egg     |  30
(8 rows)
```

◆ ***ROLLUP***

ROLLUP は指定したカラムを順に含むグループ化を行います。例えば、ROLLUP (customer, product) は GROUPING SETS ((), (customer), (customer, product)) に等しい表現になります。

```
db1=> SELECT customer, product, sum(num) FROM sales GROUP BY ROLLUP (customer, product);
                -- (), (customer), (customer, product) のグループ化を同時に行なっている。
 customer | product | sum
----------+---------+-----
 John     | egg     |  20
 John     |         |  20
 Tom      | apple   |  20
 Tom      | egg     |  10
 Tom      |         |  30
          |         |  50
(6 rows)
```

4.8. JSONB に関する演算子と関数

JSONB に関する演算子と関数が追加されました。

4.8.1. jsonb || jsonb (結合もしくは上書き)

"||" 演算子によって、jsonb と jsonb の結合もしくはキーに対して値の上書きの演算が可能になりました。

◆ **結合の場合**

```
db1=> SELECT '{"name": "John", "age": 25}'::jsonb || '{"gender": "male"}'::jsonb;
              ?column?
-----------------------------------------------
 {"age": 25, "name": "John", "gender": "male"}
(1 row)
```

◆ **上書きの場合**

```
db1=> SELECT '{"name": "John", "age": 25}'::jsonb || '{"age": 26}'::jsonb;
       ?column?
----------------------------
 {"age": 26, "name": "John"}
(1 row)
```

4.8.2. jsonb - text / int (キー、配列番号を指定して値を削除)

"-" 演算子によって、キーもしくは配列番号を指定することで対応する値を削除する演算が可能になりました。

◆ キーを指定する場合

```
db1=> SELECT '{"name": "John", "age": 25}'::jsonb - 'age';
    ?column?
------------------
 {"name": "John"}
(1 row)
```

◆ 配列番号を指定する場合

```
db1=> SELECT '["apple", "orange", "melon"]'::jsonb - 1;
     ?column?
--------------------
 ["apple", "melon"]
(1 row)
```

4.8.3. jsonb #- text[] (キー、配列番号でパス指定して値を削除)

"#-" 演算子にキー、配列番号を用いた text [] でパスを指定し、対応する値を削除する演算が可能になりました。

```
db1=> SELECT
'{"name": "John", "contact": {"phone": "0123", "email":"John@foohoge.com"}}'::jsonb
 #-  '{contact, email}'::text[];
              ?column?
-----------------------------------------------
 {"name": "John", "contact": {"phone": "0123"}}
(1 row)

db1=> SELECT '{"name": "Thomas", "aliases": ["Tom", "Tommy"]}'::jsonb
              #-   '{aliases,  1}'::text[];
           ?column?
```

```
----------------------------------------
 {"name": "Thomas", "aliases": ["Tom"]}
(1 row)
```

4.8.4. jsonb_set 関数

jsonb に対して text[] でパス指定し、対応する値を更新する jsonb_set 関数が追加されました。

指定したパスに対応する値がなかった場合、省略可能な 4 番目の引数が真ならば「値を挿入する」、偽ならば「何もしない」動作となります。デフォルトでは「値を挿入する」動作となります。

◆ 指定したパスに値が存在せず、値を挿入させる場合

```
 db1=> SELECT jsonb_set(
'{"name": "James", "contact": {"phone": "01234 567890", "fax": "01987 543210"}}'::jsonb,
'{contact,skype}',  '"myskypeid"'::jsonb,  true);

                                jsonb_set

------------------------------------------------------------------------------------------
------------
   {"name": "James", "contact": {"fax": "01987 543210", "phone": "01234 567890", "skype":
"myskypeid"}}
  (1 row)
```

◆ 指定したパスに値が存在せず、値を挿入させない場合

```
 db1=> SELECT jsonb_set(
'{"name": "James", "contact": {"phone": "01234 567890", "fax": "01987543210"}}'::jsonb,
'{contact,skype}',  '"myskypeid"'::jsonb,  false);

                        jsonb_set
-----------------------------------------------------------------------------
 {"name": "James", "contact": {"fax": "01987 543210", "phone": "01234 567890"}}
(1 row)
```

◆ **指定したパスに値が存在し、値を更新させる場合**

```
db1=> SELECT jsonb_set(
      '{"name": "James", "skills": ["design","snowboarding","mechnaicalengineering"]}',
      '{skills,2}',    '"mechanical engineering"'::jsonb,    true);

                          jsonb_set
-------------------------------------------------------------------------------------
 {"name": "James", "skills": ["design", "snowboarding", "mechanical engineering"]}
(1 row)
```

4.8.5. jsonb_pretty 関数

jsonb を見やすい text に変換する jsonb_pretty 関数が追加されました。

```
db1=> SELECT jsonb_pretty(jsonb_set('{"name": "James", "contact": {"phone": "01234
567890", "fax": "01987 543210"}}'::jsonb, '{contact,phone}', '"07900 112233"'::jsonb));
          jsonb_pretty
---------------------------------
 {                               +
    "name": "James",             +
    "contact": {                 +
       "fax": "01987 543210", +
       "phone": "07900 112233"+
    }                            +
 }
(1 row)
```

5. 既存機能の変更点

既存機能の変更点で注意が必要な点をいくつか照会します。

5.1. pg_ctl stop のデフォルト動作の変更

pg_ctl stop のデフォルト動作が smart から fast に変更されました。

- smart ：クライアント接続が全て切断されてから正常終了する。
- fast ：クライアント接続の切断を待たず正常終了する。

5.2. checkpoint_segments の廃止、max_wal_size,min_wal_size の導入

9.5 から checkpoint_segments パラメータが廃止されました。その代わりに、保持する wal ファイルの最大データ量を指定する max_wal_size （デフォルト 1GB）, 最小データ量を指定する min_wal_size（デフォルト 80MB）パラメータが導入されました。

WAL サイズは、従来は小さすぎるデフォルトパラメータ値を大きくするのが性能チューニングでよくある調整方法でしたが、9.5 バージョンではデフォルト値がそれなりに大きいので、資源に乏しい環境に導入する際には注意が必要です。

6. 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。