PostgreSQL 12 検証レポート

1.0 版
2019 年 10 月 04 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

1. はじめに

本文書は PostgreSQL 12 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 12 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。
2019 年 9 月 12 日にリリースされた PostgreSQL 12 beta4 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 12 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。

- テーブルアクセスメソッド
- SQL/JSON Path
- 生成列
- COLLATE（文字照合）機能の拡張
- インデックスの機能追加・性能改善
- パーティショニングの改善
- WITH 句の最適化
- MCV 拡張統計情報
- ページチェックサムの改善
- VACUUM 動作指定追加

この他にも、認証処理の拡充をはじめ、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 12 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/12/release-12.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 12 （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

3.2. 検証環境

検証環境として、仮想化基盤上の CentOS 7.2 (x86_64) の仮想マシンを使用しました。

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。性能を検証する場合も、旧バージョンや新機能を使わない場合との比較を行います。

3.3. インストール

gcc、zlib-devel、readline-devel、libicu-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 ディレクトリを用意したうえで 、postgres ユーザにて実行しました。

```
（以下、postgres ユーザで実行）
$ wget https://ftp.postgresql.org/pub/source/v12beta4/postgresql-
12beta4.tar.bz2
$ tar jxf postgresql-12beta4.tar.bz2
$ cd postgresql-12beta4
$ ./configure --prefix=/usr/local/pgsql/12 --enable-debug --with-icu
$ 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 > ~/pg12.env <<'EOF'
VER=12
PGHOME=/usr/local/pgsql/${VER}
export PATH=${PGHOME}/bin:${PATH}
export LD_LIBRARY_PATH=${PGHOME}/lib:${LD_LIBRARY_PATH}
export PGDATA=/var/lib/pgsql/data_${VER}
EOF
$ . ~/pg12.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
```

検証用のデータベースを作成します。

```
$ createdb -U postgres db1
```

以降の各検証は db1 データベースに postgres ユーザで接続して行います。

```
$ psql -U postgres -d db1
psql (12beta4)
Type "help" for help.

db1=#
```

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/12/share/doc/html/
```

また、以下 URL にて PostgreSQL 12 のドキュメント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/12/static/
```

4.1. テーブルアクセスメソッド

PostgreSQL12 からテーブルのアクセスメソッド（AM）を追加、適用するインタフェースが追加されました。11 以前でもインデックスに対するアクセスメソッドは追加可能でしたが、同様のことがテーブルに対しても可能になりました。

テーブルから行データを読み書きする操作は、共有バッファを通してのヒープファイルの読み書き、WALファイルへの書き出し、といった低レベルのコーディングによって実現されています。テーブルアクセスメソッドを新たに用意して、差し換えることで、この部分を別の実装に変更することができます。これまでにもPostgreSQLのテーブルデータ読み書き部分を様々に拡張したソフトウェアがありましたが、それらはPostgreSQL本体コードを改変する必要がありました。これからは、そういった機能を付け外しが容易な拡張（EXTENSION）として提供することができます。

4.1.1. テーブルアクセスメソッドの確認・作成・適用

現在のサーバ内で使用できるテーブルアクセスメソッドは、インデックスアクセスメソッドと共にpsqlの\dAコマンドで確認できます。「heap」がPostgreSQLに標準で備わっている（今のところ）唯一のテーブルアクセスメソッドです。

（テーブルアクセスメソッドを確認）

```
db1=# \dA+
                              List of access methods
  Name  | Type  |        Handler        |              Description
--------+-------+-----------------------+----------------------------------------
 brin   | Index | brinhandler           | block range index (BRIN) access method
 btree  | Index | bthandler             | b-tree index access method
 gin    | Index | ginhandler            | GIN index access method
 gist   | Index | gisthandler           | GiST index access method
 hash   | Index | hashhandler           | hash index access method
 heap   | Table | heap_tableam_handler  | heap table access method
 spgist | Index | spghandler            | SP-GiST index access method
(7 rows)
```

アクセスメソッドを定義するにはCREATE ACCESS METHOD文に「TYPE TABLE」を指定して以下のように実行します。この例はハンドラ関数にheapと同じ「heap_tableam_handler」を指定していますので、実質的な意味はありません。結局heapと同じ働きをするアクセスメソッドになります。C言語で新たなハンドラ関数を用意して、それを指定するのが本来の意味のある使い方です。

（テーブルアクセスメソッド作成）

```
db1=# CREATE ACCESS METHOD myheap TYPE TABLE HANDLER heap_tableam_handler;
CREATE ACCESS METHOD
```

CREATE TABLEにUSING句を付けて、テーブルごとに使用するアクセスメソッドを指定できます 。

SELECT 文等で異なるアクセスメソッドのテーブルを組み合わせて使用することに制限はありません。

（テーブルアクセスメソッドを指定する）

```
db1=# CREATE TABLE t_myheap (id int, v text) USING myheap;
CREATE TABLE
```

（デフォルトのテーブルアクセスメソッドは設定パラメータで決められている）

```
db1=# SHOW default_table_access_method;
 default_table_access_method
-----------------------------
 heap
(1 row)
```

テーブルアクセスメソッドのデフォルト値は新たな設定パラメータ default_table_access_method で規定されます。

4.2. SQL/JSON Path

JSON から要素を抽出する記述方式に SQL/JSON Path を利用できるようになりました。JSON Path は SQL:2016 で定められている SQL 標準の方式です。PostgreSQL 11 以前でも JSON の要素を抽出する機能はありましたが、それは PostgreSQL 固有の関数・演算子に基づいていました（とはいえ JSON Path においても一部に PostgreSQL 独自の実装が含まれます）。

JSON Path を使用できる型は jsonb のみで、json 型には対応していません。

JSON Path 問い合わせ文字列は jsonpath 型のデータとして扱われます。

4.2.1. JSON Path を使用する

JSON Path を使用して JSON から要素を取得するには jsonb_path_query 関数や jsonb_path_query_array 関数を用います。以下に例を示します。

まずはサンプルデータを用意します。

（サンプルデータ作成）

```
db1=# CREATE TABLE tjson (id int PRIMARY KEY, j jsonb);
CREATE TABLE
db1=# INSERT INTO tjson VALUES (1, '[{"name":"PostgreSQL", "type":"RDB",
"since": 1996}, {"name":"MariaDB", "type":"RDB", "since": 2009},
{"name":"CouchBase", "type":"KVS", "since": 2010}]');
INSERT 0 1
```

```
db1=# INSERT INTO tjson VALUES (2, '[{"name":"VoltDB", "type":"RDB",
"since":2009}, {"name":"Cassandra", "type":"KVS", "since": 2008}]');
INSERT 0 1
db1=# SELECT * FROM tjson;
 id |    j
----+-------------------------------------------------------------------------
  1 | [{"name": "PostgreSQL", "type": "RDB", "since": 1996}, {"name":
"MariaDB", "type": "RDB", "since": 2009}, {"name": "CouchBase", "type":
"KVS", "since": 2010}]
  2 | [{"name": "VoltDB", "type": "RDB", "since": 2009}, {"name":
"Cassandra", "type": "KVS", "since": 2008}]
(2 rows)
```

JSON Path 式で JSON データの指定部分を取り出す例を示します。

（特定行から問い合わせ結果を配列で取得する）

```
db1=# SELECT jsonb_path_query_array(j, '$[*]?(@.type == "RDB")."name"')
        FROM tjson WHERE id = 1;
  jsonb_path_query_array
---------------------------
 ["PostgreSQL", "MariaDB"]
(1 row)
```

（複数行から問い合わせ結果を行で取得する）

```
db1=# SELECT id, jsonb_path_query(j, '$[*]?(@.since > 2000)')
        FROM tjson;
 id |                 jsonb_path_query
----+--------------------------------------------------
  1 | {"name": "MariaDB", "type": "RDB", "since": 2009}
  1 | {"name": "CouchBase", "type": "KVS", "since": 2010}
  2 | {"name": "VoltDB", "type": "RDB", "since": 2009}
  2 | {"name": "Cassandra", "type": "KVS", "since": 2008}
(4 rows)
```

「$[*]?(@.type == "RDB")."name"」は、一番外側（$）にある配列要素全て（[*]）の中で type が RDB であるもの（@.type=="RDB"）の name を取り出す（.name）という意味になります。JSON Path 中では JSON

Path むけに用意された限られた演算子と関数のみが使用できることに注意してください。使用可能な演算子等の一覧は PostgreSQL12 ドキュメントの 9.15.2.SQL/JSON Path Operators and Methods 節にあります。

JSON Path の問い合わせに GIN インデックスを使うことができます。以下にインデックスが使われる動作を確認します。

```
（jsonb 型列にインデックスを作成する）
db1=# CREATE INDEX ON tjson USING gin (j jsonb_path_ops);
CREATE INDEX

（JSON Path で該当行があるかを調べる「@?」演算子で検索）
db1=# SELECT id FROM tjson WHERE j @? '$[*]?(@.name == "PostgreSQL")';
 id
----
  1
(1 row)

（インデックスを使うようにシーケンシャルスキャンを off にして、プラン確認）
db1=# SET enable_seqscan TO off;
SET
db1=# explain
      SELECT id FROM tjson WHERE j @? '$[*]?(@.name == "PostgreSQL")';
                           QUERY PLAN
-----------------------------------------------------------------------------
 Bitmap Heap Scan on tjson  (cost=16.00..20.01 rows=1 width=4)
   Recheck Cond: (j @? '$[*]?(@."name" == "PostgreSQL")'::jsonpath)
   ->  Bitmap Index Scan on tjson_j_idx  (cost=0.00..16.00 rows=1 width=0)
         Index Cond: (j @? '$[*]?(@."name" == "PostgreSQL")'::jsonpath)
(4 rows)
```

インデックス作成時に指定した jsonb_path_ops は、以前から在った JSON 内を調べる「@>」演算子むけの演算子クラスです。jsonb_path_ops は JSON Path 問い合わせに対しても同様に効率的に働きます。なお 、jsonb 型にインデックスを作ったときにデフォルトで使われる jsonb_ops 演算子クラスであっても、JSON Path 問い合わせでインデックスを使うことは可能です。

4.3. 生成列

テーブルに指定した式の値が自動的に格納される列を定義できるようになりました。これを生成列（Generated Column）と呼びます。CREATE TABLE や ALTER TABLE の列定義で GENERATED オプションを使って設定します。GENERATED オプション自体は従来からありましたが、連番を自動付与して IDENTITY 列とする機能のみがサポートされていました。

以下に生成列の使用例を示します。

```
（生成列を含むテーブルを作成／height_in にインチ単位の値が自動計算される）
db1=# CREATE TABLE t_people (
        id int primary key,
        height_cm float4,
        height_in float4 GENERATED ALWAYS AS (height_cm / 2.54) STORED);
CREATE TABLE

（値を挿入）
db1=# INSERT INTO t_people (id, height_cm)
         VALUES (1, 156), (2, 168), (3, 172), (4, 185), (5, 191);
INSERT 0 5

（height_in の値が自動計算されている）
db1=# SELECT * FROM t_people ;
 id | height_cm | height_in
----+-----------+-----------
  1 |       156 | 61.417324
  2 |       168 |  66.14173
  3 |       172 |  67.71654
  4 |       185 |  72.83465
  5 |       191 |  75.19685
(5 rows)

（生成列を直接、挿入・更新することはできません）
=# UPDATE t_people SET height_in = 61.4 WHERE id = 1;
ERROR: column "height_in" can only be updated to DEFAULT
DETAIL: Column "height_in" is a generated column.
```

4.3.1. 生成列と更新可能ビューの違い

以下例のように従来から存在する更新可能ビューの機能を用いても似た動作を実現できます。

（更新可能ビューで同じことを行う）

```
db1=# CREATE TABLE t_people_base
        (id int primary key, height_cm float4);
CREATE TABLE
db1=# CREATE VIEW v_people AS
        SELECT id, height_cm, (height_cm/2.54)::float4 AS height_in
          FROM t_people_base;
CREATE VIEW
db1=# INSERT INTO v_people (id, height_cm)
         VALUES (1, 156), (2, 168), (3, 172), (4, 185), (5, 191);
INSERT 0 5
db1=# SELECT * FROM v_people;
 id | height_cm | height_in
----+-----------+-----------
  1 |       156 | 61.417324
  2 |       168 |  66.14173
  3 |       172 |  67.71654
  4 |       185 |  72.83465
  5 |       191 |  75.19685
(5 rows)
```

生成列を持つテーブルと更新可能ビューの特徴を以下表に示します。

	生成列を持つテーブル	更新可能ビュー
列を計算するタイミング	データ挿入・更新時	データ参照時
計算する列を変更しないデータ更新処理	INSERT、UPDATE、DELETE が可能	INSERT、UPDATE、DELETE が可能
計算する列の格納データ	実体を持つ	実体を持たない
計算する列に対するインデックスの作成	可能	不可（元となるテーブルにインデックスを作ることは可能）

PostgreSQL12 の生成列は「STORED」指定のみサポートされていて必ず実体を持ちます。テーブルの別の列の値から計算した値を持つ生成列があると、データ格納に必要なストレージ容量がその分だけ増加します。参照時に計算を行う、PostgreSQL12 でも未だサポートされていない「VIRTUAL」指定に相当する動作にしたい場合には、更新可能ビューで代替してください。

4.4. COLLATE(文字照合)機能の拡張

PostgreSQL12 で COLLATE（文字照合）機能が拡張されて、柔軟な同一性比較が可能になりました。

文字列比較をする際に COLLATE の指定によって、大文字、小文字、アクセント記号の有無を同一視したり、日本語であればひらがなや全角カタカナ、半角カタカナを同一視する動作を取ることができるようになりました。ICU ライブラリの機能を用いるので、ビルドには ICU ライブラリを利用できる環境で--with-icu の指定が必要です。

以下に例を示します。

```
（サンプルデータのテーブルを作成）
db1=# CREATE TABLE t_case(id int PRIMARY KEY, txt text);
CREATE TABLE
db1=# INSERT INTO t_case VALUES (1, 'ああ'), (2, 'アア'), (3, 'ｱｱ');
INSERT 0 3

（仮名文字の比較基準を緩めた日本語の COLLATION を作成します）
db1=# CREATE COLLATION ja_case_insensitive (provider = icu,
        locale = 'ja-u-ks-level2', deterministic = false);

（ja_case_insensitive を指定して検索すると平仮名、片仮名、半角カナが同一視されます）
db1=# SELECT * FROM t_case WHERE txt = 'ああ' COLLATE ja_case_insensitive;
 id | txt
----+------
  1 | ああ
  2 | アア
  3 | ｱｱ
(3 rows)
```

この動作は ICU ライブラリで提供されているロケールでいくつかの文字種を同一視するように定義されていることと、COLLATE 作成時に新たなオプション deterministic に false を指定していることで実現します。

（deterministic = true の場合の例）

```
db1=# CREATE COLLATION ja_case_insensitive_deterministic (provider = icu,
        locale = 'ja-u-ks-level2', deterministic = true);
CREATE COLLATION
db1=# SELECT * FROM t_case WHERE txt = 'ああ'
       COLLATE ja_case_insensitive_deterministic;
 id | txt
----+------
  1 | ああ
(1 row)
```

ラテン文字（アルファベット）で大文字小文字やアクセント記号を無視する例は PostgreSQL12 ドキュメントの 23.2 Collation Support（照合順序サポート）の節に記載されていますので、そちらを参照してください。

4.5. インデックスの機能追加・性能改善

PostgreSQL12 では様々なインデックスの改善が適用されています。

4.5.1. Btree インデックスの改善

Btree インデックスがいくつか改善されています。

- ***重複データを持つ Btree インデックスのサイズ縮小***

重複データを多く持つ Btree インデックスがより小さいサイズで済むようになりました。

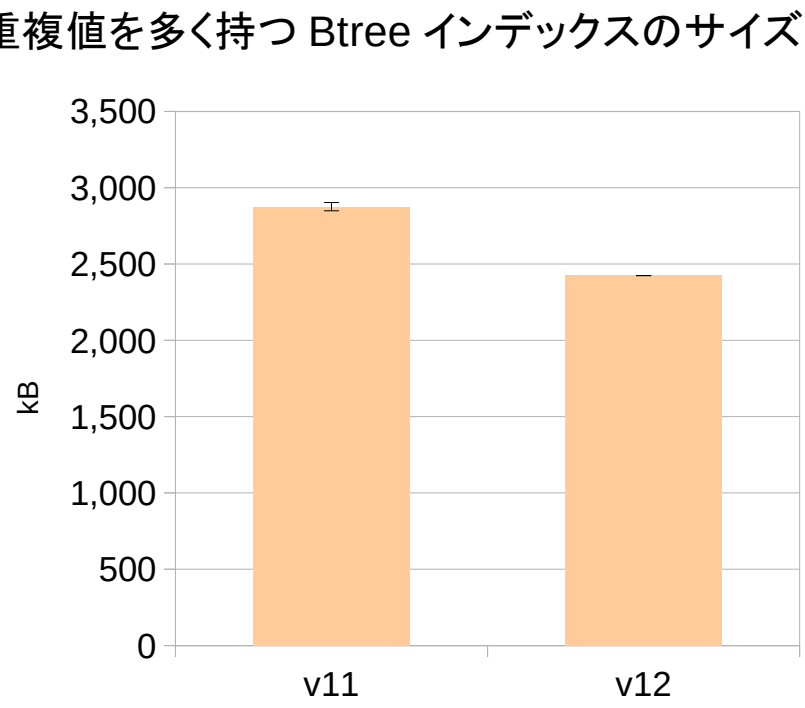
検証のために人工的なデータを作ってサイズを計測しました。10 万件のデータで値は 0 から 100 の乱数ですので、core 列には重複値が多数格納されるはずです。

（重複データを持つ Btree インデックスを作りサイズを測る）

```
db1=# CREATE TABLE t_idxtest1 (id int primary key, code bigint);
CREATE TABLE
db1=# CREATE INDEX t_idxtest1_code_idx ON t_idxtest1 (code);
CREATE INDEX
db1=# INSERT INTO t_idxtest1 SELECT g, (random() * 100)::bigint
        FROM generate_series(1, 100000) as g;
INSERT 0 100000
```

```
db1=# SELECT pg_relation_size('t_idxtest1_code_idx');
 pg_relation_size
------------------
         2482176
(1 row)
```

これを繰り返し行い PostgreSL 11 と比較すると以下のようになります。棒グラフは平均値と標準偏差を示しています。



PostgreSQL12 でバラつきが無くなり、サイズが 15%縮小しました。

◆ *複合 Btree インデックスのサイズ縮小*

複数の列を含む複合インデックスにおいて、Btree インデックスのサイズが小さくなりました。

検証のために比較的大きなデータを含む、4 列のインデックスを作って、サイズを計測しました。

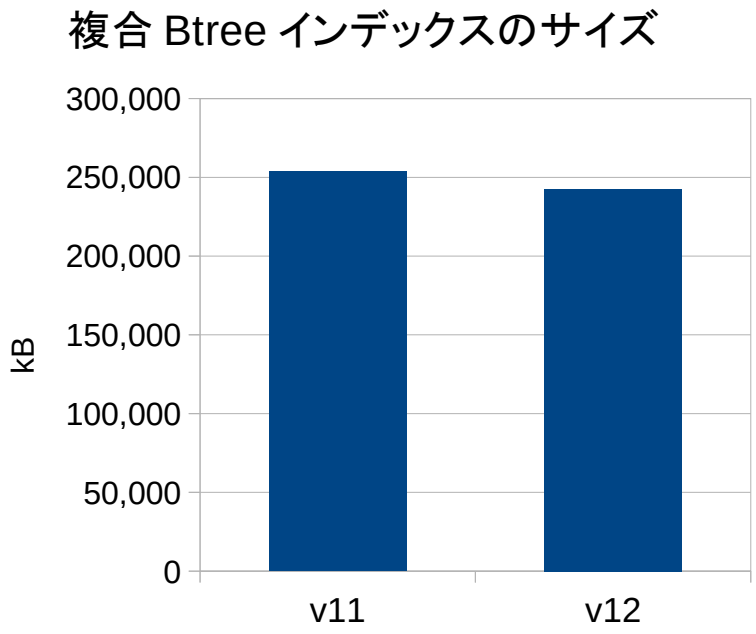
（複合 Btree インデックスを作りサイズを測る）

```
db1=# CREATE TABLE t_idxtest2 (k1 text, k2 text, k3 text, k4 text);
CREATE TABLE
db1=# CREATE INDEX ON t_idxtest2 (k1, k2, k3, k4);
CREATE INDEX
```

```
db1=# INSERT INTO t_idxtest2 SELECT repeat('A', 30) || to_char((g / 10000),
'00'),  repeat('B', 30) || to_char(((g / 100) % 100),'00'), repeat('C', 30)
|| ((g / 10) % 10), repeat('D', 30) || (g % 10) FROM generate_series(1,
500000) as g;
db1=# INSERT INTO t_idxtest2 SELECT repeat('A', 30) || to_char((g / 10000),
'00'),  repeat('B', 30) || to_char(((g / 100) % 100),'00'), repeat('C', 30)
|| ((g / 10) % 10), repeat('D', 30) || (g % 10) FROM generate_series(1,
500000) as g;
db1=# SELECT pg_relation_size('t_idxtest2_k1_k2_k3_k4_idx');
 pg_relation_size
------------------
       248545280
(1 row)
```

上記の INSERT 文は複雑ですが、意図しているところは適当な値を 50 万件ずつ 2 回投入するということだけです。

インデックスサイズを比較すると以下のようになりました。本ケースで 5%程度小さくなっていることが確認できます。



本テストでは 11、12 バージョンともサイズにバラつきは生じませんでした。

◆ ***Btree 同時実行性の改善***

Btree インデックスの内部ロックが軽減されました。多 CPU コアマシンによる Btree インデックスに対する同時実行処理が効率化されることが期待できます。

本検証は大型マシンを使ったものではないため、本修正は実証確認の対象外といたしました。

4.5.2. インデックス作成時の *WAL* 量削減

GIN インデックス、GiST インデックス、SP-GiST インデックスでインデックス作成時の WAL 出力量が削減されました。レプリケーションやアーカイブロギングをしているときに大量 WAL 発生が生じる問題を軽減できます。

以下のようにインデックスを作成する前後で WAL 位置を計測して、WAL 出力量を調べました。これは、他の WAL 書き込みを最小化するため自動 VACUUM を無効にして行います。

（サンプルデータを作成）

```
db1=# CREATE TABLE t_idx_wal_test
        (id int primary key, arr int[], tr tsrange, txt text);
CREATE TABLE
db1=# INSERT INTO t_idx_wal_test
         SELECT g, ARRAY[(g % 13), ((g + 7) % 59), ((g + 12) % 117) ],
           tsrange(now()::timestamp, now()::timestamp + '1hour'::interval),
           md5(g::text) FROM generate_series(1, 100000) as g;
INSERT 0 100000
```

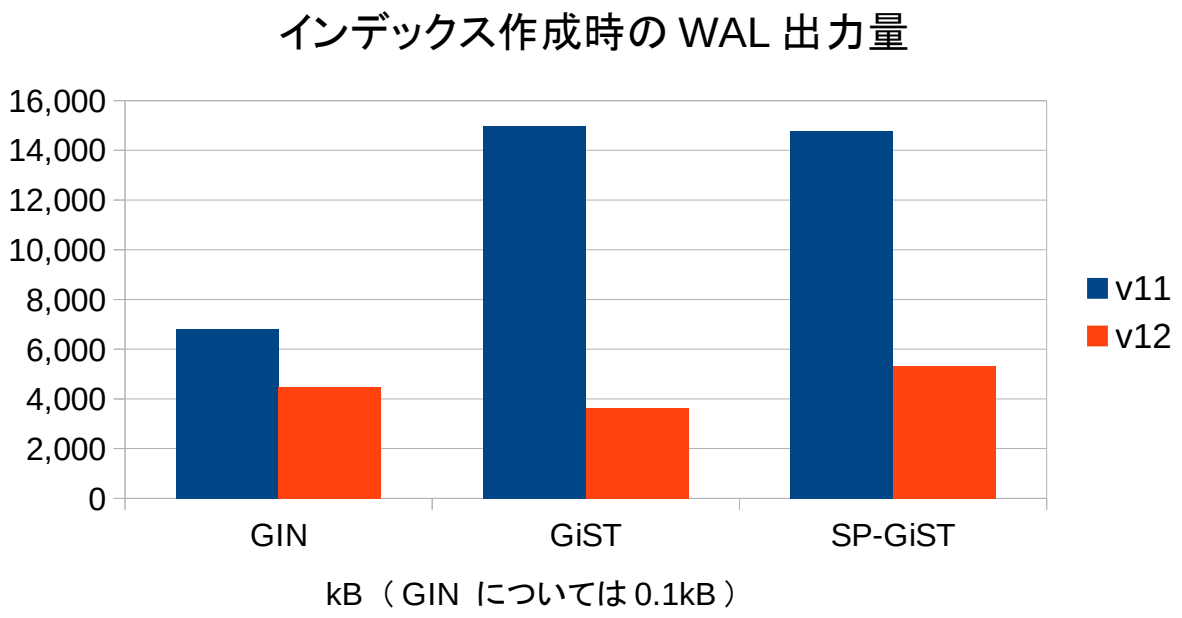
（インデックスを作成する前後で WAL 位置を計測）

```
db1=# SELECT pg_current_wal_lsn();
 pg_current_wal_lsn
--------------------
 0/20F88D30
(1 row)
db1=# CREATE INDEX ON t_idx_wal_test USING gin (arr);
CREATE INDEX
db1=# SELECT pg_current_wal_lsn();
 pg_current_wal_lsn
```

```
--------------------
 0/20FF8618
(1 row)

（以下も同様に行う／出力は省略）
db1=# CREATE INDEX ON t_idx_wal_test USING gist (tr);
db1=# SELECT pg_current_wal_lsn();
db1=# CREATE INDEX ON t_idx_wal_test USING spgist (txt);
db1=# SELECT pg_current_wal_lsn();
```

以下に結果を示します。



いずれも WAL 出力量が大幅に減っていることが確認できます。本テストは同条件で実行した結果にバラつきがほとんど生じません。

4.5.3. REINDEX が同時作成に対応

REINDEX コマンドが「インデックスの同時作成」に対応しました。以前から CREATE INDEX では CONCURRENTLY オプションを付与して、インデックスの同時作成が可能でしたが、REINDEX でも同様に実行できるようになります。

これにより、読み書きの処理と並行してインデックス再作成が可能になります。従来は CREATE INDEX で

実行する必要があったため、同じインデックス名を維持したり、制約に紐付けたりするのに、追加でいくつかの ALTER コマンドの実行が必要でした。このような手間が軽減されます。

以下の例では 4.5.1 節で作成したインデックスを同時再作成しています。

```
db1=# REINDEX INDEX CONCURRENTLY t_idxtest1_code_idx;
REINDEX
```

REINDEX の CONCURRENTLY オプションは Btree に限らず、GIN、GiST、SP-GiST のインデックスでも同様に使用できます。CONCURRENTLY オプションを付けないときに比べて所要時間が増えたり、失敗することもあってそのときはやり直さなければならない、といった制約は CREATE INDEX CONCURRENTLY と同様です。

4.5.4. GiST が INCLUDE に対応

GiST インデックスでも CREATE INDEX の INCLUDE 指定により非キー列を含めることができるようになりました。11 バージョンまでは Btree インデックスのみに対応していました。

インデックス定義で非キー列を含めるのは、インデックスオンリースキャンで情報が取り出せるようにするためです。排他制約むけのインデックスなど、必ず GiST インデックスが必要な場合があり、そのときに非キー列を含める技法を併用できるメリットがあります。

以下に例を示します。

```
（point 型に非キー列 dsc を加えた GiST インデックスを作成する）
db1=# CREATE TABLE t_point (p point, dsc text);
CREATE TABLE
db1=# CREATE INDEX ON t_point USING gist (p) INCLUDE (dsc);
CREATE INDEX

（点 p が矩形範囲 (0,0)-(1,1) に含まれるかを調べて、付帯説明 dsc も出力する）
db1=# explain SELECT p, dsc FROM t_point WHERE p <@ '(0,0,1,1)'::box;
                                QUERY PLAN
---------------------------------------------------------------------------
 Index Only Scan using t_point_p_dsc_idx on t_point  (cost=0.14..8.16 rows=1
width=48)
   Index Cond: (p <@ '(1,1),(0,0)'::box)
(2 rows)
《Index Only Scan が使われる実行プランになってる》
```

4.5.5. SP-GiST が kNN 近傍探索に対応

SP-GiST インデックスが kNN 近傍探索に対応しました。

これまで GiST インデックスしか対応していなかったために、地理情報、幾何データに対して必ず GiST インデックスが使われていた場面で、SP-GiST の使用も選択肢に加えることができるようになります。

以下に例を示します。

```
（point 型の列に SP-GiST インデックスを持つテーブルを作成）
db1=# CREATE TABLE t_point_spgist (p point, dsc text);
CREATE TABLE
db1=# CREATE INDEX ON t_point_spgist USING spgist (p) ;
CREATE INDEX

（(0,0) 地点からの距離が近い順にソートする問い合わせでインデックス使用を確認）
db1=# explain SELECT * FROM t_point_spgist ORDER BY p <-> '(0,0)';
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using t_point_spgist_p_idx on t_point_spgist  (cost=0.14..73.54
rows=1070 width=56)
   Order By: (p <-> '(0,0)'::point)
(2 rows)
```

4.6. パーティショニングの改善

PostgreSQL12 ではテーブルパーティショニングについて性能、機能の両面で改善が加えられています。

4.6.1. パーティション性能改善

パーティションテーブルに対して 2 つの大きな性能改善があります。特に多数の子パーティションを含むパーティションテーブルに効果があります。

◆ *多パーティションに COPY データ投入*

多数のパーティションを含むパーティションテーブルに COPY でデータを投入する際に性能が改善されました。

以下の手順にて、3000 個の子パーティションを含むパーティションテーブルを作成してデータ投入する検証を実施しました。各パーティションのデータは 10 件だけで、合計 30000 件が格納されています。

（投入用 COPY データを作成）

```
db1=# CREATE TABLE single (id int primary key, v text);
db1=# INSERT INTO single SELECT g, md5(g::text)
        FROM generate_series(0, 30000 - 1) as g;
db1=# COPY single TO '/var/lib/pgsql/dat.csv' CSV;
```

《上記いずれも出力省略》

（3000 子の子パーティションを持つパーティションテーブル oya を作成）

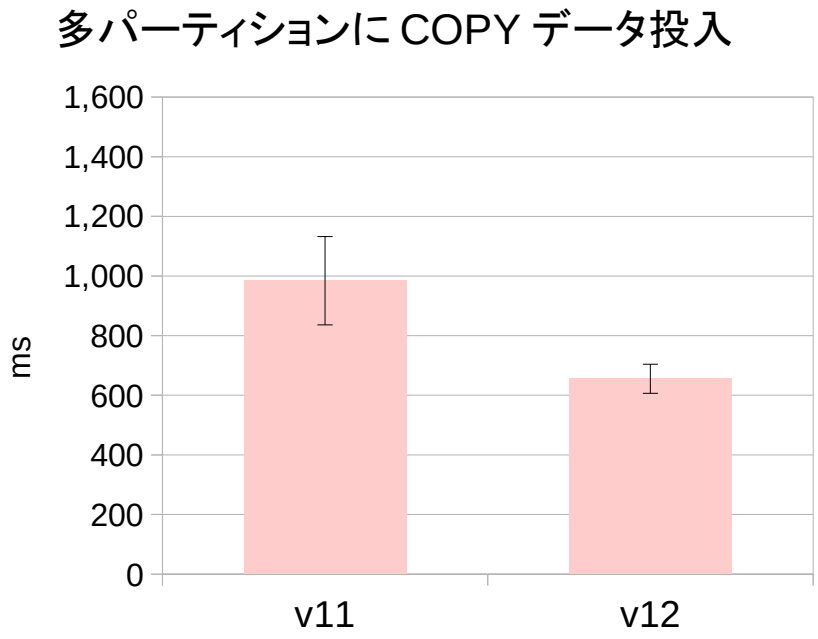
```
db1=# CREATE TABLE oya (id int primary key, v text) PARTITION BY RANGE (id);
CREATE TABLE
db1=# SELECT 'CREATE TABLE ko' || g || ' PARTITION OF oya FOR VALUES FROM ('
      || g * 10 - 10 || ') TO (' || g * 10 || ');'
      FROM generate_series(1, 3000) as g;
db1=# \gexec
```

《出力省略》

（パーティションテーブル oya に COPY でデータ投入して時間を計測する）

```
db1=# \timing on
db1=# COPY oya FROM '/var/lib/pgsql/dat.csv' CSV;
COPY 30000
Time: 586.044 ms
```

以下の結果が得られました。



所要時間が 30～40%ほど短縮され、また、所要時間のバラつきも小さくなっていることが確認できます。

- ***多パーティションに対する検索***

多数のパーティションを含むパーティションテーブルを検索して、実際にデータを取り出す対象のパーティションが少ない場合に、大幅に性能が改善しました。

前節で作ったパーティションテーブルを使って以下のように計測しました。

```
（プランの差異が出ないように Bitmap Index Scan に限定する）
db1=# SET enable_seqscan TO off;
SET
db1=# SET enable_indexscan TO off;
SET

（必要なパーティションだけを走査する実行プランになっていることを確認）
db1=# explain SELECT * FROM oya WHERE id IN (100, 1000, 10000);
                                QUERY PLAN
-------------------------------------------------------------------------------
 Append  (cost=8.43..37.45 rows=9 width=37)
  ->  Bitmap Heap Scan on ko11  (cost=8.43..12.47 rows=3 width=37)
        Recheck Cond: (id = ANY ('{100,1000,10000}'::integer[]))
```

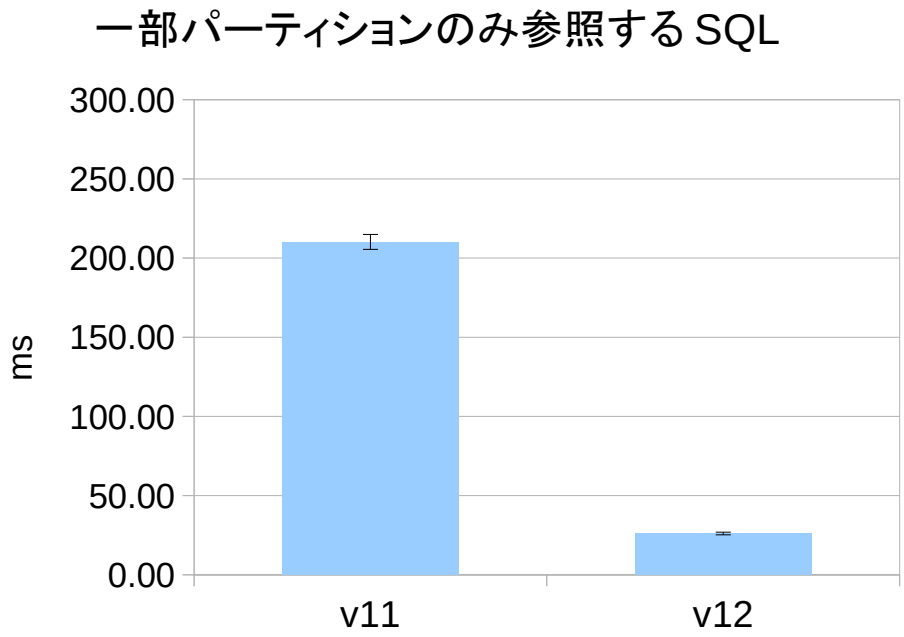
```
        ->  Bitmap Index Scan on ko11_pkey  (cost=0.00..8.43 rows=3 width=0)
              Index Cond: (id = ANY ('{100,1000,10000}'::integer[]))
   ->  Bitmap Heap Scan on ko101  (cost=8.43..12.47 rows=3 width=37)
         Recheck Cond: (id = ANY ('{100,1000,10000}'::integer[]))
         ->  Bitmap Index Scan on ko101_pkey  (cost=0.00..8.43 rows=3
width=0)
               Index Cond: (id = ANY ('{100,1000,10000}'::integer[]))
   ->  Bitmap Heap Scan on ko1001  (cost=8.43..12.47 rows=3 width=37)
         Recheck Cond: (id = ANY ('{100,1000,10000}'::integer[]))
         ->  Bitmap Index Scan on ko1001_pkey  (cost=0.00..8.43 rows=3
width=0)
               Index Cond: (id = ANY ('{100,1000,10000}'::integer[]))
(13 rows)
```

（所要時間を計測する）

```
db1=# \timing on
db1=# SELECT * FROM oya WHERE id IN (100, 1000, 10000);
  id   |                v
-------+---------------------------------
   100 | f899139df5e1059396431415e770c6dd
  1000 | a9b7ba70783b617e9998dc4dd82eb3c5
 10000 | b7a782741f667201b54880c925faec4b
(3 rows)

Time: 219.307 ms
```

結果は以下の通りです。



11 バージョンと比較して所要時間が 5 分の 1 以下になっています。

4.6.2. パーティション機能改善

パーティションテーブルに対していくつか機能が加わっています。重要な項目を取り上げます。

◆ ***外部キー制約対応***

これまではパーティションテーブルに外部キー制約を付与する場合に、被参照テーブルは非パーティションテーブルに限定されていました。12 バージョンから、この制限がなくなりました。以下のようにパーティションテーブル同士で外部キー制約を構成できます。

（パーティションとパーティション間の外部キー制約を定義）

```
db1=# CREATE TABLE t_log (
        id int PRIMARY KEY, lev text, ts timestamp, mes text)
        PARTITION BY RANGE (id);
CREATE TABLE
db1=# CREATE TABLE t_log_detail (id int PRIMARY KEY, detail text)
        PARTITION BY RANGE (id);
CREATE TABLE
db1=# ALTER TABLE t_log_detail ADD FOREIGN KEY (id) REFERENCES t_log (id);
ALTER TABLE
```

◆ ***パーティション定義に式を使用***

パーティションテーブルの定義で範囲の値を指定する際に定数に限らず任意の式を利用できるようになりました。式の値は CREATE の実行時点で決まります。

例えば以下のように記述することができるようになります。

```
（今日時点から１年ごとにパーティションを作る）
db1=# BEGIN;
db1=# CREATE TABLE t_log2 (ts timestamp, mes text) PARTITION BY RANGE (ts);
CREATE TABLE
db1=# CREATE TABLE t_log2_0 PARTITION OF t_log2 FOR VALUES
        FROM (CURRENT_DATE::timestamp)
        TO (CURRENT_DATE::timestamp + '1 year');
CREATE TABLE
db1=# CREATE TABLE t_log2_1 PARTITION OF t_log2 FOR VALUES
        FROM (CURRENT_DATE::timestamp + '1 year')
        TO (CURRENT_DATE::timestamp + '2 year');
CREATE TABLE
《以下略、最後に COMMIT する》
```

4.7. WITH 句の最適化

WITH 句による共通テーブル式（CTE と呼ばれます）を使った問い合わせで、プラン作成に際して CTE 部分についても最適化できるようになりました。オプションにより、従来の最適化しない動作も指定できます。

まず、PostgreSQL11 での CTE 部分が最適化されない動作を確認します。

```
（テスト用のテーブルとデータを作成）
db1=# CREATE TABLE t_customer (customer_id int primary key, info text);
db1=# CREATE TABLE t_sales
        (sales_id int primary key, customer_id int, info text);
db1=# INSERT INTO t_customer SELECT g, md5(g::text)
         FROM generate_series(1, 100000) as g;
INSERT 0 100000
db1=# INSERT INTO t_sales SELECT g, g % 100000, md5(g::text)
         FROM generate_series(1, 1000000) as g;
```

```
INSERT 0 1000000
db1=# CREATE INDEX ON t_customer (info);
db1=# ANALYZE;

（PostgreSQL11 で実行）
db1=# explain WITH cte AS (SELECT * FROM t_customer)
        SELECT * FROM t_sales JOIN cte USING (customer_id)
          WHERE cte.info ~ '^abcde';
                              QUERY PLAN
------------------------------------------------------------------------------
 Hash Join  (cost=42470.00..53574.77 rows=4927 width=73)
   Hash Cond: (cte.customer_id = t_sales.customer_id)
   CTE cte
     ->  Seq Scan on t_customer  (cost=0.00..1834.00 rows=100000 width=37)
   ->  CTE Scan on cte  (cost=0.00..2250.00 rows=500 width=36)
         Filter: (info ~ '^abcde'::text)
   ->  Hash  (cost=19346.00..19346.00 rows=1000000 width=41)
         ->  Seq Scan on t_sales  (cost=0.00..19346.00 rows=1000000 width=41)
(8 rows)
```

11 バージョンで実行すると CTE 部分が独立して実行されて、その結果を結合して、info 列をフィルターして、インデックスも使わないという実行プランになりました。これは CTE 部分は最適化されないためです。最適化のためには WITH 句ではなくサブクエリを使った書き方が必要でした。

12 バージョンでは同様に実行したとき、以下のように CTE 部分も最適化されます。CTE 内の t_customer テーブルのスキャンに外側にある「cte.info ~ '^abcde'」条件でのインデックススキャンが選択されています。

```
（PostgreSQL12 で実行）
db1=# EXPLAIN WITH cte AS (SELECT * FROM t_customer)
        SELECT * FROM t_sales JOIN cte USING (customer_id)
          WHERE cte.info ~ '^abcde';
                              QUERY PLAN
------------------------------------------------------------------------------
 Gather  (cost=1008.57..15625.03 rows=100 width=74)
   Workers Planned: 2
   ->  Hash Join  (cost=8.56..14615.03 rows=42 width=74)
         Hash Cond: (t_sales.customer_id = t_customer.customer_id)
```

```
        ->  Parallel Seq Scan on t_sales  (cost=0.00..13512.67 rows=416667
width=41)
        ->  Hash  (cost=8.44..8.44 rows=10 width=37)
              ->  Index Scan using t_customer_info_idx on t_customer
(cost=0.42..8.44 rows=10 width=37)
                    Index Cond: ((info >= 'abcde'::text) AND (info <
'abcdf'::text))
                    Filter: (info ~ '^abcde'::text)
(9 rows)
```

12 バージョンで WITH 句に MATERIALIZED オプションを指定すると、11 までと同様の動作になります。

```
（PostgreSQL12 で実行）
db1=# EXPLAIN WITH cte AS MATERIALIZED (SELECT * FROM t_customer)
        SELECT * FROM t_sales JOIN cte USING (customer_id)
          WHERE cte.info ~ '^abcde';
《出力省略／11 での実行と同様の実行プランになる》
```

上記例の「MATERIALIZED」部分は、何も指定しない、MATERIALIZED を指定、NOT MATERIALIZED を指定のいずれかになり、以下表の通り、三者で動作が異なります。

指定	意味
指定無し	最適化を許す。
MATERIALIZED	CTE 部分は必ず独立してプラン作成される。 （11 バージョンまでの従来動作と同じ）
NOT MATERIALIZED	CTE が複数個所に登場して、複数回評価される場合についても最適化を許す。

NOT MATERIALIZED 指定が最も強く最適化を指向します。CTE が複数個所で参照される場合には、CTE 部分を独立して 1 回実行して、その結果を使いまわす方が効率的である可能性が高くなるため、通常は「指定無し」で十分であるはずです。NOT MATERIALIZED 指定は意図をもって SQL をチューニングする手段としては有用です。

4.8. MCV 拡張統計情報

拡張統計情報に mcv（Most Common Value、最頻値）が加わりました。拡張統計情報とは、CREATE STATISTICS 文による明示的な定義に基づいて追加で採取される統計情報です。特に複数の列にまたがる統計情報を扱います。これまで、複数列の組み合わせに対する値の種類数（ndistinct）と、列と列との間の関数的依存関係（dependencies）の統計に対応していました。

以下に例を示します。

```
（サンプルテーブル／loc:都市名 'Tokyo'、'Osaka' など、gen:年代 20、30 など）
db1=# CREATE TABLE t_mcvex (id serial, loc text, gen int);
CREATE TABLE
db1=# INSERT INTO t_mcvex (loc, gen)
        SELECT 'Kyoto', 40 FROM generate_series(1,1000);
INSERT 0 1000
db1=# INSERT INTO t_mcvex (loc, gen)
         SELECT 'Tokyo', 10 FROM generate_series(1,1000);
INSERT 0 1000
《以下、適当に都市と年代を投入していく》

（プランナ統計情報を確認する）
db1=# ANALYZE;
db1=# SELECT attname, most_common_vals, most_common_freqs
        FROM pg_stats WHERE tablename = 't_mcvex';
 attname |                                          most_common_vals
|                                                        most_common_freqs
---------
+-------------------------------------------------------------------------
-----------------
+-------------------------------------------------------------------------
-------------------------------------------------
 id      |                          |
 gen     | {20,40,30,50,10,60,70}                          |
{0.28596666,0.21253334,0.17873333,0.107666664,0.10763333,0.0727,0.034766667}
 loc     |
{Tokyo,Osaka,Kyoto,Koube,Sapporo,Nagasaki,Yokohama,Kumamoto,Fukuoka,Sendai,N
iigata,Nagoya} |
```

```
{0.25046667,0.2174,0.17623334,0.071166664,0.0366,0.0359,0.035866667,0.035766
665,0.035733335,0.0357,0.035033334,0.034133334}
(3 rows)
```

《列ごと単独で見ると、loc は Tokyo が多く、gen は 20 が多いことがわかる》

（二つの列を合わせた条件で検索すると、プランナの推定行数と実際の行数とが大きく異なる）

```
db1=# explain analyze SELECT * FROM t_mcvex
        WHERE loc = 'Kyoto' AND gen = '40';
                                              QUERY PLAN
------------------------------------------------------------------------------
 Seq Scan on t_mcvex  (cost=0.00..1252.00 rows=2024 width=14) (actual
time=0.036..23.549 rows=10000 loops=1)
   Filter: ((loc = 'Kyoto'::text) AND (gen = 40))
   Rows Removed by Filter: 50000
 Planning Time: 0.404 ms
 Execution Time: 26.109 ms
(5 rows)
```

（mcv の拡張統計情報を作成して、ANALYZE で統計情報収集を行う）

```
db1=# CREATE STATISTICS es_t_mcvex_gen_loc (mcv) ON gen, loc FROM t_mcvex;
CREATE STATISTICS
db1=# ANALYZE;
ANALYZE
```

（拡張統計情報を確認する）

```
db1=# SELECT m.* FROM pg_statistic_ext JOIN pg_statistic_ext_data
        ON (oid = stxoid), pg_mcv_list_items(stxdmcv) m
        WHERE stxname = 'es_t_mcvex_gen_loc';
 index |    values     | nulls |      frequency       |    base_frequency
-------+---------------+-------+----------------------
+----------------------
     0 | {Kyoto,40}    | {f,f} |  0.16573333333333334 |  0.03268261333333333
     1 | {Osaka,10}    | {f,f} |               0.0689 |           0.02074341
     2 | {Osaka,50}    | {f,f} |  0.06586666666666667 |           0.02037627
     3 | {Osaka,20}    | {f,f} |  0.03523333333333333 | 0.053976378888888885
```

```
    4 | {Yokohama,30} | {f,f} |                0.0347 | 0.008134836666666668
    5 | {Tokyo,70}    | {f,f} |  0.03453333333333333 |           0.00800828
    6 | {Tokyo,50}    | {f,f} |  0.03403333333333333 |           0.02316681
    7 | {Osaka,60}    | {f,f} | 0.033966666666666666 | 0.013788146666666664
    8 | {Niigata,20}  | {f,f} | 0.033666666666666664 | 0.008909322222222222
    9 | {Tokyo,60}    | {f,f} | 0.033633333333333335 |           0.01567644
```

《Kyoto 所在，40 代の組み合わせが多いことが分かる》

（拡張統計情報を収集後に再度プラン確認）

```
db1=# explain analyze
        SELECT * FROM t_mcvex WHERE loc = 'Kyoto' AND gen = '40';
                                                QUERY PLAN
-----------------------------------------------------------------------------------
 Seq Scan on t_mcvex  (cost=0.00..1252.00 rows=9944 width=14) (actual
time=0.039..28.905 rows=10000 loops=1)
   Filter: ((loc = 'Kyoto'::text) AND (gen = 40))
   Rows Removed by Filter: 50000
 Planning Time: 0.515 ms
 Execution Time: 31.766 ms
(5 rows)
```

《今度は行数予測の一致度が高い》

4.9. ページチェックサムの改善

ページチェックサムが後から変更できるようになりました。

これまでは initdb 実行時点で指定するほかありませんでしたが、以下コマンドでチェックサムを有効、無効を変更できます。

（後からチェックサムを有効にする／PostgreSQL サービス停止状態で行う）

```
$ pg_checksums --enable --progress -D $PGDATA
32/32 MB (100%) computed
Checksum operation completed
Files scanned:  1306
Blocks scanned: 4157
pg_checksums: syncing data directory
```

```
pg_checksums: updating control file
Checksums enabled in cluster
《無効に変えるには --disable を指定する》
```

この操作は PostgreSQL サービスを停止した状態で行う必要があります。また、テーブルに対応した各ファイルごとの書き換え処理になるため、データの大きさに応じた所要時間が必要となります。

この pg_checksums には、11 バージョンで pg_verify_checksums コマンドとして提供されていたチェックサムを検査する機能が統合されています。以下のように検査を実行することができます。

```
（チェックサムの検査を実行／PostgreSQL サービス停止状態で行う）
$ pg_checksums --check -P -D $PGDATA
32/32 MB (100%) computed
Checksum operation completed
Files scanned:  1306
Blocks scanned: 4157
Bad checksums:  0
Data checksum version: 1
```

4.10. VACUUM 動作指定追加

VACUUM コマンドに以下のオプションが追加されました。INDEX_CLEANUP と TRUNCATE については テーブルのストレージオプションでも指定できます。

オプション	意味	対応ストレージパラメータ
INDEX_CLEANUP	インデックスも処理するか？ （デフォルトはストレージパラメータで規定）	vacuum_index_cleanup （デフォルト true）
SKIP_LOCKED	ロック待ちを要する処理をスキップするか？ （デフォルト off）	
TRUNCATE	末尾の空ページを切捨てするか？ （デフォルトはストレージパラメータで規定）	vacuum_truncate （デフォルト true）

このうち TRUNCATE 指定について動作を確認してみます。

```
（テストテーブルとデータを作成／自動 VACUUM を止めて行う）
db1=# CREATE TABLE t_vacuum (id int, v text);
```

```
CREATE TABLE
db1=# INSERT INTO t_vacuum SELECT g, md5(g::text)
        FROM generate_series(1, 10000) g;
INSERT 0 10000
db1=# SELECT pg_relation_size('t_vacuum');
 pg_relation_size
------------------
          688128
(1 row)

（DELETE と TRUNCATE off 指定の VACUUM を実行）
db1=# DELETE FROM t_vacuum;
DELETE 10000
db1=# VACUUM (TRUNCATE off) t_vacuum;
VACUUM

（全データ削除後の VACUUM でも物理サイズがゼロになっていない）
db1=# SELECT pg_relation_size('t_vacuum');
 pg_relation_size
------------------
          688128
(1 row)

（TRUNCATE on なら物理サイズがゼロに縮小する）
db1=# VACUUM (TRUNCATE on) t_vacuum;
VACUUM
db1=# SELECT pg_relation_size('t_vacuum');
 pg_relation_size
------------------
                0
(1 row)
```

5. さまざまな仕様変更

12 バージョンではいくつか仕様が変更されています。本章ではそれらのうち重要なものをいくつか取り上げます。

5.1. recovery.conf 廃止

PostgreSQL12 から物理オンラインバックからのリカバリや、物理ストリーミングレプリケーションでスタンバイサーバを起動するときに使用する recovery.conf 設定ファイルが廃止されました。

recovery.conf 設定ファイル内にあった設定項目は postgresql.conf に記述するようになります。リカバリ起動やスタンバイ起動であることを示すためには、代わりに recovery.signal ファイル、standby.signal ファイルを配置します。

また、これと関連して、これまで recovery.conf ファイル内の設定項目を変更して反映するには、スタンバイを必ずサービス再起動しなければなりませんでしたが、以下の項目が reload で反映できるようになりました。

- archive_cleanup_command
- promote_trigger_file
- recovery_end_command
- recovery_min_apply_delay

5.2. 行ごとの OID 廃止

PostgreSQL12 から行ごとの OID 付番が廃止されました。CREATE TABLE 文の WITH OID オプションが無くなっています。これは長らく非推奨とされていて、デフォルトでは使われない機能でした。

これと連動して、oid 列を含むテーブルに選択リストに * をして SELECT を実行した場合に oid 列が省略される動作がなくなりました。例えば oid 列を持つ pg_class システムテーブルに SELECT を実行したときに明示的に oid 列名を指定しなくとも oid 列が出力されるようになります。

（PostgreSQL12 では明示指定が無くとも oid 列が出力される）

```
db1=# SELECT * FROM pg_class LIMIT 1;
  oid   |    relname     | relnamespace | reltype | reloftype | relowner |
relam | relfilenode | reltablespace | relpages | reltuples | relallvisible |
reltoastrelid | relhasindex | relisshared | relpersistence | relkind |
relnatts | relchecks | relhasrules | relhastriggers | relhassubclass |
relrowsecurity | relforcerowsecurity | relispopulated | relreplident |
relispartition | relrewrite | relfrozenxid | relminmxid | relacl |
reloptions | relpartbound
```

```
-------+---------------+-------------+---------+----------+----------
+-------+------------+--------------+---------+----------
+---------------+--------------+-------------+-------------
+----------------+---------+----------+----------+-------------
+----------------+----------------+----------------+---------------------
+----------------+-------------+----------------+------------
+--------------+-----------+--------+-----------+--------------
 16389 | pg_toast_16386 |            99 |   16390 |          0 |        10 |
16385 |        16389 |              0 |         0 |          0 |              0 |
0 | t             | f            | p               | t        |         3 | 0 | f
| f               | f               | f               | f           | t
| n             | f               |           0 |   488 |           1 |
(1 row)
```

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。