PostgreSQL13 検証レポート

1.2 版
2020 年 12 月 3 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 13 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 13 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。
2020 年 5 月 21 日にリリースされた PostgreSQL 13 beta1 を使用して検証を行い、その後 2020 年 6 月 25 日にリリースされた PostgreSQL 13 beta2 での変更内容と、2020 年 9 月 24 日に正式リリースされた PostgreSQL 13 での変更内容を反映して、本文書を作成しています。

PostgreSQL 13 beta 2 での変更点は、enable_hashagg_disk 設定パラメータが hashagg_avoid_disk_plan となりデフォルト値も off に変更されたことと、enable_groupingsets_hash_disk 設定パラメータが廃止となったことの二点です。また正式にリリースされた PostgreSQL 13 での変更点は、enable_incrementalsort 設定パラメータが enable_incremental_sort に変更されたことと、hashagg_avoid_disk_plan が廃止されたことの二点です。

2. 概要

PostgreSQL 13 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。

- B-Tree インデックスの性能向上
- 新しい実行プラン
- パーティションテーブル機能の拡張
- パラレル VACUUM
- 進捗レポートビューの拡張
- pgbench の拡張
- pg_rewind の拡張
- SQL 機能の追加

この他にも、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 13 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/13/release-13.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 13（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 7.3 (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 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/v13beta1/postgresql-
13beta1.tar.bz2
                                                    《実際は 1 行》
$ tar jxf postgresql-13beta1.tar.bz2
$ cd postgresql-13beta1
$ ./configure --prefix=/usr/local/pgsql/13 --enable-debug
$ make world
$ su -c "make install-world"
```

環境変数を設定するファイルを書き出して、適用します。

```
$ cat > ~/pg13.env <<'EOF'
VER=13
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
EOF
$ . ~/pg13.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。これによりログメッセージがファイルに蓄積されます。

```
$ 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 (13beta1)
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、併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/13/share/doc/html/
```

また、以下 URL にて PostgreSQL 13 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/13/static/
```

4.1. B-Tree インデックスの性能向上

PostgreSQL の標準インデックスである B-Tree インデックスに重複排除(deduplication)の機能が追加されました。

4.1.1. 重複排除*(deduplication)*とは

テーブルには任意の列で同じ値となる行が複数存在する場合があります。その列をインデックスのキーとしたとき、インデックスには同じキー値でテーブル上の異なる行の場所 TID を指し示すインデックスタプルが複数箇所存在することになります。これはインデックス領域を肥大化させる非効率な表現でした。重複排除処理が働くことにより、重複したインデックスタプルをマージした効率の良い表現に変換し、インデックスサイズを縮小します。

以下に重複排除処理の簡易的な模式図を示します。下の図ではキー値が data_2 で重複している行がマージされています。

PostgreSQL 12以前

キー値	TID
data_1	tid_1
data_2	tid_2
data_2	tid_3
data_2	tid_4
data_3	tid_5

PostgreSQL 13

キー値	TID
data_1	tid_1
data_2	tid_2, tid_3, tid_4
data_3	tid_5

ところで、重複するインデックスタプルとは同時に見えるデータに限りません。TID が同じ行データでも更新前後の（バージョン違いの）インデックスタプルが存在するのでキー値が一致すればそれも重複するインデックスタプルとなります。ゆえに、ユニークインデックスでも重複排除は有用となります。

さらに、NULL 値同士は一般的に同じ値としては扱われませんが、重複排除の枠組みでは重複値としてみなされるので、重複排除の対象となります。

4.1.2. メリット・デメリット

重複排除のメリットはインデックスサイズが縮小されることにより、単にデータ容量削減だけでなく、クエリのパフォーマンス向上やバキュームのオーバーヘッド削減が見込まれます。

重複排除のデメリットは重複排除処理発生によるオーバーヘッドが生じることです。ただし、重複排除処理は遅延処理アプローチが採用されており、データ挿入時に毎回実行されることはなく、インデックスの既存のリーフページに収まりきらない場合にのみ実行されます。

4.1.3. パフォーマンス検証

インデックスの重複排除の設定によりインデックスサイズ、パフォーマンスがどのように変化するのかを検証しました。

本ケースでは、インデックスはいずれも単一カラムを対象としており、テーブルには同一の値を挿入し続けています。

- text 型のカラム一つを持ったテーブルを定義します。
  ここでは直接関係ありませんが、後に示す通り非決定的照合を使用した text 型は重複排除を利用できないので collate “C” としています。

```
db1=# CREATE TABLE btree_text (f1 text collate "C")
       WITH (autovacuum_enabled=false);
```

- インデックスを定義します。
  （パフォーマンス比較のため各インデックスは同時に存在しません）

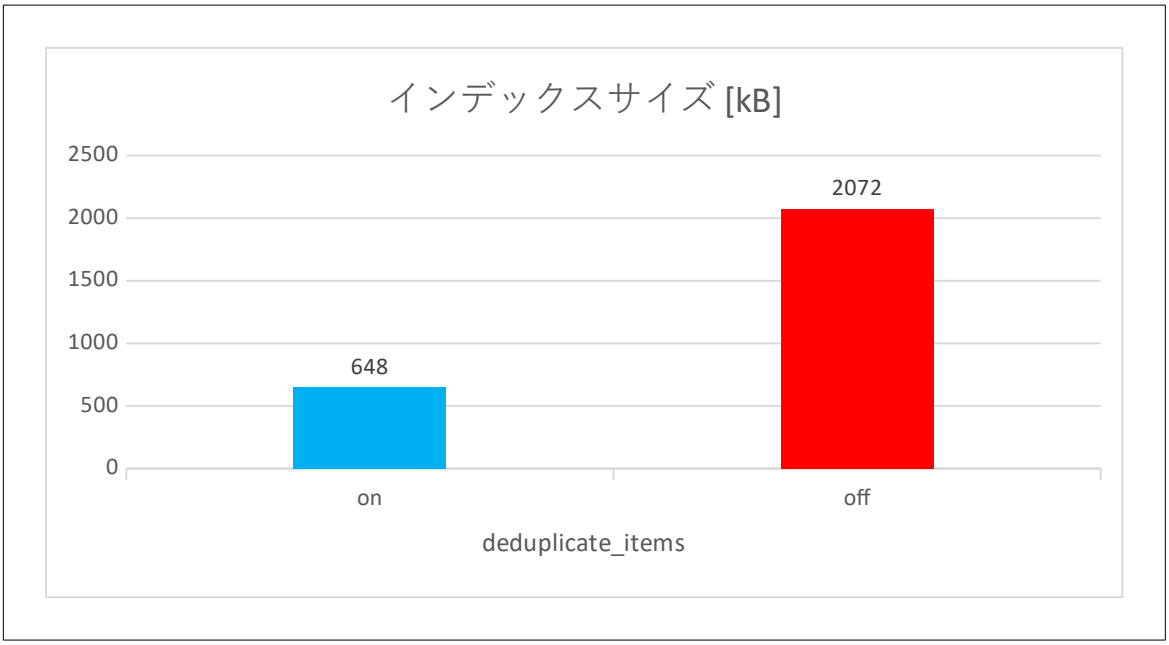
```
（重複排除を有効とするインデックスの場合）
db1=# CREATE INDEX dedup_idx ON btree_text(f1)
        WITH (deduplicate_items=on);

（重複排除を無効とするインデックスの場合）
db1=# CREATE INDEX dedup_idx ON btree_text(f1)
        WITH (deduplicate_items=off);
```

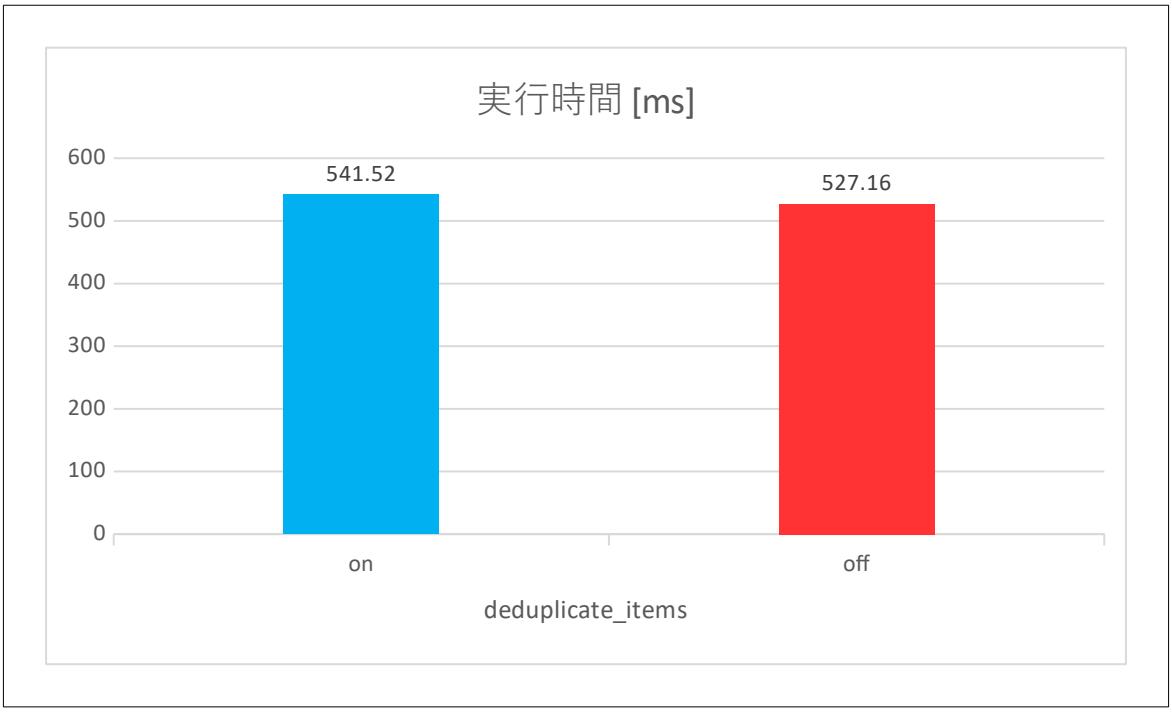
- データ挿入クエリを実行します。

```
db1=# INSERT INTO btree_text SELECT 'foo' FROM generate_series(1,100000);
```

結果、インデックスサイズは以下の図が示す通り、重複排除を有効にしたインデックスが無効にしたインデックスに比べ 1/3 以下になりました。



対して、データ挿入クエリの実行時間はわずかに重複排除を有効にした方が長くなりました。



今回はテーブルに大量に同一の値を挿入するという、重複排除のメリットとデメリットがはっきりするで

あろうケースを検証しました。結果として、インデックスサイズは圧倒的に小さくなり、データ挿入のパフォーマンスは重複排除のオーバーヘッドがわずかに存在することが示されました。

4.1.4. 重複排除の有効無効を決定する設定

あるインデックスで重複排除が適用されるかどうかはストレージパラメータ deduplicate_items で決定されます。デフォルトでは deduplicate_items = on で、重複排除は有効になっています。

インデックス作成時に重複排除を無効にするには CREATE INDEX で以下のように指定します。

```
CREATE INDEX title_idx ON films (title) WITH (deduplicate_items = off);
```

インデックス作成後に重複排除を無効にするには ALTER INDEX で以下のように指定します。この際、将来の重複排除は無効化されるものの、インデックス内部の表現は重複排除が適用されたインデックスタプルの状態が残存します。（重複排除を無効化してもインデックスサイズが増えることはありません。）

```
ALTER INDEX title_idx ON films (title) SET (deduplicate_items = off);
```

インデックスに定義したストレージパラメータは pg_class の reloptions 列で確認可能です。

```
SELECT reloptions FROM pg_class WHERE relname = 'title_idx';
      reloptions
-------------------------
 {deduplicate_items=off}
(1 row)
```

4.1.5. 重複排除が利用できない条件

B-Tree インデックスで deduplicate_items = on としても、重複排除が利用できない条件があります。

重複排除が利用できないキーの型は以下の通りです。

- 非決定的照合を使用した text, varchar, char
- numeric
- jsonb
- float4, float8
- コンテナ型(配列、複合型、範囲型)

また、INCLUDE 構文を用いて定義したインデックスでも重複排除が利用できません。

4.1.6. 過去の PostgreSQL バージョンからの移行

PostgreSQL12 以前のインデックスは pg_upgrade による PostgreSQL13 移行後も利用自体は可能ですが、重複排除の機能を利用するためには REINDEX の実行が必要になります。

4.2. 新しい実行プラン

4.2.1. インクリメンタルソート

インクリメンタルソートは複数キーのソート処理で使える新たなプラン要素です。既にソートされている部分結果を使って、残る部分を追加でソートする動作をします。EXPLAIN 文で実行プランを確認すると、インクリメンタルソートが使われた場合には「Sort」の代わりに「Incremental Sort」が現れます。このプラン要素は、設定 enable_incremental_sort が on（デフォルト）であるとき、選択されるようになります。

実際に使われる場合を確認してみます。

以下のようにログメッセージを格納するテーブルを作成します。日付時刻列（dt）、深刻度列（severity）、メッセージ列(message)を持ちます。適当なデータを日付時刻順に投入して、dt にインデックスを作ります。また、プランナ統計情報を更新しつつ、テーブル内容をバッファに載せるために、VACUUM ANALYZE 文も実行しておきます。

（テーブル作成）

```
db1=# CREATE TABLE t_log (dt timestamp(0), severity text, message text);
```

（ダミーデータを 1 万件投入）

```
db1=# INSERT INTO t_log SELECT
       now() + ((g / 2) || 's')::interval,
       CASE g % 3 WHEN 0 THEN 'LOG' WHEN 1 THEN 'ERROR' ELSE 'FATAL' END,
       md5(g::text)
       FROM generate_series(1, 10000) g;
```

（テーブル内容）

```
db1=# SELECT * FROM t_log;
         dt          | severity |              message
---------------------+----------+----------------------------------
 2020-05-25 14:09:51 | ERROR    | c4ca4238a0b923820dcc509a6f75849b
 2020-05-25 14:09:52 | FATAL    | c81e728d9d4c2f636f067f89cc14862c
```

```
 2020-05-25 14:09:52 | LOG      | eccbc87e4b5ce2fe28308fd9f2a7baf3
 2020-05-25 14:09:53 | ERROR    | a87ff679a2f3e71d9181a67b7542122c
 2020-05-25 14:09:53 | FATAL    | e4da3b7fbbce2345d7772b0674a318d5
 : 後略

 （インデックス作成、VACUUM ANALYZE）
db1=# CREATE INDEX ON t_log (dt);
db1=# VACUUM (ANALYZE) t_log;
```

ここで、t_log テーブルを dt、severity、message の 3 列でソートして、ある日時以降の先頭 1000 件を取り出す以下の問い合わせを実行することにします。

```
  SELECT * FROM t_log WHERE dt > ' 2020-05-25 14:10'
    ORDER BY dt, severity, message LIMIT 1000;
```

dt 列は B-Tree インデックスがあるため、既にソートされている状態といえます。このような場合でも従来は指定した列の組み合わせについて改めて全体をソートする以下のような実行プランになっていました。

```
 （インクリメンタルソートを使わない場合）
db1=# SET enable_incremental_sort TO off;
db1=# EXPLAIN (ANALYZE)
        SELECT * FROM t_log WHERE dt > ' 2020-05-25 14:10'
          ORDER BY dt, severity, message LIMIT 1000;
                                                  QUERY PLAN
-----------------------------------------------------------------------------
 Limit  (cost=765.86..768.36 rows=1000 width=46)
        (actual time=3.021..3.197 rows=1000 loops=1)
   ->  Sort  (cost=765.86..790.80 rows=9974 width=46)
             (actual time=3.019..3.085 rows=1000 loops=1)
         Sort Key: dt, severity, message
         Sort Method: top-N heapsort  Memory: 127kB
         ->  Seq Scan on t_log  (cost=0.00..219.00 rows=9974 width=46)
                                (actual time=0.013..1.470 rows=9981 loops=1)
               Filter: (dt > '2020-05-25 14:10:00'::timestamp without time zone)
               Rows Removed by Filter: 19
 Planning Time: 0.148 ms
```

```
 Execution Time: 3.309 ms
(9 rows)
```

インクリメンタルソートにより、ソート指定のうち dt 列が既にソート済みであることを認識して活用する以下のような実行プランになります。実行時の所要時間や使用するメモリ量が小さくなっていることが分かります。

（インクリメンタルソートを使った場合）

```
db1=# SET enable_incremental_sort TO on;
db1=# EXPLAIN (ANALYZE)
        SELECT * FROM t_log WHERE dt > ' 2020-05-25 14:10'
          ORDER BY dt, severity, message LIMIT 1000;
                                  QUERY PLAN
------------------------------------------------------------------------------
 Limit  (cost=0.38..74.06 rows=1000 width=46)
        (actual time=0.031..0.599 rows=1000 loops=1)
   ->  Incremental Sort  (cost=0.38..735.25 rows=9974 width=46)
                         (actual time=0.030..0.534 rows=1000 loops=1)
         Sort Key: dt, severity, message
         Presorted Key: dt
         Full-sort Groups: 32  Sort Method: quicksort  Average
           Memory: 27kB  Peak Memory: 27kB
         ->  Index Scan using t_log_dt_idx on t_log
               (cost=0.29..379.83 rows=9974 width=46)
               (actual time=0.011..0.333 rows=1001 loops=1)
               Index Cond:
                 (dt > '2020-05-25 14:10:00'::timestamp without time zone)
 Planning Time: 0.107 ms
 Execution Time: 0.649 ms
(9 rows)
```

「Incremental Sort」プラン要素部分のコストを見ますと、全件を返すまでのコスト（cost の 2 番目の値）は、通常のソートを使った場合の「Sort」プラン要素と変わらないように見えます。しかし、本問い合わせは LIMIT 句を伴いますので、全件返すまでの処理は不要です。「Sort」の場合には最初の 1 件を返すまでのコスト・所要時間（cost や actual time の 1 番目の値）が大きいため、SQL 全体のコスト・所要時間も大きくなっていることがわかります。

4.2.2. ストレージ上のハッシュ集約

ストレージを用いたハッシュ集約（HashAggregate）が可能になりました。

ハッシュ集約は GROUP BY を使った集約の問い合わせで使われるプラン要素です。これまでは処理過程で使用するハッシュテーブルを全てメモリ上に配置できる、すなわち、グルーピングする値の種類数に対して十分なメモリ量が work_mem で設定されている場合にのみ使用できました。work_mem が不足すると代わりにグループ集約（GroupAggregate）のプランが使われますが、これは事前にソート処理が必要となるため、しばしば性能が劣りました。

動作検証用にユーザのログインを管理するテーブル t_login を考えます。主キー列（id）とユーザ ID 列（uid）、日付時刻列（dt）を持ちます。10 万ユーザが 10 回ずつログインしたことにして 100 万行のデータを投入しておきます。

（テーブル作成）

```
db1=# CREATE TABLE t_login (id bigint primary key, uid int, dt timestamp);
```

（ダミーデータ作成と VACUUM ANALYZE）

```
db1=# INSERT INTO t_login SELECT g, g % 100000,
       now() + (g || 'sec')::interval FROM generate_series(1, 1000000) g;
db1=# VACUUM ANALYZE t_login;
```

このテーブルでログイン数を調べる問い合わせを実行し、ストレージハッシュ集約を有効にした場合と無効にした場合の実行プランを確認します。

> 注意
>
> hashagg_avoid_disk_plan 設定パラメータは 13 正式版では廃止されていますので、下記の検証結果は 13 正式版とは異なります。

（ログイン数を調べる問い合わせ）

```
db1=# SELECT uid, count(1) FROM t_login GROUP BY uid;
  uid   | count
--------+-------
 11233 |    10
 26264 |    10
 29007 |    10
```

```
 5468 |    10
  :  後略
```

（ストレージハッシュ集約が無効で work_mem が小さい場合）

```
db1=# SET hashagg_avoid_disk_plan TO on;
db1=# SET work_mem TO '4MB';
db1=# EXPLAIN (ANALYZE) SELECT uid, count(1) FROM t_login GROUP BY uid;
                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Finalize GroupAggregate  (cost=56127.50..85622.84 rows=101094 width=12)
                          (actual time=225.932..487.340 rows=100000 loops=1)
   Group Key: uid
   ->  Gather Merge  (cost=56127.50..83600.96 rows=202188 width=12)
                     (actual time=225.927..459.767 rows=300000 loops=1)
         Workers Planned: 2
         Workers Launched: 2
         ->  Partial GroupAggregate
               (cost=55127.48..59263.42 rows=101094 width=12)
               (actual time=201.999..260.557 rows=100000 loops=3)
               Group Key: uid
               ->  Sort  (cost=55127.48..56169.15 rows=416667 width=4)
                         (actual time=201.989..228.997 rows=333333 loops=3)
                     Sort Key: uid
                     Sort Method: external merge  Disk: 5624kB
                     Worker 0:  Sort Method: external merge  Disk: 4072kB
                     Worker 1:  Sort Method: external merge  Disk: 4128kB
                     ->  Parallel Seq Scan on t_login
                         (cost=0.00..10536.67 rows=416667 width=4)
                         (actual time=0.011..77.704 rows=333333 loops=3)
 Planning Time: 0.092 ms
 Execution Time: 491.311 ms
(15 rows)
```

（ストレージハッシュ集約が無効で work_mem が大きい場合）

```
db1=# SET hashagg_avoid_disk_plan TO on;
```

```
db1=# SET work_mem TO '64MB';
db1=# EXPLAIN (ANALYZE) SELECT uid, count(1) FROM t_login GROUP BY uid;
                                QUERY PLAN
---------------------------------------------------------------------------
 HashAggregate  (cost=21370.00..22380.94 rows=101094 width=12)
                (actual time=251.453..269.335
rows=100000 loops=1)
   Group Key: uid
   Peak Memory Usage: 14353 kB
   ->  Seq Scan on t_login  (cost=0.00..16370.00 rows=1000000 width=4)
                            (actual time=0.017..50.057 rows=1000000 loops=1)
 Planning Time: 0.091 ms
 Execution Time: 272.738 ms
(6 rows)
```

（ストレージハッシュ集約が有効で work_mem が小さい場合）

```
db1=# SET hashagg_avoid_disk_plan TO off;
db1=# SET work_mem TO '4MB';
db1=# EXPLAIN (ANALYZE) SELECT uid, count(1) FROM t_login GROUP BY uid;
                                    QUERY PLAN
---------------------------------------------------------------------------
 HashAggregate  (cost=36995.00..41912.19 rows=101094 width=12)
                (actual time=198.841..295.113 rows=100000 loops=1)
   Group Key: uid
   Planned Partitions: 4
   Peak Memory Usage: 4145 kB
   Disk Usage: 30376 kB
   HashAgg Batches: 4
   ->  Seq Scan on t_login  (cost=0.00..16370.00 rows=1000000 width=4)
                            (actual time=0.013..47.859 rows=1000000 loops=1)
 Planning Time: 0.092 ms
 Execution Time: 301.152 ms
(9 rows)
```

ストレージハッシュ集約は設定 enable_hashagg_disk で有効化・無効化を制御できます。デフォルトは

on （有効）です。

設定を変えて実行プランを調べた結果、work_mem が少なくて enable_hashagg_disk が有効な場合にストレージハッシュ集約が使われていることが分かります。実行コストおよび所要時間は、

グループ集約 ＞ ストレージハッシュ集約 ＞ ハッシュ集約

の順です。本例のグループ集約はパラレル処理が使われていますが、それでも最も遅い結果でした。

ストレージハッシュ集約のプランが候補にあることで、固定の work_mem で運用しているときにデータの変動によりハッシュ集約が使われなくなったとしても、極端な性能劣化が突然生じるのを防ぐことが期待できます。

4.3. パーティションテーブル機能の拡張

4.3.1. パーティションテーブルのロジカルレプリケーション

パーティションテーブルに対してロジカルレプリケーションを構成することができるようになりました。これは PostgreSQL12 でもできましたが、パーティションテーブルに属する各パーティションごとにレプリケーションを設定する必要がありました。PostgreSQL13 からは、シンプルにパーティションテーブルを指定してパブリケーション作成、サブスクリプション作成ができるようになります。

以下のパーティションテーブルを使って動作確認をしていきます。pt_member は何らかの会員リストを管理するテーブルで主キーの id 列の値の範囲でパーティショニングが行われていて、3 つのパーティションが所属しています。

（サンプルのパーティションテーブルを作成）

```
db1=# CREATE TABLE pt_member (id int primary key, email text, passwd text,
       status int) PARTITION BY RANGE (id);
db1=# CREATE TABLE p_member_1 PARTITION OF pt_member
       FOR VALUES FROM (1) TO (10001);
db1=# CREATE TABLE p_member_10001 PARTITION OF pt_member
       FOR VALUES FROM (10001) TO (20001);
db1=# CREATE TABLE p_member_20001 PARTITION OF pt_member
       FOR VALUES FROM (20001) TO (30001);
```

（ダミーデータを投入）

```
db1=# INSERT INTO pt_member
       SELECT g, 'u' || g || '@example.com', 'pass', 0
       FROM generate_series(1, 30000) g;
```

ロジカルレプリケーションを行うため、postgresql.conf の設定 wal_level を logical に変更します。

```
（設定を変更）
$ vi $PGDATA/postgresql.conf
          wal_level = logical;
（反映にはサービス再起動が必要）
$ pg_ctl restart
```

ロジカルレプリケーションのサブスクライバ側となるデータベースインスタンスを作成して、別ポートで起動します。また、サブスクライバとなるデータベース db2 も作成します。

```
（データベースクラスタ作成）
$ initdb -D /var/lib/pgsql/13/data2 --no-locale -E UTF8
    {出力省略}

（サービス起動）
$ pg_ctl start -D /var/lib/pgsql/13/data2 -o '-c port=5433'
    {出力省略}

（データベース db2 を作成）
$ createdb -p 5433 -U postgres db2
```

パブリケーション側 PostgreSQL（5432 ポート）に接続して、パーティションテーブルに対してパブリケーションを作成します。pg_publication_tables ビューを参照すると、パーティションテーブルを指定してパブリケーションを作成した結果、属する各パーティションがパブリケーション対象になっていることが確認できます。

```
（パブリケーション側データベースに接続）
$ psql -p 5432 -U postgres -d db1

（パブリケーション作成）
db1=# CREATE PUBLICATION pub1 FOR TABLE pt_member;

（パブリケーションされているテーブルを確認）
db1=# SELECT * FROM pg_publication_tables;
 pubname | schemaname |       tablename
---------+------------+------------------------
```

```
 pub1     | public     | p_member_1
 pub1     | public     | p_member_10001
 pub1     | public     | p_member_20001
(6 rows)
```

サブスクリプション側 PostgreSQL（5433 ポート）の db2 データベースに接続して、同定義のパーティションテーブルを作成して、サブスクリプションを作成します。

（サブスクリプション側データベースに接続）

```
$ psql -p 5433 -U postgres -d db2
```

（同定義のパーティションテーブルを作成）

```
db2=# CREATE TABLE pt_member (id int primary key, email text, passwd text,
         status int) PARTITION BY RANGE (id);
db2=# CREATE TABLE p_member_1 PARTITION OF pt_member
        FOR VALUES FROM (1) TO (10001);
db2=# CREATE TABLE p_member_10001 PARTITION OF pt_member
        FOR VALUES FROM (10001) TO (20001);
db2=# CREATE TABLE p_member_20001 PARTITION OF pt_member
        FOR VALUES FROM (20001) TO (30001);
```

（サブスクリプションを作成）

```
db2=# CREATE SUBSCRIPTION sub1 CONNECTION
        'host=localhost port=5432 user=postgres dbname=db1'
        PUBLICATION pub1;
NOTICE:  created replication slot "sub1" on publisher
```

（サブスクリプション対象テーブルを確認）

```
db2=# SELECT * FROM pg_subscription_rel;
 srsubid | srrelid | srsubstate |  srsublsn
---------+---------+------------+------------
   16444 |   16419 | r          | 0/2617D6E8
   16444 |   16427 | r          | 0/2617D6E8
   16444 |   16395 | r          | 0/2617D720
(3 rows)
```

```
（初期データ同期が行われていることを確認）
db2=# SELECT * FROM pt_member;
 id |       email       | passwd | status
----+-------------------+--------+--------
  1 | u1@example.com    | pass   |      0
  2 | u2@example.com    | pass   |      0
  3 | u3@example.com    | pass   |      0
  4 | u4@example.com    | pass   |      0
  :  {後略}
```

サブスクリプション作成においても、パーティションテーブルを指定するだけで所属する各パーティションをサブスクリプションに含めることができました。

◆ *publish_via_partition_root 指定*

パブリケーション側のパーティションテーブルに対して、サブスクリプション側ではパーティション全体を一つのテーブルとしてみなすようにロジカルレプリケーションを構成することもできます。以下のようにCREATE PUBLICATION のオプションで publish_via_partition_root = true を指定すると、パブリケーションテーブルとして登録されるのは、親のパーティションテーブル一つとなります。publish_via_partition_root はPostgreSQL13 からの新たなオプションでデフォルトは false です。

```
（パブリケーション側データベースに接続）
$ psql -p 5432 -U postgres -d db1

（パブリケーション削除）
db1=# DROP PUBLICATION pub1;

（単一テーブルとしてパブリケーション作成）
db1=# CREATE PUBLICATION pub2 FOR TABLE pt_member
       WITH (publish_via_partition_root = true);

（パブリケーションテーブルを確認）
db1=# SELECT * FROM pg_publication_tables;
 pubname | schemaname | tablename
---------+------------+-----------
 pub2    | public     | pt_member
(1 row)
```

サブスクリプション側で先ほど作成したサブスクリプション sub1 とパーティションテーブル pt_member を削除して、単体テーブルとして pt_member を作り、サブスクリプションを作成します。以下のように pg_subscription_rel ビューを参照するとサブスクリプション対象が sub1 のときには 3 件であったものが 1 件になっていることが確認できます。

```
（サブスクリプション側データベースに接続）
$ psql -p 5433 -U postgres -d db2

（サブスクリプションとパーティションテーブルを削除）
db2=# DROP SUBSCRIPTION sub1;
NOTICE:  dropped replication slot "sub1" on publisher
db2=# DROP TABLE pt_member;

（単体テーブルとして pt_member テーブルを再作成）
db2=# CREATE TABLE pt_member (id int primary key, email text,
        passwd text, status int);

（サブスクリプション作成）
db2=# CREATE SUBSCRIPTION sub2 CONNECTION
        'host=localhost port=5432 user=postgres dbname=db1'
         PUBLICATION pub2;
NOTICE:  created replication slot "sub2" on publisher

（サブスクリプションテーブルを確認）
db2=# SELECT * FROM pg_subscription_rel;
 srsubid | srrelid | srsubstate | srsublsn
---------+---------+------------+----------
   16448 |   16385 | d          |
(1 row)
```

パーティションテーブルから単体テーブルにロジカルレプリケーションしているときに、パブリケーション側で個別のパーティションを指定して行の更新を行うとどうなるでしょうか。結論としては、サブスクリプション側に問題なく更新が伝搬します。以下に動作確認を示します。

```
（パブリケーション側データベースでの実行）
$ psql -p 5432 -U postgres -d db1

（パーティションテーブルを通した行データ更新）
db1=# UPDATE pt_member SET passwd = 'NEW PASS' WHERE id = 123;

（個別パーティションを通した行データ更新）
db1=# UPDATE p_member_1 SET passwd = 'NEW PASS' WHERE id = 456;
```

```
（サブスクリプション側データベースでの実行）
$ psql -p 5433 -U postgres -d db2

（サブスクリプション側ではどちらの行も更新が反映されている）
db2=# SELECT * FROM pt_member WHERE id IN (123,456);
 id  |       email        |  passwd  | status
-----+--------------------+----------+--------
 123 | u123@example.com | NEW PASS |      0
 456 | u456@example.com | NEW PASS |      0
(2 rows)
```

4.3.2. パーティションテーブル同士の結合の改善

PostgreSQL11 から、パーティションテーブル同士の結合を同じ範囲を担当するパーティション毎に結合することで実現する実行方式が可能になっていました。しかしながら、これは二つのパーティションテーブルの各パーティションの境界が完全に一致していなければなりませんでした。PostgreSQL13 ではこの制限が緩和されました。

以下に動作を確認していきます。

4.3.1 節で作成したパーティションテーブル pt_member に加えて、それと結合させるパーティションテーブル pg_special_member を新たに作成します。会員のうち一部（本例では 30000 人に対して 6000 人だけ）が特別会員で、特別会員の場合に必要となる追加の情報（フルネームや電話番号、住所）を格納するテーブルという位置づけです。

```
（パーティションテーブル pt_special_member と子パーティションを作成）
db1=# CREATE TABLE pt_special_member(id int primary key, fullname text,
       phone text, addr text) PARTITION BY RANGE (id);
db1=# CREATE TABLE p_special_member_1 PARTITION OF pt_special_member
       FOR VALUES FROM (1) TO (10001);
db1=# CREATE TABLE p_special_member_10001 PARTITION OF pt_special_member
       FOR VALUES FROM (10001) TO (20001);
db1=# CREATE TABLE p_special_member_20001 PARTITION OF pt_special_member
       FOR VALUES FROM (20001) TO (30001);

（ダミーデータを 6000 件投入）
db1=# INSERT INTO pt_special_member SELECT
       g, 'fullname' || g, '0123-456-789', md5(g::text)
       FROM generate_series(1, 30000, 5) g;
INSERT 0 6000
```

まずは、パーティション境界が完全一致している状態で、パーティションごとの結合動作を確認します。

```
（デフォルトでは無効なパーティション毎の結合を有効化）
db1=# SET enable_partitionwise_join TO on;

（パーティション毎の結合になっていることを確認）
db1=# EXPLAIN (COSTS off)
       SELECT * FROM pt_member m LEFT JOIN pt_special_member s USING (id);
                          QUERY PLAN
-------------------------------------------------------------
 Append
   ->  Hash Join
         Hash Cond: (m_1.id = s_1.id)
         ->  Seq Scan on p_member_1 m_1
         ->  Hash
               ->  Seq Scan on p_special_member_1 s_1
   ->  Hash Join
         Hash Cond: (m_2.id = s_2.id)
         ->  Seq Scan on p_member_10001 m_2
         ->  Hash
```

```
            ->  Seq Scan on p_special_member_10001 s_2
  ->  Hash Join
        Hash Cond: (m_3.id = s_3.id)
        ->  Seq Scan on p_member_20001 m_3
        ->  Hash
              ->  Seq Scan on p_special_member_20001 s_3
(16 rows)
```

ここで、pt_memberパーティションテーブルにだけ新たな子パーティションを追加します。必要になったときに子パーティションを追加するという運用をしていて、通常会員のみが追加された場合を考えると、これはありうる状況です。

（pt_memberにパーティションとデータを追加）

```
db1=# CREATE TABLE p_member_30001 PARTITION OF pt_member
       FOR VALUES FROM (30001) TO (40001);
db1=# INSERT INTO pt_member
       VALUES (30001, 'new_member@example.com', 'pass', 0);
```

ここで再びパーティションテーブル同士の結合を試みると、内部結合では依然としてパーティション毎の結合をする動作になります。しかし、パーティション追加した側を残す外部結合ではパーティションテーブルを統合してから、それらを結合する動作に変わりました。

（内部結合ではパーティション毎の結合になる）

```
db1=# EXPLAIN (COSTS off)
       SELECT * FROM pt_member m JOIN pt_special_member s USING (id);
                    QUERY PLAN
-----------------------------------------------------------
 Append
  ->  Hash Join
        Hash Cond: (m_1.id = s_1.id)
        ->  Seq Scan on p_member_1 m_1
        ->  Hash
              ->  Seq Scan on p_special_member_1 s_1
  ->  Hash Join
        Hash Cond: (m_2.id = s_2.id)
```

```
        ->  Seq Scan on p_member_10001 m_2
        ->  Hash
              ->  Seq Scan on p_special_member_10001 s_2
   ->  Hash Join
         Hash Cond: (m_3.id = s_3.id)
        ->  Seq Scan on p_member_20001 m_3
        ->  Hash
              ->  Seq Scan on p_special_member_20001 s_3
(16 rows)
```

（左外部結合では Append してから結合するプランになる）

```
db1=# EXPLAIN (COSTS off)
       SELECT * FROM pt_member m LEFT JOIN pt_special_member s USING (id);
                                QUERY PLAN
-------------------------------------------------------------------------------
 Merge Right Join
   Merge Cond: (s.id = m.id)
    ->  Append
          ->  Index Scan using p_special_member_1_pkey
                on p_special_member_1 s_1
          ->  Index Scan using p_special_member_10001_pkey
                on p_special_member_10001 s_2
          ->  Index Scan using p_special_member_20001_pkey
                on p_special_member_20001 s_3
    ->  Materialize
          ->  Append
              ->  Index Scan using p_member_1_pkey on p_member_1 m_1
              ->  Index Scan using p_member_10001_pkey on p_member_10001 m_2
              ->  Index Scan using p_member_20001_pkey on p_member_20001 m_3
              ->  Index Scan using p_member_30001_pkey on p_member_30001 m_4
(12 rows)
```

内部結合（および、追加パーティションが無い側を残す外部結合）では新たに追加したパーティションは結合処理に加える必要がないので、合理的な動作といえます。PostgreSQL12 以前で同様の試験を行うと、内部結合であってもパーティション毎の結合が行われない動作になりました。

また、本例は RANGE パーティショニングでしたが、LIST パーティショニングでも同様に動作します。なお、本動作は HASH パーティショニングには対応していません。

4.3.3. 行単位 *BEFORE* トリガ対応

PostgreSQL 13 ではパーティションテーブルに行単位の BEFORE トリガが作成できるようになりました。

本節では 4.3.1 節、4.3.2 節で使ったパーティションテーブル pt_member にトリガを作成して、動作を確認していきます。メールアドレスを小文字に揃えるトリガを設定することにします。

```
（トリガ関数を作成）
db1=# CREATE FUNCTION tf_regularize() RETURNS TRIGGER LANGUAGE plpgsql
       AS $$ BEGIN NEW.email := lower(NEW.email); RETURN NEW; END; $$;

（トリガを作成）
db1=# CREATE TRIGGER trg_regularize_pt_member BEFORE INSERT OR UPDATE
       ON pt_member FOR ROW EXECUTE FUNCTION tf_regularize();

（メールアドレスを変更してトリガ動作を確認）
db1=# UPDATE pt_member SET email = 'U100_CHANGE@example.COM' WHERE id = 100;

db1=# SELECT * FROM pt_member WHERE id = 100;
 id  |          email           | passwd | status
-----+--------------------------+--------+--------
 100 | u100_change@example.com | xxxx   |      0
(1 row)
```

トリガが働いてメールアドレスが小文字に変換されていることが確認できました。これはごく一般的なトリガの使用方法の一つですが PostgreSQL12 までは、パーティションテーブルに行単位の BEFORE トリガを定義できず、以下のようなエラーになりました。

```
（PostgreSQL12 での動作）
db1=# CREATE TRIGGER trg_regularize_pt_member BEFORE INSERT OR UPDATE
       ON pt_member FOR ROW EXECUTE FUNCTION tf_regularize ();
ERROR:  "pt_member" is a partitioned table
DETAIL:  Partitioned tables cannot have BEFORE / FOR EACH ROW triggers.
```

なお、PostgreSQL13 においても、パーティションテーブルの行単位 BEFORE トリガで対象行データの属するパーティションを変える変更を加えることはできません。これは以下のようにトリガの実行時にエラーが生じます。

（トリガ関数を id を変更する内容に変更する）

```
db1=# CREATE OR REPLACE FUNCTION tf_regularize() RETURNS TRIGGER
       LANGUAGE plpgsql AS $$
         BEGIN NEW.id = NEW.id + 30000; RETURN NEW; END; $$;

db1=# UPDATE pt_member SET email = 'U500@example.COM' WHERE id = 500;
ERROR:  moving row to another partition during a BEFORE trigger is not
supported
DETAIL:  Before executing trigger "trg_regularize_pt_member", the row was to
be in partition "public.p_member_1".
```

4.4. パラレル *Vacuum*

4.4.1. 概要

PostgreSQL13 に Vacuum を並列に実行する機能が追加されました。これにより Vacuum 処理時間の短縮が期待できます。

ここでの並列実行とは複数のインデックスが存在する一つのテーブルに対して、各インデックスごとに Vacuum が実行されることを指します。

4.4.2. 並列数設定

Vacuum の並列数(parallel_degree)を指定するための構文は以下の通りです。

(SQL で実行する場合)

```
 VACUUM (PARALLEL parallel_degree) table
```

(シェルで実行する場合)

```
 vacuumdb --parallel=parallel_degree --table table
```

デフォルトでは、並列数はテーブル内のインデックス数と max_parallel_maintenance_workers で指定さ

れた数を比較して小さい方が採用されます。

並列処理を無効化したい場合は parallel_degree を 0 に指定する必要があります。このとき、max_parallel_maintenance_workers を 0 に設定しても良いですが、他のパラレル Vacuum およびパラレル CREATE INDEX の動作にも影響がでます。

その他の注意点は以下の通りです。

- parallel_degree にテーブル内のインデックス以上の数を指定しても動作しますが、実際にインデックスごとに動作するワーカープロセスは 1 つずつです。
- 並列オプションは FULL オプションとは同時に使用できません。
- インデックスのサイズが min_parallel_index_scan_size よりも大きい場合にのみ並列実行の対象内となります。

4.4.3. PostgreSQL9.5 から実装されたパラレル *Vacuum* との違い

以前、PostgreSQL9.5 から同様にパラレル Vacuum と呼ばれる機能が追加されていましたが、PostgreSQL13 で追加された機能とは異なります。

PostgreSQL13 で追加されたパラレル Vacuum は一つのテーブル内の複数のインデックスを対象に並列処理を行うのに対し、PostgreSQL9.5 で追加されたパラレル Vacuum は複数のテーブルを対象に並列処理を行います。

PostgreSQL9.5 で追加されたパラレル Vacuum を利用するには vacuumdb コマンドの --jobs オプションで並列数を指定します。

```
$ vacuumdb --jobs njobs
```

4.5. 進捗報告ビューの拡張

PostgreSQL9.6 から各データベース管理コマンドの進捗報告ビューが定義されるようになりました。これまでに進捗が確認できるようになったコマンドとビューの組み合わせは以下の通りです。

進捗報告対象コマンド	進捗報告ビュー
VACUUM (VACUUM FULL は含まれない)	pg_stat_progress_vacuum
CLUSTER, VACUUM FULL	pg_stat_progress_cluster
CREATE INDEX, REINDEX	pg_stat_progress_create_index

PostgreSQL13 では pg_basebackup と ANALYZE の進捗が確認できるようになりました。

4.5.1. pg_stat_progress_basebackup

pg_basebackup コマンドの進捗を確認できるようになりました。
ただし、pg_stat_backup() ~ pg_stop_backup() を直接利用したバックアップの進捗は確認できません。

pg_stat_progress_basebackup ビューの各列の意味は以下の通りです。

列名	意味
pid	walsender プロセス ID
phase	バックアップの状態。詳細は別表に示します。
backup_total	バックアップ合計データ推定サイズ
backup_streamed	現在バックアップされたデータサイズ
tablespaces_total	バックアップされる合計テーブル空間数
tablespaces_streamed	現在バックアップされたテーブル空間数

pg_stat_progress_basebackup ビューの phase 列に示される状態の意味は以下の通りです。

phase	意味
initializing	初期化中
waiting for checkpoint to finish	チェックポイント完了待ち
estimating backup size	バックアップサイズ推定中

phase	意味
streaming database files	バックアップデータストリーミング中
waiting for wal archiving to finish	WAL アーカイブ待ち
transferring wal files	WAL 送信中 (pg_basebackup -X f オプションで指定時)

以下に、pg_basebackup コマンドを実行して進捗を確認する例を示します。

(pg_stat_progress_basebackup ビューの参照を定期実行)

```
$ psql db1
db1=# \x
db1=# SELECT * FROM pg_stat_progress_basebackup;
(0 rows)

db1=# \watch
...
```

(別端末から pg_basebackup コマンド実行)

```
$ pg_basebackup -D /tmp/bak -Pv
pg_basebackup: initiating base backup, waiting for checkpoint to complete
pg_basebackup: checkpoint completed
pg_basebackup: write-ahead log start point: 0/75000028 on timeline 1
pg_basebackup: starting background WAL receiver
pg_basebackup: created temporary replication slot "pg_basebackup_8782"
1572815/1572815 kB (100%), 1/1 tablespace
pg_basebackup: write-ahead log end point: 0/75000138
pg_basebackup: waiting for background process to finish streaming ...
pg_basebackup: syncing data to disk ...
pg_basebackup: renaming backup_manifest.tmp to backup_manifest
pg_basebackup: base backup completed
```

(元の端末から進捗確認)

(\watch 実行中)

```
...
2020年05月26日 17時14分39秒 (every 2s)

-[ RECORD 1 ]--------+-------------------------
pid                  | 8781
phase                | streaming database files
backup_total         | 1610553344
backup_streamed      | 32807936
tablespaces_total    | 1
tablespaces_streamed | 0

...

2020年05月26日 17時14分45秒 (every 2s)

-[ RECORD 1 ]--------+-------------------------
pid                  | 8781
phase                | streaming database files
backup_total         | 1610553344
backup_streamed      | 1490317312
tablespaces_total    | 1
tablespaces_streamed | 0

(0 rows) →完了したので行が表示されなくなった

(0 rows)

...
```

backup_streamed が増え続けて最後には行が表示されなくなるのが確認できます。途中でキャンセルされていない限り、この時点でコマンド実行完了となります。完了状態を示す行が残るわけではないので注意が必要です。たとえば、backup_total = backup_streamed となったら完了などと判断してはいけません。

バックアップの進捗を大まかに進捗率として確認したい場合は、以下のように backup_streamed / backup_total を計算すればよいです。ただし、ここでの進捗率はあくまで phase が streaming database files 中のデータ受け渡しに関してのみで他の phase の進捗は無視しています。

```
=# SELECT phase, backup_streamed / backup_total::float * 100 as
"progress_ratio(%)" FROM pg_stat_progress_basebackup;
…
=# \watch

2020年05月26日 17時25分04秒 (every 2s)

-[ RECORD 1 ]-----+-------------------------
phase             | streaming database files
progress_ratio(%) | 32.19553708609419

2020年05月26日 17時25分06秒 (every 2s)

-[ RECORD 1 ]-----+-------------------------
phase             | streaming database files
progress_ratio(%) | 72.74181534727889

2020年05月26日 17時25分08秒 (every 2s)

-[ RECORD 1 ]-----+-----------------------------------
phase             | waiting for wal archiving to finish
progress_ratio(%) | 100

2020年05月26日 17時25分10秒 (every 2s)

-[ RECORD 1 ]-----+-----------------------------------
phase             | waiting for wal archiving to finish
progress_ratio(%) | 100
```

4.5.2. pg_stat_progress_analyze

ANALYZE コマンドの進捗を確認できるようになりました。

pg_stat_progress_analyze ビューの各列の意味は以下の通りです。

列名	意味

pid	ANALYZE を実行しているバックエンドプロセス ID
datid	ANALYZE 対象のデータベース OID
datname	ANALYZE 対象のデータベース名
relid	ANALYZE 対象のテーブル OID
phase	バックアップの状態。各状態の詳細は別表に示します。
sample_blks_total	サンプルブロック総数
sample_blks_scanned	スキャンが完了したサンプルブロック数
ext_stats_total	拡張統計の総数
ext_stats_computed	計算が完了した拡張統計数
child_tables_total	子テーブル数
child_tables_done	スキャンが完了した子テーブル数
current_child_table_relid	現在スキャンしている子テーブル OID

pg_stat_progress_analyze ビューの phase 列に示される状態の意味を示します。

phase	意味
initializing	初期化注
acquiring sample rows	サンプル行のスキャン中
acquiring inherited sample rows	子テーブルのサンプル行のスキャン中
computing statistics	スキャンした情報から統計情報を計算中
computing extended statistics	スキャンした情報から拡張統計情報を計算中
finalizing analyze	pg_class の更新中

以下に、ANALYZE コマンドを実行して進捗を確認する例を示します。

(pg_stat_progress_analyze ビューの参照を定期実行)

```
$ psql db1
db1=# \x
db1=# SELECT * FROM pg_stat_progress_analyze;
db1=# \watch
```

(別の端末で ANALYZE コマンド実行)

```
$ psql db1
db1=# ANALYZE;
ANALYZE
```

(元の端末に戻り進捗確認)

(\watch 実行中)

```
2020年05月26日 19時08分38秒 (every 2s)

-[ RECORD 1 ]-------------+----------------------
pid                       | 10714
datid                     | 16385
datname                   | db1
relid                     | 16544
phase                     | acquiring sample rows
sample_blks_total         | 30000
sample_blks_scanned       | 1765
ext_stats_total           | 0
ext_stats_computed        | 0
child_tables_total        | 0
child_tables_done         | 0
current_child_table_relid | 0

...

2020年05月26日 19時08分48秒 (every 2s)

-[ RECORD 1 ]-------------+----------------------
pid                       | 10714
datid                     | 16385
datname                   | db1
relid                     | 16544
phase                     | acquiring sample rows
sample_blks_total         | 30000
```

```
sample_blks_scanned       | 22669
ext_stats_total           | 0
ext_stats_computed        | 0
child_tables_total        | 0
child_tables_done         | 0
current_child_table_relid | 0

(0 rows)

(0 rows)

...
```

sample_blks_scanned が増え続けて最後には行が表示されなくなることが確認できます。途中でキャンセルされていない限り、この時点でコマンド実行完了となります。完了状態を示す行が残るわけではないので注意が必要です。たとえば、sample_blks_total = sample_blks_scanned となったら完了などと、判断してはいけません。

ところで、ANALYZE のサンプル行数は default_statistics_target(デフォルト 100) * 300 で決定しますが 、sample_blks_total(=30000) がそれと一致することが確認できます。テーブルの総行数がこれを下回る場合、サンプル行数はテーブル総行数に一致します。

注意としては、 ANALYZE の対象が複数テーブルだと全体の進捗が分かりにくいことがあります 。ANALYZE 対象テーブルの総数と ANALYZE が完了したテーブル数を示す列などは存在しません。

4.6. pgbench の拡張

PostgreSQL の標準ベンチマークツールである pgbench の機能が拡張されました。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。

4.6.1. 標準シナリオのテストデータをパーティションテーブル化

pgbench の組み込みシナリオで用いるテーブルの初期化時(-i オプション使用時)に `--partition` オプションで 1 以上の数を指定すると指定した数の子テーブルを持つパーティションテーブルを生成することができるようになりました。

また、`--partition-method` オプションで、パーティションの方法をレンジパーティション（RANGE）

かハッシュパーティション（HASH）かを指定可能です。デフォルトはレンジパーティションです。

以下に、pgbench 用のテーブルをパーティションテーブルとして初期化した例を示します。

（ベンチマーク用データベース作成）

```
$ createdb pgbench
```

（ベンチマーク用テーブル初期化）

```
$ pgbench -i -s 50 --partitions 5 pgbench
dropping old tables...
creating tables...
creating 5 partitions...
generating data (client-side)...
5000000 of 5000000 tuples (100%) done (elapsed 19.86 s, remaining 0.00 s)
vacuuming...
creating primary keys...
done in 36.53 s (drop tables 0.19 s, create tables 0.03 s, client-side
generate 20.21 s, vacuum 4.33 s, primary keys 11.76 s).

$ psql pgbench
```

（ベンチマーク用テーブルがパーティションテーブルとなっていることを確認）

```
pgbench=# \dt+ pgbench_accounts
                                  List of relations
 Schema |       Name       |       Type        |  Owner   | Persistence |
Size    | Description
--------+------------------+-------------------+----------+-------------
+---------+-------------
 public | pgbench_accounts | partitioned table | postgres | permanent   | 0
bytes |
(1 row)
```

（5 個の子テーブルを持ったレンジパーティションとなっていることを確認）

```
pgbench=# \d+ pgbench_accounts
                        Partitioned table "public.pgbench_accounts"
  Column  |     Type      | Collation | Nullable | Default | Storage  |
Stats target | Description
----------+---------------+-----------+----------+---------+----------
+--------------+-------------
```

```
 aid      | integer       |           | not null |         | plain    |
|
 bid      | integer       |           |          |         | plain    |
|
 abalance | integer       |           |          |         | plain    |
|
 filler   | character(84) |           |          |         | extended |
|
Partition key: RANGE (aid)
Indexes:
    "pgbench_accounts_pkey" PRIMARY KEY, btree (aid)
Partitions: pgbench_accounts_1 FOR VALUES FROM (MINVALUE) TO (1000001),
          pgbench_accounts_2 FOR VALUES FROM (1000001) TO (2000001),
          pgbench_accounts_3 FOR VALUES FROM (2000001) TO (3000001),
          pgbench_accounts_4 FOR VALUES FROM (3000001) TO (4000001),
          pgbench_accounts_5 FOR VALUES FROM (4000001) TO (MAXVALUE)
```

パーティションテーブル化の効果がこの機能により簡単にテストできるようになりました。

例えば、パーティションテーブルの効果を見るためにインデックスのないテーブルに対して、通常のテーブルとパーティションテーブルでどの程度パフォーマンスが異なるのかを以下で比較してみます。

（ベンチマーク用テーブル作成）

```
$ pgbench -i -I dtgv -s 120 pgbench
                            List of relations
 Schema |       Name       | Type  |  Owner   | Persistence |  Size   |
Description
--------+------------------+-------+----------+-------------+---------
+-------------
 public | pgbench_accounts | table | postgres | permanent   | 1537 MB |
 public | pgbench_branches | table | postgres | permanent   | 40 kB   |
 public | pgbench_history  | table | postgres | permanent   | 0 bytes |
 public | pgbench_tellers  | table | postgres | permanent   | 88 kB   |
(4 rows)

pgbench=# \d+ pgbench_accounts
                              Table "public.pgbench_accounts"
```

```
  Column   |     Type      | Collation | Nullable | Default | Storage  |
Stats target | Description
----------+---------------+-----------+----------+---------+----------
+--------------+-------------
 aid      | integer       |           | not null |         | plain    |
|
 bid      | integer       |           |          |         | plain    |
|
 abalance | integer       |           |          |         | plain    |
|
 filler   | character(84) |           |          |         | extended |
|
Access method: heap
Options: fillfactor=100
```

（テーブルデータをメモリに乗せるため pg_prewarm 実行）

```
pgbench=# CREATE EXTENSION pg_prewarm; --pg_prewarm 拡張追加
pgbench=# SELECT pg_prewarm('pgbench_accounts');
 pg_prewarm
------------
     196722
(1 row)
```

（pgbench 参照シナリオ実行）

```
$ pgbench -T 300 -S -n pgbench
transaction type: <builtin: select only>
scaling factor: 120
```

：《中略》

```
number of transactions actually processed: 162
```

（ベンチマーク用パーティションテーブル作成）

```
$ pgbench -i -I dtgv -s 120 --partitions 10 pgbench
$ psql pgbench
pgbench=# \d+ pgbench_accounts
```

```
                        Partitioned table "public.pgbench_accounts"
  Column  |     Type      | Collation | Nullable | Default | Storage  |
Stats target | Description
----------+---------------+-----------+----------+---------+----------
+--------------+-------------
 aid      | integer       |           | not null |         | plain    |
|
 bid      | integer       |           |          |         | plain    |
|
 abalance | integer       |           |          |         | plain    |
|
 filler   | character(84) |           |          |         | extended |
|
Partition key: RANGE (aid)
Partitions: pgbench_accounts_1 FOR VALUES FROM (MINVALUE) TO (1200001),
          pgbench_accounts_10 FOR VALUES FROM (10800001) TO (MAXVALUE),
          pgbench_accounts_2 FOR VALUES FROM (1200001) TO (2400001),
          pgbench_accounts_3 FOR VALUES FROM (2400001) TO (3600001),
          pgbench_accounts_4 FOR VALUES FROM (3600001) TO (4800001),
          pgbench_accounts_5 FOR VALUES FROM (4800001) TO (6000001),
          pgbench_accounts_6 FOR VALUES FROM (6000001) TO (7200001),
          pgbench_accounts_7 FOR VALUES FROM (7200001) TO (8400001),
          pgbench_accounts_8 FOR VALUES FROM (8400001) TO (9600001),
          pgbench_accounts_9 FOR VALUES FROM (9600001) TO (10800001)

pgbench=# \dt+
 Schema |        Name         |       Type        |  Owner   | Persistence |
Size    | Description
--------+---------------------+-------------------+----------+-------------
+---------+-------------
 public | pgbench_accounts    | partitioned table | postgres | permanent   |
0 bytes |
 public | pgbench_accounts_1  | table             | postgres | permanent   |
154 MB  |
 public | pgbench_accounts_10 | table             | postgres | permanent   |
```

```
154 MB  |
 public | pgbench_accounts_3  | table             | postgres | permanent   |
154 MB  |
 public | pgbench_accounts_4  | table             | postgres | permanent   |
154 MB  |
 public | pgbench_accounts_5  | table             | postgres | permanent   |
154 MB  |
 public | pgbench_accounts_6  | table             | postgres | permanent   |
154 MB  |
 public | pgbench_accounts_7  | table             | postgres | permanent   |
154 MB  |
 public | pgbench_accounts_9  | table             | postgres | permanent   |
154 MB  |
 public | pgbench_branches    | table             | postgres | permanent   |
40 kB   |
 public | pgbench_history     | table             | postgres | permanent   |
0 bytes |
 public | pgbench_tellers     | table             | postgres | permanent   |
88 kB   |
(14 rows)
```

（各パーティションテーブルに対して pg_prewarm 実行）

```
$ psql pgbench
pgbench=# SELECT pg_prewarm('pgbench_accounts_1');
 : 《中略》
pgbench=# SELECT pg_prewarm('pgbench_accounts_10');
```

（pgbench 参照シナリオ実行）

```
$ pgbench -T 300 -S -n pgbench
transaction type: <builtin: select only>
scaling factor: 120
partition method: range
partitions: 10
...
number of transactions actually processed: 1353
```

上記の例では、pgbench の参照シナリオにおいて、パーティションテーブルの方は通常テーブルに対し約 8 倍のトランザクション数が処理できています。

4.6.2. サーバ側で標準シナリオのテストデータを生成

以前はクライアント側で標準シナリオのテストデータを生成してサーバに送信する方法しかありませんでしたが、PostgreSQL13 ではサーバ側で直接データを生成できるようになりました。

具体的な挙動で比較すると、これまでクライアント側で作成したデータを COPY でサーバ側に送信していた方式に対し、クライアント側から INSERT クエリを発行しサーバ側で実行する方式が追加されました。

サーバ側でのデータ生成方式はサーバ・クライアント間の通信帯域が狭い場合に有効とされています。

どちら側でデータを生成するかは初期化時に初期化ステップを指定する -I,--init-steps オプションでクライアント側での生成であれば g, サーバ側での生成であれば G を指定します。デフォルト動作は以前からのクライント側での生成となります。

以下で、クライアント側とサーバ側での生成の比較した例を示します。

（クライアント側でデータ生成）

```
$ pgbench -i -I dtgvp -s 100 pgbench
dropping old tables...
creating tables...
generating data (client-side)...
10000000 of 10000000 tuples (100%) done (elapsed 21.61 s, remaining 0.00 s)
vacuuming...
creating primary keys...
done in 47.98 s (drop tables 0.05 s, create tables 0.01 s, client-side
generate 22.00 s, vacuum 5.10 s, primary keys 20.82 s).
```

（サーバ側でデータ生成）

```
$ pgbench -i -I dtGvp -s 100 pgbench
dropping old tables...
creating tables...
generating data (server-side)...
vacuuming...
creating primary keys...
done in 114.01 s (drop tables 0.21 s, create tables 0.01 s, server-side
generate 79.51 s, vacuum 2.45 s, primary keys 31.84 s).
```

この例ではクライアント側でデータ生成した方が早いという結果になりました。これは本検証環境ではサーバとクライアントが同一ホストであり、通信帯域が十分であるため INSERT よりも COPY によるデータ生成速度が勝ったものと思われます。

また、クライアント側でのデータ生成では 10 万行生成するごとに進捗情報が表示されましたが、サーバ側でのデータ生成は進捗情報が表示されません。

4.6.3. 標準シナリオのテストデータ生成中の進捗表示の一部変更

クライアント側の標準シナリオのテストデータ生成中の進捗の表示が以下の内容で変更されました。

- 以前は進捗を示す行が複数行に渡って表示されていましたが、PostgreSQL13 では 1 行で示されるようになりました。
- 以前はデータ生成にかかった総時間のみが表示されていましたが、PostgreSQL13 ではデータ生成中の状態（drop tables, vacuum など）ごとにかかった時間も表示するようになりました。

以下に、PostgreSQL12,13 の比較実行例を示します。

```
（PostgreSQL12 で実行）
$ pgbench -i -s 300 pgbench
dropping old tables...
creating tables...
generating data...
100000 of 30000000 tuples (0%) done (elapsed 0.41 s, remaining 123.41 s)
200000 of 30000000 tuples (0%) done (elapsed 0.71 s, remaining 105.07 s)
…
30000000 of 30000000 tuples (100%) done (elapsed 189.45 s, remaining 0.00 s)
vacuuming...
creating primary keys...
done.

（PostgreSQL13 で実行）
$ pgbench -i -s 300 pgbench
dropping old tables...
creating tables...
generating data...
30000000 of 30000000 tuples (100%) done (elapsed 104.95 s, remaining 0.00 s)
→100000 of 30000000 tuples  から始まり、同じ行で表示が変化している
vacuuming...
creating primary keys...
done in 270.98 s (drop tables 0.74 s, create tables 0.02 s, client-side
generate 105.20 s, vacuum 89.84 s, primary keys 75.19 s).
```

4.7. pg_rewind の拡張

pg_rewind コマンドは、ストリーミングレプリケーションで追随したいプライマリ PostgreSQL に合わせてデータディレクトリを補正するものです。PostgreSQL13 で pg_rewind の機能がいくつか拡張されました。

4.7.1. 自動クラッシュリカバリ

pg_rewind を使う際の注意点は、対象のデータディレクトリは正常に PostgreSQL を停止したものでなければならないということです。したがって、非正常に終了していた場合には一度 PostgreSQL サービスを起動・停止して、クラッシュリカバリ処理を適用させてから、pg_rewind を使う必要がありました。

これに相当する処理が自動的に行われるようになりました。--no-ensure-shutdown オプションを与えると従来通りの振る舞いになります。

以下に動作を確認します。スタンバイサーバを作って、それを 1 回昇格して、更新を加えて、非正常停止をしたものを、pg_rewind で巻き戻すことを試みます。

```
（pg_rewind を使うため wal_log_hints を有効にします）
$ vi $PGDATA/postgresql.conf
   wal_log_hints = on
$ pg_ctl restart

（スタンバイサーバを作り、少なくとも 1 回はチェックポイントを経た状態にします）
$ pg_basebackup -D stdby_data --write-recovery-conf --checkpoint=fast
$ pg_ctl start -D stdby_data -o '-c port=5434'
$ psql -c "CHECKPOINT"

（スタンバイサーバを昇格して、更新して、immediate モードで停止します）
$ pg_ctl promote -D stdby_data
$ psql -p 5434 -d db1 -c "CREATE TABLE t_new (id int)"
$ pg_ctl stop -m immediate -D stdby_data

（オプションを指定して従来の pg_rewind の振る舞いにするとエラーが返ります）
$ pg_rewind -D stdby_data --source-server="host=localhost port=5432" \
    --no-ensure-shutdown
pg_rewind: fatal: target server must be shut down cleanly
```

```
（デフォルト動作ではクラッシュリカバリ処理が自動で最初に実行されます）
$ pg_rewind -D stdby_data \
    --source-server="host=localhost port=5432"
pg_rewind: executing "/usr/local/pgsql/13/bin/postgres" for target server to
complete crash recovery
2020-06-03 18:06:45.083 JST [3907] LOG:  database system was interrupted; last
known up at 2020-06-03 18:06:27 JST
2020-06-03 18:06:45.084 JST [3907] LOG:  database system was not properly shut
down; automatic recovery in progress
2020-06-03 18:06:45.084 JST [3907] LOG:  redo starts at 0/36000178
2020-06-03 18:06:45.084 JST [3907] LOG:  invalid record length at 0/36012060:
wanted 24, got 0
2020-06-03 18:06:45.084 JST [3907] LOG:  redo done at 0/36011EB0

PostgreSQL stand-alone backend 13beta1
backend> pg_rewind: servers diverged at WAL location 0/36000148 on timeline 1
pg_rewind: rewinding from last common checkpoint at 0/36000098 on timeline 1

pg_rewind: Done!
```

4.7.2. リカバリ設定作成オプション

pg_rewind にオプション `--write-recovery-conf` が追加されました。

pg_rewind で巻き戻したデータディレクトリはスタンバイ PostgreSQL として使われます。したがって、次に行うことはスタンバイとして動作するように設定することです。これを自動的に行ってくれます。具体的には、standby.signal ファイルが追加され、ソースサーバに指定したサーバをプライマリとして接続する設定が postgresql.auto.conf に追記されます。

途中まで先ほどと同じ手順で進めて動作を確認します。

```
（スタンバイサーバを昇格して、更新するところまでは、手前の例と同じ手順で、
 停止するところで、今度は fast モードで停止します）
$ pg_ctl promote -D stdby_data
$ psql -p 5434 -d db1 -c "CREATE TABLE t_new (id int)"
$ pg_ctl stop -m fast -D stdby_data
```

```
（--write-recovery-conf を指定するとスタンバイ用の設定が自動で付加されています）
$ pg_rewind -D stdby_data --source-server="host=localhost port=5432" \
  --write-recovery-conf
pg_rewind: servers diverged at WAL location 0/3A000148 on timeline 1
pg_rewind: rewinding from last common checkpoint at 0/3A000098 on timeline 1

pg_rewind: Done!

$ ls stdby_data/standby.signal
stdby_data/standby.signal

$ cat stdby_data/postgresql.auto.conf
# Do not edit this file manually!
# It will be overwritten by the ALTER SYSTEM command.
primary_conninfo = 'user=postgres passfile=''/var/lib/pgsql/.pgpass''
channel_binding=prefer host=localhost port=5432 sslmode=prefer
sslcompression=0 gssencmode=disable krbsrvname=postgres
target_session_attrs=any'
```

4.7.3. WAL アーカイブの使用

巻き戻すべき更新の量が多い場合には pg_rewind は失敗します。そのような場合には巻き戻したいターゲット PostgreSQL の WAL アーカイブから WAL ファイルを pg_wal ディレクトリに手動コピーして対処することになっていました。これを自動的に行うことができるようになりました。

以下に動作を確認します。

```
（WAL アーカイブを有効にします）
$ mkdir /var/lib/pgsql/arc_13 /var/lib/pgsql/arc_13r
$ vi $PGDATA/postgresql.conf
   archive_mode = on
   archive_command = 'cp %p /var/lib/pgsql/arc_13/%f'
$ pg_ctl restart

（スタンバイサーバを作り、スタンバイ側も WAL アーカイブを有効にしておきます）
$ pg_basebackup -D stdby_data --write-recovery-conf --checkpoint=fast
$ vi stdby_data/postgresql.conf
```

```
   archive_command = 'cp %p /var/lib/pgsql/arc_13r/%f'
$ pg_ctl start -D stdby_data -o '-c port=5434'
$ psql -c "CHECKPOINT"

（スタンバイサーバを昇格して、pgbench で大量更新して、fast モードで停止します）
$ pg_ctl promote -D stdby_data
$ pgbench -i -p 5434 db1
$ pgbench -t 1000 -p 5434 db1
$ pg_ctl stop -m fast -D stdby_data

（pg_rewind は必要な WAL ファイルが無いためにエラーになります）
$ pg_rewind -D stdby_data --source-server="host=localhost port=5432"
pg_rewind: servers diverged at WAL location 0/44000148 on timeline 1
pg_rewind: error: could not open file
"stdby_data/pg_wal/000000020000000000000044": そのようなファイルやディレクトリはあ
りません
pg_rewind: fatal: could not find previous WAL record at 0/44000148

（restore_command を設定して、--restore-target-wal オプションを与えると成功します）
$ vi stdby_data/postgresql.conf
   restore_command = 'cp /var/lib/pgsql/arc_13r/%f %p'

$ pg_rewind -D stdby_data --source-server="host=localhost port=5432" \
    --restore-target-wal
pg_rewind: servers diverged at WAL location 0/44000148 on timeline 1
pg_rewind: rewinding from last common checkpoint at 0/44000098 on timeline 1

pg_rewind: Done!
```

巻き戻しのための WAL アーカイブのコピーには restore_command の指定が使われます。

このとき restore_command で指定するのは、昇格したスタンバイ PostgreSQL 自身が出力した WAL アーカイブをオンライン WAL ディレクトリ（pg_wal）に書き戻すコマンドです。プライマリ側（ソース側）PostgreSQL の WAL アーカイブからではないことに注意してください。

この後、スタンバイとして起動するときに restore_command を設定する場合にはプライマリ側 PostgreSQL の WAL アーカイブディレクトリからコピーするコマンドを指定しなければなりません。

4.8. SQL 機能の追加

問い合わせに新たな機能を提供する新たな SQL 構文や関数がいくつか追加されています。本節ではこれらのうち重要なものを取り上げます。

4.8.1. FETCH FIRST WITH TIES

問い合わせ結果を一部の行だけ返す SELECT 文のオプションとして、PostgreSQL は LIMIT … OFFSET … と OFFSET … FETCH … の二種類の構文をサポートしています。前者が PostgreSQL 固有の構文で、後者が SQL 標準に準拠した構文です。このうち OFFSET … FETCH … 構文について、新たに WITH TIES オプションがサポートされました。

以下に挙動を確認していきます。各行にスコア値の列（score）を持つテーブルをスコア順にソートしてトップ 3 件を取得します。

（検証用のテーブルとデータを作ります）

```
db1=# CREATE TABLE t_rank (id int, score int);
db1=# INSERT INTO t_rank VALUES (1, 100), (2, 90), (3, 90), (4, 90), (5,
85), (6, 85), (7, 80), (8, 80), (9, 75), (10, 60);
db1=# SELECT * FROM t_rank;
 id | score
----+-------
  1 |   100
  2 |    90
  3 |    90
  4 |    90
  5 |    85
  6 |    85
  7 |    80
  8 |    80
  9 |    75
 10 |    60
(10 rows)
```

（従来からサポートされている ONLY 指定の場合、指定した行数通りが返ります）

```
db1=# SELECT * FROM t_rank ORDER BY score DESC
```

```
       OFFSET 0 ROWS FETCH FIRST 3 ROWS ONLY;
 id | score
----+-------
  1 |   100
  3 |    90
  2 |    90
(3 rows)

（WITH TIES 指定の場合、指定行数を超えても同順位の全行を含んだ結果が返ります）
db1=# SELECT * FROM t_rank ORDER BY score DESC
       OFFSET 0 ROWS FETCH FIRST 3 ROWS WITH TIES;
 id | score
----+-------
  1 |   100
  2 |    90
  3 |    90
  4 |    90
(4 rows)
```

WITH TIES を指定した場合、同順位の場合には例え指定行数を超えても、それらを全て含んだ結果が返ることとが確認できました。ONLY 指定の場合には、同じ score 値のどの行が選ばれるかは不定でした。

4.8.2. JSON Path で datetime()メソッドに対応

JSON Path 式の中で datetime() メソッドに対応しました。これは日付、時刻、または、日付時刻（タイムスタンプ）をあらわす文字列を JSON Path の中でデータ型を付与して大小比較を可能にするものです。

以下のように使用することができます。

```
（配列の中から 12 時より大きい時刻の値を選択）
db1=# SELECT jsonb_path_query('["12:30:54", "9:10:00", "13:20:10"]',
       '$[*].datetime() ? (@ > "12:00:00".datetime())');
 jsonb_path_query
------------------
 "12:30:54"
 "13:20:10"
(2 rows)
```

datetime メソッドに整形テンプレートを引数で与えて、任意の日付時刻表現に対応させることもできます。指定できる整形要素は SQL 関数 to_timestamp と同じです。

（整形テンプレートを指定する例）

```
db1=# SELECT jsonb_path_query(
        '{"timestamp": "2020-06-06 05:05:05 PM +09:00"}'::jsonb,
        '$.timestamp.datetime("YYYY-MM-DD HH:MI:SS AM TZH:TZM")');
      jsonb_path_query
-----------------------------
 "2020-06-06T17:05:05+09:00"
(1 row)
```

また、JSON Path 問い合わせを処理する各 SQL 関数に「_tz」という接尾辞が付加されたものが加わっています。これらは datetime メソッドを使った処理で、タイムゾーンが省略されているデータに対して、実行環境の現在タイムゾーンをデフォルトとして付与します。

（+8 タイムゾーンの 12 時より手前の時刻を選択、
JSON 配列の時刻には実行環境から日本の+9 タイムゾーンが適用される）

```
db1=# SELECT jsonb_path_query_tz(
        '["12:30:54", "9:10:00", "13:20:10"]',
        '$[*].datetime() ? (@ < "12:00:00 +08".datetime())');
 jsonb_path_query_tz
---------------------
 "12:30:54"
 "09:10:00"
(2 rows)
```

5. さまざまな仕様変更

PostgreSQL13 では以前のバージョンとは非互換の仕様変更があります。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. SIMILAR TO ... ESCAPE NULL が NULL を返すよう変更

SIMILAR TO … ESCAPE NULL が必ず NULL を返すようになりました。この新しい動作は、標準 SQL の仕

様と一致しています。これは SQL 関数 substring(text FROM pattern ESCAPE text)にも適用されます。

以前の動作では、エスケープがデフォルトのバックスラッシュ文字に設定されていました。元々定義していたビューでは元の関数を変更せずに維持することで、以前の動作が保持されます。

```
（PostgreSQL12 で実行）
db1=# SELECT 'hello\nworld' similar to 'hello\\nworld' escape null;
 ?column?
----------
 t
(1 row)

db1=# SELECT 'hello\nworld' similar to 'hello\\nworld' escape '\';
 ?column?
----------
 t
(1 row)

（PostgreSQL13 で実行）
db1=# SELECT 'hello\nworld' similar to 'hello\\nworld' escape null;
 ?column?
----------
            《null が返る》
(1 row)

db1=# SELECT 'hello\nworld' similar to 'hello\\nworld' escape '\';
 ?column?
----------
 t
(1 row)
```

5.2. *effective_io_concurrency* 値の意味が変更

I/O 並列動作に影響する effective_io_concurrency 値は、以前はディスクドライブ数を前提に指定することで、RAID スピンドルと確率に関する理論に基づいた式を通してビットマップヒープスキャンでプリフェッチ

するページ数を並列実行数として設定されていました。

しかし、PostgreSQL13 からは effective_io_concurrency 値で直接ビットマップヒープスキャンでプリフェッチするページ数を設定するようになりました。

以前の設定値から新しい値への変換式は以下の通りです。

$$NEW = round\left(\sum_{i=1}^{OLD} \frac{OLD}{i}\right)$$

また、以下のクエリで変換可能です。

```
SELECT
  round(sum(《以前の値》/ n::float) FROM generate_series(1, 《以前の値》) s(n);
```

例えば、以前の値が 1 であれば新しい値は 1, 以前の値が 10 であれば新しい値は 29 となります。

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。