# PostgreSQL 16 検証レポート


1.2 版
2023 年 9 月 29 日

SRA OSS LLC
〒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 16 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 16 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。
2023 年 5 月 25 日にリリースされた PostgreSQL 16 beta1 を使用して検証を行いました。その後、2023 年 6 月にリリースされた beta2、2023 年 8 月にリリースされた beta3 での変更内容を反映して、本文書を作成しています。

# 2. 概要

PostgreSQL 16 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。

性能向上
- パラレルクエリ対応追加
- SELECT DISTINCT でインクリメンタルソート
- ウィンドウ関数の最適化
- 同時 COPY 性能向上
- SIMD CPU アクセラレータ対応

SQL 機能
- SQL/JSON 対応
- ANY_VALUE 集約関数
- 整数リテラル表現
- 照合順序の拡張

ロジカルレプリケーション機能追加
- スタンバイ上でのパブリケーション
- 大トランザクションの並列適用
- 主キー以外インデックスの利用
- 初期コピーでバイナリフォーマット利用
- 双方向ロジカルレプリケーション

クライアント機能
- libpq ロードバランス
- psql で拡張プロトコル対応
- pg_dump 圧縮オプション

運用管理
- 新たなモニタリング項目

- 新たな定義済みロール
- ページ凍結の改善
- VACUUM リングバッファ指定
- Meson ビルド

これらに加えて、非互換の変更点についても解説します。

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 16 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/16/release-16.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 16（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 RHEL 8.x (x86_64) 互換環境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、openssl-devel、libzstd-devel、lz4-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。事前に postgres ユーザで読み書き可能な /usr/local/pgsql/16 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/v16beta2/postgresql-
16beta2.tar.bz2
                                                    《実際は 1 行》
$ tar jxf postgresql-16beta2.tar.bz2
$ cd postgresql-16beta2
$ ./configure --prefix=/usr/local/pgsql/16 --enable-debug \
  --with-openssl --with-zstd --with-lz4
$ make world
$ make install-world
```

環境変数を設定するファイルを書き出して、適用します。postgres ユーザで読み書き可能な /var/lib/pgsql ディレクトリが在るものとします。

```
$ cat > ~/pg16.env <<'EOF'
VER=16
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
$ . ~/pg16.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 (16beta2)
Type "help" for help.

db1=#
```

# 4. 主な機能追加

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/16/share/doc/html/
```

また、以下 URL にて PostgreSQL 16 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。

```
https://www.postgresql.org/docs/16/
```

## 4.1. 性能向上

### 4.1.1. パラレルクエリ対応追加

PostgreSQL 16 ではパラレル処理が可能な SQL 実行プランの要素がいくつか追加されました。

#### ◆ *Hash Full Join ／ Hash Right Join への対応*

これまでパラレルのハッシュ結合（Hash Join）は、INNER JOIN と LEFT JOIN にしか対応していませんでしたが、本バージョンから RIGHT JOIN と FULL JOIN についてもサポートされました。

以下のように動作を確認しました。ある番号がある番号を指すという関係を示すテーブル用意して、パラレル問い合わせが選択されるようにパラメータを与えたうえで LEFT、RIGHT、FULL の 3 種類の方法で自己

結合して、その実行プランを調べています。

```
（参照関係を示す t_arrow テーブルに 5 万件弱のデータ投入）
db1=# CREATE TABLE t_arrow (origin int, point_to int);
db1=# INSERT INTO t_arrow SELECT g, (random() * 50000)::int
        FROM generate_series (1, 50000) g WHERE g % 19 != 0;

（パラレル実行プランを選択させる設定、プランナ統計情報を最新化）
db1=# SET debug_parallel_query TO on;          ← PostgreSQL 16 の場合
db1=# SET force_parallel_mode TO on;           ← PostgreSQL 15 の場合
db1=# SET min_parallel_table_scan_size TO '10kB';
db1=# VACUUM ANALYZE;

（PostgreSQL 15 で LEFT OUTER JOIN の問い合わせプラン出力）
db1=# EXPLAIN SELECT count(*) FROM t_arrow o
         LEFT OUTER JOIN t_arrow p ON (o.origin = p.point_to);
                                        QUERY PLAN
-------------------------------------------------------------------------------
 Finalize Aggregate  (cost=2341.28..2341.29 rows=1 width=8)
   ->  Gather  (cost=2341.06..2341.27 rows=2 width=8)
         Workers Planned: 2
         ->  Partial Aggregate  (cost=1341.06..1341.07 rows=1 width=8)
               ->  Parallel Hash Left Join
                    (cost=654.08..1291.72 rows=19737 width=0)
                     Hash Cond: (o.origin = p.point_to)
                     ->  Parallel Seq Scan on t_arrow o
                          (cost=0.00..407.37 rows=19737 width=4)
                     ->  Parallel Hash
                          (cost=407.37..407.37 rows=19737 width=4)
                           ->  Parallel Seq Scan on t_arrow p
(cost=0.00..407.37 rows=19737 width=4)
(9 rows)

（PostgreSQL 15 で RIGHT OUTER JOIN の問い合わせプラン出力）
db1=# EXPLAIN SELECT count(*) FROM t_arrow o
         RIGHT OUTER JOIN t_arrow p ON (o.origin = p.point_to);
```

```
                                        QUERY PLAN
-------------------------------------------------------------------------------
 Finalize Aggregate  (cost=2267.27..2267.28 rows=1 width=8)
   ->  Gather  (cost=2267.05..2267.26 rows=2 width=8)
         Workers Planned: 2
         ->  Partial Aggregate  (cost=1267.05..1267.06 rows=1 width=8)
               ->  Parallel Hash Left Join
                   (cost=654.08..1217.71 rows=19737 width=0)
                    Hash Cond: (p.point_to = o.origin)
                    ->  Parallel Seq Scan on t_arrow p
                        (cost=0.00..407.37 rows=19737 width=4)
                    ->  Parallel Hash
                        (cost=407.37..407.37 rows=19737 width=4)
                          ->  Parallel Seq Scan on t_arrow o
                              (cost=0.00..407.37 rows=19737 width=4)
(9 rows)
```

**（PostgreSQL 15 で FULL OUTER JOIN の問い合わせプラン出力）**

```
db1=# EXPLAIN SELECT count(*) FROM t_arrow o
        FULL OUTER JOIN t_arrow p ON (o.origin = p.point_to);
                                QUERY PLAN
-------------------------------------------------------------------------------
 Gather  (cost=3729.24..3729.35 rows=1 width=8)
   Workers Planned: 1
   Single Copy: true
   ->  Aggregate  (cost=2729.24..2729.25 rows=1 width=8)
         ->  Hash Full Join  (cost=1275.80..2610.82 rows=47369 width=0)
               Hash Cond: (p.point_to = o.origin)
               ->  Seq Scan on t_arrow p
                   (cost=0.00..683.69 rows=47369 width=4)
               ->  Hash  (cost=683.69..683.69 rows=47369 width=4)
                     ->  Seq Scan on t_arrow o
                         (cost=0.00..683.69 rows=47369 width=4)
(9 rows)
```

**（PostgreSQL 16 で LEFT OUTER JOIN の問い合わせプラン出力）**

```
db1=# EXPLAIN SELECT count(*) FROM t_arrow o
        LEFT OUTER JOIN t_arrow p ON (o.origin = p.point_to);
                                        QUERY PLAN
-------------------------------------------------------------------------------
 Finalize Aggregate  (cost=2267.27..2267.28 rows=1 width=8)
   ->  Gather  (cost=2267.05..2267.26 rows=2 width=8)
         Workers Planned: 2
         ->  Partial Aggregate  (cost=1267.05..1267.06 rows=1 width=8)
               ->  Parallel Hash Right Join
                     (cost=654.08..1217.71 rows=19737 width=0)
                     Hash Cond: (p.point_to = o.origin)
                     ->  Parallel Seq Scan on t_arrow p
                         (cost=0.00..407.37 rows=19737 width=4)
                     ->  Parallel Hash
                           (cost=407.37..407.37 rows=19737 width=4)
                           ->  Parallel Seq Scan on t_arrow o
(cost=0.00..407.37 rows=19737 width=4)
(9 rows)
```

**（PostgreSQL 16 で RIGHT OUTER JOIN の問い合わせプラン出力）**

```
db1=# EXPLAIN SELECT count(*) FROM t_arrow o
        RIGHT OUTER JOIN t_arrow p ON (o.origin = p.point_to);
                                        QUERY PLAN
-------------------------------------------------------------------------------
 Finalize Aggregate  (cost=2267.27..2267.28 rows=1 width=8)
   ->  Gather  (cost=2267.05..2267.26 rows=2 width=8)
         Workers Planned: 2
         ->  Partial Aggregate  (cost=1267.05..1267.06 rows=1 width=8)
               ->  Parallel Hash Left Join
                     (cost=654.08..1217.71 rows=19737 width=0)
                     Hash Cond: (p.point_to = o.origin)
                     ->  Parallel Seq Scan on t_arrow p
                           (cost=0.00..407.37 rows=19737 width=4)
                     ->  Parallel Hash
```

```
                        (cost=407.37..407.37 rows=19737 width=4)
                         ->  Parallel Seq Scan on t_arrow o
                              (cost=0.00..407.37 rows=19737 width=4)
(9 rows)
```

**（PostgreSQL 16 で FULL OUTER JOIN の問い合わせプラン出力）**

```
db1=# EXPLAIN SELECT count(*) FROM t_arrow o
        FULL OUTER JOIN t_arrow p ON (o.origin = p.point_to);
                                      QUERY PLAN
-------------------------------------------------------------------------------
 Finalize Aggregate  (cost=2267.27..2267.28 rows=1 width=8)
   ->  Gather  (cost=2267.05..2267.26 rows=2 width=8)
         Workers Planned: 2
         ->  Partial Aggregate  (cost=1267.05..1267.06 rows=1 width=8)
               ->  Parallel Hash Full Join
                    (cost=654.08..1217.71 rows=19737 width=0)
                     Hash Cond: (p.point_to = o.origin)
                     ->  Parallel Seq Scan on t_arrow p
                          (cost=0.00..407.37 rows=19737 width=4)
                     ->  Parallel Hash
                          (cost=407.37..407.37 rows=19737 width=4)
                           ->  Parallel Seq Scan on t_arrow o
(cost=0.00..407.37 rows=19737 width=4)
(9 rows)
```

PostgreSQL 15 では RIGHT OUTER JOIN も LEFT OUTER JOIN も実行プランとしては Parallel Hash Left Join になり、FULL OUTER JOIN は パラレルでない Hash Full Join を使ったプラン実行になりました。

一方、PostgreSQL 16 では LEFT OUTER JOIN に対して（問い合わせとは違いますが最適化された結果）Parallel Hash Right Join が選択されています。 また、FULL OUTER JOIN に対しては Parallel Hash Full Join が選択されました。

### ◆ *string_agg と array_agg のパラレル対応*

PostgreSQL 16 から集約関数 string_agg、array_agg が部分的な集約処理 Partial Aggregate に対応して、パラレル実行することができるようになりました。

以下の通りサンプルテーブルを作って、実行プランを確認しました。

```
（サンプルテーブルを作成 ／ special_flg 列は 1000 件に 1 件だけランダムな 1 文字を持つ）
db1=# CREATE TABLE t_agg_test(id int PRIMARY KEY, special_flg text);
db1=# INSERT INTO t_agg_test SELECT g, CASE
        WHEN random() < 0.001 THEN chr((random()*100)::int) ELSE null END
        FROM generate_series(1, 100000) g;

（プランナ統計情報を更新して、パラレル処理を選択するように設定を与える）
db1=# VACUUM ANALYZE t_agg_test;
db1=# SET debug_parallel_query TO on;          → PostgreSQL 16 の場合
db1=# SET force_parallel_mode TO on;           → PostgreSQL 15 の場合
db1=# SET min_parallel_table_scan_size TO '10kB';
db1=# SET parallel_setup_cost TO 0;
db1=# SET parallel_tuple_cost TO 0;

（PostgreSQL 15 で集約関数 array_agg / string_agg の実行プランを確認）
db1=# explain SELECT array_agg(special_flg), string_agg(special_flg, ',')
        FROM t_agg_test WHERE special_flg IS NOT NULL;
                              QUERY PLAN
-----------------------------------------------------------------------------
 Aggregate  (cost=859.91..859.92 rows=1 width=32)
   ->  Gather  (cost=0.00..859.67 rows=97 width=2)
         Workers Planned: 2
         ->  Parallel Seq Scan on t_agg_test
              (cost=0.00..859.67 rows=40 width=2)
               Filter: (special_flg IS NOT NULL)
(5 rows)

（PostgreSQL 16 で集約関数 array_agg / string_agg の実行プランを確認）
db1=# explain SELECT array_agg(special_flg), string_agg(special_flg, ',')
        FROM t_agg_test WHERE special_flg IS NOT NULL;
                                QUERY PLAN
-----------------------------------------------------------------------------
 Finalize Aggregate  (cost=859.80..859.81 rows=1 width=32)
   ->  Gather  (cost=859.78..859.79 rows=2 width=32)
```

```
        Workers Planned: 2
        ->  Partial Aggregate  (cost=859.78..859.79 rows=1 width=32)
              ->  Parallel Seq Scan on t_agg_test
                    (cost=0.00..859.67 rows=43 width=2)
                     Filter: (special_flg IS NOT NULL)
(6 rows)
```

どちらのバージョンでも t_agg_test テーブルのスキャンはパラレル実行になります。PostgreSQL 15 ではその後、パラレル実行されたスキャン結果を Gather で取りまとめてから、Aggregate を単一プロセスで実行しています。一方、PostgreSQL 16 では、パラレルのまま部分的な集約処理 Partial Aggregate を行って、最後にパラレル実行結果を Gather で取りまとめたあと Finalize Aggregate を行っています。テストした 2 つの集約関数について、集約処理をパラレルに実行するプランになっていることが確認できました。

### *4.1.2. SELECT DISTINCT* でインクリメンタルソート

インクリメンタルソートは PostgreSQL 13 から登場した機能です。インデックスにより既にソート済みであることを活かして、残り部分だけをソートする実行プランが使用できます。これまで SELECT DISTINCT を実行するためのソートについては、インクリメンタルソートが使えませんでしたが、PostgreSQL 16 で対応しました。

以下のように PostgerSQL 15 と 16 の動作を比較して確認しました。

```
（ソートするテーブルとデータを作成 ／ num1、num2 列にはインデックスを作成）
db1=# CREATE TABLE t_sort (num1 int, num2 int, num3 int, v text);
db1=# CREATE INDEX ON t_sort (num1, num2);
db1=# INSERT INTO t_sort SELECT g / 1000, (g / 10) % 100, g % 10, 'v'
         FROM generate_series(1, 100000) g;
db1=# VACUUM ANALYZE t_sort;

db1=# SELECT * FROM t_sort LIMIT 11;
 num1 | num2 | num3 | v
------+------+------+---
    0 |    0 |    1 | v
    0 |    0 |    2 | v
    0 |    0 |    3 | v
《中略》
    0 |    0 |    9 | v
```

```
   0 |    1 |    0 | v
   0 |    1 |    1 | v
(11 rows)

（PostgreSQL 15: ORDER BY でソートを要する問い合わせ）
db1=# explain SELECT * FROM t_sort ORDER BY num1, num2, num3;
                                      QUERY PLAN
------------------------------------------------------------------------------
 Incremental Sort  (cost=0.91..7726.21 rows=100000 width=14)
   Sort Key: num1, num2, num3
   Presorted Key: num1, num2
   ->  Index Scan using t_sort_num1_num2_idx on t_sort
       (cost=0.29..3221.04 rows=100000 width=14)
(4 rows)

（PostgreSQL 15: SELECT DISTINCT でソートを要する問い合わせ）
db1=# explain SELECT DISTINCT ON (num1, num2, num3) * FROM t_sort;
                             QUERY PLAN
--------------------------------------------------------------------------------
 Unique  (cost=9845.82..10845.82 rows=10000 width=14)
   ->  Sort  (cost=9845.82..10095.82 rows=100000 width=14)
         Sort Key: num1, num2, num3
         ->  Seq Scan on t_sort  (cost=0.00..1541.00 rows=100000 width=14)
(4 rows)
```

PostgreSQL 15 では ORDER BY 句によってソートが必要となる問い合わせの実行プランは Incremental Sort を使うものになりましたが、DISTINCT 句によってソートを要する問い合わせの実行プランでは、そうなりませんでした。

以下は同様の手順を PostgreSQL 16 で実行した結果です。最後の SELECT DISTINCT のテスト結果だけ記載します。

```
（PostgreSQL 16: SELECT DISTINCT でソートを要する問い合わせ）
db1=# explain SELECT DISTINCT ON (num1, num2, num3) * FROM t_sort;
                                      QUERY PLAN
------------------------------------------------------------------------------
```

```
 Unique  (cost=0.78..7082.01 rows=10000 width=14)
   ->  Incremental Sort  (cost=0.78..6332.01 rows=100000 width=14)
         Sort Key: num1, num2, num3
         Presorted Key: num1, num2
         ->  Index Scan using t_sort_num1_num2_idx on t_sort
             (cost=0.29..3221.04 rows=100000 width=14)
(5 rows)
```

PostgreSQL 16 では、SELECT DISTINCT の問い合わせでもソート処理に Incremental Sort を使った実行プランが選ばれることが確認できました。

### *4.1.3.* ウィンドウ関数の最適化

本バージョンでウィンドウ関数を使った問い合わせに対するプランナ最適化が 2 つ加わりました。

サンプルデータとして試験毎の生徒の成績のテーブルを作成して、動作の違いを確認しました。以下に手順を示します。

```
（生徒 ID、試験 ID、点数のテーブルを作成）
db1=# CREATE TABLE t_score (sid int, examid int, score int);

（100 人 × 3 試験分の乱数データ投入／値を揃えるため乱数シードも指定）
db1=# SELECT setseed (1.0);
db1=# INSERT INTO t_score
        SELECT g, 101, ( (random() + random() + random() ) * 10 + 60)::int
        FROM generate_series(1, 100) g;
db1=# INSERT INTO t_score
        SELECT g, 102, ( (random() + random() + random() ) * 10 + 60)::int
        FROM generate_series(1, 100) g;
db1=# INSERT INTO t_score
        SELECT g, 201, ( (random() + random() + random() ) * 10 + 60)::int
        FROM generate_series(1, 100) g;
```

**◆ *RANGE* モードを内部的に *ROWS* モードに自動変換**

PostgreSQL16 では RANGE モードを使ったウィンドウ関数問い合わせについて、ROWS モードに置換できるものは内部で自動的に ROWS モードとして処理できるようになりました。

例として、作成したデータを生徒（sid）ごとに各試験の点数と 3 種類の順位を出力する問い合わせを使います。生徒と試験ごとに rn（高得点順に並べたときの行番号）、rnk（順位／同点があった次順位は欠番）、drnk（順位／同点の次順位も欠番にしない）を算出します。これらには OVER 句のフレーム化オプションにそれぞれ異なる指定を与えています。指定を省略している row_number() は RANGE UNBOUNDED PRECEDING AND CURRENT ROW として扱われます。これらはどの指定でも問い合わせ結果は結局一緒になります。

**（生徒（sid）ごとに各試験の点数と 3 種類の順位を出力する）**

```
db1=# SELECT sid, examid, score,
        row_number() OVER (PARTITION BY examid ORDER BY score DESC) rn,
        rank() OVER (PARTITION BY examid ORDER BY score DESC
           ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) rnk,
        dense_rank() OVER (PARTITION BY examid ORDER BY score DESC
           RANGE BETWEEN CURRENT ROW AND CURRENT ROW) drnk
        FROM t_score ORDER BY sid, examid;
 sid | examid | score | rn  | rnk | drnk
-----+--------+-------+-----+-----+------
   1 |    101 |    75 |  54 |  50 |   11
   1 |    102 |    73 |  70 |  65 |   14
   1 |    201 |    79 |  17 |  17 |    8
   2 |    101 |    75 |  52 |  50 |   11
   2 |    102 |    77 |  43 |  39 |   10
   2 |    201 |    86 |   2 |   1 |    1
《以下略》
```

この問い合わせは、フレーム化オプション指定がそれぞれ異なっているため、PostgreSQL15 で実行すると以下のように WindowAgg プランノードが 3 つ連なった実行プランになりました。

**（PostgreSQL 15.x での実行プラン）**

```
db1=# explain (ANALYZE, COSTS OFF) SELECT sid, examid, score,
        row_number() OVER (PARTITION BY examid ORDER BY score DESC) rn,
        rank() OVER (PARTITION BY examid ORDER BY score DESC
           ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) rnk,
        dense_rank() OVER (PARTITION BY examid ORDER BY score DESC
           RANGE BETWEEN CURRENT ROW AND CURRENT ROW) drnk
        FROM t_score ORDER BY sid, examid;
```

```
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Sort (actual time=1.214..1.232 rows=300 loops=1)
   Sort Key: sid, examid
   Sort Method: quicksort  Memory: 48kB
   ->  WindowAgg (actual time=0.238..1.027 rows=300 loops=1)
         ->  WindowAgg (actual time=0.227..0.643 rows=300 loops=1)
               ->  WindowAgg (actual time=0.224..0.426 rows=300 loops=1)
                     ->  Sort (actual time=0.215..0.239 rows=300 loops=1)
                           Sort Key: examid, score DESC
                           Sort Method: quicksort  Memory: 41kB
                           ->  Seq Scan on t_score
                                 (actual time=0.008..0.107 rows=300 loops=1)
 Planning Time: 0.281 ms
 Execution Time: 1.500 ms
(12 rows)
```

これが PostgreSQL16 で実行すると以下のように一つの WindowAgg プランノードにまとめられます。実行時間も短くなっています。これは問い合わせ上は RANGE モードとなっている row_number() と dense_rank() が ROWS モードであっても結果がかわらないため、内部的に ROWS モードの同一のフレーム化オプションに置き換えているためです。

**（PostgreSQL 16.x での実行プラン）**

```
db1=# explain (ANALYZE, COSTS OFF) SELECT sid, examid, score,
        row_number() OVER (PARTITION BY examid ORDER BY score DESC) rn,
        rank() OVER (PARTITION BY examid ORDER BY score DESC
           ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) rnk,
        dense_rank() OVER (PARTITION BY examid ORDER BY score DESC
           RANGE BETWEEN CURRENT ROW AND CURRENT ROW) drnk
        FROM t_score ORDER BY sid, examid;
                                QUERY PLAN
-------------------------------------------------------------------------------
 Sort (actual time=0.588..0.607 rows=300 loops=1)
   Sort Key: sid, examid
   Sort Method: quicksort  Memory: 43kB
```

```
  ->  WindowAgg (actual time=0.179..0.483 rows=300 loops=1)
       ->  Sort (actual time=0.164..0.182 rows=300 loops=1)
            Sort Key: examid, score DESC
            Sort Method: quicksort  Memory: 36kB
            ->  Seq Scan on t_score
                  (actual time=0.021..0.066 rows=300 loops=1)
 Planning Time: 0.255 ms
 Execution Time: 0.704 ms
(10 rows)
```

現在この最適化の対象となっている関数は row_number()、 rank()、 dense_rank()、 percent_rank()、 cume_dist()、 ntile() です。

### ◆ *単調増加ウィンドウ関数最適化の追加*

PostgreSQL16 から、単調増加のウィンドウ関数 ntile()、cume_dist()、 percent_rank() に対して、条件に応じて処理を打ち切るプラン最適化が適用されるようになりました。

サンプルとして、以下のように試験種別を問わず、点数（score）が上位３%に含まれる試験結果を取り出す問い合わせを実行します。

**（score 上位 100 分の 3 の結果を取り出し）**

```
db1=# SELECT * FROM
      (SELECT *, ntile(100) OVER (ORDER BY score DESC) nt FROM t_score) s1
       WHERE nt <= 3;
 sid | examid | score | nt
-----+--------+-------+----
  96 |    102 |    87 |  1
  19 |    102 |    87 |  1
  32 |    201 |    86 |  1
  53 |    201 |    86 |  2
   2 |    201 |    86 |  2
  50 |    101 |    85 |  2
 100 |    102 |    85 |  3
  14 |    101 |    85 |  3
  62 |    201 |    85 |  3
(9 rows)
```

この問い合わせの実行プランを確認すると以下のようになります。

**（PostgreSQL 15.x での実行プラン）**

```
db1=# explain (ANALYZE, COSTS OFF) SELECT * FROM
       (SELECT *, ntile(100) OVER (ORDER BY score DESC) nt FROM t_score) s1
       WHERE nt <= 3;
                                QUERY PLAN
-------------------------------------------------------------------------
 Subquery Scan on s1 (actual time=0.171..0.295 rows=9 loops=1)
   Filter: (s1.nt <= 3)
   Rows Removed by Filter: 291
   ->  WindowAgg (actual time=0.170..0.278 rows=300 loops=1)
         ->  Sort (actual time=0.089..0.106 rows=300 loops=1)
               Sort Key: t_score.score DESC
               Sort Method: quicksort  Memory: 41kB
               ->  Seq Scan on t_score
                     (actual time=0.011..0.041 rows=300 loops=1)
 Planning Time: 0.087 ms
 Execution Time: 0.329 ms
(10 rows)
```

**（PostgreSQL 16.x での実行プラン）**

```
db1=# explain (ANALYZE, COSTS OFF) SELECT * FROM
       (SELECT *, ntile(100) OVER (ORDER BY score DESC) nt FROM t_score) s1
       WHERE nt <= 3;
                                QUERY PLAN
-------------------------------------------------------------------------
 WindowAgg (actual time=0.162..0.165 rows=9 loops=1)
   Run Condition: (ntile(100) OVER (?) <= 3)
   ->  Sort (actual time=0.116..0.132 rows=300 loops=1)
         Sort Key: t_score.score DESC
         Sort Method: quicksort  Memory: 36kB
         ->  Seq Scan on t_score (actual time=0.025..0.063 rows=300 loops=1)
 Planning Time: 0.105 ms
 Execution Time: 0.198 ms
```

```
(8 rows)
```

PostgreSQL 15.x では全件データに WindowAgg 行った結果に対して「nt <= 3」の条件に合う行を取り出しているのに対して、PostgreSQL 16.x では Run Condition を使って条件に合う行のみを処理していることが分かります。単調増加であるため、ntile(100)の結果が一度 3 よりも大きくなったなら、以降を見る必要はありませんので、無駄のない処理ができています。所要時間も短くなっています。

PostgreSQL15.x でも row_number() 、rank()、count() 関数については、この最適化に対応していました。

### *4.1.4.* 同時 *COPY* 性能向上

PostgreSQL の標準のヒープ格納方式では、テーブルやインデックスはページ（デフォルト 8KB）単位で領域が確保されます。行データが追加されると必要に応じてページ追加（リレーション拡張と呼ばれます）が行われます。PostgreSQL 16 では行データ追加にともなうページ追加について改善されました。一度に多くのデータを投入する場合と、同時ページ追加が行われようとしている場合に、複数ページをまとめて追加するようになりました。以前から複数ページをまとめて追加するコードはありましたが活用される場面が限られていました。

本改善により、同一テーブルへの並行した大規模データ投入の性能改善が期待できます。以下のように性能検証を行いました。

**（WAL 出力時間を分離するため UNLOGGED でテーブル作成）**

```
db1=# CREATE UNLOGGED TABLE t_copy1 (n int, c1 text, c2 text, c3 text);
```

**（データファイルと COPY 文の SQL ファイルを作成、サイズは 1MB 弱）**

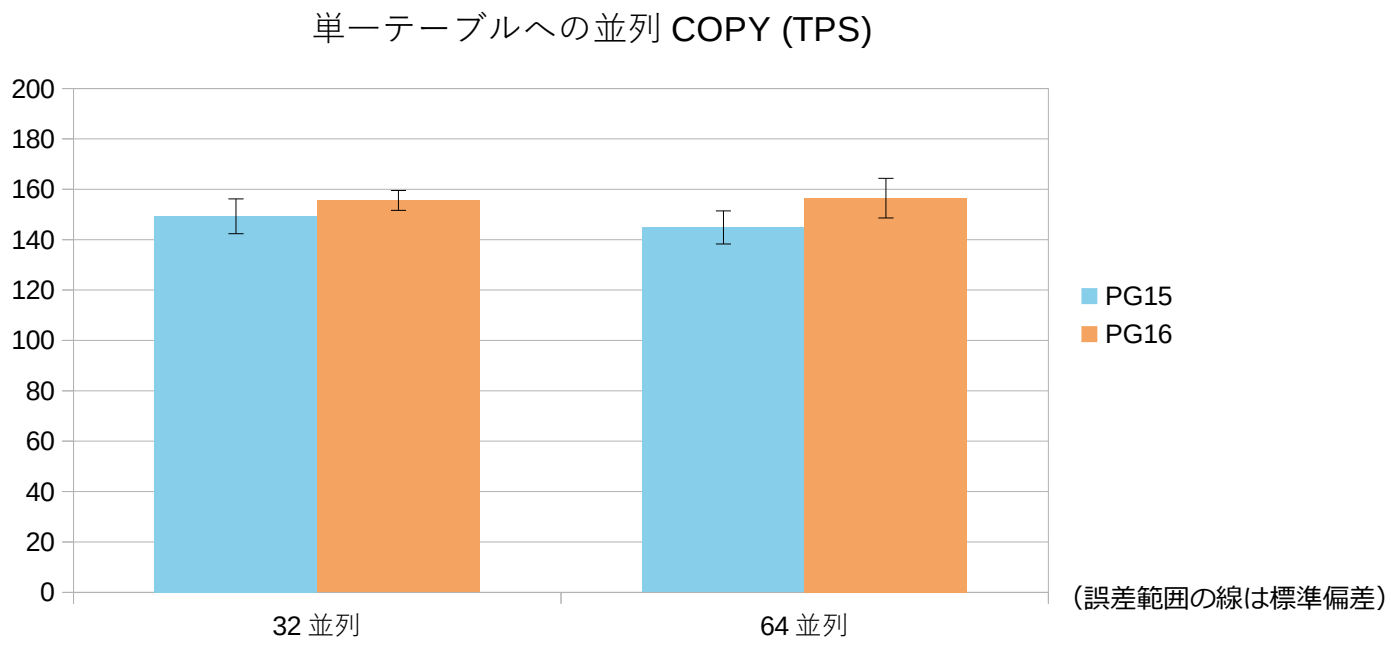
```
db1=# INSERT INTO t_copy1 SELECT g, md5(g::text), md5(g::text), md5(g::text)
        FROM generate_series(1, 10000) g;
db1=# COPY t_copy1 TO '/tmp/t_copy1.dat';
db1=# \q

$ ls -lh /tmp/t_copy1.dat
-rw-r--r-- 1 postgres postgres 1015K  6月 14 12:05 /tmp/t_copy1.dat
$ cat > copy.sql <<EOS
COPY t_copy1 FROM '/tmp/t_copy1.dat';
EOS
```

**（pgbench を通して COPY を並列実行 - PostgreSQL 15 と 16 で繰り返し実行）**

```
$ psql db1 -c "TRUNCATE t_copy1"
$ pgbench -n -f copy.sql -c 32 -t 64 -j 32 db1
pgbench (16beta2)
transaction type: copy.sql
scaling factor: 1
query mode: simple
number of clients: 32
number of threads: 32
maximum number of tries: 1
number of transactions per client: 64
《中略》
initial connection time = 63.800 ms
tps = 150.224097 (without initial connection time)
```

32 並列で各セッションが 64 回 COPY を実行する場合と、64 並列で各セッションが 32 回（-t 32）COPY を実行する場合を計測しました。どちらも合計で 2,623MB のデータをテーブルに読み込みます。WAL 出力性能や WAL 出力と連動したチェックポイントの影響を除外するため UNLOGGED テーブルを使用しました。本試験では shared_buffers = 1GB としました。

以下のグラフに結果を示します。

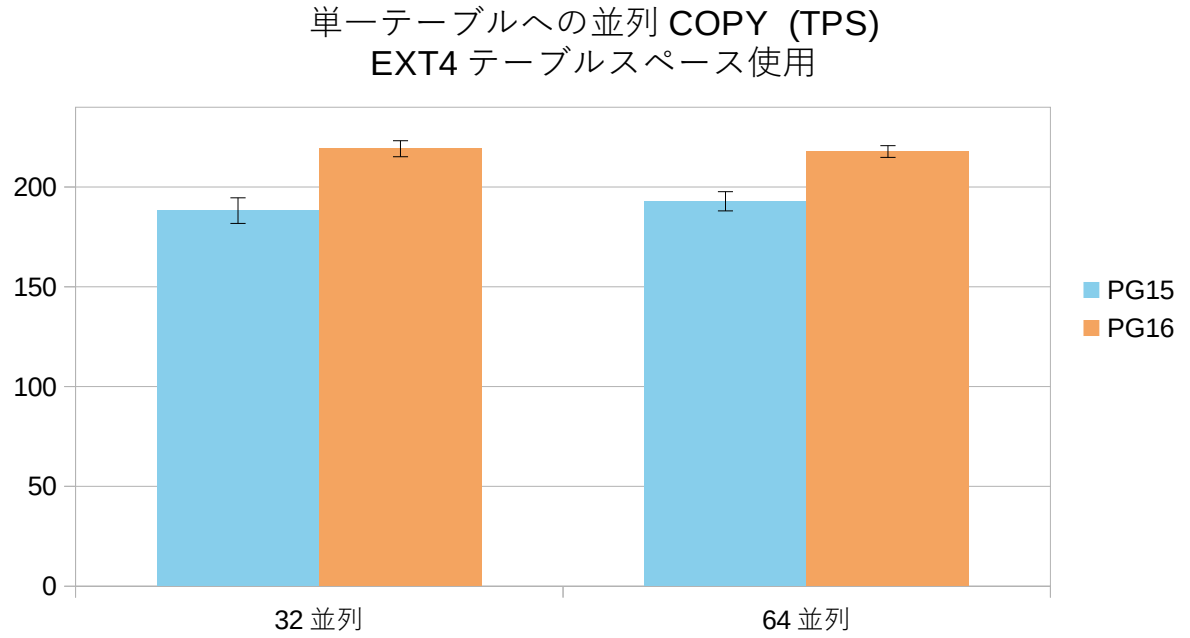


PostgreSQL 16 の方が 5～10%程度高速になりましたが、バラつきも大きく、はっきりした結果とは言えません。本例以外で、並列度や COPY 文 1 回で投入するデータ量、その他の条件によっては、ほとんど差がでなかったり、PostgreSQL15 が速くなることもありました。

### ◆ *ファイルシステム／システムコール特性の影響*

本検証で PostgreSQL 16 の性能向上が明確にはあらわれなかったのは、PostgreSQL 16beta2 の COPY に別の性能劣化する変更が含まれていたことと、ファイルシステム特性に原因がありました。PostgreSQL 16 では複数ページを追加するときに、追加するページ数に応じて使用するシステムコールを変えるように実装されているのですが、本試験で使用した環境の XFS ファイルシステムでは異なるシステムコールを混在させて繰り返し使うと性能劣化が生じるようです。そのため、システムコールを切り替える境界を前後する数のページ追加が発生するテスト条件だと結果が悪くなりました。本原因分析につきましては、PostgreSQL 開発者でコミッターの澤田雅彦氏から情報提供いただきました。リリースまでに COPY について性能改善のさらなる修正が適用される見込みです。

以下は EXT4 ファイルシステムにテーブルと COPY 元ファイルを配置して beta2 で試験した場合の結果です。こちらの方が PostgreSQL 16 の性能向上がはっきりとあらわれています。差異は +13% と+16%で、バラつきも小さいです。




（誤差範囲の線は標準偏差）

### 4.1.5. SIMD CPU アクセラレータ対応

PostgreSQL16 から x86_64 アーキテクチャと ARM アーキテクチャ CPU に対する SIMD（Single Instruction Multiple Data）のサポートが追加されました。

x86_64 アーキテクチャに対しては現在使われているほとんどの CPU でサポートしている SSE2（Streaming SIMD Extensions 2）命令に対応しています。一連のデータを一度に処理する場合、あるいはこれまで順に処理していたものを一度に処理するように変えることで、性能向上が期待できます。

SSE2 命令を使用する基盤機能を使うように変更されたのは以下の箇所です。

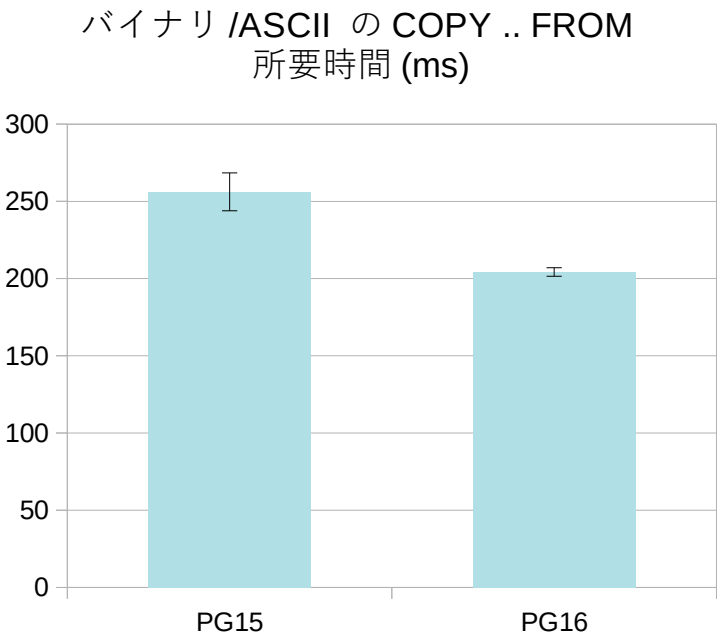
- ASCII 文字列を検査する処理
- JSON の文字列を検査する処理
- スナップショット処理やプロセス配列の処理などの内部制御処理

BINARY かつ ASCII の COPY ... FROM コマンドで ASCII 文字列検査の性能を比較してみます。ASCII 文字のみ 1600 字の列が 1 つ × 10 万行のテーブルをバイナリ形式で出力して、それを読み込む性能を比較します。1600 字なのは TOAST 格納にならない範囲の長い文字列という意味です。また、WAL 出力性能の比較にならないように UNLOGGED テーブルとしています。

```
（テスト用データを作成）
db1=# CREATE UNLOGGED TABLE t5 (t text);
CREATE TABLE
db1=# INSERT INTO t5 SELECT repeat(md5(g::text), 50)
        FROM generate_series(1, 100000) g;
INSERT 0 100000
db1=# COPY t5 TO '/tmp/t5.bin' WITH (FORMAT binary, ENCODING 'sql_ascii');
COPY 100000

（COPY 読み込み所要時間を計測 - PostgreSQL 15 と 16 で繰り返し実行）
db1=# \timing on
db1=# TRUNCATE t5;
TRUNCATE TABLE
Time: 244.458 ms
db1=# COPY t5 FROM '/tmp/t5.bin' WITH (FORMAT binary, ENCODING 'sql_ascii');
COPY 100000
Time: 244.458 ms
```

以下グラフのように COPY ... FROM 所要時間のバージョンによる違いが確認できました。

PostgreSQL 15 と比較して、PostgreSQL 16 は所要時間が 20%短縮されています。




次に JSON 読み込み性能を比較してみます。以下のように、でたらめな内容の 200 文字の JSON 文字列を jsonb 型で読み込ませる SQL を用意して、pgbench で繰り返し実行します。

**（テスト用 SQL を用意）**

```
$ cat <<EOS > q5.sql
SELECT
'"1234567890-^^asdghjkl;:zxcvbnm,./QWERTYUIOPPASDFGHJKLZXCVBNBM<;@ejeak452hg
0snkdkbnor09g309ae09jv0a9sdn0af9j0a9nglsllagsdfasdf5gaj09j0jioib908()sdlkg57
hJKeuh!#sdalllSgf___igjoiwdyaohfd-=eijgu[][]sxxQQ"'::jsonb;
EOS
```

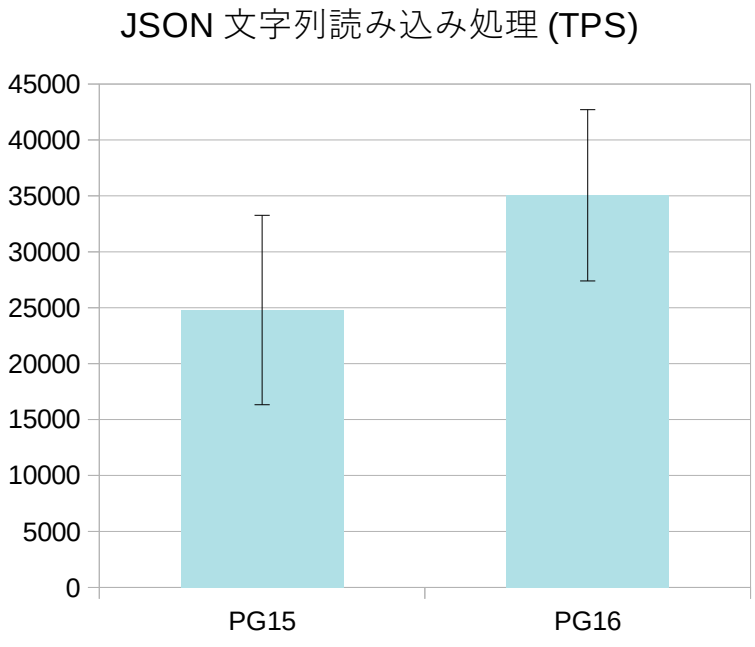
**（pgbench で繰り返し実行して性能計測をする - PostgreSQL 15 と 16 に繰り返し実行）**

```
$ pgbench -f q5.sql -c 1 -t 100000 -n -p 5432 db1
pgbench (16beta2)
transaction type: q5.sql
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
```

（誤差範囲の線は標準偏差）

```
maximum number of tries: 1
number of transactions per client: 100000
number of transactions actually processed: 100000/100000
number of failed transactions: 0 (0.000%)
latency average = 0.051 ms
initial connection time = 2.392 ms
tps = 19662.609287 (without initial connection time)
```

以下グラフのようにバージョン性能差が確認できました。
性能にバラつきはあるものの、PostgreSQL15 と比べて PostgreSQL16 が 40%高速化しています。




（誤差範囲の線は標準偏差）

## 4.2. SQL 機能

### 4.2.1. SQL/JSON 対応

本バージョンでは、SQL/JSON 標準に基づく JSON データを生成するコンストラクタ関数といくつかの JSON データに対する述語が追加されました。

追加されたコンストラクタ関数を以下表に示します。戻り値型は RETURNING 句で json、jsonb、bytea、お

よび各種テキストデータ型を指定できます。また、 WITH UNIQUE KEYS で同じキーを複数持つことを禁止できます。

| 関数名 | 説明 |
|---|---|
| JSON_OBJECT | テキスト配列から JSON オブジェクトを構築します。 |
| JSON_OBJECTAGG | 提供されたデータを JSON オブジェクトに集約します。 |
| JSON_ARRAY | 提供された SQL または JSON データから JSON 配列を構築します。 |
| JSON_ARRAYAGG | 提供された SQL または JSON データを JSON 配列に集約します。 |

以下にこれら関数の使用例を示します。

```
（JSON_OBJECT でスカラ値からオブジェクトを構成）
db1=# SELECT JSON_OBJECT('num' VALUE 123, 'dt' : CURRENT_DATE
        RETURNING jsonb);
          json_object
---------------------------------
 {"dt": "2023-05-31", "num": 123}

（同じ値を bytea で出力）
db1=# SELECT JSON_OBJECT('num' VALUE 123, 'dt' : CURRENT_DATE
        RETURNING bytea);
                          json_object
------------------------------------------------------------------------
 \x7b226e756d22203a203132332c2022647422203a2022323032332d30352d3331227d

（WITH UNIQUE KEYS を指定すると同じキーを複数持つ JSON は生成できない）
db1=# SELECT JSON_OBJECT('id' VALUE 1, 'id' VALUE 2);
    json_object
----------------------
 {"id" : 1, "id" : 2}

db1=# SELECT JSON_OBJECT('id' VALUE 1, 'id' VALUE 2 WITH UNIQUE KEYS);
ERROR:  duplicate JSON key "id"
```

```
（JSON_OBJECTAGG は複数行データから 1 つのオブジェクトを構成）
db1=# SELECT g AS n, g^3 AS cube FROM generate_series(1, 3) g;
 n | cube
---+------
 1 |    1
 2 |    8
 3 |   27

db1=# SELECT JSON_OBJECTAGG(n:cube RETURNING json) FROM (
        SELECT g AS n, g^3 AS cube FROM generate_series(1, 3) g) v;
        json_objectagg
---------------------------------
 { "1" : 1, "2" : 8, "3" : 27 }

（JSON_ARRAY はスカラ値や json/jsonb データの並びから JSON 配列を構成）
db1=# SELECT JSON_ARRAY(1, true, JSON '{"a":null}');
     json_array
-----------------------
 [1, true, {"a":null}]

（JSON_ARRAYAGG は複数行から JSON 配列を構成、並び順の指定も可能）
db1=# SELECT JSON_ARRAYAGG(v ORDER BY v DESC) FROM generate_series(1, 3) v;
 json_arrayagg
---------------
 [3, 2, 1]
```

追加された述語構文を以下に示します。

| 構文 | 説明 |
|---|---|
| IS JSON [VALUE] | 文字列が JSON として解釈できるなら真を返す。 |
| IS JSON ARRAY | 文字列が JSON 配列として解釈できるなら真を返す。 |
| IS JSON OBJECT | 文字列が JSON オブジェクト（いくつかのキーと値）として解釈できるなら真を返す。 |
| IS JSON SCALAR | 文字列が JSON のスカラ値として解釈できるなら真を返す。 |

以下に実行例を示します。

```
db1=# SELECT x, x IS JSON VALUE "JSON VALUE", x IS JSON ARRAY "JSON ARRAY",
             x IS JSON OBJECT "JSON OBJECT", x IS JSON SCALAR "JSON SCALAR"
      FROM (VALUES ('1'),('"A"'),('A'),(''),('["A","B"]'),('{"a":1,"b":2}'))
      x(x);
       x        | JSON VALUE | JSON ARRAY | JSON OBJECT | JSON SCALAR
----------------+------------+------------+-------------+-------------
 1              | t          | f          | f           | t
 "A"            | t          | f          | f           | t
 A              | f          | f          | f           | f
                | f          | f          | f           | f
 ["A","B"]      | t          | t          | f           | f
 {"a":1,"b":2}  | t          | f          | t           | f
```

ダブルクォートで囲われていない文字列の「A」や長さゼロの文字列は、JSON VALUE ではないという結果になっています。これらの述語は文字列データ型か JSON または JSONB データ型のみ受け付けます。

### 4.2.2. ANY_VALUE 集約関数

新たな集約関数 any_value が追加されました。any_value は引数で指定した列について集約対象の行のどれかの値を返します。以下のような用途に使います。

**（システムテーブル pg_class、pg_namespace に対する問い合わせ）**

```
db1=# SELECT c.relnamespace AS scm_oid, any_value(n.nspname) AS scm_name,
             count(1) FROM pg_class c
        JOIN pg_namespace n ON c.relnamespace = n.oid
        WHERE c.relkind = 'r' GROUP BY c.relnamespace;
 scm_oid |       scm_name       | count
---------+----------------------+-------
    2200 | public               |    24
   14130 | information_schema   |     4
   16791 | appscm1              |     3
      11 | pg_catalog           |    64
(4 rows)
```

これは現在データベース中にあるスキーマの OID とスキーマ名、そこに属するテーブルの数を出力する問い合わせです。テーブル等を管理するシステムテーブル pg_class と スキーマを管理するシステムテーブル pg_namespace を結合して、 スキーマの OID（c.relnamespace ）でグルーピングしています。

このとき、スキーマの OID に対してスキーマ名は関数従属的に決まるため、any_value(nspname) としてグループ内の任意の値を出力させれば、目的の出力が得られます。SELECT リストに単に nspname と書くとエラーになります。

なお、以前の PostgreSQL バージョンでも、テーブル定義から関数従属性が明らかである場合には 、any_value 関数を使わなくとも、集約の問い合わせの SELECT リストにグループ化していない列を指定することが可能でした。即ち以下のように実行することができます。

```
db1=# SELECT n.oid AS scm_oid, n.nspname AS scm_name, count(1)
        FROM pg_class c JOIN pg_namespace n ON c.relnamespace = n.oid
        WHERE relkind = 'r' GROUP BY n.oid;
 scm_oid |       scm_name       | count
---------+----------------------+-------
    2200 | public               |    24
   14130 | information_schema   |     4
   16791 | appscm1              |     3
      11 | pg_catalog           |    64
(4 rows)
```

問い合わせの意味は同じですが、グループ化に pg_namespace の主キーである oid 列を指定しているので、pg_namespace の主キー以外の列である nspname が関数従属であることは自明であると判断されました。

## 4.2.3. 数値リテラル表現

PostgreSQL 16 では数値のリテラル表現について 2 つ拡張が加わりました。

### ◆ *10 進数以外の整数リテラル*

整数リテラルとして、16 進数、8 進数、2 進数を書くことが可能になりました。

以下のように、0（ゼロ）の後に x、o、b を続けて、その後に 16 進数、8 進数、2 進数の値を記述します。マイナスは手前に付けます。

```
db1=# SELECT 0x42F, 0o273, -0b100101;
 ?column? | ?column? | ?column?
----------+----------+----------
     1071 |      187 |      -37
(1 row)
```

◆ ***桁区切り表記***

数値リテラルに_（アンダースコア）で区切りを書くことが可能になりました。

値の解釈としては単に無視されます。3 桁区切りで記述するつもりが誤って 4 桁区切りとしてしまってもエラーにはなりません。逆に言えば、意図して 4 桁区切りにする使い方もできるということです。

```
db1=# SELECT 1_000_000_000.000;
    ?column?
----------------
 1000000000.000
(1 row)

（3 桁区切りでは無い箇所に「_」を書いても、同様に無視される）
db1=# SELECT 1_0000;
 ?column?
----------
    10000
(1 row)
```

## *4.2.4. ICU* 照合順序の拡張

PostgreSQL 16 は ICU ライブラリによる照合順序を有効にするビルドが、ビルド時のデフォルトになりました。これまで ICU を使うには configure で --with-icu を指定する必要がありましたが、これからは除外するときに --without-icu の指定が必要になります。また、ICU を使った照合順序について、いくつかの機能拡張が適用されています。

◆ ***新たな定義済み照合順序***

PostgreSQL 15 までは ICU が有効なビルドであっても、ICU を使った照合順序（collation）は必要に応じて CREATE COLLATION コマンドで作成しなければならず、libc による照合順序のみが最初から定義済みでした。PostgreSQL 16 では、OS で定義されているロケールに対応した ICU による照合順序もあらかじめ作成

されています。

```
（PostgreSQL 16 で実行）
db1=# SELECT collname, collprovider FROM pg_collation;
       collname        | collprovider
-----------------------+--------------
 default               | d
 C                     | c
 POSIX                 | c
《中略》
 zh-Hant-TW-x-icu      | i
 zu-x-icu              | i
 zu-ZA-x-icu           | i
(1732 rows)
```

新たなデータベースに対して上記のように実行すると、collprovider が i （ICU という意味）の照合順序が多数あることが確認できます。PostgreSQL 15.x だと ICU の照合順序は含まれず、件数も 1000 件ほどです。

また、PostgreSQL 16 では ICU むけに以下表に示す 2 つの特別な照合順序も用意されています。

| 照合順序 | 説明 |
|---|---|
| ucs_basic | ユニコードのコードポイント順にソートする。libc の C ロケール指定をしたときと同じ振る舞いになる。UTF8 文字エンコーディング専用。 |
| unicode | 言語地域が未定義のときの妥当そうな順序でソートする。<br>ICU の「und-x-icu」ロケール指定が使われる。UTF8 以外の主要ないくつかの文字エンコーディング（latin1、EUC_JP など）でも指定可能。 |

以下に簡単な使用例を示します。

```
（C ロケール、UTF8 エンコーディングのデータベースで適当な文字列データのテーブルを作成）
db1=# CREATE TABLE t_coll (id int, v text);
db1=# INSERT INTO t_coll VALUES
        (1, 'AAA'), (2, 'bbb'), (3, 'CCC'),
        (101, 'かき'), (102, 'クケ'), (103, 'ｺ'),
        (104, 'ｻ'), (105, 'じず'), (106, 'せそ');

（ユニコードのコード順でソート）
db1=# SELECT * FROM t_coll ORDER BY v COLLATE ucs_basic;
```

```
 id  |  v
-----+------
   1 | AAA
   3 | CCC
   2 | bbb
 101 | かき
 105 | じず
 106 | せそ
 102 | クケ
 103 | ｺ
 104 | ｻ
(9 rows)

（照合順序に unicode を指定して、妥当そうな順でソート）
db1=# SELECT * FROM t_coll ORDER BY v COLLATE unicode;
 id  |  v
-----+------
   1 | AAA
   2 | bbb
   3 | CCC
 101 | かき
 102 | クケ
 103 | ｺ
 104 | ｻ
 105 | じず
 106 | せそ
(9 rows)
```

### ◆ *ICU のルール機能をサポート*

ICU ライブラリにはルールを付加して照合順序をカスタマイズする機能があります。PostgreSQL 16 から、CREATE COLLATION で照合順序を作成するときに ICU のルールを指定できるようになりました。以下に例を示します。

```
（トランプの数をあらわすデータを作成）
db1=# CREATE TABLE t_card (v text);
```

```
db1=# INSERT INTO t_card VALUES ('A'), ('J'), ('Q'), ('K'),
      ('2'), ('3'), ('4'), ('5'), ('6'), ('7'), ('8'), ('9'), ('10');

（ルール付きの照合順序を作成、locale の und は言語地域未定義を意味する）
db1=# CREATE COLLATION col_card_rule (provider = icu, locale = 'und',
       rules = '& 1 < A  & 9 < 10  & 10 < J  & J < Q  & Q < K');
```

トランプの A → 1 → 2 → ... → 8 → 9 → 10 → J → Q → K というソート順序になるようにルールを作成しています。& から次の & の手前までが一つのルールで、「& 1 < A」は「A のソート位置を 1 の次にする」という意味です。「& 9 < 10」は「10」という文字列をひと塊として「9」の次であるものと指定しています。

以下のように作成した照合順序を使用します。

```
（トランプの数のための照合順序でソート）
db1=# SELECT * FROM t_card ORDER BY v COLLATE col_card_rule;
 v
----
 A
 2
 3
 4
 5
 6
 7
 8
 9
 10
 J
 Q
 K
(13 rows)

（文字列比較であっても 9 より 10 が大きくなる）
db1=# SELECT '9'::text < '10'::text COLLATE col_card_rule;
 ?column?
----------
 t
```

```
(1 row)
```

異なる文字を同じ文字と見做すという動作もルールで指定できます。この場合、CREATE COLLATION に「deterministic = false」も指定します。

```
（「高」と「髙（はしご高）」を同じと見做すルールを使用する）
db1=# CREATE COLLATION col_my_ident_rule (provider = icu, locale = 'und',
        deterministic = false, rules = '& 高 = 髙');

db1=# SELECT '高' = '髙' COLLATE col_my_ident_rule;
 ?column?
----------
 t
(1 row)
```

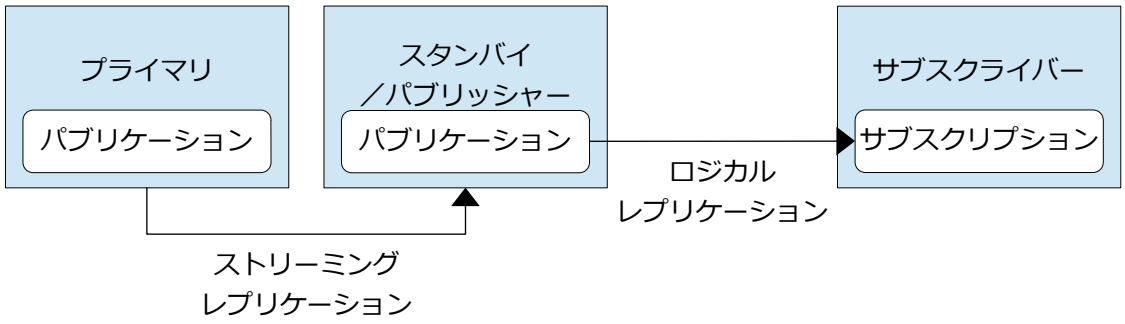
rule に指定できる構文の詳細は、ICU ライブラリのドキュメント[1]に記載されています。

## 4.3. ロジカルレプリケーション

本バージョンではロジカルレプリケーションに様々な機能追加が行われました。それらのうち主要なものについて機能を確認しました。

### 4.3.1. スタンバイからのパブリケーション

本バージョンでは、ストリーミングレプリケーションのスタンバイをパブリケーションとしてロジカルレプリケーションを構築できるようになりました。以下に模式図を示します。




1 https://unicode-org.github.io/icu/userguide/collation/customization/#collation-rule

スタンバイをパブリケーションとする場合、スタンバイは参照しか受け付けず、直接パブリケーションを作成できないため、プライマリでパブリケーションを作成し、ストリーミングレプリケーションでスタンバイに複製する必要があります。以下に構築例を示します。

**（プライマリでロジカルレプリケーションの設定）**

```
$ vi $PGDATA/postgresql.conf
   wal_level = logical
$ pg_ctl restart
```

**（パブリケーション用 DB を作成し、テーブルを作成し、データを登録し、パブリケーションを作成）**

```
$ createdb -p 5432 pubdb
$ psql -p 5432 pubdb
pubdb=# CREATE TABLE t1 (id int PRIMARY KEY, val text);
pubdb=# INSERT INTO t1 VALUES (1, 'foo');
pubdb=# CREATE PUBLICATION pub FOR ALL TABLES;
pubdb=# \q
```

**（スタンバイを構築し、5433 ポートで起動）**

```
$ pg_basebackup -p 5432 -D ${PGDATA}_rep -R -c fast
$ pg_ctl start -D ${PGDATA}_rep -o '-c port=5433'
```

**（サブスクライバを構築し、5434 ポートで起動）**

```
$ initdb --no-locale --encoding=UTF8 -D ${PGDATA}_sub
$ cp $PGDATA/postgresql.conf ${PGDATA}_sub
$ pg_ctl start -D ${PGDATA}_sub -o '-c port=5434'
```

**（サブスクリプション用 DB を作成し、スキーマを作成）**

```
$ createdb -p 5434 subdb
$ pg_dump -p 5432 -s pubdb | psql -p 5434 subdb
$ psql -p 5434 subdb
subdb=# SELECT * FROM t1;
 id | val
----+-----
(0 rows)
```

```
（サブスクリプションを作成し、データが複製されることを確認）
subdb=# CREATE SUBSCRIPTION sub
          CONNECTION 'port=5433 dbname=pubdb' PUBLICATION pub;
    《ここで応答がない場合はプライマリで SELECT pg_log_standby_snapshot() を実行》
NOTICE:  created replication slot "sub" on publisher
CREATE SUBSCRIPTION
subdb=# SELECT * FROM t1;
 id | val
----+-----
  1 | foo
(1 row)
    《データが同期されるまで少し時間がかかるかもしれない》

（次試験のためにサブスクライバ、スタンバイを停止して削除、パブリケーションも削除）
$ pg_ctl stop -D ${PGDATA}_sub
$ rm -rf ${PGDATA}_sub
$ pg_ctl stop -D ${PGDATA}_rep
$ rm -rf ${PGDATA}_rep
$ psql -p 5432 -d pubdb -c "DROP PUBLICATION pub"
```

サブスクリプション作成時にパブリッシャではレプリケーションスロットが作成されますが、パブリッシャがスタンバイの場合にはスナップショットを格納した WAL レコードの適用が必要となります。このような WAL レコードは通常はチェックポイント処理やバックグラウンドライターの処理を通して出力されます。スタンバイのパブリッシャに対するサブスクリプション作成に際してこれを待ち続けなければなりません。

それを回避するため、本バージョンで pg_log_standby_snapshot()関数が追加されました。この関数をプライマリで実行すると、すぐにスナップショットを取得して WAL 書き込みが行われ、サブスクライバにおける CREATE SUBSCRIPTON を完了することができます。

## 4.3.2. トランザクションの並列適用

本バージョンでは、ロジカルレプリケーションでデータをストリーミング送信したときにサブスクライバに並列に適用できるようになりました。これはサブスクリプションの streaming オプションで指定します。これまではトランザクション完了後に送信する off 、完了を待たずに順次送信する on のみでしたが、完了をまたずに送信したうえで並列に適用も行う parallel が追加されました。デフォルトは off です。トランザクショ

ンが大きい場合にデータ同期の遅延軽減に役立ちます。並列数は本バージョンで追加されたパラメータ max_parallel_apply_workers_per_subscription で指定します。デフォルトは 2 です。

以下のように性能検証を行いました。

**（サブスクライバを構築し、5434 ポートで起動）**

```
$ initdb --no-locale --encoding=UTF8 -D ${PGDATA}_sub
$ cp $PGDATA/postgresql.conf ${PGDATA}_sub
$ pg_ctl start -D ${PGDATA}_sub -o '-c port=5434'
```

**（サブスクリプション用 DB を作成し、スキーマを作成）**

```
$ createdb -p 5434 subdb
$ pg_dump -p 5432 -s pubdb | psql -p 5434 subdb
```

**（サブスクライバでデータ同期を 10 万件ごとにログに記録するトリガを定義）**

```
$ psql -p 5434 subdb
subdb=# CREATE FUNCTION log_rows() RETURNS trigger AS $$
          BEGIN
            IF new.id % 100000 = 0 THEN
              RAISE WARNING '% rows processed', new.id;
            END IF;
            RETURN new;
          END;
        $$ LANGUAGE plpgsql;
subdb=# CREATE TRIGGER t1_log_trg AFTER INSERT ON t1
          FOR EACH ROW EXECUTE FUNCTION log_rows();
subdb=# ALTER TABLE t1 ENABLE REPLICA TRIGGER t1_log_trg;
subdb=# \q
```

**（パブリケーション・サブスクリプションを作成）**

```
$ psql -p 5432 -d pubdb -c "CREATE PUBLICATION pub FOR ALL TABLES"
$ psql -p 5434 -d subdb -c "CREATE SUBSCRIPTION sub CONNECTION 'port=5432
dbname=pubdb' PUBLICATION pub;"
```

**《実際は 1 行》**

**（パブリッシャでデータを 50 万件登録）**

```
$ psql -p 5432 pubdb
pubdb=# TRUNCATE t1;
pubdb=# SET log_min_duration_statement TO 0;
pubdb=# INSERT INTO t1
          SELECT g, md5(g::text) FROM generate_series(1, 500000) AS g;
pubdb=# \q
```

**（ログから、**
**データ登録時間、以下例では 2913.911 ミリ秒と、**
**データ同期時間、以下例では 21:42:24.938 - 21:42:17.636 = 7302.000 ミリ秒を確認）**

```
$ cat $PGDATA/log/postgresql*.log | tail -100
2023-07-17 21:42:17.636 JST [5266] LOG:  duration: 2913.911 ms  statement:
INSERT INTO t1 SELECT g, md5(g::text) from generate_series(1, 500000) as g;
```

**《該当行のみ抜粋》**

```
$ cat ${PGDATA}_sub/log/postgresql*.log | tail -100
2023-07-17 21:42:24.938 JST [5259] WARNING:  500000 rows processed
```

**《該当行のみ抜粋》**

**（サブスクリプションの streaming オプションを on に設定）**

```
$ psql -p 5434 subdb
subdb=# ALTER SUBSCRIPTION sub SET (streaming = on);
subdb=# \q
```

**《記載省略／同様にパブリッシャでデータを 50 万件登録し、ログから登録・同期時間を確認》**

**（サブスクリプションの streaming オプションを parallel に設定）**

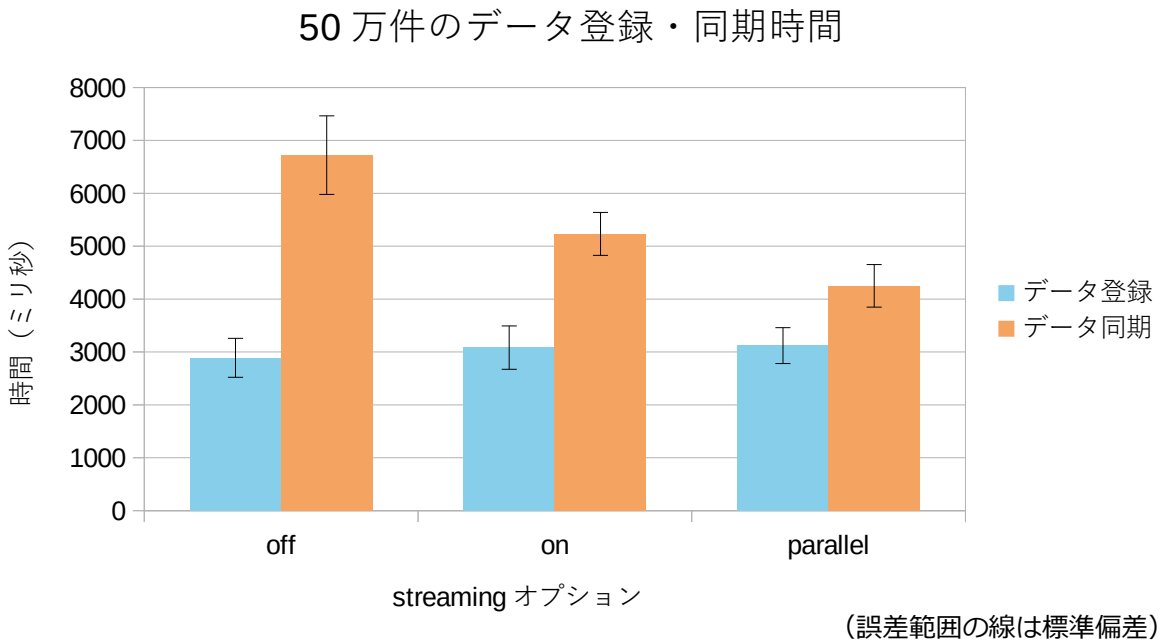
```
$ psql -p 5434 subdb
subdb=# ALTER SUBSCRIPTION sub SET (streaming = parallel);
subdb=# \q
```

**《記載省略／同様にパブリッシャでデータを 50 万件登録し、ログから登録・同期時間を確認》**

以上の手順により、ロジカルレプリケーションが構成されているテーブルに 50 万件を投入したときの INSERT 文の所要時間（データ登録時間）と、そこからサブスクライバ側にデータ投入されるまでの所要時間（データ同期時間）を計測しました。

以下グラフのようにサブスクリプションの streaming オプションによる性能差が確認できました。データ

登録時間ほとんど差異がありませんが、データ同期時間は on が 22%、parallel が 36%短くなっています。ここでの parallel はデフォルトの並列数 2 の場合の結果です。




（誤差範囲の線は標準偏差）

◆ ***並列適用の注意点***

streaming = parallel でロジカルレプリケーションを並列適用した場合の注意点としては、パブリッシャとサブスクライバでデータ更新順序が一致しないことがあり得ることと、並列適用にあたってロック競合があるということです。

以下のようにデータ投入を行ってみました。

```
（t2 テーブルを作成して、streaming = parallel のロジカルレプリケーションを構成）
$ psql -p 5432 pubdb
pubdb=# CREATE TABLE public.t2 (id serial PRIMARY KEY, ts timestamp);
pubdb=# CREATE PUBLICATION pub2 FOR TABLE public.t2;
pubdb=# \q
$ psql -p 5434 subdb
subdb=# CREATE TABLE t2 (id serial PRIMARY KEY, ts timestamp);
subdb=# CREATE SUBSCRIPTION sub2 CONNECTION 'port=5432 dbname=pubdb'
          PUBLICATION pub2 WITH (streaming = parallel);
subdb=# \q
```

**（t2 テーブルに pgbench で 10 並列にデータ投入）**

```
$ echo "INSERT INTO t2 (ts) VALUES (now())" > insert_t2.sql
$ pgbench -n -c 10 -t 10000 -f insert_t2.sql pubdb
```

そのうえで pubdb、subdb データベースで t2 テーブルをソートせずに出力すると、データ順序が異なっており、物理格納順序が異なっていることが確認できました。もちろん、id 列と ts 列の値の組み合わせは一致していますし、本来、テーブルは明示的にソートしない限り順序は保証しないものですので、これは誤動作ではありません。

**（両データベースで t2 テーブルの行データを確認）**

```
$ psql -p 5432 pubdb -c "SELECT * FROM t2 LIMIT 5 OFFSET 0"
 id |             ts
----+----------------------------
  1 | 2023-07-20 09:56:12.34631
  3 | 2023-07-20 09:56:12.346477
  4 | 2023-07-20 09:56:12.346299
  5 | 2023-07-20 09:56:12.346327
 11 | 2023-07-20 09:56:12.358304
(5 rows)
$ psql -p 5434 subdb -c "SELECT * FROM t2 LIMIT 5 OFFSET 0"
 id |             ts
----+----------------------------
  1 | 2023-07-20 09:56:12.34631
  4 | 2023-07-20 09:56:12.346299
  5 | 2023-07-20 09:56:12.346327
  3 | 2023-07-20 09:56:12.346477
  2 | 2023-07-20 09:56:12.346475
(5 rows)
```

このほか、並列にデータ更新を適用するために互いにロック競合するかもしれないことにも注意が必要です。例えば、インデックスの定義状況によってはインデックス更新におけるロック競合で並列化の効果が限定的になる場合が考えられます。

### 4.3.3. 主キー以外インデックスの利用

ロジカルレプリケーションでは、更新・削除の対象行を特定するため、レプリカアイデンティティが必要で、デフォルトでは主キーがそれにあたります。適当なキーがない場合は REPLICA IDENTITY FULL で全列をキーにすることもできますが、その場合、これまで全行をスキャンする必要がありました。本バージョンでは Btree インデックスを利用できるようになり、性能が改善されました。

以下のように性能検証を行いました。

**（パブリッシャで主キーなしテーブルを作成し、レプリカアイデンティティを全列に設定し、インデックスを作成し、パブリケーションを作成）**

```
$ psql -p 5432 pubdb
pubdb=# CREATE TABLE t3 (id int, val text);
pubdb=# ALTER TABLE t3 REPLICA IDENTITY FULL;
pubdb=# CREATE INDEX ON t3 (id);
pubdb=# CREATE PUBLICATION pub3 FOR TABLE t3;
pubdb=# \q
```

**（サブスクライバでテーブルを作成し、データ同期を 10 万件ごとにログに記録するトリガを定義し、サブスクリプションを作成）**

```
$ pg_dump -p 5432 -s -t t3 pubdb | psql -p 5434 subdb
subdb=# CREATE TRIGGER t3_log_rows AFTER UPDATE ON t3
          FOR EACH ROW EXECUTE FUNCTION log_rows();
```

**《↑この関数は 4.3.2. 節で作成したものと同じ》**

```
subdb=# ALTER TABLE t3 ENABLE REPLICA TRIGGER t3_log_rows;
subdb=# CREATE SUBSCRIPTION sub3
          CONNECTION 'port=5432 dbname=pubdb' PUBLICATION pub3;
subdb=# \q
```

**（パブリッシャでデータを 50 万件登録し、全件を更新）**

```
$ psql -p 5432 pubdb
pubdb=# INSERT INTO t3
          SELECT g, md5(g::text) FROM generate_series(1, 500000) AS g;
pubdb=# SET log_min_duration_statement TO 0;
pubdb=# UPDATE t3 SET val = md5(val);
pubdb=# \q
```

（ログから、
データ更新時間、以下例では 6274.850 ミリ秒と、
データ同期時間、以下例では 13:25:29.105 - 13:25:13.553 = 15552.000 ミリ秒を確認）

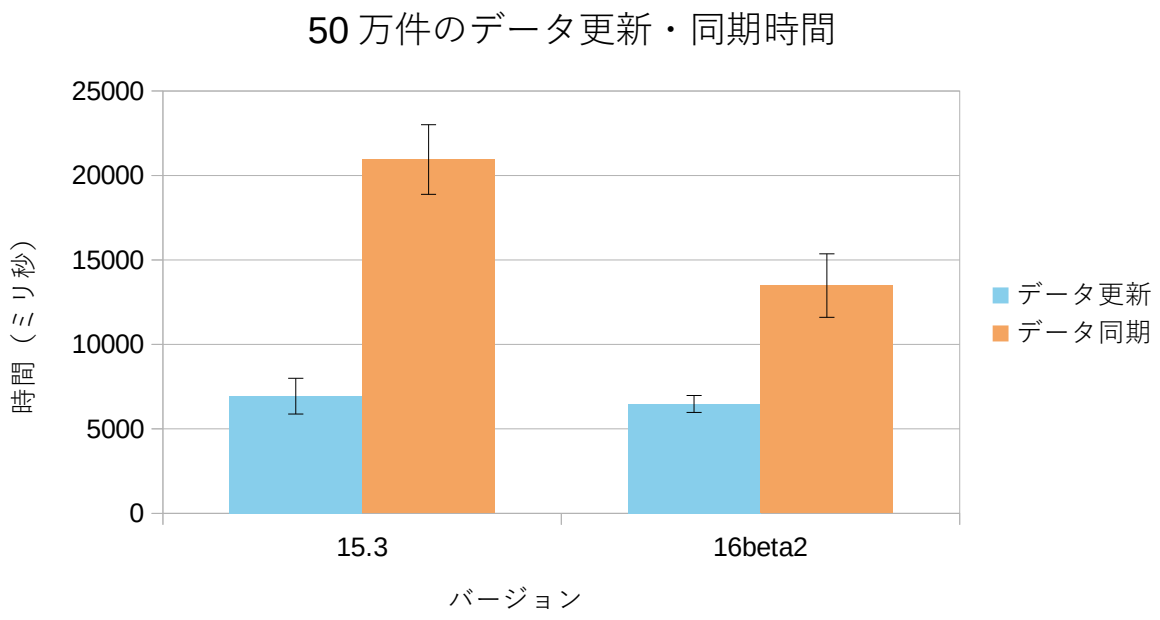
```
$ cat $PGDATA/log/postgresql*.log | tail -100
2023-07-20 13:25:13.553 JST [3859] LOG:  duration: 6274.850 ms  statement:
UPDATE t3 SET val = md5(val);
```

《該当行のみ抜粋》

```
$ cat ${PGDATA}_sub/log/postgresql*.log | tail -100
2023-07-20 13:25:29.105 JST [3849] WARNING:  500000 rows processed
```

《該当行のみ抜粋》

以上の手順により、PostgreSQL 16 と PostgreSQL 15 でロジカルレプリケーションが構成されているテーブルの 50 万件を更新したときの UPDATE 文の所要時間（データ更新時間）と、そこからサブスクライバ側にデータ更新されるまでの所要時間（データ同期時間）を計測しました。




（誤差範囲の線は標準偏差）

その結果、上記グラフのように PostgreSQL 16 での主キー以外のインデックスの利用による性能差が確認できました。データ登録時間はほとんど差異がありませんが、データ同期時間は PostgreSQL 16 が PostgreSQL 15 に比べて 35%短くなっています。

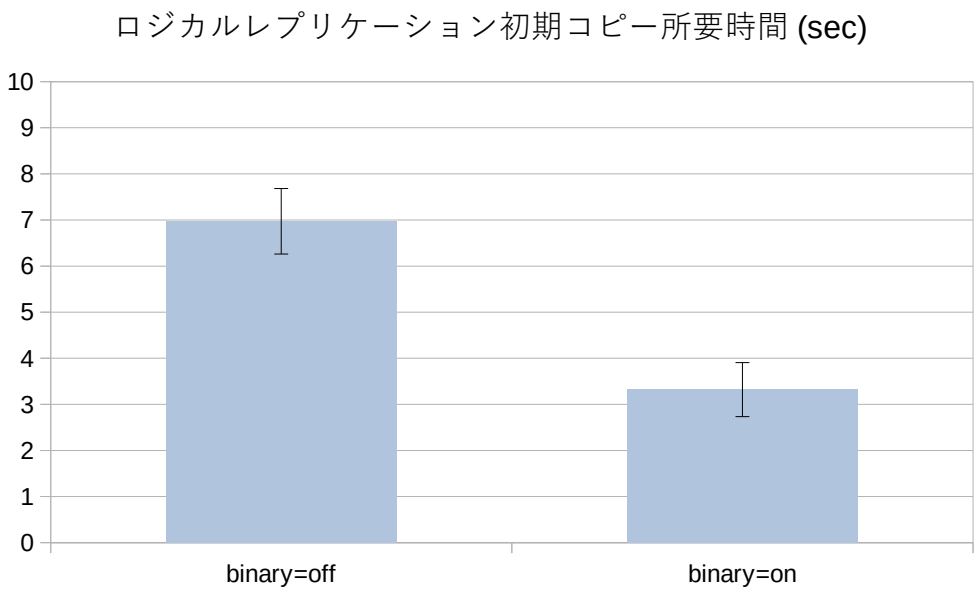
### 4.3.4. 初期コピーでバイナリ形式

ロジカルレプリケーションのデータ転送にはテキスト形式とバイナリ形式があり、これまでバイナリ形式は初期コピー後のデータ同期時にしか使えませんでしたが、本バージョンでは初期コピー時にも使えるようになりました。timestamp 型や bytea 型など、一部のデータ型ではバイナリ形式のほうが効率的に処理でき、所要時間が短くて済みます。バイナリ形式はサブスクリプションの binary オプションで指定し、デフォルトは off でテキスト形式になります。

以下の手順で PostgreSQL 16 でのテキスト形式とバイナリ形式の初期コピーの所要時間を比較しました。

```
（100 万件の bytea 型、timestamp 型のデータをもつ t4 テーブルをレプリケーションする）
$ psql pubdb
pubdb=# CREATE TABLE t4 (id int primary key, b bytea, ts timestamp);
pubdb=# INSERT INTO t4 SELECT g, repeat(md5(g::text), 10)::bytea, now()
         FROM generate_series(1, 100000) g;
pubdb=# CREATE PUBLICATION pub4 FOR TABLE public.t4;
pubdb=# \q
$ psql -p 5434 subdb
subdb=# CREATE TABLE t4 (id int primary key, b bytea, ts timestamp);
subdb=# CREATE SUBSCRIPTION sub4 CONNECTION 'port=5432 dbname=pubdb'
         PUBLICATION pub4 WITH (binary = on);
subdb=# \q                    《↑バイナリ形式なら on 指定、テキスト形式なら off 指定》

（サブスクライバ側のログから初期コピー所要時間 05.530 - 02.652 = 2.878 秒を確認）
$ cat ${PGDATA}_sub/log/* | tail -10
2023-07-21 09:57:02.642 JST [2370] LOG:  logical replication apply worker
for subscription "sub4" has started
2023-07-21 09:57:02.652 JST [2372] LOG:  logical replication table
synchronization worker for subscription "sub4", table "t4" has started
2023-07-21 09:57:05.530 JST [2372] LOG:  logical replication table
synchronization worker for subscription "sub4", table "t4" has finished
      《該当行のみ抜粋》
```

その結果、以下グラフのようにバイナリモードで高速になることが確認できました。所要時間がおよそ半分になっています。




（誤差範囲の線は標準偏差）

### 4.3.5. 双方向ロジカルレプリケーション

これまで PostgreSQL のロジカルレプリケーションでは双方向、あるいは、マルチマスタのレプリケーションには対応していませんでした。そのように構築すると、サーバ 1 での変更をサーバ 2 に反映した後、その変更がサーバ 1 に送られてきて、UPDATE が無限に繰り返されたり、INSERT が主キー重複のエラーになってレプリケーションが止まったりしてしまいます。

PostgreSQL 16 からサブスクリプションに origin オプションが追加されました。デフォルト値は any で、これは PostgreSQL 15 以前と同じ振る舞いになります。origin = none と指定すると、オリジンを持たない変更、すなわち、相手先サーバ内で生じた更新命令による変更差分のみを受け取る動作になります。これにより、双方向レプリケーションが実現できます。

以下の手順で動作確認を実施しました。

**（前節までのロジカルレプリケーションを除去）**

```
$ pg_ctl stop -D ${PGDATA}_sub
$ rm -rf ${PGDATA}_sub
$ dropdb -p 5432 pubdb
```

**（双方向ロジカルレプリケーション検証用に追加インスタンスを 2 つ作成）**

```
$ initdb --no-locale --encoding=UTF8 ${PGDATA}_n33
$ initdb --no-locale --encoding=UTF8 ${PGDATA}_n34
$ cp $PGDATA/postgresql.conf ${PGDATA}_n33/
```

```
$ cp $PGDATA/postgresql.conf ${PGDATA}_n34/
$ pg_ctl start -D ${PGDATA}_n33 -o '-c port=5433'
$ pg_ctl start -D ${PGDATA}_n34 -o '-c port=5434'
```

**（各 PostgreSQL インスタンスにデータベース db32、db33、db34 とテストテーブル t5 を作成）**

```
$ createdb db32
$ createdb -p 5433 db33
$ createdb -p 5434 db34
$ psql -c "CREATE TABLE t5 (id int PRIMARY KEY, v text, ts timestamp)" db32
$ pg_dump -p 5432 -s db32 | psql -p 5433 db33
$ pg_dump -p 5432 -s db32 | psql -p 5434 db34
```

**（3 つのデータベースの間で双方向ロジカルレプリケーションを定義）**

```
$ psql -c "CREATE PUBLICATION pub32 FOR TABLE public.t5" db32
$ psql -c "CREATE PUBLICATION pub33 FOR TABLE public.t5" -p 5433 db33
$ psql -c "CREATE PUBLICATION pub34 FOR TABLE public.t5" -p 5434 db34
$ psql db32
db32=# CREATE SUBSCRIPTION sub32_33 CONNECTION 'dbname=db33 port=5433'
        PUBLICATION pub33 WITH (origin = none);
db32=# CREATE SUBSCRIPTION sub32_34 CONNECTION 'dbname=db34 port=5434'
        PUBLICATION pub34 WITH (origin = none);
db32=# \q
$ psql -p 5433 db33
db33=# CREATE SUBSCRIPTION sub33_32 CONNECTION 'dbname=db32 port=5432'
        PUBLICATION pub32 WITH (origin = none);
WARNING:  subscription "sub33_32" requested copy_data with origin = NONE but
might copy data that had a different origin
DETAIL:  Subscribed publication "pub32" is subscribing to other
publications.
HINT:  Verify that initial data copied from the publisher tables did not
come from other origins.
NOTICE:  created replication slot "sub33_32" on publisher
CREATE SUBSCRIPTION
```

**《初期コピーに関する WARNING や HINT が出力される》**

```
db33=# CREATE SUBSCRIPTION sub33_34 CONNECTION 'dbname=db34 port=5434'
        PUBLICATION pub34 WITH (origin = none);
db33=# \q
$ psql -p 5434 db34
db34=# CREATE SUBSCRIPTION sub34_32 CONNECTION 'dbname=db32 port=5432'
        PUBLICATION pub32 WITH (origin = none);
db34=# CREATE SUBSCRIPTION sub34_33 CONNECTION 'dbname=db33 port=5433'
        PUBLICATION pub33 WITH (origin = none);
            《記載を省略しているが db33 と同様の WARNING や HINT が出力される》
```

ここまでで 3 データベース間の双方向ロジカルレプリケーションが定義できました。同じテーブルに対して 2 つのレプリケーション元があるため、初期コピーでデータが失われてしまう可能性があり、その旨の警告が出力されます。本手順は 0 行の状態から始めていますが、既存データがある場合には、初期コピーをする特定経路以外のサブスクリプションには copy_data = false オプションを与えれば良いでしょう。

続いて、以下の手順で実際に行データの挿入、更新を行いました。

```
（INSERT のレプリケーション動作確認）
$ psql db32
db32=# INSERT INTO t5 VALUES (1, 'AAA', now());      《← db32 でデータ投入》
db32=# SELECT * FROM t5;
 id |  v  |             ts
----+-----+----------------------------
  1 | AAA | 2023-07-25 16:08:16.938802
(1 row)
db32=# \q
$ psql -p 5433 -d db33 -c 'SELECT * FROM t5;'
 id |  v  |             ts
----+-----+----------------------------
  1 | AAA | 2023-07-25 16:08:16.938802
(1 row)
$ psql -p 5434 -d db34 -c 'SELECT * FROM t5;'
 id |  v  |             ts
----+-----+----------------------------
  1 | AAA | 2023-07-25 16:08:16.938802
(1 row)
$ psql -p 5433 -d db33
```

```
db33=# INSERT INTO t5 VALUES (2, 'BBB', now());      《← db33 でデータ投入》
db33=# SELECT * FROM t5;
 id |  v  |             ts
----+-----+----------------------------
  1 | AAA | 2023-07-25 16:08:16.938802
  2 | BBB | 2023-07-25 16:09:53.411778
(2 rows)
db33=# \q
$ psql -p 5432 -d db32 -c 'SELECT * FROM t5'
 id |  v  |             ts
----+-----+----------------------------
  1 | AAA | 2023-07-25 16:08:16.938802
  2 | BBB | 2023-07-25 16:09:53.411778
(2 rows)
$ psql -p 5434 -d db34 -c 'SELECT * FROM t5'
 id |  v  |             ts
----+-----+----------------------------
  1 | AAA | 2023-07-25 16:08:16.938802
  2 | BBB | 2023-07-25 16:09:53.411778
(2 rows)          《↑ 3 つのデータベースが同じデータになっている》

（UPDATE、DELETE のレプリケーション動作確認）
$ psql -p 5434 -d db34
db34=# UPDATE t5 SET v = 'BBBBB', ts = now() WHERE id = 2;
db34=# DELETE FROM t5 WHERE id = 1;          《← db34 でデータ更新と削除》
db34=# SELECT * FROM t5;
 id |   v   |             ts
----+-------+----------------------------
  2 | BBBBB | 2023-07-25 16:11:38.261006
(1 row)
db34=# \q
$ psql -p 5432 -d db32 -c 'SELECT * FROM t5'
 id |   v   |             ts
----+-------+----------------------------
  2 | BBBBB | 2023-07-25 16:11:38.261006
```

```
(1 row)
$ psql -p 5433 -d db33 -c 'SELECT * FROM t5'
 id |   v   |             ts
----+-------+----------------------------
  2 | BBBBB | 2023-07-25 16:11:38.261006
(1 row)            《↑ 3つのデータベースが同じデータになっている》
```

以上のように、いずれか一つのデータベースへの INSERT、UPDATE、DELETE が各データベースに反映されていることが確認できました。

### ◆ *双方向ロジカルレプリケーションが機能しないケース*

双方向ロジカルレプリケーションで問題が出るケースもあります。

以下のように同じ主キー値を持つ行を各データベースに同時に挿入すると何が起きるでしょうか。

```
（db32、db33、db34 に同時に同じ主キーの行を挿入する）
$ psql -p 5432 -d db32 -c "INSERT INTO t5 VALUES (5, '5432', now())" & \
  psql -p 5433 -d db33 -c "INSERT INTO t5 VALUES (5, '5433', now())" & \
  psql -p 5434 -d db34 -c "INSERT INTO t5 VALUES (5, '5434', now())" &

（db32、db33、db34 で行内容を確認）
$ psql -p 5432 -d db32 -c 'SELECT * FROM t5 WHERE id = 5'
 id |   v   |             ts
----+-------+----------------------------
  5 | 5432  | 2023-07-25 16:17:13.182098
(1 row)
$ psql -p 5433 -d db33 -c 'SELECT * FROM t5 WHERE id = 5'
 id |   v   |             ts
----+-------+----------------------------
  5 | 5433  | 2023-07-25 16:17:13.182142
(1 row)
$ psql -p 5434 -d db34 -c 'SELECT * FROM t5 WHERE id = 5'
 id |   v   |             ts
----+-------+----------------------------
  5 | 5434  | 2023-07-25 16:17:13.182127
(1 row)      《↑不一致／それぞれ自データベースで実行した INSERT 結果の値を持っている》
```

**（各インスタンスのサーバログを確認するとレプリケーションエラーが発生している）**

```
$ cat ${PGDATA}/log/* | tail -20
2023-07-25 16:19:03.325 JST [2902] LOG:  logical replication apply worker
for subscription "sub32_33" has started
2023-07-25 16:19:03.332 JST [2902] ERROR:  duplicate key value violates
unique constraint "t5_pkey"
2023-07-25 16:19:03.332 JST [2902] DETAIL:  Key (id)=(5) already exists.
2023-07-25 16:19:03.332 JST [2902] CONTEXT:  processing remote data for
replication origin "pg_16979" during message type "INSERT" for replication
target relation "public.t5" in transaction 770, finished at 0/19E1880
```

**《該当箇所のみ抜粋／1 つのサーバの 1 つのサブスクリプションのみ例示》**

**（サーバログから LSN を特定してエラーを出している更新をスキップ）**

```
$ psql -p 5432 -d db32
db33=# ALTER SUBSCRIPTION sub32_33 SKIP (LSN = '0/19E1880');
ALTER SUBSCRIPTION
```

**《実際には 6 つのサブスクリプション全てにこの処置が必要》**

上記の動作例では、各データベースで行内容が異なるものとなり、なおかつ、全てのサブスクリプションがエラーに陥ってしまいました。

非同期で双方向にロジカルレプリケーションを行うため、タイミングによっては競合して、レプリケーションエラーに陥ってしまうことがあり得ます。また、エラーにならなくとも、テーブルの行データがデータベース間で異なった内容になってしまうこともあり得ます。

PostgreSQL 16 の新たなサブスクリプションのオプション origin = none で、双方向レプリケーションを構築することはできるけれども、どのサーバでも任意のタイミングで安全に更新ができるマルチマスタクラスタというわけではありません。基本的に同じデータの同時更新は安全ではありません。同時データ更新に対して、レプリケーションエラーが起きるケースや、各データベース間でのデータが一貫しなくなるケースについては、利用者側（アプリケーション側）で考慮する必要があります。

# 4.4. クライアント機能

## 4.4.1. libpq ロードバランス

PostgreSQL のクライアントライブラリである libpq にロードバランス機能が追加されました。本機能を利用することで、複数の接続先を指定したデータベースサーバへの接続を確率的に均等に分散させることが可能になり、システム全体のパフォーマンスと耐久性の向上が見込めます。

libpq のロードバランス機能は、接続文字列の load_balance_hosts オプションによって制御されます 。load_balance_hosts には、disable と random のいずれかの値を設定することができます。それぞれの値の意味は以下表のとおりです。

| 設定値 | 説明 |
|---|---|
| disable（デフォルト） | ロードバランスは行われず、接続先のホストは指定された順序で試みられ、アドレスは DNS やホストファイルから得られる順序で試みられます。 |
| random | 接続先のホストとアドレスはランダムな順序で決定されます。これにより、複数の PostgreSQL サーバー間で接続が分散され、ロードバランスが実現されます。 |

なお、環境変数 PGLOADBALANCEHOSTS でも同じ指定ができます。pgbench など、任意の接続オプションが指定できないクライアントコマンドでは環境変数を利用してください。

これまでも PostgreSQL 10 以降であれば libpq では複数の接続先を指定することが可能でした。しかし 、PostgreSQL 15 以前では、指定された順序でホストへの接続を試み、最初に接続が成功したホストに対してのみクエリが送信されていました（disable と同じ）。つまり、複数の接続先を指定可能であってもロードバランシングが実現されていませんでした。これに対して、PostgreSQL 16 では機能が拡張され、接続を複数のホスト間で確率的に均等に分散させることが可能になりました。なお、複数接続先指定を前提とした場合でも、接続タイムアウト時間を設定する connect_timeout の指定は依然として推奨されます。

### ◆ *動作例*

libpq のロードバランスを実際に試しました。

まずサービスは、ローカルにポートが 5432 の DB サービスと、ポートが 5433 の DB サービスの 2 つが動作しているものとします。

次にクライアントは、libpq による接続が可能な psql から予め用意した複数の接続先を指定し、および load_balance_hosts=random を指定します。このとき、接続先の複数指定は -h (host)、-p (port) にカンマ区切りで記載します（今回は-h localhost,localhost -p 5432,5433）。ホストやポートが共通する場合でも、接続先のホストとポートがそれぞれ対応するように記載する必要があるので注意してください。

psql で発行するクエリには "SHOW port" を指定し、接続先のポートを表示することで実際の接続先の確認を行います。

以上のように指定した psql の実行を 10 回行いました。

**（接続先を 2 つ指定し、接続を 10 回試みる）**

```
$ for i in $(seq 1 10); do psql -h localhost,localhost -p 5432,5433
'load_balance_hosts=random' -c "SHOW port" -tA; done
5433
5432
5433
5433
5433
5433
5432
5433
5432
5433
```

今回はポート 5432 には 3 回、5433 には 7 回接続される結果となりました。libpq のロードバランスはランダムで接続先を決定しているので必ずしも各接続先の接続が同じ回数行われるわけではないことがわかります。

### ◆ *応用例　ホットスタンバイ*

先の例では複数の独立したデータベースを接続先に指定しましたが、一般的にロードバランスで取得されるデータは接続先に依らず同じデータが取得できることが求められます。

PostgreSQL において、常に同じデータを提供する複数のデータベースサービスはホットスタンバイ機能により構築可能です。これまでは、プライマリとスタンバイに対してクライアントが単一の接続先を設定してロードバランスを行うか、Pgpool-II などのツールにより自動的に接続先を振り分けるロードバランスが可能でしたが、今回から新たに libpq によるロードバランスという選択肢が増えました。

ただし、ホットスタンバイに対して libpq のロードバランスを利用するには少し制約があります。それを見ていきましょう。まずは、スタンバイを作成し、扱うデータを準備します。

**（スタンバイの作成とデータの準備）**

```
$ pg_basebackup -p 5432 -D data_standby -c fast -R -Pv
$ pg_ctl start -D data_standby -o '-p 6432'
$ psql -p 5432
=# CREATE TABLE t1 (i int);
```

```
=# INSERT INTO t1 VALUES (1);
```

次に、プライマリとスタンバイを接続先として指定し、10 回接続を試みます。

**（接続先を 2 つ指定し、接続を 10 回試みる）**

```
$ for i in $(seq 1 10); do psql -h /tmp,/tmp -p 5432,6432
'load_balance_hosts=random' -c "SHOW PORT" -tA; done
6432
5432
6432
6432
5432
5432
5432
6432
5432
6432
```

ホットスタンバイを接続先に指定した場合でも、libpq のロードバランスが動作していることがわかります。では更新クエリを用いた次の例はどうでしょう。

**（接続先を 2 つ指定し、更新クエリの発行を 10 回試みる）**

```
$ for i in $(seq 1 10); do psql -h /tmp,/tmp -p 5432,6432
'load_balance_hosts=random' -c "UPDATE t1 SET i = 1"; done
UPDATE 1
ERROR:  cannot execute UPDATE in a read-only transaction
UPDATE 1
ERROR:  cannot execute UPDATE in a read-only transaction
ERROR:  cannot execute UPDATE in a read-only transaction
UPDATE 1
ERROR:  cannot execute UPDATE in a read-only transaction
UPDATE 1
UPDATE 1
UPDATE 1
```

更新クエリを発行したこちらの例では、エラーが発生しているものとそうでないものがあります。これは

そもそもスタンバイが参照クエリしか受け付けられないので、スタンバイに更新クエリを発行したものはエラーが返ってしまいます。

更新クエリを発行するには最初から接続先にプライマリのみを指定するか、接続文字列 target_session_attrs に primary と指定する必要があります。以下に後者の例を示します。

**（接続先を 2 つ指定し、target_session_attrs=primary を指定し、更新クエリの発行を 10 回試みる）**

```
$ for i in $(seq 1 10); do psql -h /tmp,/tmp -p 5432,6432
'load_balance_hosts=random target_session_attrs=primary' -c "UPDATE t1 SET i
= 1"; done
UPDATE 1
UPDATE 1
UPDATE 1
UPDATE 1
UPDATE 1
UPDATE 1
UPDATE 1
UPDATE 1
UPDATE 1
```

更新クエリは全て成功し、エラーが発生していないことがわかります。これは target_session_attrs=primary の指定により、全ての更新クエリがプライマリ側で発行されているためです。

なお、target_session_attrs に standby を指定することで、プライマリに参照クエリをロードバランスさせないことも可能です。

**（接続先を 2 つ指定し、target_session_attrs=standby を指定し、ポートを表示する参照クエリの発行を 10 回試みる）**

```
$ for i in $(seq 1 10); do psql -h /tmp,/tmp -p 5432,6432
'load_balance_hosts=random target_session_attrs=standby' -c "SHOW port" -tA;
done
6432
6432
6432
6432
6432
6432
```

```
6432
6432
6432
6432
```

今度は target_session_attrs=primary の指定により、全てのクエリがスタンバイ側(port=6432)で発行されていることが見て取れます。なお、スタンバイが複数存在する場合にはスタンバイ間でロードバランスが行われます。

以上までの例の通り、接続中に発行するクエリの内容によってクライアントは接続文字列を調整しなくてはいけないので、ホットスタンバイを用いたロードバランスはクライアント側は接続先の考慮が依然として必要です。クライアント側が複数の接続先や発行するクエリを考慮することなくロードバランスを実現したいということであれば、Pgpool-II の利用を検討してください。

また、注意すべき点として libpq の複数接続先指定では、最初の接続試行に失敗したら別の接続先への接続を試みますが、その後にサーバ側からの切断などでクエリ実行が失敗しても別の接続先への接続が試みることはありません。

### 4.4.2. psql で拡張問い合わせプロトコル対応

psql コマンドで、拡張問い合わせプロトコルを使って SQL 文のパラメータに値を渡す \bind メタコマンドがサポートされました。これまで、psql 上の SQL 実行は簡易問い合わせしかできませんでした。そのため、拡張問い合わせで実行したときの検証を行うときには、代替に pgbench が使われていました。

使い方は、\bind メタコマンドで値を設定した後に、$1、$2 などのバインド変数を使った SQL 文を実行します。psql コマンドはバインド変数に\bind メタコマンドで設定した値を設定したクエリが拡張クエリプロトコルを使ってバックエンドプロセスに渡され処理されます。その後は \bind メタコマンドで異なる値を設定して、\g メタコマンドで同じ SQL 文を実行することが可能です。

以下に使用例を示します。

```
（\bind メタコマンドでバインド変数 $1、$2 に値を設定）
db1=# \bind 'foo' 'bar'

（バインド変数 $1、$2 を使った SQL 文を実行）
db1=# SELECT $1, $2;
 ?column? | ?column?
----------+----------
 foo      | bar
(1 row)
```

```
（バインド変数に異なる値を設定しなおして、\g メタコマンドで同じ SQL 文を再実行）
db1=# \bind 'abc' 'xyz'
db1=# \g
 ?column? | ?column?
----------+----------
 abc      | xyz
(1 row)
```

postgresql.conf で log_min_duration_statement = 0 を設定してある状態で、psql 上で上記のコマンドを実行したときの PostgreSQL サーバログ出力は、以下のようになります。

```
2023-07-12 10:04:23.445 JST [6250] LOG:  duration: 0.205 ms  parse
<unnamed>: SELECT $1, $2;
2023-07-12 10:04:23.445 JST [6250] LOG:  duration: 0.086 ms  bind <unnamed>:
SELECT $1, $2;
2023-07-12 10:04:23.445 JST [6250] DETAIL:  parameters: $1 = 'foo', $2 =
'bar'
2023-07-12 10:04:23.446 JST [6250] LOG:  duration: 0.006 ms  execute
<unnamed>: SELECT $1, $2;
2023-07-12 10:04:23.446 JST [6250] DETAIL:  parameters: $1 = 'foo', $2 =
'bar'
2023-07-12 10:04:29.942 JST [6250] LOG:  duration: 0.052 ms  parse
<unnamed>: SELECT $1, $2;
2023-07-12 10:04:29.942 JST [6250] LOG:  duration: 0.039 ms  bind <unnamed>:
SELECT $1, $2;
2023-07-12 10:04:29.942 JST [6250] DETAIL:  parameters: $1 = 'abc', $2 =
'xyz'
2023-07-12 10:04:29.942 JST [6250] LOG:  duration: 0.007 ms  execute
<unnamed>: SELECT $1, $2;
2023-07-12 10:04:29.942 JST [6250] DETAIL:  parameters: $1 = 'abc', $2 =
'xyz'
```

名前なし（<unnamed>）プリペアドステートメントが順に parse、bind、execute されていて、拡張問い合わせプロトコルが使用されていることが分かります。

### 4.4.3. pg_dump 圧縮オプション

pg_dump の出力ファイルを圧縮するオプション -Z / --compress の圧縮方式で、従来利用可能であった gzip 以外に新しい圧縮方式として lz4 と zstd が追加されました。オプション引数で指定する圧縮方式は 、gzip、lz4、zstd、または none(非圧縮)で、追加オプションとして圧縮レベルの数字や long が追加指定できます。圧縮が可能な出力形式は前バージョンと同じく tar 形式以外のプレインテキスト、カスタム形式、ディレクトリ形式です。本検証では、出力形式がプレインテキストの場合に、各圧縮方式でレベルを指定しない場合と最大レベルを指定した場合の出力ファイルサイズと pg_dump の実行時間を確認しました。

各圧縮方式で指定可能な圧縮レベルの数字の範囲は、gzip が 1～9、lz4 が 1～12、zstd が -131072～22 となっています。

**（pg_dump で使用するサンプルデータベースを作成）**

```
$ pgbench -i db1
dropping old tables...
NOTICE:  table "pgbench_accounts" does not exist, skipping
：(略)
creating primary keys...
done in 0.29 s (drop tables 0.00 s, create tables 0.01 s, client-side
generate 0.15 s, vacuum 0.04 s, primary keys 0.08 s).
```

**（圧縮方式 none、gzip、lz4、zstd について、プレインテキスト形式の pg_dump で圧縮レベルを指定せずに実行[下記は lz4 の場合]）**

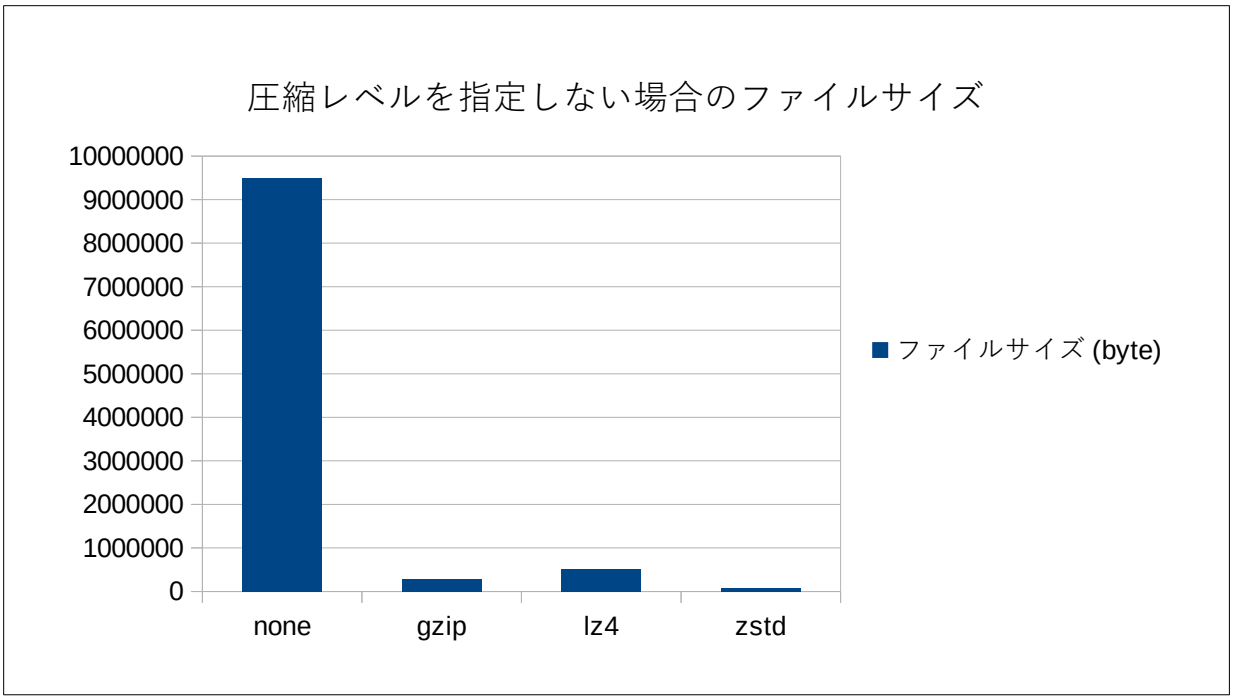
```
$ time pg_dump db1 -f db1.dmp.lz4 -Z lz4
real    0m0.218s
user    0m0.077s
sys     0m0.007s
```

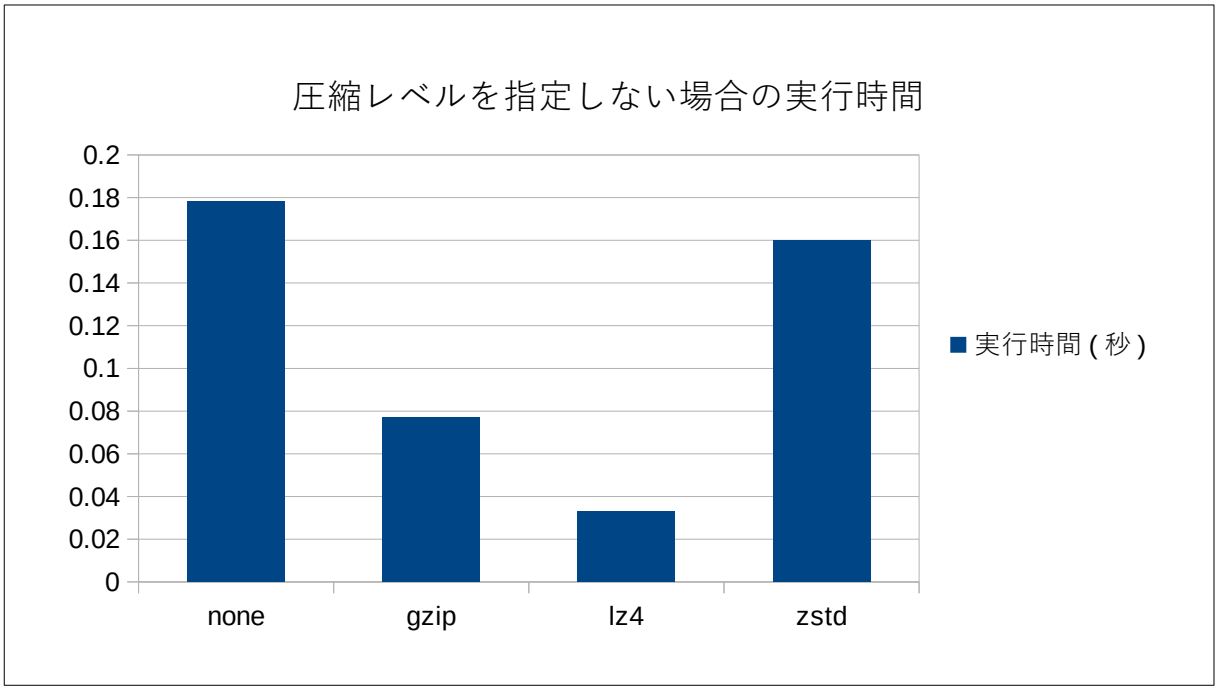
**（圧縮方式 none、gzip、lz4、zstd について、プレインテキスト形式の pg_dump で最大の圧縮レベルを指定して実行[下記は zstd の場合]）**

```
$ time pg_dump db1 -f db1.dmp.zstd:22 -Z zstd:22
real    0m15.496s
user    0m11.730s
sys     0m0.448s
```

各圧縮方式でレベルを指定しない場合のファイルサイズと実行時間は下記のようになりました。
ファイルサイズの圧縮率は高いものから zstd、gzip、lz4、none の順で、実行時間は短いものから

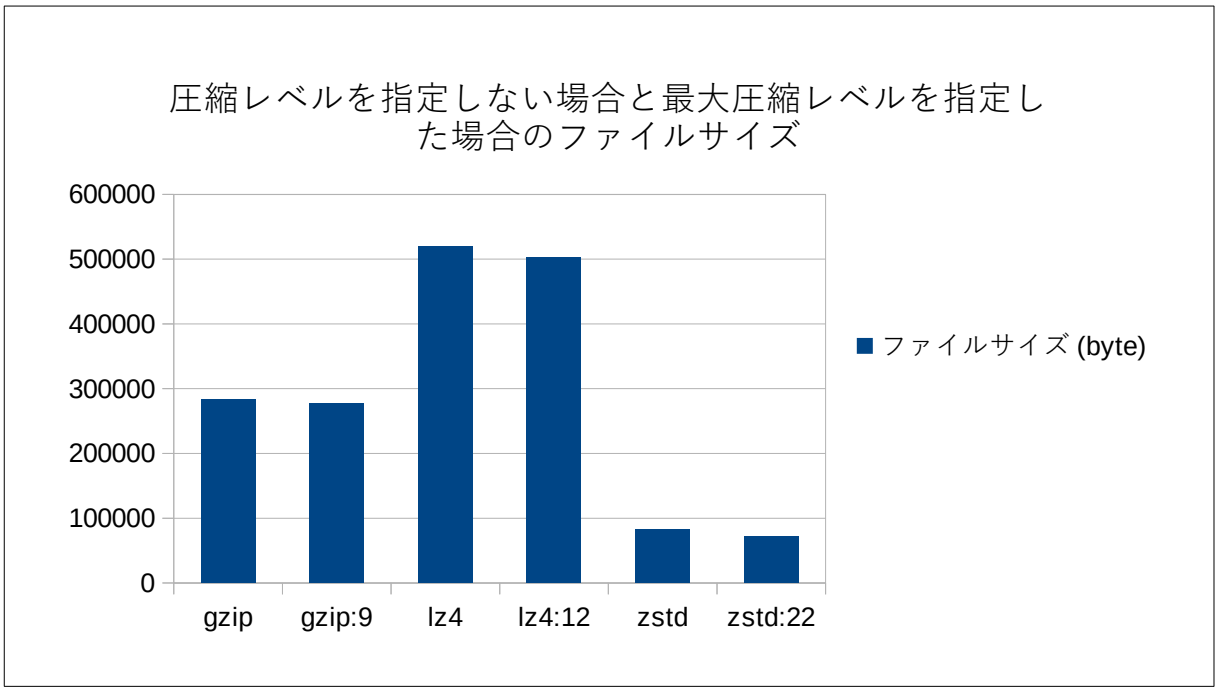
lz4、zstd、none、gzip の順となりました。

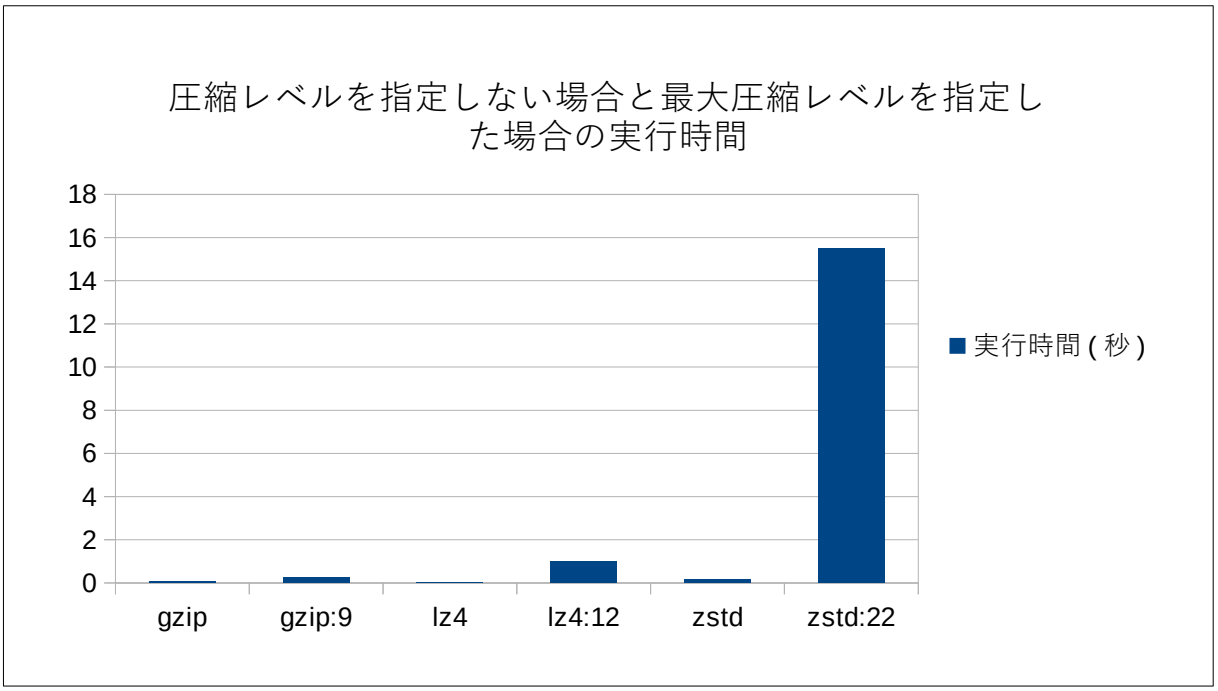



また、レベルを指定しない場合の圧縮レベルは、gzip はレベル 6/7 と同一サイズ、lz4 と zstd はレベル 0 と同一サイズとなりました。

各圧縮方式で最大レベルを指定した場合のファイルサイズと実行時間を、レベルを指定しない場合と並べた結果は下記のようになりました。




ファイルサイズはいずれもレベルを指定しない場合より小さくなりましたが、レベルを指定しない場合とのファイルサイズの差異はあまり無いことに比べて実行時間は大幅に増加する圧縮方式もあることから、レベル指定が無しでも実用的な使い勝手のファイルサイズと実行時間であると言えます。

## 4.5. 運用管理

### 4.5.1. 新たなモニタリング項目

本バージョンでは、データベースサーバの状態を取得するためのビューや関数、ビューが出力する情報がいくつか追加されています。以下の表示に示します。

| 関数／ビュー | 説明 |
|---|---|
| ビュー pg_stat_io | I/O に関する統計情報を出力。 |
| ビュー pg_stat_*_tables/indexes の<br>列 last_seq_scan、last_idx_scan | 最後にシーケンシャルスキャン、インデックススキャンが実行された時刻を出力。 |
| ビュー pg_stat_*_tables の<br>列 n_tup_newpage_upd | 新しいページへの行バージョン作成を伴って更新された行数を出力。 |
| 関数<br>pg_stat_get_backend_subxact(backendid) | バックエンドのサブトランザクションキャッシュ内のサブトランザクション数、そのキャッシュがオーバーフローしているかを出力。引数は pid ではなく backendid。 |

ビュー pg_stat_io について、実行例を示します。バックエンド種別、対象オブジェクト種別、実行コンテキストの組み合わせごとにストレージ I/O の各種統計値を得ることができます。

```
db1=# SELECT * FROM pg_stat_io;
《中略》
-[ RECORD 8 ]--+------------------------------
backend_type   | client backend
object         | relation
context        | normal
reads          | 13443
read_time      | 0
writes         | 0
write_time     | 0
writebacks     | 0
writeback_time | 0
extends        | 613
extend_time    | 0
```

```
op_bytes        | 8192
hits            | 1466159
evictions       | 1077
reuses          |
fsyncs          | 0
《後略》
```

関数 pg_stat_get_backend_subxact についても使用例を示します。

```
（例外処理付き関数の再帰呼び出しでサブトランザクションを多用する pgbench シナリオを作成）
$ psql db1
db1=# CREATE TABLE t_subtx_log (id int, cnt int, ts timestamp);
db1=# INSERT INTO t_subtx_log SELECT g, 0, null
        FROM generate_series(1, 10000) g;
db1=# CREATE FUNCTION f_subtx(i int, c int) RETURNS boolean
        LANGUAGE plpgsql AS $$
  BEGIN
    UPDATE t_subtx_log SET cnt = c, ts = now() WHERE id = i;
    IF c > 0 THEN RETURN f_subtx(i, c - 1); ELSE RETURN true END IF;
  EXCEPTION WHEN others THEN RETURN false;
  END; $$;

db1=# \q
$ cat > f_subtx.sql <<EOS
\set id random(1, 10000)
SELECT f_subtx( :id , 100);
EOS
$ pgbench -f f_subtx.sql -n -c 5 -t 100 db1

（別の psql セッションで並行して実行）
db1=# SELECT v2.pid, wait_event_type, wait_event, s.* FROM (
      SELECT a.*, v.backendid FROM pg_stat_activity a JOIN (
       SELECT backendid, pg_stat_get_backend_pid(backendid) AS pid
        FROM pg_stat_get_backend_idset() backendid) v ON v.pid = a.pid) v2,
        LATERAL pg_stat_get_backend_subxact(v2.backendid) s
          WHERE backend_type = 'client backend';
```

```
 pid  | wait_event_type |  wait_event   | subxact_count | subxact_overflowed
------+-----------------+---------------+---------------+--------------------
 3958 | *null*          | *null*        |             0 | f
 4196 | LWLock          | SubtransSLRU  |            61 | f
 4197 | LWLock          | SubtransSLRU  |            64 | t
 4198 | *null*          | *null*        |            31 | f
 4199 | *null*          | *null*        |            47 | f
 4200 | LWLock          | SubtransSLRU  |             1 | f
(6 rows)
```

pg_stat_get_backend_subxact 関数はセッションごとの情報をレコード型で返します。引数は pid ではなくバックエンド ID を使用します。バックエンド ID とバックエンドプロセスの PID を対応付けるため、本例では、pg_stat_get_backend_pid 関数と pg_stat_get_backend_idset 関数を使っています。また、レコード型の返し値を扱うため、LATERAL 結合を使っています。戻り値レコードに含まれるには、subxact_count 列（キャッシュ上のサブトランザクション数）と subxact_overflowed 列（キャッシュが溢れたかどうか）が含まれています。

多数の同時実行プロセスでサブトランザクションを多用されることで、サブトランザクションのキャッシュが溢れてファイルアクセスが生じたり、ロック待ちが生じて動作が遅くなるケースがあります。本関数を使って、実際にセッションのサブトランザクションが多いか、キャッシュのオーバーフローが起きているのかを確認することができました。

### 4.5.2. 新たな定義済みロール

本バージョンでは、以下の 3 つの定義済みシステムロールが追加されました。いずれもあらかじめ作成されているのみならず、特別な機能を持っています。

| ロール名 | 説明 |
|---|---|
| pg_maintain | 全てのテーブルのメンテナンス操作（VACUUM 、ANALYZE 、CLUSTER、REFRESH MATERIALIZED VIEW、REINDEX、LOCK TABLE）を実行できるロール。以前はメンテナンス操作はテーブルの所有者またはスーパーユーザしか実行できませんでした。<br>**→ 本ロールの追加は beta3 で取り消されました** |

| ロール名 | 説明 |
|---|---|
| pg_create_subscription | データベースの CREATE 権限を持つユーザがサブスクリプションを作成できるロール。以前はサブスクリプションはスーパーユーザしか作成できませんでした。 |
| pg_use_reserved_connections | パラメータ reserved_connections で予約した接続数を使って接続できるロール。reserved_connections は本バージョンで追加されたパラメータで、スーパーユーザ用の superuser_reserved_connections とは別に予約する接続数を指定し、デフォルトは 0 です。 |

以下の動作確認を行いました。

```
（pg_maintain メンバーとして maintainer を作成し、VACUUM を実行できるかを確認）
db1=# CREATE ROLE maintainer LOGIN IN ROLE pg_maintain;
CREATE ROLE
db1=# \c - maintainer
You are now connected to database "db1" as user "maintainer".
db1=> VACUUM pgbench_accounts;
VACUUM

（通常ユーザ、pg_use_reserved_connections メンバー、スーパーユーザが
 それぞれ接続数を使い切った場合の動作を確認）
《通常ユーザ alice と pg_use_reserved_connections メンバーとして bob を作成》
db1=# CREATE ROLE alice LOGIN;
CREATE ROLE
db1=# CREATE ROLE bob LOGIN IN ROLE pg_use_reserved_connections;
CREATE ROLE
db1=# \q

《最大接続数 max_connections を 10 に減らし、reserved_connections を 3 に設定》
$ vi $PGDATA/postgresql.conf
  max_connections = 10
  reserved_connections = 3
  superuser_reserved_connections = 3
$ pg_ctl restart
```

```
《別端末で通常ユーザ alice として接続》
$ psql -U alice db1
$ psql -U alice db1
$ psql -U alice db1
$ psql -U alice db1

《一般ユーザは
  max_connections - (superuser_reserved_connections + reserved_connections)
  が 4 を超える接続はエラーになる》
$ psql -U alice db1
psql: error: connection to server on socket "/tmp/.s.PGSQL.5432" failed:
FATAL:  remaining connection slots are reserved for roles with privileges of
the "pg_use_reserved_connections" role

《別端末で pg_use_reserved_connections メンバー bob として接続》
$ psql -U bob db1
$ psql -U bob db1
$ psql -U bob db1

《pg_use_reserved_connections メンバーは reserved_connections = 3 の接続数は
  予約されているが、一般ユーザと合わせて
  max_connections - superuser_reserved_connections = 7 を超える接続は
  エラーになる》
$ psql -U bob db1
psql: error: connection to server on socket "/tmp/.s.PGSQL.5432" failed:
FATAL:  remaining connection slots are reserved for roles with SUPERUSER

《別端末でスーパーユーザ postgres として接続》
$ psql -U postgres db1
$ psql -U postgres db1
$ psql -U postgres db1

《スーパーユーザは superuser_reserved_connections = 3 の接続数は予約されているが、
  スーパーユーザ以外も含めて max_connections = 10 を超える接続はエラーになる》
$ psql -U postgres db1
```

```
psql: error: connection to server on socket "/tmp/.s.PGSQL.5432" failed:
FATAL:  sorry, too many clients already
```

以上の通り、新たな定義済みロールの働きを確認できました。

## 4.5.3. ページ単位でのタプル凍結

本バージョンから VACUUM 処理におけるページ単位でのタプル凍結（フリーズ、FREEZE）処理が行われるようになりました。

PostgreSQL のテーブルの物理的な行（タプル）は、そのタプルを作ったときのトランザクション ID（XID)をメタデータとして保持しています。これは同時実行される行の挿入、更新、削除、参照に対して、トランザクション分離レベルに応じた一貫性のあるデータ読み取りを実現するために使われます。一方で、XID は今のところ 32bit 整数であるため番号の周回が発生しますので、古い XID、新しい XID の関係が壊れないように、定期的に古い XID を持つタプルを凍結する（＝最も古い XID で作られたタプルと印付けする）必要があり、これは VACUUM 処理の中で行われます。

これまでの VACUUM 処理では、前回の凍結処理からの XID 発番の進行量を基準に「積極的な VACUUM」を行うと判断された場合にテーブル全体について凍結処理を行うという動作をしていました（ページ単位で凍結済みであるかを記録しておいて処理を省略することはできました）。

本バージョンからは「積極的な VACUUM」と判断されない場合でも、WAL へのフルページ書き込みをするときには、ページ単位での凍結処理も一緒に行うようになりました。また、凍結処理に関する WAL 出力量を重複を無くすことで小さくする改善も適用されています。

### ◆ *ページ単位の凍結*

以下の手順でページ単位の凍結動作を確認しました。

contrib 配布の拡張 pageinspect を使用して、テーブルの各タプルが凍結されているかを確認することができます。

t_xmin がタプル作成時の XID 値です。タプルが凍結されているかどうかは infomask フラグビットで印付けされます。「(i.t_infomask & 768) = 768 frozen」の式はそれを調べるものです。

```
（pageinspect 拡張を導入し、テストテーブル・テストデータを作成）
db1=# CREATE EXTENSION pageinspect;
db1=# CREATE TABLE t_freeze (id int, v text);
db1=# INSERT INTO t_freeze SELECT g, substr(md5(g::text),1,10)
        FROM generate_series(1, 10) g;
```

（テーブル t_freeze の 0 ページ目における各タプルの凍結状態と格納データを参照する）

```
db1=# SELECT i.lp, i.t_xmin, i.t_infomask,
       (i.t_infomask & 768) = 768 frozen, t.*
       FROM heap_page_items(get_raw_page('public.t_freeze', 0)) i
       LEFT JOIN public.t_freeze t ON ('(0,' || i.lp || ')')::tid = t.ctid;
 lp | t_xmin | t_infomask | frozen | id |     v
----+--------+------------+--------+----+------------
  1 |  50976 |       2050 | f      |  1 | c4ca4238a0
  2 |  50976 |       2050 | f      |  2 | c81e728d9d
  3 |  50976 |       2050 | f      |  3 | eccbc87e4b
  《中略》
  8 |  50976 |       2050 | f      |  8 | c9f0f895fb
  9 |  50976 |       2050 | f      |  9 | 45c48cce2e
 10 |  50976 |       2050 | f      | 10 | d3d9446802
(10 rows)
```

（行を更新すると作成時 XID を示す t_xmin の値が異なるタプルが増える）

```
db1=# UPDATE t_freeze SET v = upper(v);
db1=# SELECT i.lp, i.t_xmin, i.t_infomask,
       (i.t_infomask & 768) = 768 frozen, t.*
       FROM heap_page_items(get_raw_page('public.t_freeze', 0)) i
       LEFT JOIN public.t_freeze t ON ('(0,' || i.lp || ')')::tid = t.ctid;
 lp | t_xmin | t_infomask | frozen |   id   |     v
----+--------+------------+--------+--------+------------
  1 |  50976 |        258 | f      | *null* | *null*
  2 |  50976 |        258 | f      | *null* | *null*
  《中略》
  9 |  50976 |        258 | f      | *null* | *null*
 10 |  50976 |        258 | f      | *null* | *null*
 11 |  50977 |      10242 | f      |      1 | C4CA4238A0
 12 |  50977 |      10242 | f      |      2 | C81E728D9D
  《中略》
 19 |  50977 |      10242 | f      |      9 | 45C48CCE2E
 20 |  50977 |      10242 | f      |     10 | D3D9446802
(20 rows)
```

CHECKPOINT 処理の後に VACUUM を行うと、WAL 出力でフルページ書き込みが生じます。このときに新機能のページ単位の凍結処理が働くはずです。

```
db1=# CHECKPOINT;
db1=# VACUUM VERBOSE public.t_freeze;
INFO:  vacuuming "db1.public.t_freeze"          ←「aggressively」ではない
INFO:  finished vacuuming "db1.public.t_freeze": index scans: 0
pages: 0 removed, 1 remain, 1 scanned (100.00% of total)
tuples: 10 removed, 10 remain, 0 are dead but not yet removable
removable cutoff: 50978, which was 0 XIDs old when operation ended
new relfrozenxid: 50978, which is 3 XIDs ahead of previous value
frozen: 1 pages from table (100.00% of total) had 10 tuples frozen
                                          ↑ページ単位タプル凍結の報告
index scan not needed: 0 pages from table (0.00% of total) had 0 dead item
identifiers removed
    《以降の VACUUM VERBOSE 出力を省略》
```

VACUUM VERBOSE の出力から、「aggressively vacuuming」（積極的な VACUUM）として動作していない一方、「frozen:」という項目でタプル凍結したことが報告されていることが確認できました。さらに、以下のようにタプル状態を確認すると、フラグビットに凍結済みフラグがセットされています。

```
db1=# SELECT i.lp, i.t_xmin, i.t_infomask,
       (i.t_infomask & 768) = 768 frozen, t.*
       FROM heap_page_items(get_raw_page('public.t_freeze', 0)) i
       LEFT JOIN public.t_freeze t ON ('(0,' || i.lp || ')')::tid = t.ctid;
 lp | t_xmin | t_infomask | frozen |   id   |     v
----+--------+------------+--------+--------+------------
  1 | *null* |     *null* | *null* | *null* | *null*
  2 | *null* |     *null* | *null* | *null* | *null*
  《中略》
  9 | *null* |     *null* | *null* | *null* | *null*
 10 | *null* |     *null* | *null* | *null* | *null*
 11 |  50977 |      11010 | t      |      1 | C4CA4238A0
 12 |  50977 |      11010 | t      |      2 | C81E728D9D
 13 |  50977 |      11010 | t      |      3 | ECCBC87E4B
```

```
《中略》
 18 |  50977 |      11010 | t      |      8 | C9F0F895FB
 19 |  50977 |      11010 | t      |      9 | 45C48CCE2E
 20 |  50977 |      11010 | t      |     10 | D3D9446802
(20 rows)                    ↑タプルが凍結されている
```

テーブルの一部だけが凍結されるという動作パターンも確認しました。10 万行のデータに対して 500 行程度の更新をかけて、チェックポイント後に VACUUM を実行します。

```
db1=# TRUNCATE t_freeze;
db1=# INSERT INTO t_freeze SELECT g, substr(md5(g::text),1,10)
        FROM generate_series(1, 100000) g;
db1=# UPDATE t_freeze SET v = upper(v) WHERE id < 500;
db1=# CHECKPOINT;
db1=# VACUUM VERBOSE t_freeze;
INFO:  vacuuming "db1.public.t_freeze"
INFO:  finished vacuuming "db1.public.t_freeze": index scans: 0
pages: 0 removed, 544 remain, 544 scanned (100.00% of total)
tuples: 499 removed, 100000 remain, 0 are dead but not yet removable
removable cutoff: 50971, which was 0 XIDs old when operation ended
new relfrozenxid: 50969, which is 1 XIDs ahead of previous value
frozen: 1 pages from table (0.18% of total) had 56 tuples frozen
                    ↑ 一部だけが凍結されている
index scan not needed: 3 pages from table (0.55% of total) had 499 dead item
identifiers removed
    《以降の VACUUM VERBOSE 出力を省略》
```

この手順ですと t_freeze の大部分のページは削除済み領域を持たないために VACUUM 処理がスキップされますので、それらページには凍結処理も行われない結果となりました。

**◆ 凍結処理での *WAL* 出力量の削減**

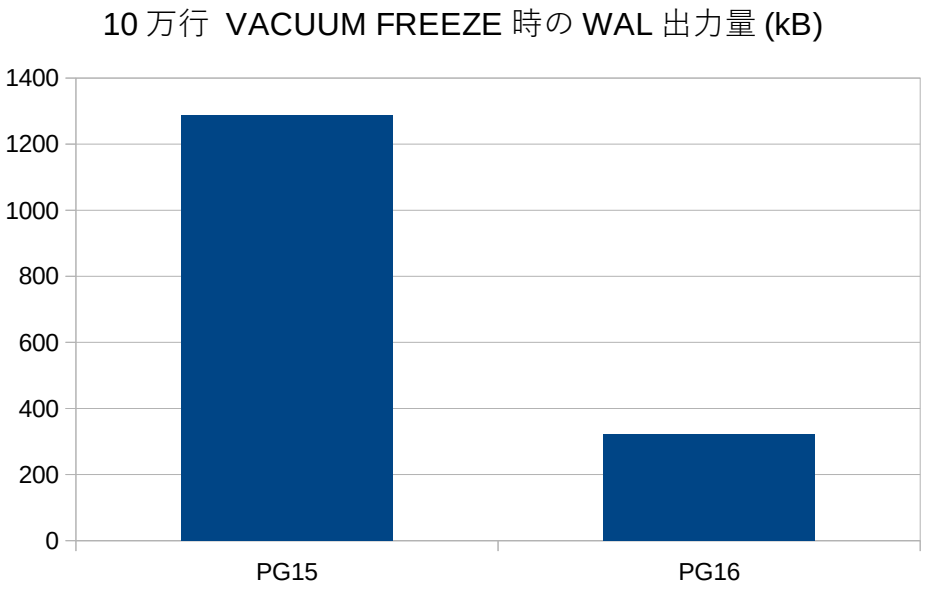
WAL 出力量の削減も以下の手順で確認しました。

10 万行のテーブルに VACUUM FREEZE を行う前後で現在の WAL 位置を採取して、進行量を調べます。

```
（10 万行テーブルを用意）
$ psql db1
```

```
db1=# TRUNCATE t_freeze;
db1=# CHECKPOINT;
db1=# INSERT INTO t_freeze SELECT g, md5(g::text)
        FROM generate_series(1, 100000) g;
db1=# \q

（pgbench で txid_current() を実行してトランザクション ID を 1 万トランサクション進める）
$ echo "SELECT txid_current();" > txid_current.sql
$ pgbench -n -f txid_current.sql -c 1 -t 10000 db1
    《出力省略》
$ psql db1
db1=# SELECT pg_current_wal_lsn();
 pg_current_wal_lsn
--------------------
 0/5F0E5E78

（テーブル全体について凍結処理を実行させる）
db1=# VACUUM FREEZE t_freeze;
db1=# SELECT pg_current_wal_lsn();
 pg_current_wal_lsn
--------------------
 0/5F136658

（VACUUM 前後の WAL 位置から、WAL 出力量を計算）
db1=# SELECT '0/5F136658'::pg_lsn - '0/5F0E5E78'::pg_lsn;
 ?column?
----------
   329696
```

これを PostgreSQL15 と PostgreSQL16 で比較すると以下の結果となりました。PostgreSQL16 では WAL 出力量が PostgreSQL15 の 4 分の 1 程度で済んでいます。本検証では結果のバラつきはほとんど生じませんでした。




## 4.5.4. VACUUM リングバッファサイズ指定

PostgreSQL ではテーブルやインデックスのデータを読み書きするときにデータを共有バッファ上に載せて処理を行います。このとき VACUUM などテーブルを一括処理する類の処理については、共有バッファ内の限定されたサイズ（VACUUM であれば 256kB）の領域のみを使うようになっています。この領域をリングバッファと呼びます。これにより、一つの処理のために共有バッファのを特定のテーブルのデータで埋めてしまうことを防止しています。

PostgreSQL16 からは、VACUUM と ANALYZE におけるリングバッファのサイズを設定パラメータと VACUUM および ANALYZE コマンドのオプション BUFFER_USAGE_LIMIT で指定できるようになりました。指定は ANALYZE 処理にも適用されます。

設定パラメータの説明は以下の通りです。

| 設定パラメータ名 | 説明 |
|---|---|
| vacuum_buffer_usage_limit | VACUUM や ANALYZE におけるリングバッファのサイズを指定。<br>デフォルトは 256kB。128kB から 16GB が指定可能。<br>実効値としては shared_buffers の 8 分の 1 サイズが上限。 |

BUFFER_USAGE_LIMIT オプション使用例も示します。

**（VACUUM コマンドでの指定例）**

```
db1=# VACUUM (BUFFER_USAGE_LIMIT '1MB');
VACUUM
```

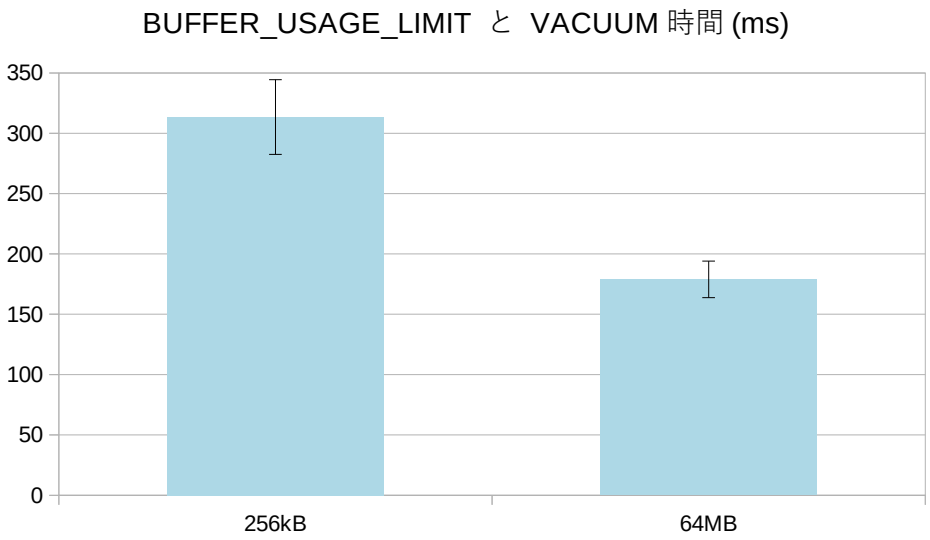
より大きなリングバッファサイズを指定することで、共有バッファを上書きしてしまう一方で、より高速に VACUUM や ANALYZE を実行することができます。PostgreSQL サービス起動直後の VACUUM ANALYZE では共有バッファは空ですので、大きなリングバッファサイズを指定するのは良い考えと言えます。

以下の手順で大きなリングバッファの性能効果を調べました。

```
（データ投入と削除でテーブルに不要領域を含むデータを作る）
db1=# CREATE TABLE t_ringbuffer (id int, val text);
db1=# INSERT INTO t_ringbuffer SELECT g, md5(g::text)
        FROM generate_series(1, 750000) g;
INSERT 0 750000
db1=# DELETE FROM t_ringbuffer WHERE id % 5 = 0;
DELETE 150000
db1=# \q

（PostgreSQL 再起動直後に VACUUM を実行する）
$ pg_ctl restart
$ psql db1
db1=# \timiming on          《← 所要時間計測》
db1=# VACUUM (BUFFER_USAGE_LIMIT '64MB') t_ringbuffer;
VACUUM                  《↑オプション有無の場合をそれぞれ実施》
Time: 176.855 ms
```

次のグラフは、shared_buffers = 1GB のときに、PostgreSQL 起動直後に 75 万行（物理サイズ 49MB） のテーブルを 15 万行削除した後に VACUUM するときの、デフォルトと BUFFER_USAGE_LIMIT '64MB' を指定した場合の所要時間です。PostgreSQL 再起動前に以下のように 75 万行の INSERT と 15 万件の DELETE を行っています。

（誤差範囲の線は標準偏差）




BUFFER_USAGE_LIMIT 64MB を指定した場合、デフォルト（256kB）の場合の 60% の時間で VACUUM が実行できました。

### 4.5.5. Meson ビルド

PostgreSQL のビルドは長らく Linux では Autoconf、Windows では Perl スクリプトによる MSVC ビルドシステムが利用されてきましたが、昨今ではいずれも古くなり開発におけるメンテナンスが難しいものとなってきました。そこで、PostgreSQL 16 からは近年広く使用されるようになったユーザフレンドリーな Meson ビルドシステムをサポートするようになりました。以下では「3. 検証のためのセットアップ」の章で記述している Linux の Autoconf によるビルドを「従来のビルド」として説明します。

#### ◆ *従来のビルドとの比較*

Meson ビルドのビルド・インストール手順は従来のビルドとよく似ています。従来のビルドでは ./configure → make → make install といったコマンドを順に実行していましたが、Meson ビルドでは meson setup → ninja → ninja install の順で実行します。

従来のビルドではソースディレクトリ上でビルドされていましたが、Meson ビルドではソースディレクトリの外部に出力するためのビルドディレクトリを指定するようになりました。これにより、同じソースディレクトリに対してビルド設定ごとにビルドを分けることが可能になりました。

従来のビルドでは並列ビルドを実行するために make に-j オプションで並列数を直接指定する必要があります。Meson ビルドではデフォルトで CPU 数を自動で検出し、それがビルドの並列数と指定されます。

従来のビルドの方法は当分の間サポートされますが、将来的には Meson ビルドのみに置き換えることが宣言されています。

以下は、従来のビルドと Meson ビルドの比較を表にまとめたものです。

| | 従来のビルド | Meson ビルド |
|---|---|---|
| ビルド・インストール手順 | ./configure<br>make<br>make install | meson setup<br>ninja<br>ninja install |
| ビルド設定ごとのビルドが可能か | 不可能 | ビルド設定ごとにビルド可能 |
| 並列ビルド | -j オプションで明示的に並列数を設定する必要がある | デフォルトで自動で検出した CPU 数を並列数とする |
| 将来性 | 当分の間サポート | 将来的に一本化 |

従来のビルド手順を実行した「3. 検証のためのセットアップ」と比較しつつ、具体的な Meson ビルドの手順を以下に示します。

### ◆ *ソフトウェア入手*

Meson ビルドは 3.1.で紹介しているディストリビューション tarball からではなく、Git チェックアウトからビルドする場合にのみ機能します。

Git チェックアウトの PostgreSQL 16 は以下 URL のページからダウンロード可能です。

```
https://github.com/postgres/postgres/releases/tag/REL_16_BETA2
```

### ◆ *検証環境*

Meson ビルドの検証環境は 3.2.で示されている環境と同様に、仮想化基盤上の RHEL 8.x (x86_64) 互換環境を使用します。

### ◆ *インストール*

gcc、zlib-devel、readline-devel、libicu-devel、openssl-devel、libzstd-devel、lz4-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、追加でビルドに必要なパッケージをインストールします。

このとき Python のバージョンは 3.7 以上である必要がありますが、この例では Python3.11 がインストールされるようにしています。

perl、 flex、 bison が必要であるのは、git リポジトリのソースコードをビルドするためです。tarball 配布物の場合には、これらツールによって生成されたファイルが始めから含まれていますので、これらツールは必須ではありませんでした。

```
# dnf install python3.11-pip
# pip3.11 install meson ninja
# dnf install perl flex bison
```

その後、3.3. 節と同様のオプション指定となるようにソースコードのビルドを行いました。事前に postgres ユーザで読み書き可能な /usr/local/pgsql/meson16 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 --no-check-certificate
https://github.com/postgres/postgres/archive/refs/tags/REL_16_BETA2.tar.gz
                                                          《実際は 1 行》
$ tar xf REL_16_BETA2.tar.gz
$ cd postgres-REL_16_BETA2
$ meson setup build --prefix=/usr/local/pgsql/meson16 --debug \
  -Dssl=openssl -Dzstd=enabled -Dlz4=enabled
                                                  《build はビルドディレクトリ》
The Meson build system
Version: 1.1.1
Source dir: /home/postgres/install/postgres-REL_16_BETA2
Build dir: /home/postgres/install/postgres-REL_16_BETA2/build
Build type: native build
Project name: postgresql
Project version: 16beta2
C compiler for the host machine: cc (gcc 8.5.0 "cc (GCC) 8.5.0 20210514 (Red
Hat 8.5.0-16)")
C linker for the host machine: cc ld.bfd 2.30-113
Host machine cpu family: x86_64
Host machine cpu: x86_64
Run-time dependency threads found: YES
Program perl found: YES (/usr/bin/perl)
Program python3 found: YES (/usr/bin/python3)
  《出力中略》          ↓  configure 同様の検査の後、ビルド時条件一式が出力される
  Data layout
    data block size         : 8 kB
    WAL block size          : 8 kB
```

```
    segment size            : 1 GB

  System
    host system             : linux x86_64
    build system            : linux x86_64

  Compiler
    linker                  : ld.bfd
    C compiler              : gcc 8.5.0
```

**《出力中略》**

```
    plperl                  : YES
    plpython                : YES 3.8
    pltcl                   : NO
    readline                : YES
    selinux                 : NO
    systemd                 : YES 239
    uuid                    : NO
    zlib                    : YES 1.2.11
    zstd                    : YES 1.4.4

  User defined options
    debug                   : True
    prefix                  : /usr/local/pgsql/meson16
    lz4                     : enabled
    ssl                     : openssl
    zstd                    : enabled

Found ninja-1.11.1.git.kitware.jobserver-1 at /usr/local/bin/ninja

$ cd build
$ ninja
[2316/2316] Linking target src/interfaces/ecpg/test/thread/alloc
```

**↑デフォルトでは１行を上書きする形で経過が出力される**

**`ninja -v` とすればコンソールに各ビルドコマンド結果が出力される**

```
$ ninja install
[0/1] Installing files.
Installing subdir
/data/pgsql/install/postgres-REL_16_BETA2/src/include/access to
/usr/local/pgsql/meson16/include/server/access
Installing
/data/pgsql/install/postgres-REL_16_BETA2/src/include/access/amapi.h to
/usr/local/pgsql/meson16/include/server/access
  《出力中略》            ↑↓１ファイル毎にインストール報告が出力される
Installing symlink pointing to libecpg_compat.so.3.16 to
/usr/local/pgsql/meson16/lib64/libecpg_compat.so.3
Installing symlink pointing to libecpg_compat.so.3 to
/usr/local/pgsql/meson16/lib64/libecpg_compat.so

$ ls /usr/local/pgsql/meson16/
bin  include  lib64  share
```

以上で、Meson によるビルドおよびインストールは完了です。従来の方法でインストールしたときと同様に PostgreSQL バイナリが利用可能になりました。

# 5. 非互換変更

PostgreSQL 16 にはいくつか非互換の変更点があります。本章ではそのうち主要なものを説明します。

### ◆ *PL/pgSQL のカーソル仕様変更*

PL/pgSQL 手続き言語ではカーソルを利用することができます。PL/pgSQL でカーソルを使うときにそのポータル名（PostgreSQL のカーソル名）に、これまでは PL/pgSQL におけるカーソル変数名をそのまま使っていました。そのため、同名のカーソル変数を同時にオープンしようとするとエラーが発生していました。

以下は PostgreSQL15 でのエラー発生例です。

```
（適当な SELECT 文にバインドされたカーソル変数 c1 を OPEN する関数 f_cur1、f_cur2 を定義）
db1=# CREATE FUNCTION f_cur1() RETURNS refcursor LANGUAGE plpgsql AS
      $$ DECLARE c1 CURSOR FOR SELECT tablename FROM pg_tables;
         BEGIN OPEN c1;  RETURN c1;  END; $$;
db1=# CREATE FUNCTION f_cur2() RETURNS refcursor LANGUAGE plpgsql AS
      $$ DECLARE c1 CURSOR FOR SELECT usename FROM pg_user;
         BEGIN OPEN c1;  RETURN c1;  END; $$;

（それら 2 つの関数を呼び出して、その時のカーソル状態を返す関数 f_cur3 を定義）
db1=# CREATE FUNCTION f_cur3() RETURNS SETOF pg_cursors LANGUAGE plpgsql AS
      $$ DECLARE c101 REFCURSOR;   c102 REFCURSOR;
         BEGIN  c101 := f_cur1();  c102 := f_cur2();
         RETURN QUERY EXECUTE 'SELECT * FROM pg_cursors'; END; $$;

（PostgreSQL 15 で実行すると同名カーソルのために失敗する）
db1=# SELECT name, statement FROM f_cur3();
ERROR:  cursor "c1" already in use
CONTEXT:  PL/pgSQL function f_cur2() line 1 at OPEN
PL/pgSQL function f_cur3() line 2 at assignment
```

上位の関数・プロシージャが、呼び出し先の各関数内で使われているカーソル名を把握していないと名前の衝突を防げませんので、この挙動は望ましくありません。

同じ処理を PostgreSQL16 で実行すると成功します。このとき、戻り値の name 列（すなわち pg_cursors ビューの name 列）で示されるのがカーソル名（ポータル名）です。「c1」というカーソル変数名とは無関係に自動的に命名されていることがわかります。

**（PostgreSQL 16 で同様に f_cur1、f_cur2、f_cur3 関数を定義して、f_cur3 関数を実行）**

```
db1=# SELECT name, statement FROM f_cur3();
       name         |            statement
--------------------+---------------------------------
 <unnamed portal 3> | SELECT tablename FROM pg_tables
 <unnamed portal 4> | SELECT usename FROM pg_user
(2 rows)
```

PostgreSQL15 で f_cur3 関数の二つ目の関数呼び出しをコメントアウトして実行してみます。

**（PostgreSQL 15 で f_cur3 関数を書き換えして実行）**

```
db1=# CREATE OR REPLACE FUNCTION f_cur3() RETURNS SETOF pg_cursors
      LANGUAGE plpgsql AS
      $$ DECLARE c101 REFCURSOR;  c102 REFCURSOR;
         BEGIN  c101 := f_cur1(); /*c102 := f_cur2();*/
         RETURN QUERY EXECUTE 'SELECT * FROM pg_cursors'; END; $$;

db1=# SELECT name, statement FROM f_cur3();
 name |            statement
------+---------------------------------
 c1   | SELECT tablename FROM pg_tables
(1 row)
```

name 列で示されるカーソルのポータル名が変数名と同じ「c1」になっていることが確認できました。

ポータル名の命名規則が変わることは非互換の変更と言えます。PL/pgSQL におけるカーソル変数名とポータル名が同じであることを前提にしているアプリケーションコードには修正が必要です。

PostgreSQL 16 では以下のように記述すると、カーソルのポータル名を明示的に指定できます。任意文字列が指定できますが、変数名と同じ名前にすれば、PostgreSQL 15 以前と同じ挙動を実現できます。

**（PostgreSQL 16 でカーソルのポータル名を明示的に指定する場合）**

```
db1=# CREATE OR REPLACE FUNCTION f_cur1() RETURNS refcursor
      LANGUAGE plpgsql AS
      $$ DECLARE c1 CURSOR FOR SELECT tablename FROM pg_tables;
         BEGIN c1 := 'portal-c1';  OPEN c1;  RETURN c1;  END; $$;

db1=# SELECT name, statement FROM f_cur3();
```

```
       name         |             statement
--------------------+---------------------------------
 <unnamed portal 1> | SELECT usename FROM pg_user
 portal-c1          | SELECT tablename FROM pg_tables
(2 rows)
```

### ◆ *主キーで NULLS NOT DISTINCT インデックスが禁止*

テーブルの主キーとして既存のユニークインデックスを使うように指定できますが、このとき、これまでは「NULLS NOT DISTINCT」指定のユニークインデックスでも使用可能でした。主キーでは NOT NULL 制約も必ず課せられますので、値が NULL の場合に関する指定は無意味です。このような主キーをダンプすると、リストア不能な誤ったダンプファイルが出力されてました。

PostgreSQL 16 ではこのようなインデックスを主キーに使うことが禁止されました。

### ◆ *REINDEX DATABASE 仕様変更*

REINDEX DATABASE コマンド（および reindexdb コマンド）の動作が変わり、システムテーブルのインデックスは再作成されなくなりました。PostgreSQL 16 で、システムテーブルのインデックスを再作成するには、REINDEX SYSTEM コマンド（reindexdb --system コマンド）を実行する必要があります。

定期実行されるバッチスクリプトで改修が必要となる可能性があります。

### ◆ *継承テーブル／パーティションテーブルにおける生成式の仕様変更*

PostgreSQL 16 では、継承テーブルまたはパーティションテーブルにおいて、親テーブルと子テーブルで同じ列について生成列であるかが必ず一致していなければならなくなりました。不一致の子テーブルを追加しようとすると、エラーになるか、自動的に親テーブルに揃えられます。

**（PostgreSQL 16 のパーティションテーブルの例 / 生成列が親子で不一致だとエラー）**

```
db1=# CREATE TABLE t_p1 (id int, c1 int, c2 int) PARTITION BY HASH (id);
db1=# CREATE TABLE t_c0 (id int, c1 int, c2 int);
db1=# CREATE TABLE t_c1 (id int, c1 int,
        c2 int GENERATED ALWAYS AS (c1 + 1) STORED);
db1=# ALTER TABLE t_p1 ATTACH PARTITION t_c0
        FOR VALUES WITH (MODULUS 2, REMAINDER 0);
db1=# ALTER TABLE t_p1 ATTACH PARTITION t_c1
        FOR VALUES WITH (MODULUS 2, REMAINDER 1);
ERROR:  column "c2" in child table must not be a generated column
```

なお、PostgreSQL 16 でも親子のテーブルで生成列の式が異なることは認められています。

**（PostgreSQL 16 の継承テーブルの例 ／ 親子で生成列の式は不一致でも良い）**

```
db1=# CREATE TABLE t_inh_p (id int, v text,
        g text GENERATED ALWAYS AS (v || 'X') STORED);
db1=# CREATE TABLE t_inh_c1 (id int, v text,
        g text GENERATED ALWAYS AS (v || 'YZ') STORED) INHERITS (t_inh_p);
db1=# INSERT INTO t_inh_p (id, v) VALUES (1, 'A');
db1=# INSERT INTO t_inh_c1 (id, v) VALUES (101, 'B');
db1=# SELECT * FROM t_inh_p;
 id  | v |  g
-----+---+-----
   1 | A | AX
 101 | B | BYZ
(2 rows)
```

### ◆ *pg_walinspect で関数廃止*

contrib 拡張モジュール pg_walinspect で、以下の関数が廃止されました。これらは end_lsn 引数をもたず、存在する WAL ファイルの最後までを処理対象とします。

- pg_get_wal_records_info_till_end_of_wal
- pg_get_wal_stats_till_end_of_wal

代替手段としては、pg_get_wal_records_info 関数や pg_get_wal_stats 関数の第二引数（end_lsn）に以下のように LSN の最大値 FFFFFFFF/FFFFFFFF を与えれば、同様の結果が得られます。

```
db1=# SELECT * FROM
        pg_get_wal_records_info('0/03000000', 'FFFFFFFF/FFFFFFFF');
```

なお、上記例の第一引数（start_lsn）は、pg_wal ディレクトリにある最も小さい番号のファイル名を持った WAL ファイルが「000000010000000000000003」であったので、その WAL ファイルの最初を意味するように、9 桁目から 16 桁目までの値「0」をスラッシュの手前に、末尾 2 桁の「03」をスラッシュの後に、その後に「000000」を置いて、LSN 文字列「0/03000000」を作って指定したという想定です。

### ◆ *CREATE RULE によるビュー作成の廃止*

CREATE RULE コマンドを使ってビューを作る方法が廃止されました。

PostgreSQL 15 までは以下のようにしてビュー定義が可能でした。

**（PostgreSQL 15.x 以前で可能であった CREATE RULE によるビュー定義）**

```
db1=# CREATE TABLE t_ruletest (id int PRIMARY KEY, c1 int, c2 int);
db1=# CREATE TABLE v_ruletest AS SELECT * FROM t_ruletest;
db1=# CREATE RULE "_RETURN" AS ON SELECT TO v_ruletest
        DO INSTEAD SELECT * FROM t_ruletest;   → ここでルール名は必ず "_RETURN"
db1=# \dv
          List of relations
 Schema |     Name     | Type |  Owner
--------+--------------+------+----------
 public | v_ruletest | view | postgres          → v_ruletest はビューに変わる
(1 row)
```

この構文が必須となる機会は無いと考えられます。

### ◆ *設定パラメータ変更*

PostgreSQL 16 で以下表に示す既存の設定パラメータが変更されました。

| パラメータ名 | 説明 |
|---|---|
| force_parallel_mode | debug_parallel_query にパラメータ名が変更されました。 |
| vacuum_defer_cleanup_age | 廃止されました。<br>プライマリにおける VACUUM 処理がレプリケーションされることでスタンバイ側の参照問い合わせと衝突が生じるのを回避するには、代替に hot_standby_feedback をスタンバイ側で on にしたり、レプリケーションスロットを使ったりすることが考えられます。 |
| promote_trigger_file | 廃止されました。<br>トリガファイルを置いてスタンバイサーバを昇格する機能自体が廃止されました。代替に pg_ctl promote コマンドや pg_promote 関数を使う必要があります。 |

| パラメータ名 | 説明 |
| --- | --- |
| lc_collate<br>lc_ctype | 両パラメータとも、廃止されました。<br>これらは参照専用のパラメータで、それぞれ現在接続しているデータベースの Collate と Ctype を返すものでした。<br>同じ情報は以下の SQL で取得できます。<br>`SELECT datcollate collate, datctype ctype FROM pg_database WHERE datname = current_database();` |

### ◆ *postmaster* コマンド廃止

postmaster コマンドが廃止されました。

PostgreSQL 8.2.x の時点から、postgres の別名であり廃止予定である旨がマニュアルに記載されていました。実体としては単なるシンボリックリンクでした。

postgres ではなく、postmaster として起動していた場合の唯一の違いは OS のプロセス一覧に現れるコマンドライン文字列に postmaster が現れることです。これまで postmaster コマンドを使っていて、OS のプロセス一覧のコマンドライン文字列を使って監視していた場合には、何らか手当が必要です。

### ◆ *ロールの CREATEUSER 属性の変更*

ロールに CREATEUSER 属性を付与した場合に、そのロールでできる操作が従来よりも限定されるように変更されました。違いを以下表に示します。

| | バージョン 15 まで | バージョン 16 から |
| --- | --- | --- |
| 属性の付与、変更（CREATE ROLE、ALTER ROLE） | BYPASSRLS、REPLICATION 、SUPERUSER はスーパーユーザ限定、それ以外は任意に可能 | 自身がその属性を持っていて、変更するロールの ADMIN OPTION 付きメンバーである場合に可能 |
| ロールメンバーシップの付与／剥奪（GRANT／REVOKE） | SUPERUSER 属性ロール以外は任意にメンバーシップ付与/剥奪が可能 | 自身がメンバーシップ付与するロールの ADMIN OPTION 付きメンバーである場合に可能 |

また、psql で ADMIN OPTION 付きでロールメンバーシップが付与されたことを確認できる \drg コマンドが追加されました。以下にコマンド実行時の出力例を示します。

```
db1=> \drg
                List of role grants
 Role name | Member of |       Options        | Grantor
-----------+-----------+----------------------+----------
 bar       | role1     | ADMIN, INHERIT, SET | postgres
 foo       | role1     | INHERIT, SET        | bar
 foo       | role2     | ADMIN, INHERIT, SET | postgres
(3 rows)
```

# 6. 免責事項

本ドキュメントは SRA OSS LLC により作成されました。しかし、SRA OSS LLC 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。