PostgreSQL 17 検証レポート

1.5 版
2024 年 10 月 21 日

株式会社 SRA OSS
〒170-0022 東京都豊島区南池袋 2-32-8
Tel. 03-5979-2701 Fax. 03-5979-2702
https://www.sraoss.co.jp/

目次

1. はじめに

本文書は PostgreSQL 17 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 17 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。
2024 年 6 月 27 日にリリースされた PostgreSQL 17 beta2 を使用して検証を行いました。その後、2024 年 8 月 8 日にリリースされた beta3 に対応して、いくつか記載を修正しています。

2. 概要

PostgreSQL 17 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
- WAL 排他処理の改良
- VACUUM 性能改善
- COPY 性能向上
- ストリーム I/O
- CTE プランの改善
- インデックス IN 句検索の改善

SQL 機能
- SQL/JSON と jsonpath の機能追加
- MERGE 文の拡張
- COPY FROM に ON_ERROR オプション追加
- EXPLAIN 文の拡張
- 接続認証のイベントトリガ
- プラットフォーム非依存の照合順序

ロジカルレプリケーション
- pg_upgrade 対応の改善
- フェイルオーバ制御
- pg_createsubscriber コマンド追加

パーティショニング
- 分割とマージ　**（※ 本機能は beta 版に含まれましたが、その後、取り下げられました）**
- 排他制約に対応
- IDENTITY 列に対応

クライアント機能

- libpq で非同期処理とクエリキャンセルが改善
- libpq で PQchangePassword API 追加

運用管理

- インクリメンタルバックアップ
- pg_combinebackup コマンド
- pg_dump の --filter オプション
- 新たなモニタリングビュー pg_stat_checkpointer、pg_wait_events
- 新たな定義済ロール pg_maintain

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 17 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/17/release-17.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 17（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っています。

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/17 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/v17beta2/postgresql-
17beta2.tar.bz2
                        《実際は 1 行、リリース後は「v17.0/postgresql-17.0.tar.bz2」》
$ tar jxf postgresql-17beta2.tar.bz2
$ cd postgresql-17beta2
$ ./configure --prefix=/usr/local/pgsql/17 --enable-debug \
  --with-openssl --with-icu --with-zstd --with-lz4
$ make world-bin
$ make install-world-bin
```

上記手順はドキュメントのビルド・インストールを省略しています。ドキュメントもビルド・インストールする場合には、docbook-dtds や docbook-style-xsl などのパッケージが OS にインストールされた状態で「make world」および「make install-world」を実行してください。

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 > ~/pg17.env <<'EOF'
VER=17
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
$ . ~/pg17.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 (17beta2)
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。インストール時にドキュメントもビルド・インストールした場合、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/17/share/doc/html/
```

また、以下 URL にて PostgreSQL 17 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/17/
```

4.1. 性能向上

4.1.1. WAL 排他処理の改良

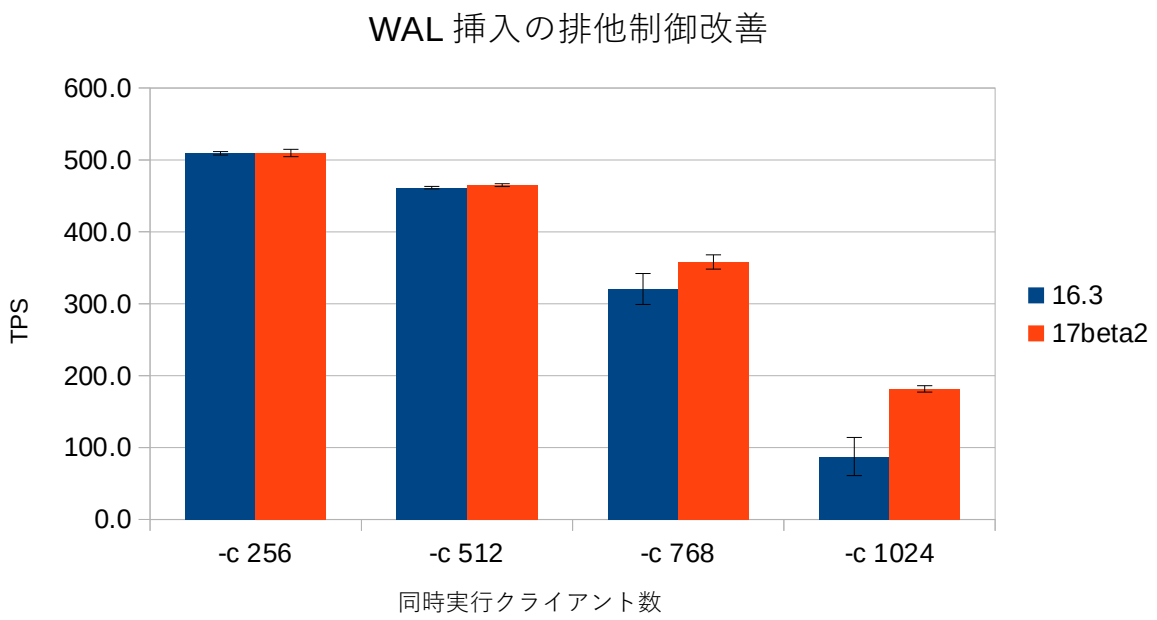
本バージョンで WAL を書き出すときの排他制御が効率化されました。64bit 値のアトミックな交換をサポートする CPU で、多数の書き込みトランザクションの同時実行において性能改善が期待できます。

2 vCPU のサーバにて、以下のように PostgreSQL 16.3 と 17beta2 で pgbench でデフォルトシナリオをいくつかの同時実行数で計測しました。

```
（以下を設定して反映）
$ vi $PGDATA/postgresql.conf
   max_connections = 1050
   shared_buffers = 512MB
   checkpoint_timeout = 30min
$ pg_ctl restart

（各試行ごとに以下を実行）
$ NCON=1024
$ pgbench -n -i -s 10 -F 70 db1
$ psql -c "VACUUM (BUFFER_USAGE_LIMIT '512MB')" db1
$ psql -c checkpoint
$ pgbench -n -c $NCON -T 300 db1
```

負荷のバラつきと WAL 以外の処理負荷を下げる狙いで、-F 70（fillfactor=70）として UPDATE 負荷を軽減し、VACUUM 時にデータを共有バッファに載せるようにして、さらにチェックポイントタイミングを揃えています。

結果は以下の通りです。誤差線は上下に標準偏差の長さをとって記載しています。



17beta2 の方が同時実行クライアント数の増加による性能ダウンが緩やかになっていて、1024 同時接続の時に大幅な差異があらわれました。一方で 256 同時接続と 512 同時接続までは、ほとんど性能向上が見られませんでした。

4.1.2. VACUUM 性能改善

16.x までのバージョンでは（FULL でない) VACUUM 処理におけるメモリの使い方について課題がありました。実装上の理由で maintenance_work_mem 設定により大きな値を設定したとしても、最大 1GB までしかメモリが使われず、そのために処理すべきデッドタプルを覚えておくメモリ領域が不足すると本来不要なテーブルの再走査が発生していました。さらに、見つけたデッドタプルを格納するデータ構造も単純な配列データであるため、件数が多い場合の動作は効率的とはいえませんでした。

17 バージョンでこれらの課題の解決を含む VACUUM 実装の以下の変更が導入されました。

- 発見したデッドタプルの記憶用に 新たに Adaptive Radix Tree データ構造を導入し、使用可能メモリサイズの上限も撤廃
- デッドタプルの整理とタプル凍結を一括で行うようにして、WAL 出力量も軽減
- インデックスの無いテーブルの VACUUM を効率化し、WAL 出力量を軽減

以下の手順で PostgreSQL 16.x と 17beta2 とで VACUUM 処理性能を比較しました。

（以下を設定、その他のパラメータは初期値とする）

```
$ vi $PGDATA/postgresql.conf
```

```
   autovacuum = off
   maintenance_work_mem = 2GB
$ pg_ctl restart

（2億行のテストデータを作成して、全件を 5%、50%、95%を UPDATE する）
$ psql db1
db1=# CREATE TABLE t412 (id int, v text);
db1=# CREATE INDEX ON t412 (id);
          《 インデックス無しの場合には省略する》
db1=# INSERT INTO t412 SELECT g, 'x' FROM generate_series(1, 200000000) g;
db1=# UPDATE t412 SET v = 'y' WHERE id % 20 != 1;
          《 上記は 95%の場合、5%なら「id % 20 = 1」、50%なら「id % 2 = 1」》

（VACUUM 所要時間と WAL 出力量を計測）
db1=# VACUUM VERBOSE t412;
INFO:  vacuuming "db1.public.t412"
INFO:  finished vacuuming "db1.public.t412": index scans: 0
pages: 0 removed, 1769912 remain, 1769912 scanned (100.00% of total)
tuples: 200000000 removed, 200000000 remain, 0 are dead but not yet
removable
removable cutoff: 3405970, which was 0 XIDs old when operation ended
new relfrozenxid: 3405969, which is 2 XIDs ahead of previous value
frozen: 0 pages from table (0.00% of total) had 0 tuples frozen
index scan not needed: 0 pages from table (0.00% of total) had 0 dead item
identifiers removed
avg read rate: 39.208 MB/s, avg write rate: 39.418 MB/s
buffer usage: 1785656 hits, 1754716 misses, 1764125 dirtied
WAL usage: 2654869 records, 879288 full page images, 179001058 bytes
system usage: CPU: user: 85.18 s, system: 38.69 s, elapsed: 349.63 s
INFO:  vacuuming "db1.pg_toast.pg_toast_17239"
  《後略》
```

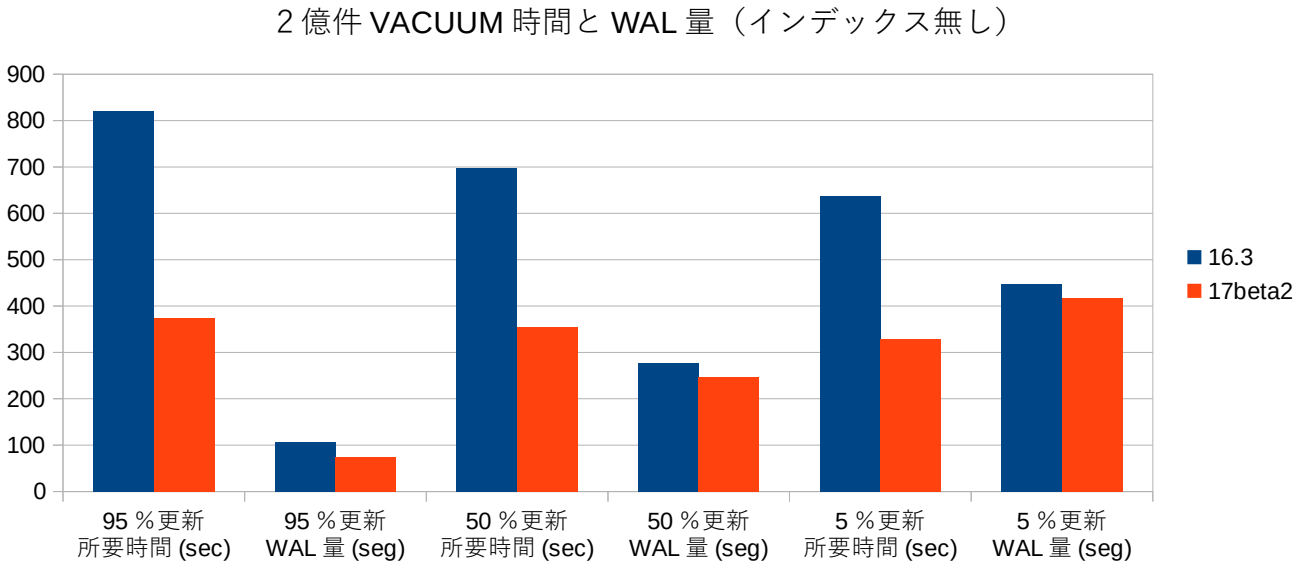
VACUUM VERBOSE の報告から VACUUM の所要時間および WAL 出力量がわかります。他に TOAST テーブルに関する WAL 量と所要時間がありますが、本テーブルに TOAST 格納される長さの列はありませんので、これはごく僅かです。

これに加えて以下のコマンドを実行して、プロセスのプライベート使用メモリ量を調査しました。OS で HugePages を利用可能でない場合には下記の書き方では共有バッファサイズも混じってしまう可能性があります。その場合には、smaps ファイルから抽出して集計する条件を更に調整する必要があります。

```
$ P=《VACUUM 実行セッションの PID》
$ while true ; do sleep 30; echo -n "private mem(kB): " ; \
  cat /proc/$P/smaps | grep Private | sed -e 's/^.*://' -e 's/kB//' | \
  awk '{ sum += $1 } ; END { print sum }'; done
private mem(kB): 4384
private mem(kB): 6164
private mem(kB): 7976
  《後略》
```

バージョン 16.3 と 17beta2 での実行結果は以下の通りです。本テストの計測値はほとんどバラつきませんでしたので、各条件で 1 回の結果を使用しています。グラフの WAL 量は、同じスケールの縦軸目盛を使うために、セグメント数単位（1 セグメント＝16MB）で記載しています。

以下はインデックス無しの場合です。

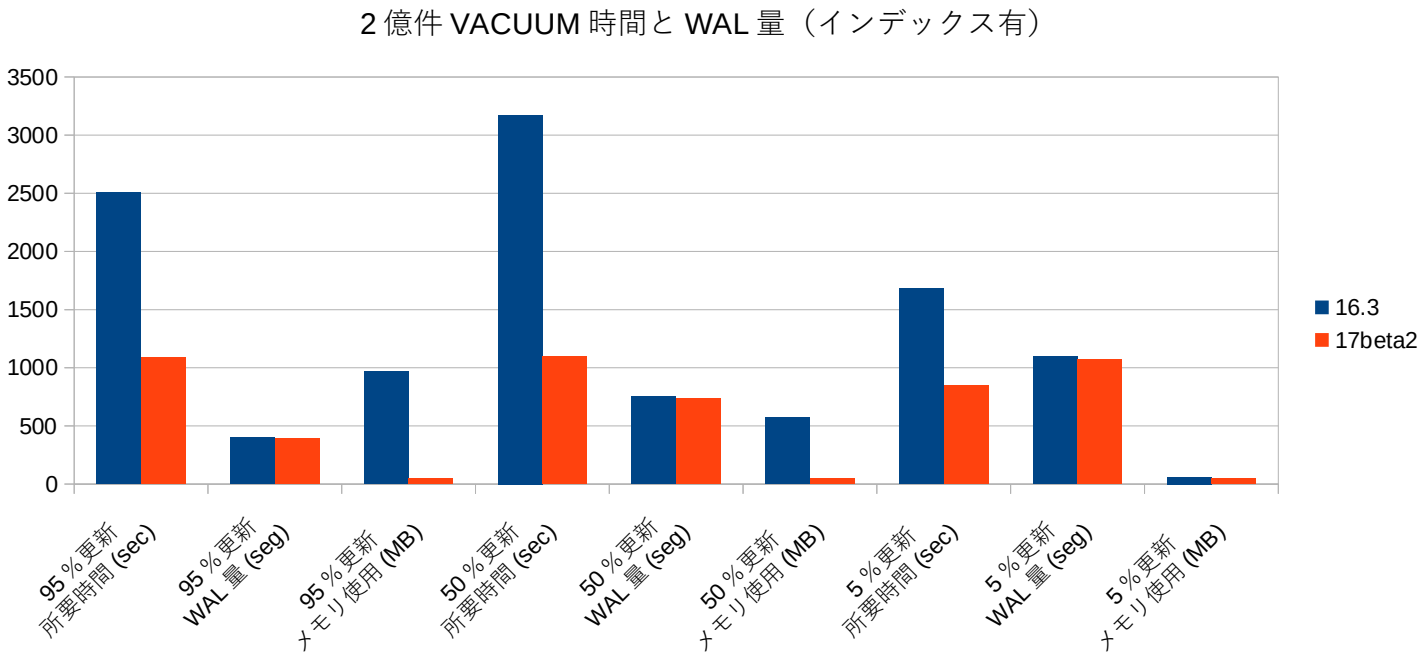


インデックス無しの場合、17beta2 の VACUUM は 95%、50%、5%のいずれの更新割合でも所要時間で半分程度となりました。WAL 量は、データの 95%を更新した後の場合に 30%程度の減少で、データの 5%のみを更新した後の場合には 7%程度の減少でした。

VACUUM 中のメモリ使用量は、いずれの場合も 4 MB 程度で大差無いため記載していません。インデック

スが無い場合の VACUUM ではメモリの大量使用は発生しないことが知られています。

以下はインデックス有（int 型列に btree インデックス 1 つ）の場合です。



インデックス有の場合でも、いずれのデータ更新割合においても 17beta2 で所要時間が半分以下になっています。メモリ使用量は、16.3 では 95%更新において 1GB 近くを使用しており、1GB の壁に当たっていることがわかります。17beta2 では使用メモリ量が大幅に減っていて、いずれも 45MB 程度です。WAL 出力量は 16.3 と 17beta2 でほとんど差がありませんでした。

いずれの性能改善も 17beta2 の VACUUM 性能改善の改修の効果と考えられます。今回の検証ケースでは 17beta2 で性能劣化するケースはありませんでした。

なお、両バージョンで更新した割合が低いほど VACUUM 時の WAL 量が多い傾向は奇妙ですが、これは 、INSERT 後の初回アクセスに伴う内部更新が VACUUM 時に生じる度合いが多いためと考えられます。

本項で検証していないケースとしては、更新行が均等に発生するのではなく特定物理ページに偏在している場合があります。ただし、このような場合には従来から、テーブルパーティショニングを用いて更新の多いパーティションだけを VACUUM することで負担を減らすことが可能でした。

4.1.3. COPY 性能向上

PostgreSQL 17 ではサーバ・クライアント間の通信で不要な内部データコピーを無くす改善が適用されました。これは特に大きなデータブロックをクライアントに送出する場合に効果が期待できます。

PostgreSQL 16.3 と 17beta3 とで、1 行のサイズが大きめのテーブルのデータを psql の \COPY コマンド

でエクスポートを行って所要時間を比較しました。

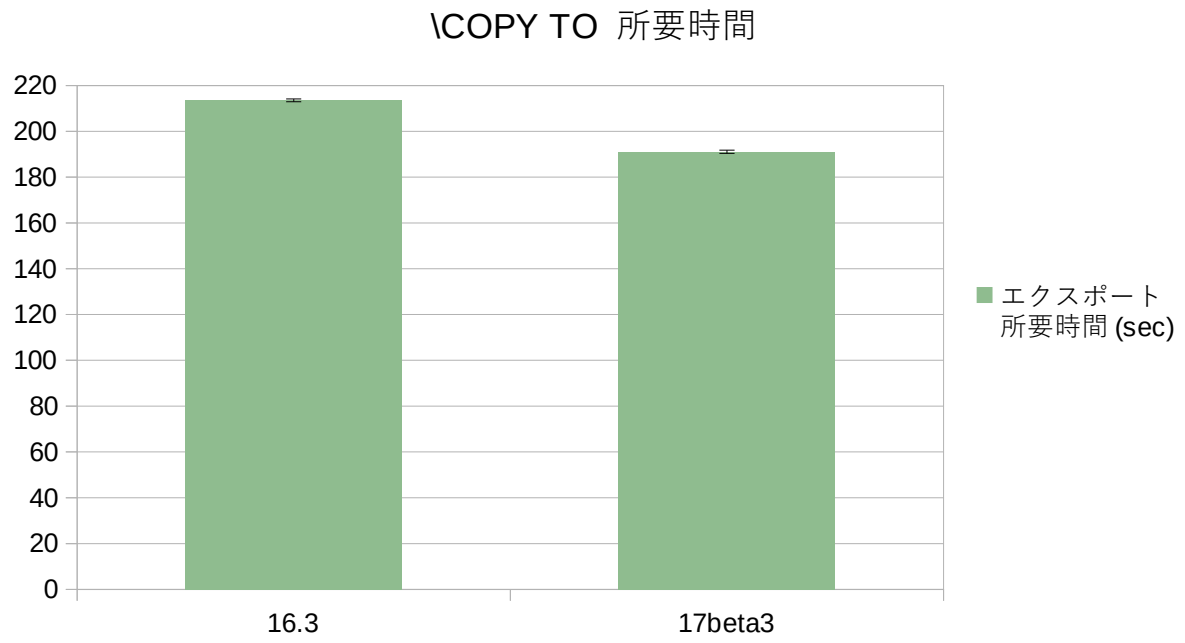
以下にコマンド実行手順を示します。

```
（以下を設定、その他のパラメータは初期値とする）
$ vi $PGDATA/postgresql.conf
   shared_buffers = 1024MB
   autovacuum = off
$ pg_ctl restart

（テーブルと 100 万行のデータを作成）
$ psql db1 <<EOS
CREATE UNLOGGED TABLE t413 (
  id int PRIMARY KEY, c1 text, c2 text, c3 text, c4 text, c5 text, c6 text,
  c7 text, c8 text, c9 text, ts1 timestamp);
INSERT INTO t413 SELECT g, repeat(md5('1' || g), 100),
  repeat(md5('2' || g), 100), repeat(md5('3' || g), 100),
  repeat(md5('4' || g), 100), repeat(md5('5' || g), 100),
  repeat(md5('6' || g), 100), repeat(md5('7' || g), 100),
  repeat(md5('8' || g), 100), repeat(md5('9' || g), 100),  now()
  FROM generate_series(1, 1000000) g;
EOS

（pg_prewarm で行データを共有バッファに載せる）
$ psql db1 <<EOS
CREATE EXTENSION pg_prewarm;
SELECT pg_prewarm('t413');
EOS
CREATE EXTENSION
 pg_prewarm
------------
    100000
(1 row)

（psql で \COPY TO を実行、出力はクライアント側で /dev/null に読み捨て）
$ time psql -h localhost -d db1 > /dev/null <<EOS
\copy t413 TO STDOUT
```

```
EOS


real    3m10.908s
user    0m15.634s
sys     0m50.599s
```

テーブルは 1 行あたりのデータ量を大きくしていて、1 行あたり内部表現で 750 バイト程度、COPY 出力行サイズで 29KB ほどになります。テーブル全体の物理サイズは 800MB 程度になります。ストレージ I/O の負荷をできるだけ除外するため、UNLOGGED テーブルを使い、事前に共有バッファに行データを載せて、さらに、自動 VACUUM を無効化しています。time コマンドでコマンドの所要時間を採取しました。

以下の結果が得られました。



エクスポート時間で 10%程度、所要時間が減っています。

本検証では対象テーブルは共有バッファに載っているところから開始しましたが、次節で述べるストリーム I/O 対応により、バッファに載っていないデータの読み取りの性能向上も行われているため、COPY 処理の総合的な性能向上も期待できます。

4.1.4. ストリーム I/O

PostgreSQL 17 ではストリーム I/O を利用したバッファ読み込みの内部インターフェイスが導入されました。いくつかの連続する複数ブロックの読み取りを 1 つのシステムコールで行い、それに続くブロックについ

てもカーネルに先読みを促します。

このときに一度に扱う最大サイズを新たな設定パラメータ io_combine_limit（デフォルト 128 kB）で指定します。128 kB は RHEL 7、8、9 (x86_64) の XFS ファイルシステムにおける最大先読みサイズですので、一般的な Linux サーバであればデフォルト設定のままで良いでしょう。

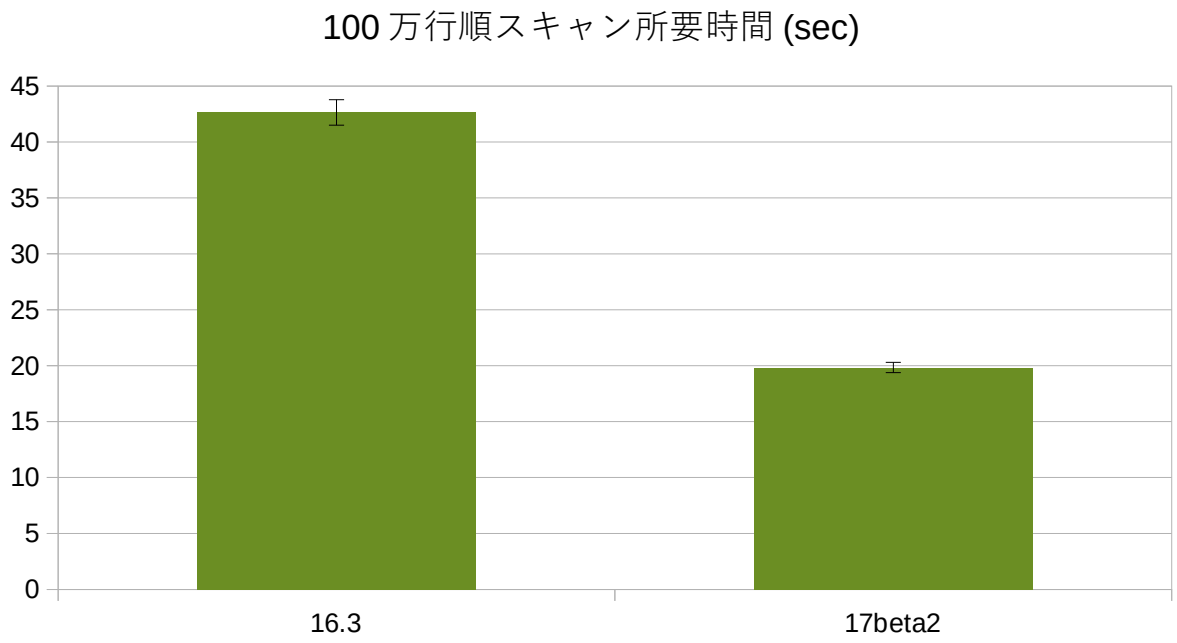
ストリーム I/O はシーケンシャルスキャンと ANALYZE で使用されます。

前節で作ったテーブル t413 を使って、以下の手順でシーケンシャルスキャンの所要時間を計測しました。

```
（root ユーザで以下コマンドを実行して、カーネルキャッシュを空にする）
# sync
# echo 3 > /proc/sys/vm/drop_caches
# su - postgres

（PostgreSQL はここで起動して、起動直後の共有バッファが空の状態で SELECT 実行）
$ . ~/pg17.env
$ pg_ctl start
$ psql db1 <<EOS
\timing on
SELECT count(1) FROM t413;
EOS
  count
---------
 1000000
(1 row)

Time: 19655.307 ms (00:19.655)

（１回実行するごとに PostgreSQL を停止して root ユーザに戻る）
$ pg_ctl stop
$ exit
#
```

t413 テーブルは、行数 100 万行、物理サイズは 2GB 弱です。データベースクラスタは XFS ファイルシステム上に配置しています。

PostgreSQL 16.3 と 17beta2 で実行した結果は以下の通りです。



17beta2 が 16.3 の半分の所要時間で実行で完了する結果となりました。

実行プランはどちらのバージョンも同一で以下の形です。3 並列 Parallel Seq Scan したデータを Partial Aggregate - Gather - Finalize Aggregate で集約したものです。性能改善は、ストレージ読み込み処理が効率化して、Seq Scan または Parallel Seq Scan の処理速度が上がったために生じたものと言えます。

（実行プランを確認 - 下記は 17beta2 での結果だが 16.3 でも同形）

```
db1(5432)=# explain SELECT count(*) FROM t413;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Finalize Aggregate  (cost=256224.88..256224.89 rows=1 width=8)
  -> Gather  (cost=256224.67..256224.88 rows=2 width=8)
     Workers Planned: 2
      -> Partial Aggregate  (cost=255224.67..255224.68 rows=1 width=8)
          -> Parallel Seq Scan on t413  (cost=0.00..254182.93 rows=416693 width=0)
(5 rows)
```

4.1.5. IN 句インデックス検索の改善

Btree インデックスを使っていくつかの値を探すときに効率的に動作できるようになりました。

典型的には「SELECT ... FROM ... WHERE c1 IN (...) AND c2 IN (...) 」のような IN 句を使った問い合わせで性能改善が見込まれます。

以下の手順で PostgreSQL 16.3 と 17beta2 の動作を比較しました。IN 句は = ANY (配列) と等価なので、ここでは = ANY (配列) を使用しています。

（テーブルとデータを作成 - 10 万件テーブル t415）

```
db1=# CREATE TABLE t415 (id int, subid int, v text,
        PRIMARY KEY (id, subid));
db1=# INSERT INTO t415 SELECT g/1000, g%1000, md5(g::text)
        FROM generate_series(1, 100000) g;
db1=# VACUUM ANALYZE t415;
```

（複合インデックス対象の列への 2 つの = ANY （配列） を含む問い合わせを実行してプラン確認）

```
db1=# \set ids '{81,84,79,40,31}'      --《これらの値は事前にランダム生成しておく》
db1=# \set subids '{659,197,370,495,251}'
db1=# explain analyze SELECT * FROM t415
        WHERE id = ANY ( :'ids' ) AND subid = ANY ( :'subids' )
        ORDER BY id DESC, subid DESC;
```

（16.3 の結果）

```
                                QUERY PLAN
--------------------------------------------------------------------------
 Sort  (cost=150.39..150.46 rows=25 width=41)
       (actual time=0.870..0.875 rows=25 loops=1)
   Sort Key: id DESC, subid DESC
   Sort Method: quicksort  Memory: 26kB
   ->  Index Scan using t415_pkey on t415
       (cost=0.29..149.81 rows=25 width=41)
       (actual time=0.165..0.841 rows=25 loops=1)
         Index Cond: ((id = ANY ('{81,84,79,40,31}'::integer[]))
                AND (subid = ANY ('{659,197,370,495,251}'::integer[])))
 Planning Time: 0.183 ms
 Execution Time: 0.914 ms
(7 rows)
```

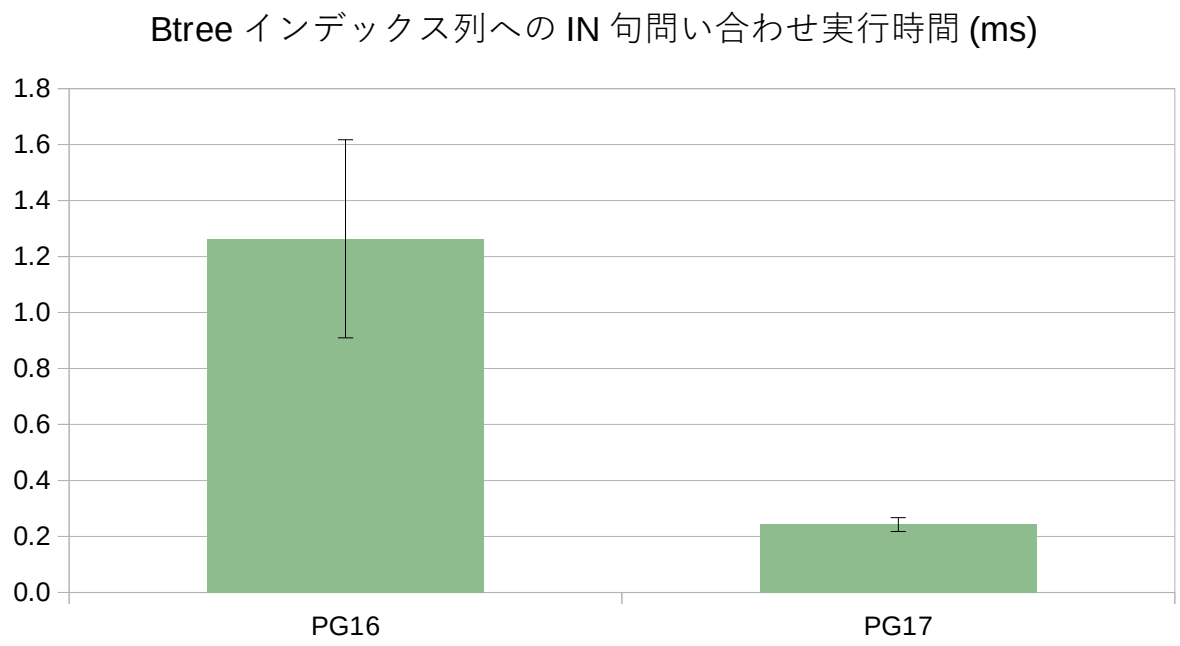
（17beta2 の結果）

```
                                QUERY PLAN
--------------------------------------------------------------------------
 Index Scan Backward using t415_pkey on t415
```

```
   (cost=0.29..149.81 rows=25 width=41)
   (actual time=0.036..0.225 rows=25 loops=1)
   Index Cond: ((id = ANY ('{81,84,79,40,31}'::integer[]))
            AND (subid = ANY ('{659,197,370,495,251}'::integer[])))
 Planning Time: 0.120 ms
 Execution Time: 0.247 ms
(4 rows)
```

両バージョンとも Index Scan を使った実行プランですが、17beta3 では ORDER BY で要求された順にインデックス走査することで Index Scan 後のソート処理が省略されています。また、Index Scan 自体にかかる時間も短くなっています。これは、重複したインデックスの読み取りを回避する改善が適用された結果です。

検索条件にあたえる値を様々にランダム生成して、同じ値を 16.3 と 17beta3 に与えて実行時間を測った結果が以下のグラフです。実行時間は EXPLAIN ANALYZE コマンドの Execution Time から採っています。



17beta2 の方が 5 倍ほど速い結果となっています。また、16.3 で所要時間のバラつきも大きくなっています。16.3 では Sort よりも Index Scan において与えられた条件によって所要時間が変動していました。

なお、ランダム値の生成は以下の SQL で実行しました。

```
SELECT array_agg(g) ids FROM
  (SELECT g FROM generate_series(0, 100) g ORDER BY random() LIMIT 5) s1 \gset
\echo '\\set ids' :'ids'
```

```
SELECT array_agg(g) subids FROM
  (SELECT g FROM generate_series(0, 999) g ORDER BY random() LIMIT 5) s1 \gset
\echo '\\set subids' :'subids'
```

4.1.6. 各種プランナの改善

PostgreSQL 17 では実行プラン作成に関する改善がいくつか適用されました。ここでは主要な改善点を取り上げます。記載したもの以外にも、パーティションへの LIMIT 句適用や MergeAppend プラン要素利用の改善、NULL 検査の省略など、幾つか変更点があります。

◆CTE プランの改善

CTE プラン要素の使い方が改善されて、統計情報と出力列のソート順序が考慮されるようになりました。以下の手順で PostgreSQL 16.3 と 17beta2 を比較しました。

```
（テーブルとデータを作成 - 2 つの 10 万件テーブル t416a と t416b）
db1=# CREATE TABLE t416a (id int PRIMARY KEY, v text);
db1=# INSERT INTO t416a
        SELECT g, md5(g::text) FROM generate_series(1, 100000) g;
db1=# CREATE TABLE t416b (id int PRIMARY KEY, v text);
db1=# INSERT INTO t416b SELECT * FROM t416a;
db1=# VACUUM ANALYZE t416a, t416b;

（実行プラン単純化して見やすくするためパラレル実行は無効化しておく）
db1=# SET max_parallel_workers_per_gather TO 0;

（CTE プラン要素が現れる SQL を実行 - 16.3 での実行結果）
db1=# EXPLAIN ANALYZE
        WITH cte1 AS MATERIALIZED (SELECT id FROM t416b ORDER BY id)
        SELECT count(*) FROM t416a WHERE id IN
          (SELECT id FROM cte1 ORDER BY id);
                                QUERY PLAN
-------------------------------------------------------------------------------
 Aggregate  (cost=14088.20..14088.21 rows=1 width=8)
       (actual time=818.386..818.391 rows=1 loops=1)
   CTE cte1
     ->  Index Only Scan using t416b_pkey on t416b
```

```
          (cost=0.29..2604.29 rows=100000 width=4)
          (actual time=0.156..38.290 rows=100000 loops=1)
        Heap Fetches: 0
  ->  Nested Loop  (cost=10805.11..11358.90 rows=50000 width=0)
                   (actual time=242.701..803.146 rows=100000 loops=1)
      ->  HashAggregate  (cost=10804.82..10806.82 rows=200 width=4)
                         (actual time=242.454..328.386 rows=100000 loops=1)
            Group Key: cte1.id
            Batches: 5  Memory Usage: 10305kB  Disk Usage: 200kB
            ->  Sort  (cost=10304.82..10554.82 rows=100000 width=4)
                      (actual time=126.534..141.190 rows=100000 loops=1)
                  Sort Key: cte1.id
                  Sort Method: quicksort  Memory: 3073kB
                  ->  CTE Scan on cte1
                     (cost=0.00..2000.00 rows=100000 width=4)
                     (actual time=0.162..106.685 rows=100000 loops=1)
      ->  Index Only Scan using t416a_pkey on t416a
          (cost=0.29..3.25 rows=1 width=4)
          (actual time=0.004..0.004 rows=1 loops=100000)
            Index Cond: (id = cte1.id)
            Heap Fetches: 0
 Planning Time: 1.505 ms
 Execution Time: 822.361 ms
(17 rows)
```

PostgresQL 16.3 で実行すると、幾つかの点で非効率な実行プランが生じました。

CTE プラン要素内で主キーインデックスの Index Only Scan を使ってデータ取得をしているのに、それを Sort しています。インデックス順なのでソート済みであるということが活かされていません。さらに、それを HashAggregate で集約しています。既にユニークであることが活かされていません。

続いて、17beta2 での結果を確認します。

（CTE プラン要素が現れる SQL を実行 - 17beta2 での実行結果）

```
db1=# EXPLAIN ANALYZE
        WITH cte1 AS MATERIALIZED (SELECT id FROM t416b ORDER BY id)
        SELECT count(*) FROM t416a WHERE id IN
          (SELECT id FROM cte1 ORDER BY id);
```

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Aggregate  (cost=8958.36..8958.37 rows=1 width=8)
            (actual time=223.932..223.936 rows=1 loops=1)
   CTE cte1
     ->  Index Only Scan using t416b_pkey on t416b
         (cost=0.29..2604.29 rows=100000 width=4)
         (actual time=0.139..35.407 rows=100000 loops=1)
           Heap Fetches: 0
   ->  Merge Semi Join  (cost=0.52..6104.07 rows=100000 width=0)
                        (actual time=0.179..209.987 rows=100000 loops=1)
         Merge Cond: (t416a.id = cte1.id)
         ->  Index Only Scan using t416a_pkey on t416a
             (cost=0.29..2604.29 rows=100000 width=4)
             (actual time=0.030..35.666 rows=100000 loops=1)
               Heap Fetches: 0
         ->  CTE Scan on cte1  (cost=0.00..2000.00 rows=100000 width=4)
                               (actual time=0.142..100.270 rows=100000 loops=1)
 Planning Time: 1.495 ms
 Execution Time: 225.715 ms
(11 rows)
```

PostgreSQL 17beta2 での実行プランでは、16.3 で非効率だった点が改善されています。コストが 14088 から 8958 に、実行時間も 822ms から 226ms と、大幅に短くなりました。

CTE プラン要素をスキャンした結果がソート済かつユニークであることが認識されていて、余計なソート処理や集約処理が無くなっています。また、データがソート済であることを使って、ソート済要素を子要素に取ることができる Merge Semi Join が選択されています。

なお、本問い合わせ例は WITH 句の MATERIALIZED 指定を外せば、CTE プラン要素を使わない実行プランになって、17beta2 でも 16.3 でも同じ程度のコスト・所要時間になります。このプランナ改善はあくまで CTE プラン要素が使われる場合における改善です。

◆ *相関 IN 句サブクエリの改善*

IN 句サブクエリが外側のテーブルを参照している相関サブクエリになっている場合に、これまでは 必ず Subplan プラン要素を使った実行プランになっていたものが、テーブル結合の実行プランに展開できるようになりました。

以下に例を示します。

（テーブルとデータ作成）

```
db1=# CREATE TABLE t416c (id int PRIMARY KEY, c1 int, c2 int);
db1=# INSERT INTO t416c
        SELECT g, g % 100, g % 2 FROM generate_series(1, 10000) g;
db1=# CREATE TABLE t416d (id int PRIMARY KEY, c1 int, c2 int);
db1=# INSERT INTO t416d SELECT * FROM t416c;
db1=# VACUUM ANALYZE t416c, t416d;
```

（相関 IN 句サブクエリの問い合わせ実行プラン - 16.3 での実行結果）

```
db1=# explain analyze SELECT * FROM t416c c WHERE c1 IN (
        SELECT c1 FROM t416d d WHERE c.c2 = d.c2);
                              QUERY PLAN
-----------------------------------------------------------------------------
 Seq Scan on t416c c  (cost=0.00..962685.00 rows=5000 width=12)
                      (actual time=0.110..227.156 rows=10000 loops=1)
   Filter: (SubPlan 1)
   SubPlan 1
     ->  Seq Scan on t416d d  (cost=0.00..180.00 rows=5000 width=4)
                              (actual time=0.005..0.017 rows=26 loops=10000)
           Filter: (c2 = c.c2)
           Rows Removed by Filter: 25
 Planning Time: 1.188 ms
 Execution Time: 228.489 ms
(8 rows)
```

（相関 IN 句サブクエリの問い合わせ実行プラン - 17beta2 での実行結果）

```
db1=# explain analyze SELECT * FROM t416c c WHERE c1 IN (
        SELECT c1 FROM t416d d WHERE c.c2 = d.c2);
                              QUERY PLAN
-----------------------------------------------------------------------------
 Hash Join  (cost=210.00..540.00 rows=10000 width=12)
            (actual time=7.493..15.476 rows=10000 loops=1)
   Hash Cond: ((c.c2 = d.c2) AND (c.c1 = d.c1))
   ->  Seq Scan on t416c c  (cost=0.00..155.00 rows=10000 width=12)
```

```
                              (actual time=0.022..1.775 rows=10000 loops=1)
  ->  Hash  (cost=207.00..207.00 rows=200 width=8) (actual
time=7.439..7.441 rows=100 loops=1)
        Buckets: 1024  Batches: 1  Memory Usage: 12kB
        ->  HashAggregate  (cost=205.00..207.00 rows=200 width=8)
                           (actual time=7.331..7.369 rows=100 loops=1)
             Group Key: d.c2, d.c1
             Batches: 1  Memory Usage: 40kB
             ->  Seq Scan on t416d d  (cost=0.00..155.00 rows=10000 width=8)
                                (actual time=0.011..2.771 rows=10000 loops=1)
 Planning Time: 0.363 ms
 Execution Time: 16.447 ms
(11 rows)
```

PostgreSQL 16.3 では Subplan を使った Seq Scan を 10000 回繰り返す実行プランが選択されていますが、17beta2 では t416d テーブルを c1 と c2 で GROUP BY したテーブルと結合する形に書き換えられていて、テーブルの Seq Scan はそれぞれ 1 回のみとなっています。コスト、実行時間ともに、962685 から 540、228.5ms から 16.5ms と、大幅に小さくなっています。

◆ *範囲データ型への対応拡充*

PostgreSQL 17 では範囲データ型に対するプランナの対応が拡充されました。

以下のように範囲データ型に含まれるかを判定する演算子が大小比較に展開されるようになりました。

（1 万行の timestamp 型データを持つテーブルを作成）

```
db1=# CREATE TABLE t416e (id int, ts timestamp);
db1=# INSERT INTO t416e SELECT g, '2024-08-02'::timestamp - (g || 'hour')
        ::interval FROM generate_series(1, 10000) g;
db1=# CREATE INDEX ON t416e (ts);
db1=# VACUUM ANALYZE t416e;
```

（範囲データでの問い合わせ - 16.3 の結果）

```
db1=# explain analyze SELECT * FROM t416e
        WHERE tsrange('-Infinity', '2023-06-15'::timestamp, '()') @> ts;
                                QUERY PLAN
-------------------------------------------------------------------------
```

```
 Seq Scan on t416e  (cost=0.00..180.00 rows=50 width=12)
                    (actual time=3.451..3.474 rows=55 loops=1)
   Filter: ('(-infinity,"2023-06-15 00:00:00")'::tsrange @> ts)
   Rows Removed by Filter: 9945
 Planning Time: 0.112 ms
 Execution Time: 3.507 ms
(5 rows)
```

（範囲データでの問い合わせ - 17beta2 での結果）

```
db1=# explain analyze SELECT * FROM t416e
        WHERE tsrange('-Infinity', '2023-06-15'::timestamp, '()') @> ts;
                                QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using t416e_ts_idx on t416e  (cost=0.29..9.36 rows=54 width=12)
                                  (actual time=0.018..0.040 rows=55 loops=1)
   Index Cond: ((ts > '-infinity'::timestamp without time zone)
           AND (ts < '2023-06-15 00:00:00'::timestamp without time zone))
 Planning Time: 0.623 ms
 Execution Time: 0.075 ms
(4 rows)
```

17beta2 では、範囲に含まれるかという「@>」演算子が、2 つの大小比較演に展開されて、インデックス利用が可能になって、処理時間が短くなっていることが分かります。

また、テーブルの範囲データ型のプランナ統計情報が拡充されました。以下のように確認しました。

（範囲型 int4range を持つテーブルを作成、ANALYZE、プランナ統計情報確認 - 17beta2 で）

```
db1=# CREATE TABLE t416f (id int, ir int4range);
db1=# INSERT INTO t416f SELECT g, ('[0,' || g || ']')::int4range
        FROM generate_series(1, 10) g;
db1=# INSERT INTO t416f SELECT g, ('[10,20]')::int4range
        FROM generate_series(1, 10) g;
db1=# ANALYZE t416f;
db1=# SELECT * FROM pg_stats WHERE tablename = 't416f' AND attname = 'ir';
-[ RECORD 1 ]----------+-----------------------------------------------------
schemaname             | public
tablename              | t416f
```

```
attname                 | ir
inherited               | f
    《中略》
range_length_histogram | {2,3,4,5,6,7,8,9,10,11,11,11,11,11,11,11,11,11,11,11}
range_empty_frac        | 0
range_bounds_histogram | {"[0,2)","[0,3)","[0,4)","[0,5)","[0,6)","[0,7)",
"[0,8)","[0,9)","[0,10)","[0,11)","[10,21)","[10,21)","[10,21)","[10,21)",
"[10,21)","[10,21)","[10,21)","[10,21)","[10,21)","[10,21)"}
```

新たな列が３つ追加されていることが分かります。それぞれ、範囲の長さの分布、空範囲の比率、範囲値の分布をあらわしています。

◆ *GROUP BY の最適化*

プランナが GROUP BY の処理順序を ORDER BY 指定やインデックス順に合わせて調整できるようになりました。これにより、Group 処理や GroupAggregate 処理で、ソート処理の重複を回避することができます。

以下に PostgreSQL 16.3 と 17beta2 を比較した例を示します。

```
（テストテーブルを作成、インデックスは k1,k2,k3 列のみに作成）
db1=# CREATE TABLE t416g (id int, k1 int, k2 int, k3 int, k4 int);
db1=# INSERT INTO t416g SELECT g, g % 10, (g / 10) % 10, (g / 100) % 10,
       (g / 1000) % 10 FROM generate_series(1, 100000) g;
db1=# CREATE INDEX ON t416g (k1, k2, k3);
db1=# VACUUM ANALYZE t416g;

（グループ集約の場合を見たいのでハッシュ集約は無効化）
db1=# SET enable_hashagg TO off;

（プランを単純化するためパラレル処理も無効化）
db1=# SET max_parallel_workers_per_gather TO 0;

（インデックス順と異なる GROUP BY 指定順 - 16.3 実行結果）
db1=# explain analyze
      SELECT k3, k2, k1, count(*) FROM t416g GROUP BY k3, k2, k1;
                            QUERY PLAN
-------------------------------------------------------------------------------
```

```
 GroupAggregate  (cost=9941.82..11201.82 rows=1000 width=20)
                 (actual time=153.078..219.239 rows=1000 loops=1)
   Group Key: k3, k2, k1          《←指定順の通り》
   ->  Sort  (cost=9941.82..10191.82 rows=100000 width=12)
             (actual time=152.986..186.330 rows=100000 loops=1)
         Sort Key: k3, k2, k1          《←Sort 処理が発生》
         Sort Method: external merge  Disk: 2160kB
         ->  Seq Scan on t416g  (cost=0.00..1637.00 rows=100000 width=12)
(actual time=0.017..34.701 rows=100000 loops=1)
 Planning Time: 0.208 ms
 Execution Time: 220.167 ms
(8 rows)
```

（インデックス順と異なる GROUP BY 指定順 - 17beta2 実行結果）

```
db1=# explain analyze
      SELECT k3, k2, k1, count(*) FROM t416g GROUP BY k3, k2, k1;
                            QUERY PLAN
-------------------------------------------------------------------------------
 GroupAggregate  (cost=0.29..2882.29 rows=1000 width=20) (actual
time=0.155..51.933 rows=1000 loops=1)
   Group Key: k1, k2, k3          《←インデックス順になっている／ソート不要》
   ->  Index Only Scan using t416g_k1_k2_k3_idx on t416g
       (cost=0.29..1872.29 rows=100000 width=12)
       (actual time=0.094..26.529 rows=100000 loops=1)
         Heap Fetches: 0
 Planning Time: 0.538 ms
 Execution Time: 52.118 ms
(6 rows)
```

（インデックス外の列も追加 - 16.3 実行結果）

```
db1=# explain analyze
      SELECT k3, k2, k1, k4, count(*) FROM t416g GROUP BY k3, k2, k1, k4;
                                QUERY PLAN
-------------------------------------------------------------------------------
 GroupAggregate  (cost=9941.82..11541.82 rows=10000 width=24)
```

```
                  (actual time=205.639..284.987 rows=10000 loops=1)
   Group Key: k3, k2, k1, k4
   ->  Sort  (cost=9941.82..10191.82 rows=100000 width=16)
             (actual time=205.610..241.823 rows=100000 loops=1)
         Sort Key: k3, k2, k1, k4      《←全列通したソートが発生》
         Sort Method: external merge  Disk: 2552kB
         ->  Seq Scan on t416g  (cost=0.00..1637.00 rows=100000 width=16)
(actual time=0.016..37.831 rows=100000 loops=1)
 Planning Time: 0.246 ms
 Execution Time: 287.777 ms
(8 rows)
```

（インデックス外の列も追加 - 17beta2 実行結果）

```
db1=# explain analyze
      SELECT k3, k2, k1, k4, count(*) FROM t416g GROUP BY k3, k2, k1, k4;
                                QUERY PLAN
-------------------------------------------------------------------------------
 GroupAggregate  (cost=8.02..10352.20 rows=10000 width=24)
                 (actual time=0.484..321.066 rows=10000 loops=1)
   Group Key: k1, k2, k3, k4
   ->  Incremental Sort  (cost=8.02..9002.20 rows=100000 width=16)
                         (actual time=0.469..277.995 rows=100000 loops=1)
         Sort Key: k1, k2, k3, k4
         Presorted Key: k1, k2, k3      《←順序が違ってもインクリメンタルソート》
         Full-sort Groups: 1000  Sort Method: quicksort  Average Memory:
27kB  Peak Memory: 27kB
         Pre-sorted Groups: 1000  Sort Method: quicksort  Average Memory:
28kB  Peak Memory: 28kB
         ->  Index Scan using t416g_k1_k2_k3_idx on t416g
            (cost=0.29..4410.28 rows=100000 width=16)
            (actual time=0.037..163.147 rows=100000 loops=1)
 Planning Time: 1.098 ms
 Execution Time: 322.312 ms
(10 rows)
```

17beta2 ではインデックスと違った列順で GROUP BY を与えた場合に、インデックス順序に合わせたグループ集約が行われました。所要時間も 220ms から 52ms と短縮しています。

追加でインデックス外の列を GROUP BY 指定に加えたときには、16.3 では全列でのソートが発生しましｔが、17beta2 では Incremental Sort が使われました。ただし、今回の SQL 例では所要時間としては改善しませんでした.。

本最適化は新たな設定パラメータ enable_group_by_reordering で有効／無効を設定できます。デフォルトは on で有効です。

4.2. SQL 機能

4.2.1. SQL/JSON の関数・メソッド追加

PostgreSQL 17 で SQL/JSON 標準に準拠した JSON データを処理する関数および jsonpath メソッドがいくつか追加されました。実現される機能としては、従来から在った json、jsonb 型データに対する関数とそれほど大きな違いはありませんが、他の DBMS との互換性向上に役立ちます。

動作確認のためのサンプルデータを用意します。

（JSONB 型データを持つテーブルを作成）

```
db1=# CREATE TABLE t_ext(id serial, js jsonb);
db1=# INSERT INTO t_ext (js) VALUES
      ($$ { "basic": {
         "name": "pg_store_plans",
         "repository": "https://github.com/ossc-db/pg_store_plans",
         "first-release": "2015-10-09",
         "main-authors": ["Kyotaro Horiguchi"]}}$$),
      ($$ { "basic": {
         "name": "pgaudit",
         "repository": "https://github.com/pgaudit/pgaudit",
         "website": "https://www.pgaudit.org",
         "first-release": "2015-12-12",
         "main-authors": ["David Steele"]}}$$),
      ($$ { "basic": {
         "name": "pg_repack",
         "repository": "https://github.com/reorg/pg_repack",
         "first-release": "2008-12-08",
         "main-authors": ["Daniele Varrazzo", "Josh Kupershmidt"]}}$$ );
```

◆ *JSON_TABLE*

JSON データを SQL のテーブルとして展開する JSON_TABLE 関数が追加されました。

以下のように使用できます。

（jsonb 列の内容を JSON_TABLE で表に展開）

```
db1=# SELECT id, jt.* FROM t_ext, LATERAL JSON_TABLE(js, '$.basic' COLUMNS (
        name text PATH '$.name' ,
        main_authors text[] PATH '$."main-authors"',
        "1st_rel" date PATH '$."first-release"')) as jt;
 id |      name      |              main_authors                |  1st_rel
----+----------------+------------------------------------------+------------
  1 | pg_store_plans | {"Kyotaro Horiguchi"}                    | 2015-10-09
  2 | pgaudit        | {"David Steele"}                         | 2015-12-12
  3 | pg_repack      | {"Daniele Varrazzo","Josh Kupershmidt"} | 2008-12-08
(3 rows)
```

jsonpath の問い合わせ結果について、さらに jsonpath で値を取り出して、それぞれ SQL としての列名とデータ型を指定することができます。上記 SELECT 例で LATERAL 結合で記述しているのは、JSON_TABLE が複数行を返す場合に対応するためです。本サンプルデータと jsonpath では該当しませんが、その場合には 1 つのテーブル行に対して複数行が出力されます。

◆ *SQL/JSON 問い合わせ関数*

SQL/JSON 準拠の問い合わせ関数 JSON_EXISTS、JSON_QUERY、JSON_VALUE が追加されました。いずれも JSON データに対して、jsonpath で結果を取り出します。

以下に使用例を示します。

（JSON_EXISTS は該当要素の有無を問い合わせる、JSON_QUERY は該当要素の取り出し）

```
db1=# SELECT id, JSON_EXISTS(js, '$.basic.website'),
        JSON_QUERY(js, '$.basic.website' RETURNING text OMIT QUOTES)
        FROM t_ext;
 id | json_exists |       json_query
----+-------------+------------------------
  1 | f           |
  2 | t           | https://www.pgaudit.org
  3 | f           |
```

```
(3 rows)

（JSON_VALUE は該当要素を SQL のデータ型として取り出し）
db1=# SELECT id,
       JSON_VALUE(js, '$.basic."first-release"' RETURNING timestamp)
       FROM t_ext WHERE id = 1;
 id |     json_value
----+---------------------
  1 | 2015-10-09 00:00:00
(1 row)

（JSON_VALUE では JSON 配列→SQL 配列の変換はできない）
db1=# SELECT id,
       JSON_VALUE(js, 'strict $.basic."main-authors"' RETURNING text[]
         ERROR ON ERROR) FROM t_ext WHERE id = 3;
ERROR:  JSON path expression in JSON_VALUE should return single scalar item

（JSON_QUERY であれば JSON 配列→SQL 配列の変換も可能）
db1=# SELECT id,
       JSON_QUERY(js, 'strict $.basic."main-authors"' RETURNING text[]
         ERROR ON ERROR) FROM t_ext WHERE id = 3;
 id |               json_query
----+----------------------------------------
  3 | {"Daniele Varrazzo","Josh Kupershmidt"}
(1 row)
```

◆ *SQL/JSON コンストラクタ関数の追加*

SQL/JSON 準拠のコンストラクタ関数 JSON_SCALAR、 JSON_SERIALIZE が追加されました。

以下に使用例を示します。

```
（JSON_SCALAR は SQL のスカラ値から JSON データを作成、出力は json 型になる）
db1=# SELECT json_scalar('t'::boolean);
 json_scalar
-------------
 true
```

```
(1 row)

db1=# SELECT json_scalar(CURRENT_TIMESTAMP) ;
          json_scalar
-------------------------------------
 "2024-08-15T17:21:51.701404+09:00"
(1 row)

（JSON_SELIALIZE は json データをバイナリ列または文字列に変換）
db1=# SELECT json_serialize('{"A":100, "B":200} '::json RETURNING bytea);
            json_serialize
------------------------------------------
 \x7b2241223a3130302c202242223a3230307d20
(1 row)
    《→ 出力は UTF8 でバイナリ列にしたもの、正規化もされない。
        末尾に空白文字に対応した 0x20 がある。》

（JSON_SELIALIZE の入力は json 型であり、jsonb 型では機能しない）
db1=# SELECT json_serialize('{"A":100, "B":200} '::jsonb RETURNING bytea);
 json_serialize
----------------
 \x02              《→ エラーにはならないが役に立たない出力 》
(1 row)
```

◆ *jsonpath の型変換メソッド追加*

jsonpath 内で使える型変換のメソッド .bigint()、.boolean()、.date()、.decimal(precision, scale)、.integer()、.number()、.string()、.time()、.time_tz()、.timestamp()、.timestamp_tz() が追加されました。JSON は数値型と文字列以外にはデータ型の区別がありませんが、jsonpath 内で比較するときには、データ型を指定することで適切な大小比較順序を定めることができます。

以下に使用例を示します。

```
（jsonpath 内で型変換メソッドを使う）
db1=# SELECT id, JSON_QUERY(js, '$.basic.name') FROM t_ext
        WHERE JSON_EXISTS (js,
           '$.basic."first-release" ? (@.date() < "2010-01-01".date())');
```

```
 id | json_query
----+-------------
  3 | "pg_repack"
(1 row)

（.decimal は指定の精度と桁位置で丸めた数値を得るメソッド）
db1=# SELECT JSON_QUERY('{"a": "123.45"}', '$.a.decimal(3, -1)');
 json_query
------------
 120
(1 row)

（適用対象文字列は変換先データ型の要件を満たしていないといけない）
db1=# SELECT JSON_QUERY('{"a": "3.14"}',
            'strict $.a.bigint()' ERROR ON ERROR);
ERROR:  argument "3.14" of jsonpath item method .bigint() is invalid for
type bigint
```

上記例で bigint 型に小数を含む数値の文字列を与えるとエラーになっているように、適用対象の JSON 文字列は変換先データ型の要件を満たしていないとエラーになります。また、.date()、 .time()、 .time_tz()、 .timestamp()、 .timestamp_tz() を適用する日付時刻の文字列は ISO 8601 拡張形式（ただし、分、秒の省略は禁止）であることが求められます。いずれも、PostgreSQL の数値や日付時刻のデータ型で受け付けられる入力文字列と比べると許容範囲が狭いことに注意が必要です。

4.2.2. *MERGE* 文の拡張

PostgreSQL 17 では MERGE 文に 3 つ拡張が加わりました。

◆ *RETURNING 句*

MERGE で RETURNING 句を指定できるようになりました。RETURNING オプションを指定すると 、MERGE の影響を受けたターゲットテーブルの行が返ります。さらに RETURNING オプションにより返された行の DML 種別（INSERT/UPDATE/DELETE）を判定するための関数 merge_action も追加されました。

以下にコマンド実行例を示します。

```
（テーブルとデータを作成）
db1=# CREATE TABLE t_tgt(id int PRIMARY KEY, v text);
```

```
db1=# INSERT INTO t_tgt VALUES (1, 'data1'),
                               (2, 'data2');
db1=# CREATE TABLE t_src(id int PRIMARY KEY, v text);
db1=# INSERT INTO t_src VALUES (1, 'data1 updated'),
                               (3, 'data3');

（RETURNING オプション付き MERGE 文の実行例：
  マッチする行は t_src の値で更新、マッチしない行は t_src の値を挿入）
db1=# MERGE INTO t_tgt t USING t_src s ON s.id = t.id
        WHEN MATCHED THEN UPDATE SET v = s.v
        WHEN NOT MATCHED THEN INSERT (id, v) VALUES (s.id, s.v)
        RETURNING merge_action(), t.* ;
 merge_action | id |       v
--------------+----+---------------
 UPDATE       |  1 | data1 updated
 INSERT       |  3 | data3
(2 rows)

MERGE 2

（id=1 の行が更新されて、id=3 の行が挿入された）
db1=# SELECT * FROM t_tgt ORDER BY id;
 id |     v
----+---------------
  1 | data1 updated
  2 | data2
  3 | data3
(3 rows)
```

◆ *WHEN NOT MATCHED BY SOURCE*

MERGE に WHEN NOT MATCHED BY SOURCE 構文が追加されました。

今までは、「WHEN NOT MATCHED」でソース側の行に対してターゲット側の行が無い場合の操作を指定できましたが、新たにターゲット側の行に対してソース側の行が無い場合も指定可能になりました。操作としては、UPDATE / DELETE / DO NOTHING が指定可能です。 これまでの「WHEN NOT MATCHED 」は「BY TARGET」を省略した形と見做されます。

以下に実行例を示します。

```
（引きつづき t_src、t_tgt テーブルを使用／t_src に行を追加する）
db1=# INSERT INTO t_src VALUES (4, 'data4');
db1=# SELECT * FROM t_src ORDER by id;
 id |       v
----+---------------
  1 | data1 updated
  3 | data3
  4 | data4
(3 rows)

db1(5432)=# SELECT * FROM t_tgt ORDER BY id;
 id |       v
----+---------------
  1 | data1 updated
  2 | data2
  3 | data3
(3 rows)

（WHEN NOT MATCHED BY SOURCE の実行例：
  t_src になく t_tgt にある行を t_tgt から削除）
db1=# MERGE INTO t_tgt t USING t_src s ON s.id = t.id
        WHEN NOT MATCHED BY SOURCE THEN DELETE
        WHEN NOT MATCHED THEN INSERT (id, v) VALUES (s.id, s.v)
        RETURNING merge_action(), t.* ;
 merge_action | id |   v
--------------+----+-------
 DELETE       |  2 | data2
 INSERT       |  4 | data4
(2 rows)

MERGE 2

（id=2 の行が削除され、id=4 の行が挿入された）
db1(5432)=# SELECT * FROM t_tgt;
```

```
 id |       v
----+---------------
  1 | data1 updated
  3 | data3
  4 | data4
(3 rows)
```

◆ *更新可能ビューのMERGE*

MERGE で指定するターゲットとして更新可能ビューを指定することができるようになりました。
以下実行例を示します。

```
（ソーステーブルにデータ作成し、ターゲットテーブルは 0 行とする）
db1=# TRUNCATE t_tgt, t_src;
db1=# INSERT INTO t_src VALUES (generate_series(1, 3), 'data1') RETURNING *;
 id |   v
----+-------
  1 | data1
  2 | data1
  3 | data1
(3 rows)

（ターゲットテーブルの更新可能ビューを作成）
db1=# CREATE VIEW v_tgt AS SELECT * FROM t_tgt;

（ターゲットにビューを指定し MERGE を実行）
db1=# MERGE INTO v_tgt v1 USING t_src s ON v1.id = s.id
        WHEN NOT MATCHED THEN INSERT (id, v) VALUES (s.id, s.v)
        RETURNING merge_action(), v1.* ;
 merge_action | id |   v
--------------+----+-------
 INSERT       |  1 | data1
 INSERT       |  2 | data1
 INSERT       |  3 | data1
(3 rows)
```

```
MERGE 3
```

16.x までのバージョンでは、指定できるターゲットとしてビューのサポートはされておらず、ビューを指定して MERGE を実行するとエラーになります。

4.2.3. COPY FROM に ON_ERROR オプション追加

PostgreSQL 17 では、新たに COPY FROM 文に ON_ERROR オプションが追加されました。設定できる値は、stop（変換エラー発生時に停止する）と、ignore（変換エラーを無視し処理を継続する）です。

PostgreSQL 16 までは、COPY FROM 文の入力ファイルあるいは入力ストリームに、取り込み先のテーブルの列のデータ型に合わない表現が含まれていたなど、データ変換に失敗した場合は、エラーで終了していました。これにより、エラー以前までの入力がすべて無駄になっていました。

PosstgreSQL 17 では、このような変換エラーの発生した入力行をスキップし、正常な入力行のみを、取り込むことができるようになりました。ただし、制約違反が発生した場合はこの限りではありません。制約違反が発生した時点で処理はエラーで終了し、データは 1 行も取り込まれません。

また、エラー発生時のエラー出力を制御する、LOG_VERBOSITY オプションも追加されました。設定できる値は、default（標準の出力）と、verbose（より詳細な出力）です。

以下に実行例を示します。

（PG16 実行例 - エラーが発生し、データは取り込まれない）

```
db1=# CREATE TABLE fruits (name text, price numeric);
db1=# COPY fruits FROM STDIN WITH (FORMAT csv);
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself, or an EOF signal.
>> orange, 100
>> apple, 200
>> melon, Market Price
>> bananas, 300
>> \.
ERROR:  invalid input syntax for type numeric: " Market Price"
CONTEXT:  COPY fruits, line 3, column price: " Market Price"

db1=# SELECT * FROM fruits;
 name | price
------+-------
(0 rows)
```

（PG17 実行例 - エラー行は除外され、それ以外の行が取り込まれ、エラー行数が報告される）

```
db1=# COPY fruits FROM STDIN WITH (FORMAT csv, ON_ERROR ignore);
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself, or an EOF signal.
>> orange, 100
>> apple, 200
>> melon, Market Price
>> bananas, 300
>> \.
NOTICE:  1 row was skipped due to data type incompatibility
COPY 3

db1=# SELECT * FROM fruits;
  name   | price
---------+-------
 orange  |   100
 apple   |   200
 bananas |   300
(3 rows)
```

（PG17 実行例 verbose ログ指定 - エラーの箇所も報告される）

```
db1=# COPY fruits FROM STDIN WITH
        (FORMAT csv, ON_ERROR ignore, LOG_VERBOSITY verbose);
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself, or an EOF signal.
>> orange, 100
>> apple, 200
>> melon, Market Price
>> bananas, 300
>> \.
NOTICE:  skipping row due to data type incompatibility at line 3 for column
price: " Market Price"
NOTICE:  1 row was skipped due to data type incompatibility
COPY 3
```

4.2.4. EXPLAIN 文の拡張

EXPLAIN 文に SERIALIZE オプション、MEMORY オプションが追加されました。また、I/O 時間の報告も可能になりました。

SERIALIZE オプションはネットワーク転送用にテキスト形式またはバイナリ形式にデータを変換するコストを報告します。指定するとデータ量 (output) と時間 (time) の情報が出力されます。ANALYZE オプションと同時に使用する必要があります。

オプションの設定値としては以下の値が使用できます。

設定値	説明
NONE	出力なし（SERIALIZE を指定しなかった場合のデフォルト）
TEXT	テキスト形式で出力した場合の値を出力（SERIALIZE のみを指定したときのデフォルト）
BINARY	バイナリ形式で出力した場合の値を出力

以下に実行例を示します。

```
（設定変更）
$ vi $PGDATA/postgresql.conf
   track_io_timing = on
$ pg_ctl reload

（SERIALIZE オプション動作例）
$ psql db1
db1=# CREATE TABLE t424 AS SELECT g id, md5(g::text) v
        FROM generate_series(1, 1000) g;
db1=# EXPLAIN (ANALYZE, SERIALIZE BINARY) SELECT * FROM t424;
                                      QUERY PLAN
------------------------------------------------------------------------------
 Seq Scan on t424  (cost=0.00..36.32 rows=2032 width=36)
   (actual time=0.024..0.336 rows=1000 loops=1)
 Planning Time: 0.103 ms
 Serialization: time=0.432 ms  output=45kB  format=binary
 Execution Time: 0.452 ms
(4 rows)
```

MEMORY オプションを指定すると、プラン作成のために割り当てられたメモリ量（allocated）と実際に使用されたメモリ量（used）の情報が出力されます。SQL 実行時に使うメモリ量ではないことに注意が必要です。

（MEMORY オプション動作例）

```
db1=# EXPLAIN (MEMORY) SELECT * FROM t424;
                       QUERY PLAN
-------------------------------------------------------------
 Seq Scan on t424  (cost=0.00..22.00 rows=1200 width=40)
 Planning:
   Memory: used=7kB  allocated=8kB
(3 rows)
```

EXPLAIN で BUFFERS オプションを指定した場合の出力に「Local I/O Read Time」と「Local I/O Write Time」が出力されるようになりました。track_io_timing 設定が有効な場合に使用できます。

また、これまでは「I/O Read Time」「I/O Write Time」として報告されていた項目は「Shared I/O Read Time」「Shared I/O Write Time」という項目名に変わりました。従来動作では、Shared と Local が合算されていたのではなく、Shared についてのみ報告されていました。

（BUFFER オプション動作例）

```
db1=# EXPLAIN (ANALYZE, BUFFERS, FORMAT json) SELECT * FROM t424;
              QUERY PLAN
--------------------------------------
 [                                    +
   {                                  +
     "Plan": {                        +
       "Node Type": "Seq Scan",       +
       "Parallel Aware": false,       +
       "Async Capable": false,        +
       "Relation Name": "t424",       +
      《中略》
       "Shared I/O Read Time": 6.692, +
       "Shared I/O Write Time": 0.000,+
       "Local I/O Read Time": 0.000,  +
       "Local I/O Write Time": 0.000, +
       "Temp I/O Read Time": 0.000,   +
```

```
      "Temp I/O Write Time": 0.000    +
    },                                +
    "Planning": {                     +
      "Shared Hit Blocks": 0,         +
      《中略》
      "Shared I/O Read Time": 3.583, +
      "Shared I/O Write Time": 0.000,+
      "Local I/O Read Time": 0.000,  +
      "Local I/O Write Time": 0.000, +
      "Temp I/O Read Time": 0.000,    +
      "Temp I/O Write Time": 0.000    +
    },                                +
    "Planning Time": 4.349,           +
    "Triggers": [                     +
    ],                                +
    "Execution Time": 7.325           +
  }                                    +
]
(1 row)
```

上記の実行例ではフォーマットを JSON としています。これは、EXPLAIN では出力フォーマットが TEXT（デフォルト）の場合には値がゼロの場合には出力自体が省略されてしまうためです。

4.2.5. 接続認証のイベントトリガ

PostgreSQL 17 では イベントトリガに 2 つ拡張が加わりました。

◆ *login イベントトリガ*

クライアントからの認証が成功した時点で実行される login イベントトリガ が追加されました。

イベントトリガプロシージャにバグがあると、ログインが失敗する可能性がありますが、そのような場合には、イベントトリガの実行を一時的に無効にすることができる設定パラメータ event_triggers（デフォルト on）を false に設定することで回避できます。また、シングルユーザモードでは、イベントトリガが無効になるため、シングルユーザモードで再起動することで回避することもできます。

以下は接続したセッションユーザ名と接続時刻をテーブルに追記するイベントトリガの実行例です。

（ログイン記録を追記するためのテーブル作成）

```
db1=# CREATE TABLE login_event (user_name text, login_timestamp timestamp);
```

（イベントトリガ関数を作成）

```
db1=# CREATE OR REPLACE FUNCTION login_event ()
        RETURNS event_trigger SECURITY DEFINER LANGUAGE plpgsql AS $$
        BEGIN
          INSERT INTO login_event (user_name, login_timestamp)
            VALUES (session_user, current_timestamp);
        END; $$;
```

（ログインイベントトリガを作成）

```
db1=# CREATE EVENT TRIGGER login_event_trigger ON login
        EXECUTE FUNCTION login_event();
```

（イベントトリガを有効化）

```
db1=# ALTER EVENT TRIGGER login_event_trigger ENABLE ALWAYS;
```

（接続し直して、イベントトリガの動作結果を確認）

```
db1=# \q
$ psql -U postgres -d db1
db1=# SELECT now();
              now
-------------------------------
 2024-08-05 13:06:29.139727+09
(1 row)

db1=# SELECT * FROM login_event;
 user_name |      login_timestamp
-----------+----------------------------
 postgres  | 2024-08-05 13:06:28.015732
(1 row)
```

《→ データベース接続をテーブルに記録することができた》

◆ *REINDEX トリガ*

イベントトリガが REINDEX コマンドで機能するようになり、ddl_command_start および ddl_command_end イベントのトリガを作成できるようになりました。

以下に実行例を示します。まずは、REINDEX 開始時イベントトリガの発行の実行例です。

```
（イベントトリガ関数を作成：
 REINDEX 開始時、トリガを発行させたイベント名とコマンドタグ名を出力させる）
db1=# CREATE OR REPLACE FUNCTION reindex_start()
       RETURNS event_trigger LANGUAGE plpgsql AS $$
       BEGIN
        RAISE NOTICE 'reindex_start: tg_event=% tg_tag=%', tg_event, tg_tag;
       END; $$;

（REINDEX イベントトリガ を作成する）
db1=# CREATE EVENT TRIGGER reindex_start_trigger ON ddl_command_start
       WHEN TAG IN ('REINDEX')
       EXECUTE PROCEDURE reindex_start() ;

（前節で作った login_event テーブルに REINDEX TABLE を実行）
db1=# REINDEX TABLE login_event;
NOTICE:  reindex_start: tg_event=ddl_command_start tg_tag=REINDEX
REINDEX
```

次はREINDEX 終了時イベントトリガの発行の実行例です。

```
（login_event テーブルに インデックスを作成）
db1=# CREATE INDEX idx_1 ON login_event (user_name);
db1=# CREATE INDEX idx_2 ON login_event (login_timestamp);

（イベントトリガ関数を作成：
 REINDEX 終了時、コマンドタグ、オブジェクトの型、識別をメッセージとして出力させる）
db1=# CREATE OR REPLACE FUNCTION reindex_end()
       RETURNS event_trigger LANGUAGE plpgsql AS $$
       DECLARE
         obj record ;
       BEGIN
```

```
          FOR obj IN SELECT * FROM pg_event_trigger_ddl_commands() LOOP
           RAISE NOTICE
            'reindex_end: tg_event=% tg_tag=% object_type=% object_id=%',
              tg_event, tg_tag, obj.object_type, obj.object_identity;
          END LOOP ;
        END; $$;

（REINDEX イベントトリガを作成）
db1=# CREATE EVENT TRIGGER reindex_end_trigger ON ddl_command_end
       WHEN TAG IN ('REINDEX')
       EXECUTE PROCEDURE reindex_end();

（REINDEX TABLE を実行してトリガ動作を確認）
db1=# REINDEX TABLE login_event;
NOTICE:  reindex_start: tg_event=ddl_command_start tg_tag=REINDEX
NOTICE:  reindex_end: tg_event=ddl_command_end tg_tag=REINDEX
object_type=index object_id=public.idx_1
NOTICE:  reindex_end: tg_event=ddl_command_end tg_tag=REINDEX
object_type=index object_id=public.idx_2
REINDEX
```

16.x までのバージョンでは、イベントトリガでは REINDEX はサポートしておらず、コマンドタグに REINDEX を指定した CREATE EVENT TRIGGER を実行すると「ERROR: event triggers are not supported for REINDEX」のエラーが発生しました。

4.2.6. プラットフォーム非依存の照合順序

プラットフォーム依存しない C 照合順序と同様の照合順序プロバイダ「builtin」が追加されました。この照合順序プロバイダでは、C および C.UTF-8 ロケールのみがサポートされています。また、照合順序プロバイダを指定するコマンドや SQL に builtin を指定できるようになりました。

以下は、CREATE DATABASE で LOCALE_PROVIDER を指定した時の実行例です。

```
(LOCALE_PROVIDERオプションにそれぞれ builtin と libc を指定してデータベース作成)
db1=# CREATE DATABASE testdb1 LOCALE_PROVIDER builtin LOCALE 'C.UTF-8'
        TEMPLATE template0;
db1=# CREATE DATABASE testdb2 LOCALE_PROVIDER libc LOCALE 'C.UTF-8'
```

```
        TEMPLATE template0;
db1=# SELECT datname, datcollate, datctype, datlocale, datcollversion
        FROM pg_database WHERE datname LIKE 'testdb%';
 datname | datcollate | datctype | datlocale | datcollversion
---------+------------+----------+-----------+----------------
 testdb1 | C.UTF-8    | C.UTF-8  | C.UTF-8   | 1
 testdb2 | C.UTF-8    | C.UTF-8  |           |
(2 rows)
```

《→ 区別しにくいが、libc プロバイダの場合には datlocale が NULL になる》

builtin プロバイダの C ロケールの動作は libc プロバイダの C ロケールと同じです。データベースエンコーディングに依存して動作が異なるかもしれません。

builtin プロバイダの C.UTF-8 ロケールは、データベースエンコーディングが UTF-8 の場合にのみ使用可能で、 Unicode に基づき動作します。builtin プロバイダの C.UTF-8 は libc 版 と比べて、ソートや大文字と小文字の変換が高速、プラットフォーム依存しない、という利点があります。

CREATE DATABASE の他にも、initdb や createdb の --locale-provider オプションで builtin を指定できます。また、builtin プロバイダを使用する時のロケール名を指定する --builtin-locale オプションも追加されています。

以下、initdb の実行例を示します。

```
$ initdb --locale-provider=builtin --builtin-locale=C.UTF8
The files belonging to this database system will be owned by user
"postgres".
This user must also own the server process.

The database cluster will be initialized with this locale configuration:
  locale provider:   builtin
  default collation: C.UTF-8
    《出力後略》
```

4.3. ロジカルレプリケーション

ロジカル（論理）レプリケーションに関しても、いくつか拡張されています。主要なものを検証しました。

4.3.1. pg_createsubscriber コマンド追加

物理ストリーミングレプリケーションのスタンバイサーバを元にして、ロジカル（論理）レプリケーションのサブスクライバサーバを作るコマンド pg_createsubscriber が追加されました。

以下のように使用します。本検証では専用のデータベースクラスタを２つ作成します。

```
（5433 ポートで動作するデータベースクラスタ ptdata1 を作成し、
 データベース db1、テーブル t43 を作成）
$ initdb -D ./pgdata1 --no-locale -E UTF8 \
   -c logging_collector=on -c wal_level=logical -c port=5433
$ pg_ctl start -D ./pgdata1
$ createdb -p 5433 db1
$ psql -p 5433 db1
db1=# CREATE TABLE t43 (id int PRIMARY KEY, v text);
db1=# INSERT INTO t43 SELECT g, md5(g::text) FROM generate_series(1, 10) g;
db1=# \q

（ストリーミングレプリケーションのスタンバイをデータベースクラスタ pgdata2 として作成）
$ pg_basebackup -p 5433 -D ./pgdata2 -R -c fast

（スタンバイを起動しない状態でプライマリで １行追加、
 スタンバイは未起動なのでこの行は未だ伝搬していない）
$ psql -p 5433 db1
db1=# INSERT INTO t43 VALUES (11, 'XI');
db1=# \q

（ pg_createsubscriber コマンドを実行、パラメータは：
 -d 対象データベース、-P パブリケーションサーバへの接続文字列、
 -p サブスクライバポート、-D サブスクライバディレクトリ、
 --publication パブリケーション名、--subscription サブスクリプション名 ）
$ pg_createsubscriber -d db1 -P "dbname=db1 user=postgres port=5433" \
   -p 5434 -D ./pgdata2 --publication=pub1 --subscription=sub1
2024-08-07 18:23:04.965 JST [344982] LOG:  redirecting log output to logging
collector process
2024-08-07 18:23:04.965 JST [344982] HINT:  Future log output will appear in
directory "log".
```

```
2024-08-07 18:23:05.525 JST [344996] LOG:  redirecting log output to logging
collector process
2024-08-07 18:23:05.525 JST [344996] HINT:  Future log output will appear in
directory "log".
```

《↑ 本コマンド処理内で 2 回サーバ起動／停止が行われてる》

（サブスクライバサーバに変換された pgdata2 を 5434 ポートで起動して、データ確認）

```
$ pg_ctl start -D pgdata2 -o '-c port=5434'
$ psql -p 5434 db1
db1=# SELECT * FROM t43;
 id |                 v
----+----------------------------------
  1 | c4ca4238a0b923820dcc509a6f75849b
  2 | c81e728d9d4c2f636f067f89cc14862c
  3 | eccbc87e4b5ce2fe28308fd9f2a7baf3
  4 | a87ff679a2f3e71d9181a67b7542122c
  5 | e4da3b7fbbce2345d7772b0674a318d5
  6 | 1679091c5a880faf6fb5e6087eb1b2dc
  7 | 8f14e45fceea167a5a36dedd4bea2543
  8 | c9f0f895fb98ab9159f51fd0297e236d
  9 | 45c48cce2e2d7fbdea1afc51c7c6ad26
 10 | d3d9446802a44259755d38e6d163e820
 11 | XI
(11 rows)
db1=# \q
```

《← 追加した行も伝搬している》

（ロジカルレプリケーション定義を確認、データベース中の全テーブルが対象となっている）

```
$ psql -p 5433 db1
db1=# \x on
db1=# SELECT * FROM pg_publication;
-[ RECORD 1 ]+------
oid          | 16394
pubname      | pub1
pubowner     | 10
puballtables | t
```

```
pubinsert     | t
pubupdate     | t
pubdelete     | t
pubtruncate   | t
pubviaroot    | f
```

上記の検証手順では、pg_basebackup でスタンバイサーバを作った後に、スタンバイサーバを起動せずにプライマリサーバにデータ追加を行って、その状態で pg_createsubscriber コマンドでサブスクライバサーバを作成しました。その結果、サブスクライバサーバには後から追加したデータも反映されていることが確認できました。このことは、サブスクライバサーバを構築するにあたり、稼働しているプライマリ（パブリッシャ）サーバの使用を止める必要がないことを意味しています。

ロジカルレプリケーションの初期コピーには、対象テーブル群を pg_dump / pg_restore するのと同程度の時間がかかります。pg_createsubscriber コマンドを使うことで、これをより高速な pg_basebackup による物理コピーで置き換えできるメリットが得られます。

4.3.2. ロジカルレプリケーションのフェイルオーバ対応

ロジカル（論理）レプリケーションのパブリッシャが、物理ストリーミングレプリケーションのプライマリからスタンバイへのフェイルオーバに対応しました。プライマリサーバに在ったロジカルレプリケーション用のスロットをスタンバイサーバに引継がせることができます。対象となるのは以下図のような構成です。

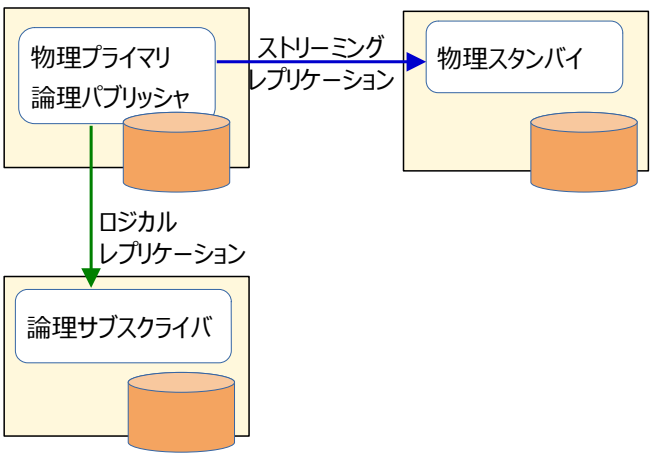


図 1: レプリケーションの組み合わせ

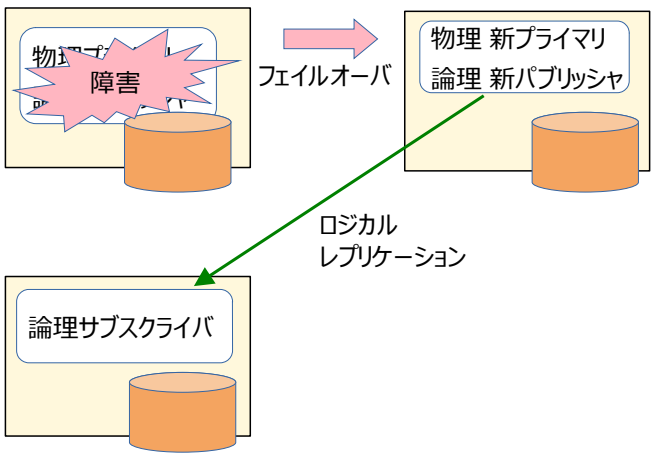


図 2: フェイルオーバした場合

◆ *レプリケーション組み合わせ構成を構築*

前節でデータベースクラスタ pgdata1（5433 ポート、パブリッシャ）、pgdata2（5434 ポート、論理サブスクライバ）、を作っています。これを使ってフェイルオーバ動作を確認しました。まずは、フェイルオーバが可能な設定を加えて pgdata3（5435 ポート、物理スタンバイ）のデータベースクラスタを作成します。

```
（サブスクリプション sub1 をフェイルオーバ有効にする）
$ psql -p 5434 db1
db1=# ALTER SUBSCRIPTION sub1 DISABLE;
db1=# ALTER SUBSCRIPTION sub1 SET (failover = true);
db1=# ALTER SUBSCRIPTION sub1 ENABLE;
db1=# \q

（パブリッシャ側のレプリケーションスロット sub1 の状態を確認）
$ psql -p 5433 db1
db1=# \x on
db1=# SELECT * FROM pg_replication_slots;
-[ RECORD 1 ]-------+-----------
slot_name           | sub1
plugin              | pgoutput
slot_type           | logical
datoid              | 16384
database            | db1
《中略》
invalidation_reason |
failover            | t     《← パブリッシャ側にあるスロットの属性に反映される》
synced              | f
db1=# \q

（スタンバイサーバを作成、ここで物理レプリケーションスロットを使用する必要がある）
$ pg_basebackup -p 5433 -D ./pgdata3 -R --create-slot --slot=pslot1 -c fast

（スロット同期に必要な設定を加える、
 primary_conninfo には接続先データベースの指定を追記する）
$ vi ./pgdata3/postgresql.conf
  sync_replication_slots = on
  hot_standby_feedback = on
```

```
$ vi ./pgdata3/postgresql.auto.conf
   primary_conninfo = 'dbname=db1 user=postgres passfile='...《後略》

（スタンバイサーバを 5435 ポートで起動して、スロット sub1 が作られていることを確認）
$ pg_ctl start -D ./pgdata3 -o '-c port=5435'
$ ls ./pgdata3/pg_replslot
sub1
```

ここでスタンバイサーバが起動できなかったり、pg_repslot ディレクトリ下が空であった場合には、スタンバイサーバのログメッセージを確認してください。不足している設定を示すエラーが出ているはずです。

◆ *synchronized_standby_slots 設定*

パブリッシャが物理ストリーミングレプリケーションのスタンバイにフェイルオーバする場合、パブリッシャ兼プライマリのサーバに加えられた変更が、スタンバイよりもサブスクライバに先に反映されていると、フェイルオーバ後にパブリッシャとサブスクライバでデータの不一致が発生してしまいます。

本設定を使うと、ロジカルレプリケーションよりも指定の物理ストリーミングレプリケーションを先にデータ同期させることができます。これはプライマリサーバに指定します。指定するのは物理ストリーミングレプリケーションで使用しているスロット名のカンマ区切りのリストです。

本検証では pgdata1 のクラスタに以下の設定を与えます。

```
$ vi ./pgdata1/postgresql.conf
   synchronized_standby_slots = 'pslot1'    《beta2 までは standby_slot_names》
$ pg_ctl reload -D ./pgdata1
```

◆ *フェイルオーバ動作確認*

次に、pgdata1（ポート 5433、プライマリ兼パブリッシャ）がダウンした想定でフェイルオーバを行います。

```
（pgdata1 を停止して、pgdata3 を昇格）
$ pg_ctl stop -D ./pgdata1
$ pg_ctl promote -D ./pgdata3

（pgdata2 のロジカルレプリケーション接続先を変更）
$ psql -p 5434 -d db1
db1=# ALTER SUBSCRIPTION sub1 DISABLE;
```

```
db1=# ALTER SUBSCRIPTION sub1
        CONNECTION 'user=postgres port=5435 dbname=db1';
db1=# ALTER SUBSCRIPTION sub1 ENABLE;
db1=# \q
```

（新パブリッシャでロジカルレプリケーション継続を確認）

```
$ psql -p 5435 -d db1
db1=# SELECT * FROM pg_stat_replication;
  pid   | usesysid | usename  | application_name | client_addr |
client_hostname | client_port |         backend_start         | backend_xmin
|   state   |  sent_lsn  | write_lsn  | flush_lsn  | replay_lsn | write_lag
| flush_lag | replay_lag | sync_priority | sync_state |          reply_time
--------+----------+----------+------------------+-------------
+-----------------+-------------+--------------------------------
+--------------+-----------+------------+------------+------------
+------------+-----------+-----------+-----------+--------------
+------------+--------------------------------
 347652 |       10 | postgres | sub1             |             |
|          -1 | 2024-08-08 11:32:27.987249+09 |              | streaming |
0/10000210 | 0/10000210 | 0/10000210 | 0/10000210 |           |           |
|             0 | async      | 2024-08-08 11:32:38.112114+09
(1 row)

db1=# SELECT * FROM pg_replication_slots;
 slot_name |  plugin  | slot_type | datoid | database | temporary | active |
active_pid | xmin | catalog_xmin | restart_lsn | confirmed_flush_lsn |
wal_status | safe_wal_size | two_phase | inactive_since | conflicting |
invalidation_reason | failover | synced
-----------+----------+-----------+--------+----------+-----------+--------
+------------+------+--------------+-------------+---------------------
+------------+---------------+-----------+----------------+-------------
+---------------------+----------+--------
 sub1      | pgoutput | logical   |  16384 | db1      | f         | t      |
347652 |      |          745 | 0/10000128  | 0/10000210          | reserved
|               | f         |                | f           |
```

```
| t         | t
(1 row)

（ロジカルレプリケーション動作を確認 - 引きつづき 5435 ポートのセッションで）
db1=# INSERT INTO t43 VALUES (12, 'XII');
db1=# \q
$ psql -p 5434 db1
db1=# SELECT * FROM t43 WHERE id = 12;
 id |  v
----+-----
 12 | XII
(1 row)
```

pgdata3（ポート 5435）サーバが新たなパブリッシャ兼プライマリとして動作し、pgdata2（ポート 5434）サーバがサブスクライバとして追随しています。

以上の手順でロジカルレプリケーションのパブリッシャがフェイルオーバできることが確認できました。

4.3.3. ロジカルレプリケーションの *pg_upgrade* 対応

PostgreSQL 17 から pg_upgrade でロジカルレプリケーションの設定を引継いでデータベースクラスタのメジャーバージョンアップを行うことができるようになりました。ただし、バージョンアップ時の旧バージョンが 17.x 以上である場合が対象です。したがって、実際にこの機能が使われるのは、PostgreSQL 18 以降が登場して、17.x バージョンからのバージョンアップをしたい時になってから、となります。

本節では開発中の 18devel バージョンを使って、pg_upgrade の動作検証を行います。前節までの手順で 、17beta2 バージョンにてロジカルレプリケーションが構成されたパブリッシャ（pgdata3 ディレクトリ、5435 ポート）とサブスクライバ（pgdata2 ディレクトリ、5434 ポート）のデータベースクラスタを作っていますので、これらを 18devel バージョンにアップグレードすることにします。

◆ *PostgreSQL 18devel のインストール*

開発中のスナップショットバージョンのソースコードは以下 URL から入手しました。本検証では「07-Aug-2024 04:22」時点のファイルを使用しました。

```
https://ftp.postgresql.org/pub/snapshot/dev/postgresql-snapshot.tar.bz2
```

「3.3. インストール」の手順で 17beta2 と同じようにインストールを行います。このときインストール先ディレクトリは /usr/local/pgsql/18 として、環境変数を書いたファイルは pg18.env とします。

◆ *pg_upgrade の適用*

以下の通り、pg_upgrade 適用手順を進めます。基本的な流れは、新バージョンで空のデータベースクラスタを作って、設定を整えて、pg_upgrade を実行することです。

```
（pgdata2、pgdata3 にそれぞれ新バージョンの空データベースクラスタを作成）
$ . pg18.env
$ initdb --no-locale -E UTF8 -D ./pgdata3_18
$ initdb --no-locale -E UTF8 -D ./pgdata2_18

（旧バージョンと同じ設定を与える）
$ vi ./pgdata3_18/postgresql.conf
    wal_level = logical
    logging_collector = on
    sync_replication_slots = on          《←必須ではないが同じ設定にしておく》
    hot_standby_feedback = on            《←必須ではないが同じ設定にしておく》

$ vi ./pgdata2_18/postgresql.conf
   wal_level = logical
   logging_collector = on

（稼働中の 2 つの旧バージョン PostgreSQL を停止）
$ . pg17.env
$ pg_ctl stop -D ./pgdata3
$ pg_ctl stop -D ./pgdata2
```

稼働中の旧バージョンの PostgreSQL を止める時点で、レプリケーション処理が完了している必要があります。途中の処理があったり、コンフリクトがあったりしてはいけません。また、一時作成のレプリケーションスロットが在ってもいけません（テーブル初期同期時に使われます）。実システムに対しては、クライアントアプリケーションからの更新を止めてしばらく時間をおいて、パブリッシャで「SELECT * FROM pg_replication_slots」で確認する、といった手順が必要になります。

続いて pg_upgrade を実行します。

（パブリッシャとサブスクライバでバージョン18のpg_upgradeを実行）

```
$ . pg18.env
$ pg_upgrade --old-datadir ./pgdata3 --new-datadir ./pgdata3_18 \
    --old-bindir /usr/local/pgsql/17/bin \
    --new-bindir /usr/local/pgsql/18/bin
Performing Consistency Checks
-----------------------------
Checking cluster versions                                   ok
Checking database user is the install user                  ok
    《中略》

Performing Upgrade
------------------
Setting locale and encoding for new cluster                 ok
Analyzing all rows in the new cluster                       ok
Freezing all rows in the new cluster                        ok
    《中略》
Setting next OID for new cluster                            ok
Restoring logical replication slots in the new cluster      ok
Sync data directory to disk                                 ok
Creating script to delete old cluster                       ok
Checking for extension updates                              ok

Upgrade Complete
----------------
Optimizer statistics are not transferred by pg_upgrade.
Once you start the new server, consider running:
    /usr/local/pgsql/18/bin/vacuumdb --all --analyze-in-stages
Running this script will delete the old cluster's data files:
    ./delete_old_cluster.sh

$ pg_upgrade --old-datadir ./pgdata2 --new-datadir ./pgdata2_18 \
    --old-bindir /usr/local/pgsql/17/bin \
    --new-bindir /usr/local/pgsql/18/bin
    《出力省略》
```

パブリッシャについては、レプリケーションスロットを移行した旨のメッセージが出ます。サブスクライバ側ではロジカルレプリケーションに関するシステムテーブルの移行が行われますが、特別なメッセージ出力はありません。

続いて、新バージョンのサービス起動をして動作確認を行います。

```
（2 つの新バージョン PostgreSQL を起動）
$ . pg18.env
$ pg_ctl start -D ./pgdata3_18 -o '-c port=5435'
$ pg_ctl start -D ./pgdata2_18 -o '-c port=5434'

（ロジカルレプリケーションが動作していることを確認）
$ psql -p 5435 db1
db1=# SELECT * FROM pg_replication_slots;
 slot_name |  plugin  | slot_type | datoid | database | temporary | active |
active_pid | xmin | catalog_xmin | restart_lsn | confirmed_flush_lsn |
wal_status | safe_wal_size | two_phase | inactive_since | conflicting |
invalidation_reason | failover | synced
-----------+----------+-----------+--------+----------+-----------+--------
+------------+------+--------------+-------------+---------------------
+------------+---------------+-----------+----------------+-------------
+---------------------+----------+--------
 sub1       | pgoutput | logical   |  16384 | db1      | f         | t      |
352203 |      |           774 | 0/80002B0   | 0/80002E8           | reserved
|               | f         |                 | f           |
| t         | f
(1 row)     《サブスクライバのレプリケーションスロットが引継がれていてる》

db1=# INSERT INTO t43 VALUES (13, 'XIII');
db1=# \q
$ psql -p 5434 db1
db1=# SELECT * FROM t43 WHERE id = 13;
 id |  v
----+------
 13 | XIII
(1 row)     《行データが同期されている》
```

```
db1=# SELECT * FROM pg_stat_subscription;
 subid | subname | worker_type |  pid   | leader_pid | relid | received_lsn
|      last_msg_send_time       |     last_msg_receipt_time     |
latest_end_lsn |        latest_end_time
-------+---------+-------------+--------+------------+-------+--------------
+-------------------------------+-------------------------------
+----------------+-------------------------------
 16401 | sub1    | apply       | 352202 |            |       | 0/8000838
| 2024-08-09 09:15:08.762152+09 | 2024-08-09 09:15:08.762189+09 | 0/8000838
| 2024-08-09 09:15:08.762152+09
(1 row)    《 → サブスクリプションが動作している 》
```

以上のように、pg_upgrade でロジカルレプリケーションのパブリッシャ、サブスクライバを、ロジカルレプリケーションを維持した状態でバージョンアップできることが確認できました。

4.4. パーティショニング

4.4.1. マージと分割

※本機能は beta 版には含まれましたが、その後、取り下げられました

PostgreSQL 17 では複数のパーティションを 1 つにマージするパーティションのマージと、1 つのパーティションを複数に分割するパーティションの分割がサポートされました。

パーティションマージとパーティション分割の構文は以下の通りです。

```
ALTER TABLE … MERGE PARTITIONS
   ( <マージ前パーティション名 1>, <マージ前パーティション名 2>, ... )
   INTO <マージ後パーティション名>;
ALTER TABLE … SPLIT PARTITION <パーティション名>
   INTO ( PARTITION <分割後パーティション名 1> FOR VALUES <境界条件 1>,
          PARTITION <分割後パーティション名 2> FOR VALUES <境界条件 2>, ... )
```

複数のパーティションを 1 つにマージする実行例と、1 つのパーティションを複数に分割する実行例を以下に示します。

```
（パーティション定義 - 約10万件× 3 で構成）
db1=# CREATE TABLE t_log (id int PRIMARY KEY, ts timestamp, mes text,
        typ int) PARTITION BY RANGE (id);
db1=# CREATE TABLE t_log_0 PARTITION OF t_log
        FOR VALUES FROM (0) TO (100000);
db1=# CREATE TABLE t_log_1 PARTITION OF t_log
        FOR VALUES FROM (100000) TO (200000);
db1=# CREATE TABLE t_log_2 PARTITION OF t_log
        FOR VALUES FROM (200000) TO (300000);
db1=# \timing on          《相対的な所要時間確認のため timing on とする》
db1=# INSERT INTO t_log SELECT g, now(), md5(g::text), 0
        FROM generate_series(0, 300000 - 1) g;
INSERT 0 300000
Time: 4346.434 ms (00:04.346)

db1=# \d+ t_log
                  Partitioned table "public.t_log"
     : 《列定義の出力を省略》
Partition key: RANGE (id)
Indexes:
    "t_log_pkey" PRIMARY KEY, btree (id)
Partitions: t_log_0 FOR VALUES FROM (0) TO (100000),
            t_log_1 FOR VALUES FROM (100000) TO (200000),
            t_log_2 FOR VALUES FROM (200000) TO (300000)

（パーティションのマージ - t_log_1 と t_log_2 を統合する）
db1=# ALTER TABLE t_log MERGE PARTITIONS (t_log_1, t_log_2) INTO t_log_12;
ALTER TABLE
Time: 1124.506 ms (00:01.125)

db1=# \d+ t_log
                   Partitioned table "public.t_log"
     : 《列定義の出力を省略》
Partition key: RANGE (id)
Indexes:
```

```
    "t_log_pkey" PRIMARY KEY, btree (id)
Partitions: t_log_0 FOR VALUES FROM (0) TO (100000),
            t_log_12 FOR VALUES FROM (100000) TO (300000)

（パーティションの分割 - t_log_12 を再び分割する）
db1=# ALTER TABLE t_log SPLIT PARTITION t_log_12 INTO
        (PARTITION t_log_1 FOR VALUES FROM (100000) TO (200000),
         PARTITION t_log_2 FOR VALUES FROM (200000) TO (300000));
ALTER TABLE
Time: 1105.019 ms (00:01.105)
db1=# \d+ t_log
                        Partitioned table "public.t_log"
     :《列定義の出力を省略》
Partition key: RANGE (id)
Indexes:
    "t_log_pkey" PRIMARY KEY, btree (id)
Partitions: t_log_0 FOR VALUES FROM (0) TO (100000),
            t_log_1 FOR VALUES FROM (100000) TO (200000),
            t_log_2 FOR VALUES FROM (200000) TO (300000)
```

以上のように ALTER TABLE コマンドでパーティションのマージと分割ができました。物理データの移動が発生しますので、これらの操作にはストレージ I/O が発生し、そのための処理時間を要します。マージと分割の実行後に使わなくなった元のパーティションは、マージ・分割の操作で削除されます。

4.4.2. 排他制約に対応

これまでパーティションテーブルでは、パーティションテーブル全体に対する B-tree インデックスを使った主キー制約、一意性制約に対応していましたが、PostgreSQL 17 からは GiST インデックスによる排他制約にも対応しました。主キー制約や一意性制約と同様に、排他制約の対象列に全てのパーティションキー列が「=」演算子の排他条件として含まれる必要があります。

以下に、ハッシュで分割されたオンラインミーティングの予約を管理するパーティションテーブルで使用例を示します。

```
（btree_gist 拡張を導入 - 整数型などに「=」で gist インデックスを使うため）
db1=# CREATE EXTENSION btree_gist;
```

（HASH によるパーティションテーブルを定義）

```
db1=# CREATE TABLE t_meeting (id int, channel int, tsr tsrange, uid int)
        PARTITION BY HASH (channel);
db1=# CREATE TABLE t_meeting_0 PARTITION OF t_meeting
        FOR VALUES WITH (MODULUS 3, REMAINDER 0);
db1=# CREATE TABLE t_meeting_1 PARTITION OF t_meeting
        FOR VALUES WITH (MODULUS 3, REMAINDER 1);
db1=# CREATE TABLE t_meeting_2 PARTITION OF t_meeting
        FOR VALUES WITH (MODULUS 3, REMAINDER 2);
```

（パーティションテーブル全体に主キー制約と排他制約を追加）

```
db1=# ALTER TABLE t_meeting ADD PRIMARY KEY (id, channel);
db1=# ALTER TABLE t_meeting
        ADD EXCLUDE USING gist (channel WITH =, tsr WITH &&);

db1=# \d t_meeting
       Partitioned table "public.t_meeting"
 Column  |  Type   | Collation | Nullable | Default
---------+---------+-----------+----------+---------
 id      | integer |           | not null |
 channel | integer |           | not null |
 tsr     | tsrange |           |          |
 uid     | integer |           |          |
Partition key: HASH (channel)
Indexes:
 "t_meeting_pkey" PRIMARY KEY, btree (id, channel)
 "t_meeting_channel_tsr_excl" EXCLUDE USING gist (channel WITH =, tsr WITH &&)
Number of partitions: 3 (Use \d+ to list them.)
```

（データを投入して、制約の動作を確認）

```
db1=# INSERT INTO t_meeting (id, channel, tsr, uid) SELECT  g,  g % 4,
  tsrange(ts0 + (g || 'hour')::interval, ts0 + (g || 'hour 30min')::interval),
  g % 100 FROM (
  SELECT g, '2024-01-01'::timestamp AS ts0 FROM generate_series(1, 1000) g) s;
```

```
db1=# SELECT * FROM t_meeting WHERE id = 201;
 id  | channel |                       tsr                        | uid
-----+---------+--------------------------------------------------+-----
 201 |       1 | ["2024-01-09 09:00:00","2024-01-09 09:30:00") |   1
(1 row)

db1=# INSERT INTO t_meeting VALUES
       (1001, 1, tsrange('2024-01-09 9:15', '2024-01-09 9:45'), 101);
ERROR:  conflicting key value violates exclusion constraint
"t_meeting_2_channel_tsr_excl"
DETAIL:  Key (channel, tsr)=(1, ["2024-01-09 09:15:00","2024-01-09
09:45:00")) conflicts with existing key (channel, tsr)=(1, ["2024-01-09
09:00:00","2024-01-09 09:30:00")).
```

channel 列の値が同じで、tsr 列のタイムスタンプ範囲に重なりのある行の挿入を試みると、期待通り制約違反エラーが発生しました。

以上のようにパーティションテーブルで排他制約を使用できることが確認できました。

4.4.3. IDENTITY 列に対応

PostgreSQL 17 ではパーティションテーブルで IDENTITY 列を使用できるようになりました。

以下に例を示します。

```
（IDENTITY 列を持つパーティション定義）
db1=# CREATE TABLE partid (i1 INTEGER,
        i2 INTEGER GENERATED ALWAYS AS IDENTITY) PARTITION BY RANGE (i1);
db1=# CREATE TABLE partid_1 PARTITION OF partid
        FOR VALUES FROM (0) TO  (10000);
db1=# CREATE TABLE partid_2 PARTITION OF partid FOR
        VALUES FROM (10000) TO  (20000);

（親テーブル、子テーブルに問題なく INSERT できる）
db1=# INSERT INTO partid (i1) VALUES (1);
db1=# INSERT INTO partid_1 (i1) VALUES (2);

db1=# SELECT * FROM partid;
```

```
 i1 | i2
----+----
  1 |  1
  2 |  2
(2 rows)
```

PostgreSQL 16 では同様に実行すると、IDENTITY 列を含むパーティションテーブルの定義自体は実行できますが、パーティション（子テーブル）に INSERT した時点で、以下のようにエラーが発生していました。

```
（PostgreSQL 16 ではここでエラーが発生）
db1=# INSERT INTO partid_1 (i1) VALUES (2);
ERROR:  null value in column "i2" of relation "partid_1" violates not-null
constraint
DETAIL:  Failing row contains (2, null).
```

4.5. クライアント機能

libpq ライブラリと psql コマンドでいくつか拡張と改善が行われました。ここでは libpq の重要と考えられる改善と機能追加のみを取り上げます。

4.5.1. libpq で非同期処理でのクエリキャンセル

クエリキャンセルをするための API 関数群が追加されました。

これまでも PQcancel（および PQgetCancel、PQfreeCancel）がありましたが、これは、ブロッキング IO を用いた実装のみとなっており、非同期処理での動作はできませんでした。また、クエリキャンセルのための接続は暗号化されないという制限がありました。

追加される API 関数は以下の通りです。

関数名	説明
PQcancelCreate	キャンセル要求用の接続オブジェクトを作成
PQcancelBlocking	ブロッキング IO にてキャンセル要求を行う
PQcancelStart	ノンブロッキング IO にてキャンセル要求を行う
PQcancelPoll	ノンブロッキング IO でのキャンセル要求後の応答確認
PQcancelSocket	キャンセル要求した接続のファイル識別子を取得

関数名	説明
PQcancelStatus	ノンブロッキング IO でのキャンセル要求応答の内容確認
PQcancelFinish	キャンセル要求の接続を閉じる
PQcancelReset	キャンセル要求用の接続オブジェクトを再利用できるようにリセット
PQcancelErrorMessage	キャンセル要求のエラーメッセージを取得

新たな API 関数でキャンセル要求した場合には、元の接続が暗号化されたものであれば、キャンセル要求も暗号化されます。これら API 関数の使い方の例は、 src/test/modules/libpq_pipeline/libpq_pipeline.c に追加されています。

従来の PQcancel （および PQgetCancel、 PQfreeCancel）は非推奨の扱いとなりました。

4.5.2. libpq で PQchangePassword API 追加

データベースユーザのパスワードを変更するための専用 API 関数 PQchangePassword が追加されました。

これまでは、SQL 文「ALTER USER ... PASSWORD ..」を使う必要がありましたが、これを API 関数で置き換えできます。

PQchangePassword は平文で与えられたパスワード文字列をクライアント側でハッシュ化したパスワードを生成したうえで「ALTER USER .. PASSWORD ..」をサーバに投げます。これにより、実行される SQL 文に平文のパスワードが現れるのを防ぎます。psql コマンドのメタコマンド \password では以前からそのような実装となっていましたが、本 API 関数により、同様の動作をする安全なクライアントプログラムを容易に実装できるようになりました。

以下にクライアントプログラム例を示します。

```
/*
  chgdbpwd.c   《パスワード変更を行うクライアントプログラム》
  build: gcc -I $PGHOME/include -lpq -L $PGHOME/lib -o chgdbpwd chgdbpwd.c
  usage: ./chgdbpwd "dbname=postgres user=postgres" "user1" "passwd1"
*/
#include "libpq-fe.h"
/*
  argv[1] connection string
  argv[2] user name to chage password
  argv[3] new password
 */
```

```
int main(int argc, char **argv)
{
  PGconn* con;
  if (argc != 4) return 1;
  if (PQstatus(con = PQconnectdb(argv[1])) != CONNECTION_OK)
  {
    fprintf(stderr, "PQconnectdb failed: %s\n", PQerrorMessage(con));
    return 2;
  }
  if (PQresultStatus(PQchangePassword(con,argv[2],argv[3])) != PGRES_COMMAND_OK)
  {
    fprintf(stderr, "PQchangePassword failed: %s\n", PQerrorMessage(con));
    PQfinish(con);
    return 3;
  }
  PQfinish(con);
  return 0;
}
```

4.6. 運用管理

4.6.1. インクリメンタルバックアップ

PostgreSQL 17 では、インクリメンタルバックアップ（増分バックアップ）が利用できるようになりました。インクリメンタルバックアップは前回取得したベースバックアップとの差分のみを取得する機能です。これにより、少ないストレージ容量で複数世代のベースバックアップを保持することができます。

インクリメンタルバックアップは pg_basebackup コマンドの機能拡張として実装されており、その実現には新たに導入された WAL サマライズ機能を利用します。そのため、インクリメンタルバックアップを使用するには WAL サマライズを有効にする必要があり、wal_level 設定も replica（デフォルト）以上でなければなりません。

WAL サマライズ機能は、データベースクラスタ下の pg_wal/summaries/ ディレクトリに、WAL ファイル内容のサマリ情報（どのリレーション・ブロックを変更するか）を報告するバイナリの .summary ファイルを継続的に出力するものです。中身を人間が読むための pg_walsummary コマンドも提供されます。

以下にインクリメンタルバックアップを取得する手順を示します。

```
（postgresql.conf を編集し summarization_wal パラメータを on にして、reload で反映）
$ vi $PGDATA/postgresql.conf
   summarize_wal = on
$ pg_ctl reload

（テスト用テーブル作成）
$ psql db1
db1=# CREATE TABLE t461 (v1 bigint, v2 text);
db1=# \q

（WAL サマリファイルの出力を確認）
$ ls $PGDATA/pg_wal/summaries/
000000010000000000100002800000000010B1D38.summary
000000010000000000010B1D38000000000152F750.summary
000000010000000000152F750000000000152F850.summary
000000010000000000152F8500000000015361F8.summary
000000010000000000015361F800000000015362F8.summary
000000010000000000015362F800000000001536630.summary

（pg_basebackup で通常のベースバックアップを取得）
$ pg_basebackup -D backup_b -c fast
                                        《-c fast はチェックポイント待ちを避けるため》
（ここでテスト用テーブルにデータ投入）
$ psql db1
db1=# INSERT INTO t461 VALUES (generate_series(1, 100000), 'data');
=# \q

（前回バックアップのマニフェストファイルを指定してインクリメンタルバックアップを取得）
$ pg_basebackup -D backup_1 --incremental=backup_b/backup_manifest -c fast
```

インクリメンタルバックアップではどのブロックをバックアップする必要があるかを判断するために、pg_wal/summaries ディレクトリ内の WAL サマリを使用します。この時、今回バックアップの開始時点直後の WAL サマリも検索するため、必要なサマリファイルが未だ作られていない可能性があり、その場合にはファイルが作られるまで待機します。実運用システムの大きなデータのバックアップであれば、

（バックアップ中にサマリファイルが作られるので）この待機を要することはないと思われます。

また、前回バックアップ時のサマリファイルがすでに削除されている場合にはインクリメンタルバックアップ取得は失敗します。wal_summary_keep_time 設定（デフォルト 10d）を、前回バックアップとの間隔より長い期間に指定する必要があります。

--incremental オプションに前回バックアップのマニフェストファイルを指定します。その前回バックアップはフルバックアップ（非インクリメンタルバックアップ）である必要もないですし、直近に取得したバックアップアップである必要もありません。何であれ、指定した前回バックアップに対する変更分がバックアップ取得されます。

以下のように取得した内容を確認しました。

（取得した backup_1 ディレクトリ構成 - 通常のベースバックアップと変らない）

```
$ ls backup_1
PG_VERSION          pg_commit_ts    pg_replslot     pg_twophase
backup_label        pg_dynshmem     pg_serial       pg_wal
backup_manifest     pg_hba.conf     pg_snapshots    pg_xact
base                pg_ident.conf   pg_stat         postgresql.auto.conf
current_logfiles    pg_logical      pg_stat_tmp     postgresql.conf
global              pg_multixact    pg_subtrans
log                 pg_notify       pg_tblspc
```

（base 以下を見るとインクリメンタルバックアップ特有のファイルがある）

```
$ ls backup_1/base/16384
1247_fsm    4157                    INCREMENTAL.2650      INCREMENTAL.3380
1249_fsm    4159                    INCREMENTAL.2651      INCREMENTAL.3394
1255_fsm    4163                    INCREMENTAL.2652      INCREMENTAL.3394_vm
1259_fsm    4165                    INCREMENTAL.2653      INCREMENTAL.3395
```

《後略》

（バックアップのサイズははっきり異なる）

```
$ du -sh backup_b
90M     backup_b
$ du -sh backup_1
61M     backup_1
```

以上のようにインクリメンタルバックアップの取得ができました。リカバリ方法は次節に記載します。

4.6.2. pg_combinebackup コマンド

PostgreSQL 17 でインクリメンタルバックアップ（増分バックアップ）の取得ができるようになりましたが、リカバリには一貫した一つのベースバックアップが必要です。PostgreSQL 17 では、完全なベースバックアップにインクリメンタルバックアップを統合する pg_combinebackup コマンドが追加されました。

pg_combinebackup コマンドには、完全なベースバックアップとインクリメンタルバックアップが格納されたディレクトリを指定し、 --output オプションに出力先のディレクトリを指定します。インクリメンタルバックアップのディレクトリは複数指定することも可能です。

```
（pg_combinebackup で統合されたベースバックアップを作る）
$ pg_combinebackup backup_b backup_1 --output=backup_c

（リカバリ起動を確認）
$ pg_ctl stop -D $PGDATA
$ pg_ctl start -D ./backup_c

（インクリメンタルバックアップのみに含まれる行挿入が反映されている）
$ psql db1
=# SELECT count(1) FROM t461;
 count
--------
 100000
(1 row)

（検証後、PostgreSQL サービスを元に戻しておく）
$ pg_ctl stop -D ./backup_c
$ pg_ctl start -D $PGDATA
```

以上でインクリメンタルバックアップを使ってリカバリができました。

統合されたベースバックアップは単純に pg_basebackup を使用して取得した完全なベースバックアップと違いはありません。したがって、その後はリカバリ手順に特別な点はありません。

◆ *pg_combinebackup の高速化オプション*

pg_combinebackup は統合した結果を別ディレクトリに作成するため、ファイルコピーの負荷がかかります。これを軽減する以下の表に示すオプションが用意されています。

オプション	説明	所要時間例 （合計 150MB）
--copy	デフォルト。ファイルをコピーする。	2.11 秒
--copy-file-range	copy_file_range システムコールを使ったファイル部分コピーを行う。Linux ではカーネル 4.5 以降で利用可能。	0.60 秒
--clone	Reflink を機能を使う。Linux ではカーネル 4.5 以降で、Btrfs か、作成時に reflink=1 指定された XFS で利用可能	0.55 秒

所要時間例の列は本検証データ（元のベースバックアップ 90MB、差分 60MB）での pg_combinebackup の所要時間です。--copy-file-range または --clone を指定することで短縮されています。多くの Linux ディストリビューションの mkfs.xfs コマンドでは reflink=1 はデフォルトではないことに注意が必要です。

17beta2 では Linux で「--clone」オプションが使用できなかったため、本計測は 17beta3 で行いました。

4.6.3. pg_dump の --filter オプション

pg_dump に --filter オプションが追加されました。

このオプションを利用すると、ダンプファイルに出力するオブジェクトを選択、あるいは除外することがまとめて指定できるようになります。指定はファイルに書いて、そのファイルをオプションに与えます。

PostgreSQL 16 までは、-t（--table）オプションや、-n（--schema）オプションなど、コマンドラインで指定していましたが、本機能の追加でこれを一括でできるようになりました。

--filter オプションで指定するファイルの書式は以下の通りです。これを複数行連ねることができます。

```
{include か exclude}　　{オブジェクトの種類}　　{パターン}
```

include と exclude はダンプに含めるか、含めないかを指定するものです。exclude だけ指定した場合には、全てを含めるという条件が冒頭に加わっていて、そこから除外するものを指定していると解釈されます。

指定できるオブジェクトの種類は以下の通りです。

種類	説明	該当するコマンドラインオプション
extension	拡張	--[exclude-]extension （PostgreSQL 17 で追加された）
foreign_data	外部テーブル	--include-foreign-data

種類	説明	該当するコマンドラインオプション
table	テーブル （シーケンスやビューも指定可）	--[exclude-]table
table_and_children	テーブルと子テーブル	--[exclude-]table-and-children
table_data	テーブルのデータ	--exclude-table-data
table_data_and_children	テーブルと子テーブルのデータ	--exclude-table-data-and-children
schema	スキーマ	--[exclude-]schema

パターンは psql の \d メタコマンドで指定できるものと同じです。「*.*」で全てのスキーマ名とオブジェクト名の組み合わせをあらわし、「abc*」で abc で始まる名前、「*xyz」で xyz で終わる名前をあらわします。スキーマ名部分が省略されていれば search_path 設定に基づいてスキーマ名が決定されます。

以下に実行例を示します。

```
（テスト用にデータベース db463 を作り、テーブルを 4 つ作成）
$ createdb db463
$ psql db463
db463=# CREATE TABLE data1 (id int);
db463=# CREATE TABLE fruits (id int);
db463=# CREATE TABLE test1 (id int);
db463=# CREATE TABLE test2 (id int);
db463=# \dt
        List of relations
 Schema |  Name  | Type  |  Owner
--------+--------+-------+----------
 public | data1  | table | postgres
 public | fruits | table | postgres
 public | test1  | table | postgres
 public | test2  | table | postgres
(4 rows)
db463=# \q

（フィルタファイル filter.txt を作成）
$ cat > filter1.txt <<EOS
include table data*
```

```
include table *1
exclude table public.test2
EOS

（フィルタファイルを指定して pg_dump を実行）
$ pg_dump --filter filter1.txt db463 > db463_1.txt
$ grep "CREATE TABLE" db463_1.txt
CREATE TABLE public.data1 (
CREATE TABLE public.test1 (
      《→ data1 と test1 の ２つのテーブルの内容が出力された》

（exclude だけ指定する場合）
$ cat > filter2.txt <<EOS
exclude table data1
EOS

$ pg_dump --filter filter2.txt db463 > db463_2.txt
$ grep "CREATE TABLE" db463_2.txt
CREATE TABLE public.fruits (
CREATE TABLE public.test1 (
CREATE TABLE public.test2 (
      《→ exclude だけ指定する場合、全部を含むとしたうえで除外を指定した扱いになる》
```

本機能により、これまでのオプション指定のみでは表現が難しかったダンプ対象オブジェクトの指定が容易に実現できるようになりました。

4.6.4. 新たなモニタリングビュー *pg_stat_checkpointer*、*pg_wait_events*

PostgreSQL サーバの活動状態を見るための新たなビューが追加されました。

◆ *pg_stat_checkpointer*

本ビューはチェックポイントに関する稼働統計を報告します。これまで pg_stat_bgwriter ビューに含まれていた列がこちらに移動しています（「5.4.1. ◆ pg_stat_bgwriter の一部の列が削除」参照）。

以下に実行例と各項目の意味を記載します。

```
db1=# SELECT * FROM pg_stat_checkpointer;
-[ RECORD 1 ]-------+------------------------------
num_timed           | 361        《タイムアウトによるチェックポイント実行回数》
num_requested       | 8          《WAL 量によるチェックポイント実行回数》
restartpoints_timed | 0          《タイムアウト等によるリスタートポイント予定数》
restartpoints_req   | 0          《WAL 量によるリスタートポイント要求数》
restartpoints_done  | 0          《リスタートポイント実行回数》
write_time          | 462102     《ファイル書き込みの所要時間累計（ms）》
sync_time           | 1062       《ストレージ同期の所要時間累計（ms）》
buffers_written     | 15823      《書き込みバッファ数累計（ページ）》
stats_reset         | 2024-08-13 14:11:25.131205+09
```

リスタートポイントに関する項目はストリーミングレプリケーションのスタンバイサーバで使われます。書き込みバッファ数累計はチェックポイント（またはリスタートポイント）処理で書き込まれたバッファ数を意味します。SQL を実行するバックエンドプロセス自身で書き出されたバッファ数は含まれません。

◆ *pg_wait_event*

待機イベントのタイプとイベント名の一覧を報告するビューです。description 列に待機イベントの説明も出力されます。待機イベントは、pg_stat_activity ビューの wait_event_type、 wait_event 列に現れます。

```
db1=# SELECT * FROM pg_wait_events;
-[ RECORD 1 ]----------------------------------------------------
type        | Activity
name        | ArchiverMain
description | Waiting in main loop of archiver process
-[ RECORD 2 ]----------------------------------------------------
type        | Activity
name        | AutovacuumMain
description | Waiting in main loop of autovacuum launcher process
-[ RECORD 3 ]----------------------------------------------------
type        | Activity
name        | BgwriterHibernate
description | Waiting in background writer process, hibernating
    《後略》
```

4.6.5. 新たな定義済ロール *pg_maintain* と *MAINTAIN* 権限

新たな定義済ロールとして pg_maintain が用意されました。また、関連して、GRANT／REVOKE 文で付与・剥奪する権限 MAINTAIN が新たに追加されました。

以下に MAINTAIN 権限の使用例を示します。

（MAINTAIN 権限の使用例）

```
db1=# CREATE USER bar;
db1=# CREATE TABLE t464 (id int);
db1=# INSERT INTO t464 VALUES (1);
db1=# GRANT MAINTAIN ON t464 TO bar;    《 bar に t464 の MAINTAIN 権限を付与》
db1=# \c db1 bar                        《ユーザ bar で接続し直す》
You are now connected to database "db1" as user "bar".
db1=> SELECT * FROM t464;
ERROR:  permission denied for table t464      《参照はできない》
db1=> VACUUM t464;
VACUUM                                        《VACUUM はできる》
db1=> VACUUM pg_class;
WARNING:  permission denied to vacuum "pg_class", skipping it
VACUUM                          《権限のないテーブルの VACUUM はできない》
```

pg_maintain ロールに属するユーザは、任意のテーブルについて MAINTAIN 権限を持っているかのように、VACUUM、ANALYZE、CLUSTER、REFRESH MATERIALIZED VIEW、REINDEX、LOCK TABLE が実行できます。

以下に使用例を示します。

（pg_maintain ロールの使用例）

```
db1=# GRANT pg_maintain TO bar;
db1=# \c db1 bar
You are now connected to database "db1" as user "bar".
db1=> VACUUM pg_class;
VACUUM
```

pg_maintain ロールに所属させるた bar ロールで、特に MAINTAIN 権限を付与していないシステムテーブル pg_class を VACUUM できていることが確認できました。

5. 非互換変更

PostgreSQL 17 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。

5.1. メンテナンス操作時の search_path

メンテナンス操作中に安全な search_path を使用するように関数が変更されました。これによりメンテナンス操作が安全でないアクセスを実行することがなくなります。

メンテナンス操作とは VACUUM 、ANALYZE、CLUSTER、REFRESH MATERIALIZED VIEW、REINDEX、および、CREATE INDEX を指します。安全でないアクセスとは、これらの処理の中で実行される SQL や関数において、想定外のデータベースオブジェクトを参照させることを指します。

以下に例を示します。実行時の search_path はデフォルトの「"$user", public」であるものとします。

```
（search_pathによる別関数実行を試みる - PostgreSQL 16.3）
db1=# CREATE USER foo;
db1=# CREATE SCHEMA foo;
db1=# ALTER SCHEMA foo OWNER TO foo;
db1=# CREATE TABLE t51 (id int PRIMARY KEY, txt text);
db1=# ALTER TABLE t51 OWNER TO foo;
db1=# CREATE FUNCTION f51(text) RETURNS text IMMUTABLE LANGUAGE sql
        AS $$ SELECT md5($1); $$;
db1=# CREATE FUNCTION my_md5(text) RETURNS text IMMUTABLE LANGUAGE sql
        AS $$ SELECT f51($1); $$;
db1=# CREATE INDEX ON t51 (my_md5(txt));
                《ここまでのテーブルや関数は publicスキーマ上に作られている》
db1=# \c db1 foo
You are now connected to database "db1" as user "foo".

db1=> INSERT INTO t51 VALUES (1, 'ABC'), (2, 'CDE');
db1=> SELECT *, my_md5(txt) FROM t51 WHERE my_md5(txt) ~ '^9';
 id | txt |                f51
----+-----+----------------------------------
  1 | ABC | 902fbdd2b1df0c4f70b4a5d23525e932
(1 row)
```

```
（search_path = "$user", public で優先選択される foo.f51(text) 関数を作成）
db1=> CREATE FUNCTION foo.f51(text) RETURNS text IMMUTABLE LANGUAGE plpgsql
        AS $$ BEGIN RAISE WARNING 'malice action'; RETURN md5($1); END; $$;

db1=> SELECT * FROM t51 WHERE my_md5(txt) ~ '^9';
WARNING:  malice action
WARNING:  malice action
 id | txt
----+-----
  1 | ABC
(1 row)              《my_md5 関数から使われる関数が foo.f51 に置き換えられている》

db1=> VACUUM ANALYZE t51;
WARNING:  malice action
WARNING:  malice action
VACUUM               《VACUUM ANALYZE で foo.f51 が実行される》

db1=> REINDEX TABLE t51;
WARNING:  malice action
WARNING:  malice action
REINDEX              《REINDEX でも foo.f51 が実行される》
```

これを 17beta2 で実行すると、最初に関数インデックスを作る時点でエラーになります。このときの search_path は「pg_catalog, pg_temp」に暗黙に置き換えられていて、それ以外のスキーマに属するオブジェクトは見つからないというエラーが発生します。回避するにはインデックスで使う関数の定義で明示的に search_path を指定するか、完全修飾されたオブジェクト名を指定する必要があります。

以下に動作例を示します。

```
（search_path により別関数実行を試みる - 17beta2）
   《CREATE USER .. から CREATE FUNCTION .. までを省略》
db1=# CREATE INDEX ON t51 (my_md5(txt));
ERROR:  function f51(text) does not exist
LINE 1:  SELECT f51($1);
                ^
HINT:  No function matches the given name and argument types. You might need
to add explicit type casts.
```

```
QUERY:   SELECT f51($1);
CONTEXT:  SQL function "my_md5" during inlining

（my_md5 関数を SET search_path を付けた定義に置き換えて、エラー回避）
db1=# CREATE OR REPLACE FUNCTION my_md5(text) RETURNS text IMMUTABLE
        SET search_path = 'public' LANGUAGE sql AS $$ SELECT f51($1); $$;
CREATE FUNCTION

db1=# CREATE INDEX ON t51 (my_md5(txt));
CREATE INDEX

    《明示的に public.f51 が指定されたので以降に foo.f51 を作っても影響を受けない》
```

この変更により、式インデックスやマテリアライズドビューで使用する関数で、システムスキーマに属さないオブジェクトを参照するものは、完全修飾したオブジェクト名を使うか、関数定義で search_path を指定する必要があります。

5.2. *interval* 型の表現文字列の厳格化

interval 型の表現文字列における「ago」の使用可能箇所が限定されるようになりました。表現文字列の最後に 1 つだけ記述可能となります。また、空の間隔単位がいくつか現れることも禁止されました。

以下に例を示します。

```
（PostgreSQL 16.x 以前では受け入れられた interval 型文字列表現）
db1=# SELECT '10 minute ago 3 second ago'::interval;
 interval
-----------
 -00:10:03
(1 row)

db1=# SELECT '10 day  month  year'::interval;
 interval
----------
 10 days
(1 row)
```

```
（PostgreSQL 17 からはエラーになる）
db1=# SELECT '10 minute ago 3 second ago'::interval;
ERROR:  invalid input syntax for type interval: "10 minute ago 3 second ago"

db1=# SELECT '10 day  month  year'::interval;
ERROR:  invalid input syntax for type interval: "10 day  month  year"
```

5.3. SET SESSION AUTHORIZATION 仕様変更

セッションユーザ識別子を変更する SET SESSION AUTHORIZATION の動作が変更されました。

これまでは接続したときにスーパーユーザであった場合にセッションユーザ識別子を変更できました。これからは SET SESSION AUTHORIZATION を実行したときにスーパーユーザである場合に、セッションユーザ識別子を変更できます。

この変更で以下のような操作ができなくなるわけではありません。2 回目以降の SET SESSION AUTHORIZATION 実行時のセッションユーザ識別子は u1 や u2 で非スーパーユーザですが、最初の実行時にスーパーユーザであったことが記憶されていて、スーパーユーザであるものと判断されます。

```
（この動作は PostgreSQL 16.x 以前でも、PostgreSQL 17 でも可能）
$ psql -d db1 -U postgres -c 'CREATE USER u1'
$ psql -d db1 -U postgres -c 'CREATE USER u2'
$ psql -d db1 -U postgres
db1=# SET SESSION AUTHORIZATION u1;
db1=> SET SESSION AUTHORIZATION u2;
db1=> SET SESSION AUTHORIZATION postgres;
db1=#
```

挙動に違いがでるのは、接続後にユーザのスーパーユーザ属性が剥奪された場合です。

以下に例を示します。

```
（接続後にスーパーユーザ権限を取り除いた場合の動作 - 17beta2）
$ psql -d db1 -U postgres -c 'CREATE USER u3 SUPERUSER'
$ psql -d db1 -U u3
db1=# ALTER USER u3 NOSUPERUSER;
db1=# SET SESSION AUTHORIZATION postgres;
```

```
ERROR:  permission denied to set session authorization "postgres"
```

《これは 16.x 以前ではエラーにならずセッションユーザ変更ができてしまう》

この例ではセッション内で「ALTER USER u3 NOSUPERUSER;」を実行していますが、別の接続からスーパーユーザ属性を剥奪された場合でも同じ結果になります。

従来の動作ですと、ずっと維持されている接続では、最初の接続時のユーザからスーパーユーザ属性を剥奪されてもその影響をまぬがれることができて、望ましくありませんでした。

5.4. WAL ファイル名関数の仕様変更

WAL ファイル名を返すシステム関数の仕様が変更されました。

2 つの WAL ファイル名関数、pg_walfile_name() および pg_walfile_name_offset() は、LSN （Log Sequence Number）がファイルセグメント境界上にある場合は、一つ前の LSN セグメント番号を報告していましたが、現在の LSN セグメントを返すようになりました。

以下の手順で動作の違いを比較します。

（pg_walfile_name_offset 関数の実行例 - PG16、PG17 共通の結果）

```
db1=# SELECT segment_number, file_offset FROM
        pg_walfile_name_offset('0/0'::pg_lsn + 16*1024*1024 - 1),
        pg_split_walfile_name(file_name);
 segment_number | file_offset
----------------+-------------
              0 |    16777215
(1 row)
```

（pg_walfile_name_offset 関数の実行例 - PG16 の結果）

```
db1=# SELECT segment_number, file_offset FROM
        pg_walfile_name_offset('0/0'::pg_lsn + 16*1024*1024 + 0),
        pg_split_walfile_name(file_name);
 segment_number | file_offset
----------------+-------------
              0 |           0
(1 row)
```

（pg_walfile_name_offset 関数の実行例 - PG17 の結果）

```
db1=# SELECT segment_number, file_offset FROM
        pg_walfile_name_offset('0/0'::pg_lsn + 16*1024*1024 + 0),
        pg_split_walfile_name(file_name);
 segment_number | file_offset
----------------+-------------
              1 |           0
(1 row)
```

（pg_walfile_name_offset 関数の実行例 - PG16、PG17 共通の結果）

```
db1=# SELECT segment_number, file_offset FROM
        pg_walfile_name_offset('0/0'::pg_lsn + 16*1024*1024 + 1),
        pg_split_walfile_name(file_name);
 segment_number | file_offset
----------------+-------------
              1 |           1
(1 row)
```

この SQL 実行結果の segment_number 列は pg_walfile_name_offset 関数から出力されるファイル名のセグメント番号部分を pg_split_walfile_name 関数で切り出したものです。

LSN が WAL セグメントファイルの境界を越えて 1 ずつ増えていくときの segment_number と file_offset の組み合わせを見ると、PostgreSQL 17 の方が自然な値と言えます。PostgreSQL 17 の出力では 、segment_number を上位桁、file_offset を下位桁とすれば単調増加しています。

これまで 1 つ手前の WAL セグメント番号を持つファイル名を報告していたのは、奇妙な振る舞いですが、ドキュメント記載もされている仕様です。現在の LSN に対してその内容の WAL アーカイブ状況を確認するときに見るべき WAL ファイル名を把握するという用途に寄せて設計されていました。

5.5 システムテーブル／ビューの変更

PosstgreSQL 17 では、システムテーブル、システムビューにいくつか変更が加えられました。なお、列が追加されたり、新たなテーブル・ビューが追加された変更はここには含めていませんが、非互換変更のあるテーブル・ビューにおいて列追加もある場合には記載しています。

◆pg_stat_bgwriter の一部の列が削除

pg_stat_bgwriter の一部の列が削除されました。システムカタログの pg_stat_bgwriter ビューから buffers_backend 列と buffers_backend_fsync 列が削除されました。これらのフィールドは、pg_stat_io ビューの同様の列と重複していると考えられたためです。また、 checkpoints_timed、 checkpoints_req、checkpoint_write_time、checkpoint_sync_time、buffers_checkpoint は、pg_stat_checkpointer ビューに移されました。

変更前（１６まで）	変更後（１７から）
buffers_backend	削除
buffers_backend_fsync	削除
checkpoints_timed	削除（pg_stat_checkpointer の num_timed に移動）
checkpoints_req	削除（pg_stat_checkpointer の num_requested に移動）
checkpoint_write_time	削除（pg_stat_checkpointer の write_time に移動）
checkpoint_sync_time	削除（pg_stat_checkpointer の sync_time に移動）
buffers_checkpoint	削除（pg_stat_checkpointer の buffers_written に移動）

◆pg_stat_statements の列名の一部が変更

pg_stat_statements ビューの I/O ブロック読み取り/書き込みタイミングに関する、統計列の名前が変更されました。新たに local_blk_read_time、local_blk_write_time が加わっていますが、16.x バージョンにおける blk_read_time、blk_write_time と同じ意味なのは shared_blk_read_time、shared_blk_write_time です。17.x の pg_stat_statements ビューでは他にも機能追加として幾つか列が追加されています。

変更前（１６まで）	変更後（１７から）
blk_read_time	shared_blk_read_time
blk_write_time	shared_blk_write_time
なし（１７で追加）	local_blk_read_time
なし（１７で追加）	local_blk_write_time
なし（１７で追加）	jit_deform_count
なし（１７で追加）	jit_deform_time
なし（１７で追加）	jit_deform_time
なし（１７で追加）	minmax_stats_since

◆ *pg_stat_progress_vacuum 列名の一部変更と追加*

pg_stat_progress_vacuum ビューの max_dead_tuples 列の列名が max_dead_tuple_bytes に 、num_dead_tuples の列名が num_dead_item_ids に変更され、また、機能追加としていくつか列が追加されました。

変更前（１６まで）	変更後（１７から）
max_dead_tuples	max_dead_tuple_bytes
num_dead_tuples	num_dead_item_ids
なし（１７で追加）	dead_tuple_bytes
なし（１７で追加）	indexes_total
なし（１７で追加）	indexes_processed

◆ *pg_stat_slru の name 列の値が変更*

システム ビュー pg_stat_slru の name 列の値が変更されました。

同時に、pg_stat_reset_slru() で受け入れられる列名も変更されました。

変更前（１６まで）	変更後（１７から）
CommitTs	commit_timestamp
MultiXactMember	multixact_member
MultiXactOffset	multixact_offset
Notify	notify
Serial	serializable
Subtrans	subtransaction
Xact	transaction
other	other（変わらず）

5.6. 設定パラメータの変更

PostgreSQL 17 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
old_snapshot_threshold	廃止されました。 本機能のために生じる障害や性能上の問題がありました。
db_user_namespace	廃止されました。 ユーザ（ロール）はインスタンスに対して作成されますが、本設定でデータベースごとのユーザをシミュレートすることができました。 本機能はほとんど使用されていなかったため廃止されました。本機能を使用していた場合の代替手段は特に提供されません。
trace_recovery_messages	廃止されました。 これを on にしたときに出ていたログメッセージは DEBUG1～DEBUG4 のメッセージとして引きつづき報告されます。
wal_sync_method	Windows で設定値 fsync_writethrough が廃止されました。 この値は Windows における fsync 指定と同じでした。

6. 免責事項

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

7. 更新履歴

1.0	2024/9/02	初期公開バージョン
1.1	2024/9/11	軽微な誤記修正
1.2	2024/9/17	軽微な誤記修正
1.3	2024/10/8	4.1.3. COPY 性能向上 の記載を置き換え
1.4	2024/10/17	4.1.2. VACUUM 性能改善 に補足記載を追加、軽微な誤記修正
1.5	2024/10/21	5.6. 設定パラメータの変更 に追記