PostgreSQL 19 検証レポート

1.0版
2026年6月30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 19 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 19 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。
2026 年 6 月 4 日にリリースされた PostgreSQL 19 beta1 を使用して検証を行いました。

2. 概要

PostgreSQL 19 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
- 各種プランナ改善
- タプルデフォーム高速化
- SIMD 利用拡大
- 外部キー制約検査の高速化
- 基数ソート

SQL 機能
- プロパティグラフ問合せ（SQL/PGQ）
- テンポラルテーブル更新（UPDATE/DELETE .. FOR PORTION OF）
- パーティションの分割／統合（ALTER TABLE .. MARGE/SPLIT PARTITIONS）
- グルーピングの拡張
- COPY コマンドの拡張
- 各種の追加関数

運用管理
- Autovacuum スコア
- 各種モニタリングビューの追加
- ログ出力の拡張
- REPACK コマンド
- バイナリ pg_dumpall

レプリケーション
- ストリーミングレプリケーションの拡張
- 論理レプリケーションの拡張

拡張モジュール

- pg_plan_advice
- pg_stash_advice
- pg_stat_statements 改善

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 19 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。リリース後は URL の「devel」が「19」になります。

https://www.postgresql.org/docs/devel/release-19.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 19（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 10.x (x86_64) 互換 OS 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、bison、perl-FindBin、zlib-devel、readline-devel 、libicu-devel、openssl-devel、libzstd-devel、lz4-devel、libuuid-devel、liburing-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、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/19 ディレクトリを用意しておきます。

また、ビルドツールとしては meson / ninja を使用しますので、最初にこれらのインストールを行います。

以下に実行コマンドを示します。

```
（meson、ninja をインストール）
# dnf install python3-pip perl-FindBin
# pip3 install meson ninja

（以下、postgres ユーザでビルドとインストールを実行）
# su - postgres
$ wget
https://ftp.postgresql.org/pub/source/v19beta1/postgresql-19beta1.tar.bz2
                        《実際は１行、リリース後は「v19.0/postgresql-19.0.tar.bz2」》
$ tar jxf postgresql-19beta1.tar.bz2
$ cd postgresql-19beta1
$ meson setup build --prefix=/usr/local/pgsql/19
$ cd build
$ ninja
$ ninja install
```

前述の事前導入するパッケージ群にはドキュメントビルドのためのパッケージを含めていないため、本導入手順では多くの場合、HTML ドキュメントのビルド・インストールは省略されるはずです。

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 ディレクトリがあるものとし、その下にデータベースクラスタディレクトリを配置します。

```
$ cat > ~/pg19.env <<'EOF'
VER=19
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

$ . ~/pg19.env
$ cat ~/pg19.env >> ~/.pgsql_profile
```

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。また、サーバログ収集を有効にする設定を与えています。

```
$ initdb --no-locale --encoding=UTF8 -c logging_collector=on
```

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 (19beta1)
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/19/share/doc/html/
```

また、以下 URL にて PostgreSQL 19 の英語ドキュメントが公開されています。リリース後は URL の「devel」部分が「19」になります。

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

4.1. 性能向上

本節では主な性能改善を取り上げます。

本節で取り上げたもの以外にも、AIO 基盤、NOTIFY コマンド、列の追加、UTF8 の大文字小文字変換、（EXPLAIN の TIMING などの）タイミング計測、PL/pgSQL インライン処理化などで、性能改善を狙った改修が行なわれています。

4.1.1. プランナ改善

本バージョンでも様々なプランナの改善が導入されました。主なものを取り上げます。これ以外にも多数の拡張や改善が行なわれています。

◆ANTI JOIN の改善

一致しないこと結合条件とする、いわゆる ANTI JOIN の問合せのプラン作成がいくつか改善されました。

NOT NULL 制約のある列での「... NOT IN (...)」を使った副問い合わせを Anti Join プラン要素で実行できるようになりました。以下に例を示します。

```
（NULL が含まれない列での NOT IN を Anti Join で処理）
db1=# CREATE TABLE t411a (id int NOT NULL, v text);
db1=# CREATE TABLE t411b (id int NOT NULL, v text);
db1=# INSERT INTO t411a SELECT g, md5(g::text) FROM generate_series(1, 1000) g;
```

```
db1=# INSERT INTO t411b SELECT g, md5(g::text) FROM generate_series(1, 500) g;
db1=# ANALYZE;

db1=# explain (costs off)
      SELECT * FROM t411a WHERE id NOT IN (SELECT id FROM t411b);
             QUERY PLAN
-----------------------------------
 Hash Anti Join
   Hash Cond: (t411a.id = t411b.id)
   ->  Seq Scan on t411a
   ->  Hash
         ->  Seq Scan on t411b
(5 rows)
```

（PostgreSQL 18.x の結果： hashed SubPlan と Filter で処理されている）

```
                      QUERY PLAN
------------------------------------------------------
 Seq Scan on t411a
   Filter: (NOT (ANY (id = (hashed SubPlan 1).col1)))
   SubPlan 1
     ->  Seq Scan on t411b
(4 rows)
```

また、より多くの場合に外部結合を ANTI JOIN と解釈できるようになりました。以下の例では、外部結合しているテーブルの NUT NULL 列に対して IS NULL 条件をつけているので、意味としては ANTI JOIN（一致しないものを選ぶ）になります。これを Anti Join のプラン要素で処理できるようになっています。

（外部結合を Anti Join で処理）

```
db1=# CREATE TABLE t411c (id int, v text);
db1=# CREATE TABLE t411d (a int NOT NULL, b int);
db1=# INSERT INTO t411c SELECT g, md5(g::text) FROM generate_series(1, 10000) g;
db1=# INSERT INTO t411d SELECT g, g FROM generate_series(1, 100) g;
db1=# ANALYZE;
db1=# explain (costs off) SELECT * FROM t411c t1
        LEFT JOIN t411d t2 ON t1.id = t2.b WHERE t2.a IS NULL;
            QUERY PLAN
```

```
---------------------------------
 Hash Anti Join
   Hash Cond: (t1.id = t2.b)
   ->  Seq Scan on t411c t1
   ->  Hash
         ->  Seq Scan on t411d t2
(5 rows)

 （PostgreSQL 18.x の結果： 外部結合と Filter で処理されている）
            QUERY PLAN
---------------------------------
 Hash Right Join
   Hash Cond: (t2.b = t1.id)
   Filter: (t2.a IS NULL)
   ->  Seq Scan on t411d t2
   ->  Hash
         ->  Seq Scan on t411c t1
(6 rows)
```

◆ *SEMI JOIN の改善*

IN 句の副問い合わせを伴う、いわゆる SEMI JOIN の問合せのプラン作成が改善されました。

SEMI JOIN では Semi Join プラン要素を使う方法の他に、副問い合わせ側から重複を除いたうえで内部結合を行う方法があります。このときのプラン選択がヒューリスティックに依存していて、その結果、不合理な実行プランが生成されることがありました。そのような場合についても、候補の方式についてコスト比較を行うように改善されました。

以下に、問題提起の際に提示されたおかしなプラン生成の例を示します。

```
（PostgreSQL 18.x でのおかしな実行プランが生じる例）
db1=# CREATE TABLE t411e (a int, b int);
db1=# INSERT INTO t411e SELECT g % 100, g FROM generate_series(1, 10000) g;
db1=# CREATE INDEX ON t411e (a);
db1=# VACUUM (ANALYZE) t411e;

db1=# explain (costs off) SELECT * FROM t411e t1 WHERE t1.a IN (
        SELECT a FROM t411e t2 WHERE a < 10) ORDER BY t1.a;
```

```
                            QUERY PLAN
------------------------------------------------------------------
 Merge Join
   Merge Cond: (t1.a = t2.a)
   ->  Index Scan using t411e_a_idx on t411e t1
   ->  Sort
         Sort Key: t2.a
         ->  HashAggregate
               Group Key: t2.a
               ->  Index Only Scan using t411e_a_idx on t411e t2
                     Index Cond: (a < 10)
(9 rows)
```

Index Only Scan をしてソートされている a 列の値をハッシュ集約（HashAggregate）して、その後、再びソートしています。これは無駄のある処理順といえます。以下のようにハッシュ集約が選択されなかったケースに限定しますと、異常さがより明確になります。

```
（PostgreSQL 18.x - ハッシュ集約を避けた場合）
db1=# SET enable_hashagg TO off;
db1=# explain (costs off) SELECT * FROM t411e t1 WHERE t1.a IN (
        SELECT a FROM t411e t2 WHERE a < 10) ORDER BY t1.a;
                            QUERY PLAN
------------------------------------------------------------------------
 Merge Join
   Merge Cond: (t1.a = t2.a)
   ->  Index Scan using t411e_a_idx on t411e t1
   ->  Sort
         Sort Key: t2.a
         ->  Unique
               ->  Sort
                     Sort Key: t2.a
                     ->  Index Only Scan using t411e_a_idx on t411e t2
                           Index Cond: (a < 10)
(10 rows)
```

上記結果では、インデックス順にスキャンした結果に対して、ソートが 2 回行なわれています。

PostgreSQL 19 では同じ手順で以下の結果が得られました。ソート済のインデックススキャン結果を Unique で一意データ化する、という無駄な処理重複の無い実行プランとなりました。

```
（PostgreSQL 19.x での結果 - enable_hashagg = on/off いずれでも）
db1=# explain (costs off) SELECT * FROM t411e t1 WHERE t1.a IN (
        SELECT a FROM t411e t2 WHERE a < 10) ORDER BY t1.a;
                        QUERY PLAN
-----------------------------------------------------------
 Merge Join
   Merge Cond: (t1.a = t2.a)
   ->  Index Scan using t411e_a_idx on t411e t1
   ->  Unique
         ->  Index Only Scan using t411e_a_idx on t411e t2
               Index Cond: (a < 10)
(6 rows)
```

◆ *Append/MergeAppend でインクリメンタルソートを組み合わせ*

UNION 句と ORDER BY 句で問い合わせ結果をソートしつつ併合するときに、下位ノードでのインクリメンタルソートを活用するプランを作れるようになりました。

以下に例を示します。

```
（a 列、b 列にインデックスがあるテーブルと a 列だけインデックスがあるテーブルを作成）
db1=# CREATE TABLE t411f (a int, b int, PRIMARY KEY (a,b));
db1=# CREATE TABLE t411g (a int, b int, PRIMARY KEY (a));
db1=# INSERT INTO t411f
        SELECT g % 1000, g / 100 FROM generate_series(1, 100000) g;
db1=# INSERT INTO t411g
        SELECT g, (random() * 1000)::int FROM generate_series(1, 10000) g;
db1=# VACUUM (ANALYZE) t411f, t411g;

（シーケンシャルスキャンにならなかった場合について実行プランを比較する）
db1=# SET enable_seqscan TO off;
db1=# explain
      SELECT a, b FROM t411f UNION ALL SELECT a, b FROM t411g ORDER BY a, b;
```

（PostgreSQL 18.x での実行プラン出力）

```
                                QUERY PLAN
---------------------------------------------------------------------------
 Merge Append  (cost=982.97..4907.97 rows=110000 width=8)
   Sort Key: t411f.a, t411f.b
   ->  Index Only Scan using t411f_pkey on t411f
       (cost=0.29..2800.29 rows=100000 width=8)
   ->  Sort  (cost=982.67..1007.67 rows=10000 width=8)
         Sort Key: t411g.a, t411g.b
         ->  Index Scan using t411g_pkey on t411g
             (cost=0.29..318.29 rows=10000 width=8)
(6 rows)
```

PostgreSQL 18.x の実行プランでは、下位ノードで t411g テーブルのインデックススキャンをした結果を再度ソートする形になっています。インデックススキャン結果は a 列についてはソート済のはずですから、本来は Incremental Sort が可能なはずです。

（PostgreSQL 19.x での実行プラン出力）

```
                                QUERY PLAN
---------------------------------------------------------------------------
 Merge Append  (cost=3.57..4621.78 rows=110000 width=8)
   Sort Key: t411f.a, t411f.b
   ->  Index Only Scan using t411f_pkey on t411f
       (cost=0.29..2796.29 rows=100000 width=8)
   ->  Incremental Sort  (cost=3.27..725.48 rows=10000 width=8)
         Sort Key: t411g.a, t411g.b
         Presorted Key: t411g.a
         ->  Index Scan using t411g_pkey on t411g
             (cost=0.29..314.29 rows=10000 width=8)
(7 rows)
```

これに対して、PostgreSQL 19.x の実行プランでは、下位ノードで一部についてソート済であることを活かして、Incremental Sort が採用されています。本例のコスト値も少なくなっています。

◆ *集約と結合に対するプランナ作成の改善*

テーブル結合した結果をグルーピング集約する問合せについて、結合に先だって集約処理を行なえるようになりました。

この振る舞いは新たな設定パラメータ enable_eager_aggregate で有効・無効になります。デフォルトは on（有効）です。また、新たな設定パラメータ min_eager_agg_group_size でこの最適化を行うためのグループのサイズ、すなわち GROUP BY で選り分けた各グループに属する平均の行数、の閾値を設定します。デフォルトは 8 です。グルーピングをした後であれば行数が 8 分の 1 以下に減る見込みがなければ、この最適化は行なわないという意味になります。

以下に例を示します。

（各行にグループ ID を持つテーブルと、グループ毎の行を持つテーブルを作成する）

```
db1=# CREATE TABLE t411h (id int PRIMARY KEY, gid int, v text);
db1=# CREATE TABLE t411i (gid int PRIMARY KEY, v text);
db1=# INSERT INTO t411h
        SELECT g, g % 100, '' FROM generate_series(1, 10000) g;
db1=# INSERT INTO t411i SELECT g, '' FROM generate_series(1, 100) g;
db1=# VACUUM (ANALYZE) t411h, t411i;
```

（結合と集約を伴う問い合わせの実行プランを出力）

```
db1=# explain SELECT gid, count(1) FROM t411h h JOIN t411i i USING (gid)
        GROUP BY gid;
                                QUERY PLAN
---------------------------------------------------------------------------
 Finalize HashAggregate  (cost=210.02..211.02 rows=100 width=12)
   Group Key: h.gid
   ->  Hash Join  (cost=208.25..209.52 rows=100 width=12)
         Hash Cond: (h.gid = i.gid)
         ->  Partial HashAggregate  (cost=205.00..206.00 rows=100 width=12)
               Group Key: h.gid
               ->  Seq Scan on t411h h (cost=0.00..155.00 rows=10000 width=4)
         ->  Hash  (cost=2.00..2.00 rows=100 width=4)
               ->  Seq Scan on t411i i  (cost=0.00..2.00 rows=100 width=4)
(9 rows)
```

上記実行プランでは「Partial HashAggregate」という新たなプラン要素が使われていて、先に t411h テーブルのグルーピングが行なわれています。そのうえで、t411i テーブルと結合して、最後に新たなプラン要素

「Finalize HashAggregate」で最終調整を行なっています。

（本最適化を無効にした場合の実行プラン）

```
db1=# SET enable_eager_aggregate TO off;
db1=# explain SELECT gid, count(1) FROM t411h h JOIN t411i i USING (gid)
        GROUP BY gid;
                                QUERY PLAN
-------------------------------------------------------------------------
 HashAggregate  (cost=235.61..236.61 rows=100 width=12)
   Group Key: h.gid
   ->  Hash Join  (cost=3.25..185.61 rows=10000 width=4)
         Hash Cond: (h.gid = i.gid)
         ->  Seq Scan on t411h h  (cost=0.00..155.00 rows=10000 width=4)
         ->  Hash  (cost=2.00..2.00 rows=100 width=4)
               ->  Seq Scan on t411i i  (cost=0.00..2.00 rows=100 width=4)
(7 rows)
```

本最適化を行なっていない場合、問合せの意味通りに、テーブルを結合して、その結果をグループピング集約するという実行プランになります。本例は Hash Join が低コストなのでコストに大きな差はありませんが、テーブル結合にコストがかかるケースですと、これは効果的な最適化になります。

◆ *拡張統計情報のリストア*

pg_restore_extended_stats() 関数で拡張統計情報のリストアも可能になりました。

これはプランナの改修ではありませんが、関連する項目であるため、本節に記載しておきます。

pg_dump でプランナ統計情報を含むダンプを取得すれば、本関数を使った拡張統計情報をリストアするコマンドも含まれるようになります。これにより、バックアップしておいた所定の統計情報で SQL を実行する手法が拡張統計情報に対しても使えることになります。

4.1.2. タプルデフォーム高速化

テーブルに格納された行データを読み出して、問合せの処理で使える形式のメモリ上のデータに変換することをタプルデフォームと呼びます。各種の問合せ実行の中で暗黙的に行われています。PostgreSQL 19 ではタプルデフォームの処理が効率化されました。

列数が多いテーブルに対する、多数の行をスキャンして後ろの方まで列の値を調べる必要のある SELECT の実行には、タプルデフォームの処理時間が大きく影響を与えます。そのような SELECT 文の所要時間が改

善しているかを以下のように確認しました。

（id 列と 100 列を持つテーブルを作成）

```
db1=# SELECT 'CREATE TABLE t412 (id int not null' || string_agg(col, ' ')
        || ');' FROM (SELECT ',c' || i || ' int not null' AS col
          FROM generate_series(1, 100) i) s1 \gexec
```

（各列に乱数値を持つ 10 万行を挿入）

```
db1=# SELECT 'INSERT INTO t412 SELECT g,' || string_agg(val, ',')
        || ' FROM generate_series(1, 100000) g;'
        FROM (SELECT (random()*100::int)::text AS val
          FROM generate_series(1, 100) i) s1 \gexec
db1=# VACUUM (ANALYZE,FREEZE) t412;
```

（後ろの方の列の値を集計する問合せを実行して、所要時間を計測）

```
db1=# \timing on
db1=# SELECT sum(c99) FROM t412;
   sum
---------
 6100000
(1 row)

Time: 63.009 ms
```

→ 15 回実行して、キャッシュヒット率が向上しているであろう最後の 5 回を計測値とする

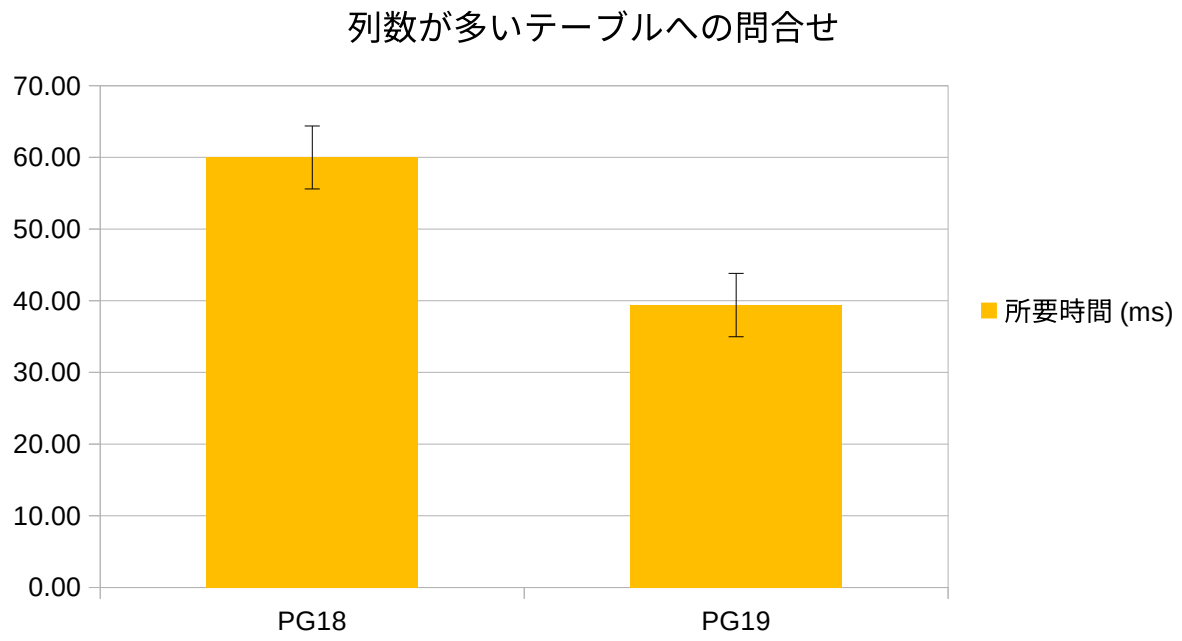
id 列と 100 個の整数列を持つテーブルを作り、99 番目の値の合計を求める問合せを実行しています。機器からのセンサーデータを採取・集計する場合には十分に有り得るテーブル・問合せです。

この手順を PostgreSQL 19beta1 と 18.4 バージョンで行なった結果を以下のグラフに示します。誤差線は標準偏差を上下に延ばしたものです。

PostgreSQL 19 で性能がはっきり改善していることが分かります。所要時間を 34%減らしています。

4.1.3. SIMD 利用拡大

一連のデータを 1 つの命令で一括処理する、CPU の SIMD（Single Instruction Multiple Data）命令の適用個



所が追加されました。text および CSV 形式からの COPY FROM と、バイナリ文字列を hex 形式でエンコード／デコードする関数で使用されるようになりました。

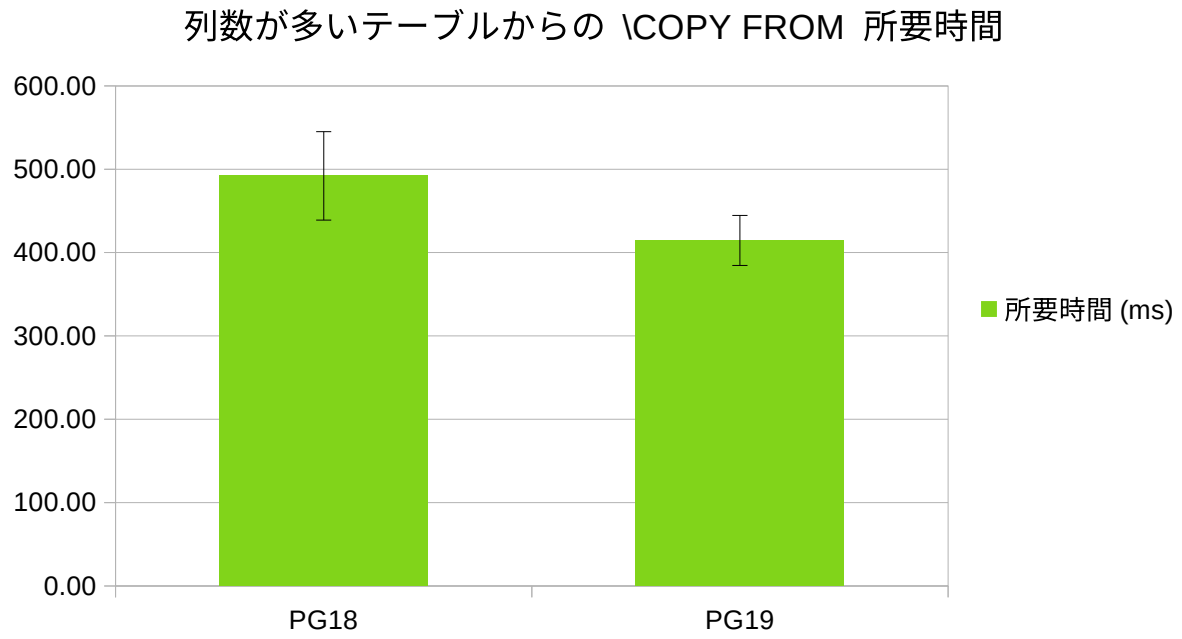
COPY FROM の性能を前節で作成した、101 個の列をもつ 10 万行の t412 テーブルを使って、以下のように比較・測定しました。

```
（テーブル t412 を CSV ファイルに保存）
db1=# \copy t412 TO t412.csv WITH (FORMAT CSV)

（テーブルの TRUNCATE と \COPY FROM を繰り返し実行）
db1=# TRUNCATE t412;
TRUNCATE TABLE
db1=# \timing on
db1=# \copy t412 FROM t412.csv WITH (FORMAT CSV)
COPY 100000
Time: 492.912 ms
  → 以下、TRUNCATE と \copy .. FROM を繰り返し実行
```

psql の \copy コマンドは内部的には COPY コマンドが使われていますので、これにより COPY FROM の性能を測ることができます。初めに CSV ファイルを作って、CSV ファイルからデータを読み込む所要時間を比較しました。

以下の結果が得られました。



PostgreSQL 19 が 15%～20%くらい高速化していることが分かります。

同様に hex 形式のエンコーディング、デコーディングについても以下のように性能を調べました。

（sha256 のバイナリ文字列を 10 万行格納したテーブルを作成）

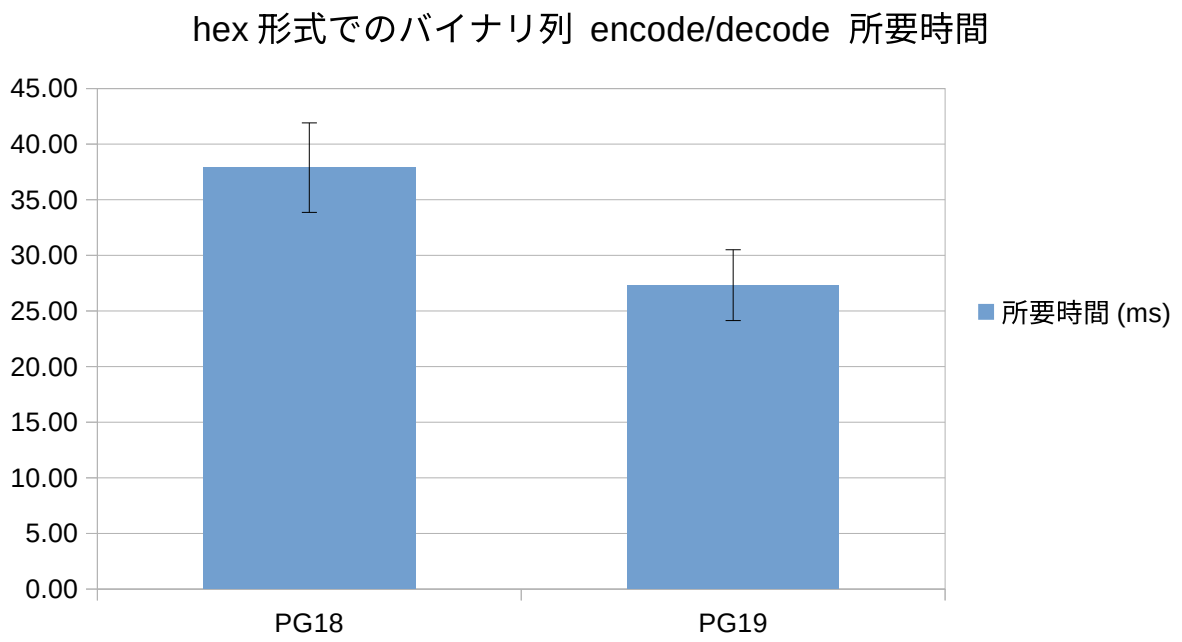
```
db1=# CREATE TABLE t413hex (b bytea);
db1=# INSERT INTO t413hex SELECT sha256(g::text::bytea)
        FROM generate_series(1, 100000) g;
db1=# VACUUM ANALYZE t413hex;
```

（エンコードしてデコードする以下の問合せを繰り返し実行）

```
db1=# \timing on
db1=# SELECT count(1) FROM t413hex
        WHERE decode(encode(b, 'hex'), 'hex') = 'x';
 count
-------
     0
(1 row)

Time: 37.351 ms
```

問合せの所要時間は以下の結果になりました。



25～30%程度、高速化されています。COPY FROM と比べて、本問合せの方が処理の中で SIMD 命令を適用するメモリ上のデータ変換処理部分が占める割合が高いワークロードですので、高速化がより大きくあらわれていると言えます。

4.1.4. 外部キー制約検査の高速化

外部キー制約の検査が高速化されました。

これまで外部キー制約の検査には、サーバプログラミングインタフェース（SPI）による内部的な SQL 実行が行なわれていました。これを SQL 実行を経ずに直接ユニークインデックスを調べるコードに書き換えることで処理が効率化されました。パーティションテーブルによる外部キー制約や、OVERLAP 句を伴う範囲データ型の検査を伴う外部キー制約の検査と、外部キー制約に伴うカスケード更新などのアクションには引き続き SPI が使われます。

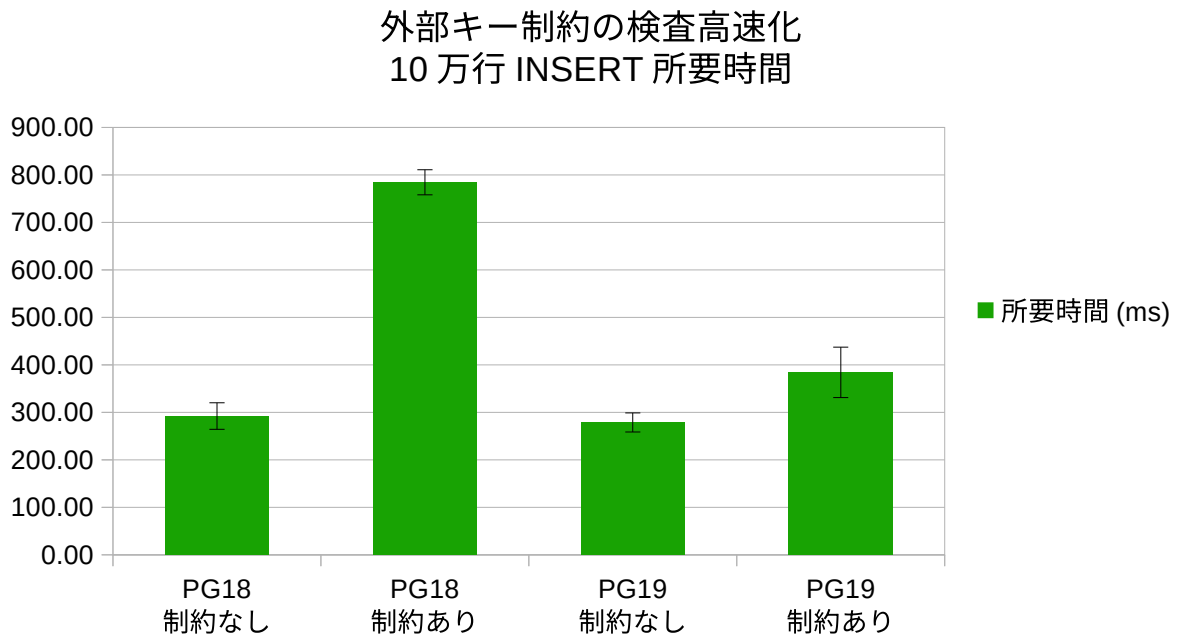
以下の手順で単純な外部キー制約において性能が改善されているかを確認しました。

（外部キー制約の被参照テーブルと参照テーブルの INSERT 所要時間を比較）

```
db1=# CREATE TABLE t414m (id int PRIMARY KEY, v text);
db1=# CREATE TABLE t414 (id int PRIMARY KEY REFERENCES t414m (id), v text);
db1=# \timing on
db1=# INSERT INTO t414m
```

```
        SELECT g, md5(g::text) FROM generate_series(1, 100000) g;
INSERT 0 100000
Time: 259.804 ms
db1=# INSERT INTO t414
        SELECT g, md5(g::text) FROM generate_series(1, 100000) g;
INSERT 0 100000
Time: 444.485 ms
```

PostgreSQL 18 と 19 の比較で以下の結果となりました。誤差線は標準偏差を上下に伸ばしたものです。



どちらのバージョンも制約の無い被参照テーブルへの INSERT と比べて、外部キー制約のある参照テーブルへの INSERT は所要時間が増えていますが、PostgreSQL 19 の方が所要時間が半分くらいに短縮されていることが分かります。一方、被参照テーブルの INSERT 所要時間はほとんど違いがありませんので、本改善による成果と言えます。

4.1.5. 基数ソート

メモリ内で行なわれるタプルのソート処理で従来からのクイックソートに加えて、基数ソート（radix sort）アルゴリズムも使われるようになりました。実行プランの表示としては「Sort Method: quicksort」で変わりませんが、内部的に条件が合えば基数ソートが使われます。

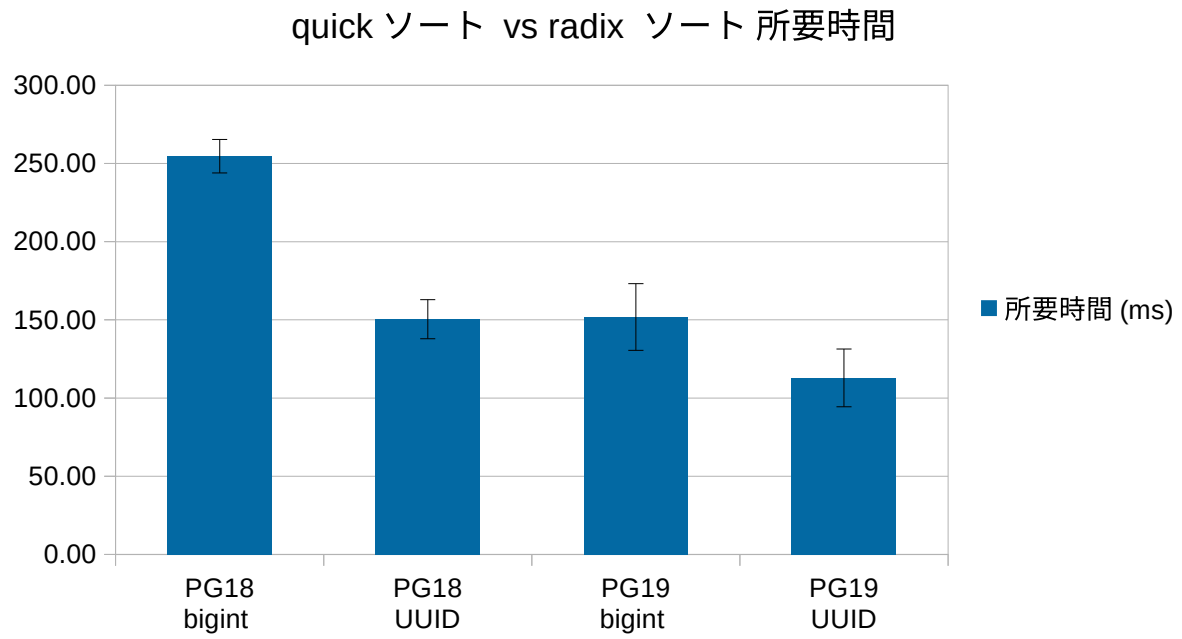
対象が整数データ型と UUID 型で 40 行以上ある場合に基数ソートが選択されます。
以下のようにソートを伴う問い合わせの所要時間を PostgreSQL 18 と 19 で比較しました。

```
（bigint 型と uuid 型を持つソート用のテーブルとデータを作成）
db1=# CREATE TABLE t415 (c1 bigint, c2 uuid);
db1=# INSERT INTO t415 SELECT (random() * 100000)::bigint, uuidv7()
        FROM generate_series(1, 100000) g;
          : 5 回繰り返して、合計 50 万行を挿入する、
 → データを揃えるため、他方バージョンには \copy TO/FROM でデータを投入

（c1 と c2 でソートして Execution Time を採取する）
db1=# SET work_mem TO '50MB';
db1=# explain (analyze) SELECT * FROM t415 ORDER BY c1;
                            QUERY PLAN
------------------------------------------------------------------------------
 Sort  (cost=55576.92..56826.92 rows=500000 width=24)
       (actual time=112.603..137.887 rows=500000.00 loops=1)
   Sort Key: c1
   Sort Method: quicksort  Memory: 31820kB   ← 表示はどちらも quicksort になる
   Buffers: shared hit=3248
   ->  Seq Scan on t415  (cost=0.00..8248.00 rows=500000 width=24)
                         (actual time=0.018..38.453 rows=500000.00 loops=1)
         Buffers: shared hit=3248
 Planning Time: 0.065 ms
 Execution Time: 209.319 ms
(8 rows)
```

ソートを行う問い合わせの EXPLAIN (ANALYZE) を繰り返し行い、バッファヒット率があがってきた最後の方から 7 回分の Execution Time を採取しました。結果は以下の通りです。

bigint でのソート、UUID でのソートともに PostgreSQL 19 の所要時間が短くなっていることがわかります。なお、bigint 列よりも UUID 列の方が所要時間が短いのは、生成順がソート順となる UUIDv7 であるためソート済みのデータとなっているからと考えられます。



4.2. SQL 機能

4.2.1. プロパティグラフ問合せ

リレーショナルテーブルをグラフ構造として扱うためのプロパティグラフ問合せ機能が追加されました。

PostgreSQL はリレーショナルデータベースであってグラフデータベースの製品ではありませんが、レコード間の関係を辿る問合せをしたい場合があります。そのような場合には、テーブル結合や再帰問合せを使って問合せを記述することが可能ですが、より直感的なグラフ問い合わせを使う選択肢が加わりました。

プロパティグラフは、リレーショナルテーブル上に作成される「読み取り専用のグラフビュー」のような仕組みです。実際のデータは通常のテーブルに保持されたままであり、グラフ構造として扱うための定義のみを作成します。

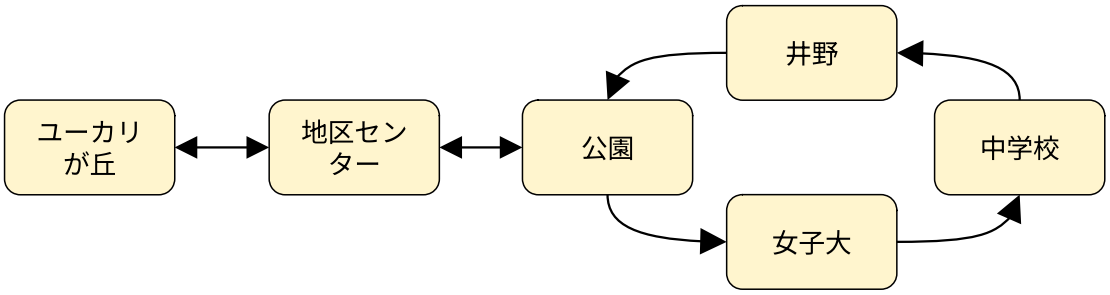
以下に例を示します。駅 (station) と駅間接続 (link) を用いて、片方向・双方向の接続を含むグラフ構造をプロパティグラフで定義し、相互接続を検出する問合せを実行してみます。

（テーブル作成とデータ挿入）

```
db1=# CREATE TABLE station (id int PRIMARY KEY, name text);
db1=# CREATE TABLE link (id1 int, id2 int, PRIMARY KEY(id1, id2));
db1=# INSERT INTO station VALUES
       (1, 'ユーカリが丘'), (2, '地区センター'), (3, '公園'),
```

```
       (4, '女子大'), (5, '中学校'), (6, '井野');
db1=# INSERT INTO link VALUES
       (1, 2), (2, 1), (2, 3), (3, 2), (3, 4), (4, 5), (5, 6), (6, 3);
```

ユーカリが丘線の路線図は以下図のように片方向のループがあります。



以下のようにプロパティグラフを定義します。station テーブルが VERTEX（頂点）で、link テーブルが EDGE（辺）であり、キーとして使う列とプロパティとして使う列をそれぞれ指定しています。

（プロパティグラフを定義）

```
db1=# CREATE PROPERTY GRAPH g1
        VERTEX TABLES (station KEY (id) PROPERTIES(id, name))
        EDGE TABLES (link
          SOURCE KEY (id1) REFERENCES station(id)
          DESTINATION KEY (id2) REFERENCES station(id));
CREATE PROPERTY GRAPH
```

（新しく追加された \dG コマンドで作成したプロパティグラフを確認できる）

```
db1=# \dG
              List of property graphs
 Schema |     Name      |      Type      |  Owner
--------+---------------+----------------+----------
 public | g1            | property graph | postgres
(1 rows)
```

プロパティグラフへの問合せは、SELECT 文の FROM 句に GRAPH_TABLE を指定して実行します。頂点と辺の組み合わせをそのまま表現した記述で繋がりを示します。 WHERE 句で追加の条件を指定して、COLUMNS 句で出力する列を指定しています。

```
（「中学校」駅から４駅進んで元の「中学校」駅に戻る経路を問合せ）
db1=# SELECT * FROM GRAPH_TABLE (g1 MATCH
        (s1 IS station)-[l1 IS link]->(s2 IS station)
                       -[l2 IS link]->(s3 IS station)
                       -[l3 IS link]->(s4 IS station)
                       -[l4 IS link]->(s1 IS station)
        WHERE s1.name = '中学校'
        COLUMNS (s1.name AS s1, s2.name AS s2, s3.name AS s3,
                 s4.name AS s4));
   s1    |  s2   |  s3   |   s4
--------+------+------+--------
 中学校  | 井野  | 公園  | 女子大
(1 row)
```

問合せの意図通りの結果が得られました。

グラフ問い合わせは、内部的にはテーブルの結合で実現されています。本問い合わせの実行計画を確認すると、以下のようになります。単純にリンクしている数だけテーブル結合をしていることが分かります。

```
db1=# explain (costs off) SELECT * FROM GRAPH_TABLE (g1 MATCH
        (s1 IS station)-[l1 IS link]->(s2 IS station)
                       -[l2 IS link]->(s3 IS station)
                       -[l3 IS link]->(s4 IS station)
                       -[l4 IS link]->(s1 IS station)
        WHERE s1.name = '中学校'
        COLUMNS (s1.name AS s1, s2.name AS s2, s3.name AS s3, s4.name AS s4));
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Nested Loop
   Join Filter: (link_2.id2 = station_3.id)
   ->  Nested Loop
         Join Filter: ((station_2.id = link_2.id1) AND (link_2.id2 = link_3.id1))
          ->  Nested Loop
                Join Filter: (link_1.id2 = station_2.id)
                ->  Nested Loop
                      Join Filter: (station_1.id = link_1.id1)
                      ->  Nested Loop
```

```
                          Join Filter: (link.id2 = station_1.id)
                          ->  Hash Join
                                Hash Cond: (link_3.id2 = station.id)
                                ->  Seq Scan on link link_3
                                ->  Hash
                                      ->  Hash Join
                                            Hash Cond: (link.id1 = station.id)
                                            ->  Seq Scan on link
                                            ->  Hash
                                                  ->  Seq Scan on station
                                                        Filter: (name = '中学校'::text)
                          ->  Seq Scan on station station_1
                    ->  Seq Scan on link link_1
              ->  Seq Scan on station station_2
        ->  Seq Scan on link link_2
  ->  Seq Scan on station station_3
(25 rows)
```

なお、PostgreSQL 19 のプロパティグラフ問合せは、現時点では固定長のパターンマッチングのみをサポートしています。可変長パスによる経路探索には対応していないため、任意の深さの探索や推移的な関係の検索には引き続き再帰共通テーブル式（CTE）を使った問合せが必要になります。

4.2.2. テンポラルテーブル更新

PostgreSQL 19 では、テンポラルテーブルに対する期間指定の更新・削除機能である UPDATE/DELETE FOR PORTION OF 構文が追加されました。テンポラルテーブルとは、データに有効期間を持たせるデータモデリング手法です。PostgreSQL では、daterange 型や tsrange 型などの範囲型、および datemultirange 型や tsmultirange 型などの多重範囲型を利用して実装できます。

従来、テンポラルデータの一部分のみを更新・削除し、残りの部分は保持する場合は、アプリケーション側で実装する必要がありました。FOR PORTION OF 構文を使用して特定の時間範囲の行を更新または削除すると、PostgreSQL は自動的に行を分割します。対象期間のデータが変更され、変更されていない期間は元の値を持つ別行として保持されます。

以下は FOR PORTION OF 構文の実行例です。

（テンポラルテーブル機能を使うには btree_gist 拡張が必要）

```
db1=# CREATE EXTENSION IF NOT EXISTS btree_gist;
```

```
（テンポラルテーブルを作成）
db1=# CREATE TABLE employee (
        emp_id int, department text, valid_period daterange,
        PRIMARY KEY(emp_id, valid_period WITHOUT OVERLAPS));

（初期データを登録）
db1=# INSERT INTO employee
         VALUES (1001, 'sales', daterange('2026-04-01', NULL));

db1=# SELECT * FROM employee;
 emp_id | department | valid_period
--------+------------+--------------
   1001 | sales      | [2026-04-01,)
(1 row)

（2026-06-01 以降のみ所属部署を変更）
db1=# UPDATE employee
        FOR PORTION OF valid_period FROM '2026-06-01' TO NULL
        SET department = 'marketing' WHERE emp_id = 1001;
UPDATE 1

db1=# SELECT * FROM employee WHERE emp_id = 1001;
 emp_id | department |      valid_period
--------+------------+-------------------------
   1001 | marketing  | [2026-06-01,)
   1001 | sales      | [2026-04-01,2026-06-01)
(2 rows)
```

実行結果は「UPDATE 1」と表示されますが、内部的には行の挿入が実行されています。また、元の department が「sales」であった行の valid_period 列の終端が NULL から '2026-06-01' に書き換えられています。

DELETE でも同様に期間を指定した削除を実行できます。

```
db1=# DELETE FROM employee
        FOR PORTION OF valid_period FROM '2026-07-01' TO NULL
```

```
        WHERE emp_id = 1001;
DELETE 1

db1=# SELECT * FROM employee WHERE emp_id = 1001;
 emp_id | department |      valid_period
--------+------------+------------------------
   1001 | sales      | [2026-04-01,2026-06-01)
   1001 | marketing  | [2026-06-01,2026-07-01)
(2 rows)
```

DELETE で「DELETE 1」と出力されますが、実際には valid_period 列の終端が NULL であったものが、「2026-07-01」に書き換えられていて、物理的な行数は 2 行のままであることがわかります。

これまで UPDATE や DELETE するときの基幹指定では TO を NULL として、ある時点からそれ以降に変更を適用するという指定をしてきましたが、過去の期間に対して指定することもできます。

（過去の期間を上書きした場合）

```
db1=# UPDATE employee
        FOR PORTION OF valid_period FROM '2026-05-25' TO '2026-06-02'
        SET department = 'holding status' WHERE emp_id = 1001;
UPDATE 2
db1=# SELECT * FROM employee;
 emp_id |   department   |      valid_period
--------+----------------+------------------------
   1001 | holding status | [2026-05-25,2026-06-01)
   1001 | sales          | [2026-04-01,2026-05-25)
   1001 | holding status | [2026-06-01,2026-06-02)
   1001 | marketing      | [2026-06-02,2026-07-01)
(4 rows
```

既存の行で重なる期間を持つものが分割されて、期間が調整されたうえで行が追加されました。2 つある department が「holding status」の行は、統合できそうなところですが、そこまでは行なってくれません。

また、主キーを変更するような問合せでは何が起きるでしょうか。「2026-06-25」以降は emp_id が 11001 に変更する、という UPDATE を実行してみます。

```
db1=# UPDATE employee
        FOR PORTION OF valid_period FROM '2026-06-25' TO NULL
```

```
        SET emp_id = 11001 WHERE emp_id = 1001;
UPDATE 1
db1=# SELECT * FROM employee;
 emp_id |   department   |      valid_period
--------+----------------+-------------------------
   1001 | holding status | [2026-05-25,2026-06-01)
   1001 | sales          | [2026-04-01,2026-05-25)
   1001 | holding status | [2026-06-01,2026-06-02)
   1001 | marketing      | [2026-06-02,2026-06-25)
  11001 | marketing      | [2026-06-25,2026-07-01)
(5 rows)
```

そうすると、emp_id = 1001 で有効期間が最も新しい行は「2026-06-25」までの期間となって、新たに emp_id = 11001 の行が追加されて、それは「2026-06-25」から、DELETE 時に指定した「2026-07-01」までとなります。この場合でも、破綻せずにデータ変更が行なわれていることが確認できました。

◆ *多重範囲型を使用する場合*

FOR PORTION OF は 多重範囲型にも対応しています。範囲型とは構文が少し変わります。以下に例を示します。

（テンポラルテーブル作成）

```
db1=# CREATE TABLE employee_multi (emp_id int, department text,
                                   valid_period datemultirange );
```

（データを登録）

```
db1=# INSERT INTO employee_multi VALUES (1001, 'sales',
          datemultirange(daterange('2026-04-01', '2026-06-01'),
          daterange('2026-07-01', NULL) ) );
```

（行内容を確認）

```
db1=# SELECT * FROM employee_multi;
 emp_id | department |                valid_period
--------+------------+-----------------------------------------
   1001 | sales      | {[2026-04-01,2026-06-01),[2026-07-01,)}
```

```
(1 row)

（データ更新 - 複数の期間を指定）
db1=# UPDATE employee_multi
         FOR PORTION OF valid_period (
           datemultirange(daterange('2026-05-01', '2026-05-15'),
                          daterange('2026-07-10', '2026-08-01')))
         SET department = 'marketing' WHERE emp_id = 1001;
UPDATE 1

db1=# \x
db1=# SELECT * FROM employee_multi;
-[ RECORD 1 ]+-----------------------------------------------------------
emp_id       | 1001
department   | sales
valid_period | {[2026-04-01,2026-05-01),[2026-05-15,2026-06-01),
 [2026-07-01,2026-07-10),[2026-08-01,)}
-[ RECORD 2 ]+-----------------------------------------------------------
emp_id       | 1001
department   | marketing
valid_period | {[2026-05-01,2026-05-15),[2026-07-10,2026-08-01)}
```

このように、多重範囲型を使うことで、同じ値を持つ複数期間を 1 行にまとめて保持することができて、範囲型を利用する場合と比較して、FOR PORTION OF を伴う行の更新時に、行数増加を抑制できるメリットがあります。

4.2.3. パーティションの併合／分割

「ALTER TABLE ... MERGE / SPLIT PARTITIONS」という構文が追加され、パーティションの併合と分割を単一のコマンドで実施できるようになりました。複数のパーティションを 1 つパーティションに併合したり、単一のパーティションを複数のパーティションへ分割することができます。

これまでは、パーティションの再編成を行う場合は、CREATE TABLE、ATTACH PARTITION、INSERT ... SELECT、DROP TABLE などを組み合わせた複雑なコマンド実行手順が必要でしたが、本機能により操作を単純化できます。

このコマンドは単一プロセスで動作し、処理中は親テーブルに対して ACCESS EXCLUSIVE ロックを取

得します。処理の完了まで他セッションから行われる DML 操作は待機状態となりますので、高負荷環境や大規模パーティションテーブルでの利用にはロックの観点から注意が必要です。

以下の検証では、パーティションの統合および分割の動作確認と、その際のロック挙動の確認を行います。

まずは、基本的なコマンドの使い方を確認します。

（2026 年 1 月～3 月の月別パーティションを作成し、各パーティションにデータを格納）

```
db1=# CREATE TABLE sales (
        id bigint, sale_date date, amount int
        ) PARTITION BY RANGE (sale_date);
db1=# CREATE TABLE sales_2026_01 PARTITION OF sales
        FOR VALUES FROM ('2026-01-01') TO ('2026-02-01');
db1=# CREATE TABLE sales_2026_02 PARTITION OF sales
        FOR VALUES FROM ('2026-02-01') TO ('2026-03-01');
db1=# CREATE TABLE sales_2026_03 PARTITION OF sales
        FOR VALUES FROM ('2026-03-01') TO ('2026-04-01');
db1=# INSERT INTO sales VALUES
        (1, '2026-01-10', 100),
        (2, '2026-02-10', 200),
        (3, '2026-03-10', 300);
```

（パーティションテーブルの行が各パーティションに格納されていることを確認）

```
db1=# SELECT tableoid::regclass, * FROM sales;
   tableoid    | id | sale_date  | amount
---------------+----+------------+--------
 sales_2026_01 |  1 | 2026-01-10 |    100
 sales_2026_02 |  2 | 2026-02-10 |    200
 sales_2026_03 |  3 | 2026-03-10 |    300
(3 rows)
```

（パーティションを併合する）

```
db1=# ALTER TABLE sales
        MERGE PARTITIONS (sales_2026_01, sales_2026_02) INTO sales_2026_q1;

db1=# SELECT tableoid::regclass, * FROM sales;
   tableoid    | id | sale_date  | amount
---------------+----+------------+--------
```

```
 sales_2026_q1 |  1 | 2026-01-10 |    100
 sales_2026_q1 |  2 | 2026-02-10 |    200
 sales_2026_03 |  3 | 2026-03-10 |    300
(3 rows)

db1=# \d+ sales_2026_q1
                          Table "public.sales_2026_q1"
 : 出力中略
Partition of: sales FOR VALUES FROM ('2026-01-01') TO ('2026-03-01')
Partition constraint: ((sale_date IS NOT NULL) AND (sale_date >= '2026-01-
01'::date) AND (sale_date < '2026-03-01'::date))
Access method: heap
```

ALTER TABLE .. MERGE PARTITIONS でパーティション sales_2026_01 と sales_2026_02 を併合して sales_2026_q1 としました。その後に SELECT で確認すると、1 月と 2 月の行が同じ sales_2026_q1 に格納されていることが確認できます。また、併合して作ったパーティション sales_2026_q1 の定義を確認すると、パーティションキーの日付の範囲が統合されて「2026-01-01」から「2026-03-01」未満となっていることがわかります。

続いて、ALTER TABLE .. SPLIT PARTITION でパーティション分割を行なってみます。

```
db1=# ALTER TABLE sales SPLIT PARTITION sales_2026_q1 INTO (
        PARTITION sales_2026_01
          FOR VALUES FROM ('2026-01-01') TO ('2026-02-01'),
        PARTITION sales_2026_02
          FOR VALUES FROM ('2026-02-01') TO ('2026-03-01'));

db1=# SELECT tableoid::regclass, * FROM sales;
   tableoid    | id | sale_date  | amount
---------------+----+------------+--------
 sales_2026_01 |  1 | 2026-01-10 |    100
 sales_2026_02 |  2 | 2026-02-10 |    200
 sales_2026_03 |  3 | 2026-03-10 |    300
(3 rows)
```

実行すると、パーティション sales_2026_q1 がなくなり、パーティション sales_2026_01、sales_2026_02 が作成されて、sale_date 列の値に応じて行がそれぞれのパーティションに格納された状態

になりました。意図通りに分割できていることが分かります。

ここまでの例では RANGE パーティションの場合を示しましたが、このほかに LIST パーティションでも「ALTER TABLE ... MERGE / SPLIT PARTITIONS」で併合と分割が可能です。HASH パーティションには対応していません。

◆ ロックと所要時間の確認

ALTER TABLE ... MERGE / SPLIT PARTITIONS は、パーティションテーブル（親テーブル）と処理対象のパーティション（子テーブル）に Access Exclusive モードのテーブルロックを取得します。したがって、参照のみの SELECT 文を含めて、対象パーティションテーブルに対する並行する全ての処理とロック競合します。

そこで ALTER TABLE ... MERGE / SPLIT PARTITIONS が比較的大規模なテーブルに対して、どの程度時間を要するかを以下のように確認しました。

```
（pgbench で各 100 万行、計 1000 万行のパーティションテーブルを作成）
$ pgbench -i --partitions=10 -s 100 db1
dropping old tables...
NOTICE:  table "pgbench_accounts" does not exist, skipping
NOTICE:  table "pgbench_branches" does not exist, skipping
NOTICE:  table "pgbench_history" does not exist, skipping
NOTICE:  table "pgbench_tellers" does not exist, skipping
creating tables...
creating 10 partitions...
generating data (client-side)...
vacuuming...
creating primary keys...
done in 43.20 s (drop tables 0.01 s, create tables 0.06 s, client-side
generate 13.09 s, vacuum 23.06 s, primary keys 6.98 s).

（バックグラウンドで pgbench 実行 - 1TPS 程度のそれほど重くない負荷）
$ pgbench -c 10 -T 600 -R 1 -n db1 &> pgbench.out &

（並行して、パーティションの併合と分割を実行）
$ psql db1
db1=# \timing
db1=# ALTER TABLE pgbench_accounts MERGE PARTITIONS
      (pgbench_accounts_1, pgbench_accounts_2) INTO pgbench_accounts_1plus2;
```

```
ALTER TABLE
Time: 2076.886 ms (00:02.077)

db1=# ALTER TABLE pgbench_accounts SPLIT PARTITION
        pgbench_accounts_1plus2 INTO (
          PARTITION pgbench_accounts_1 FOR VALUES
            FROM (MINVALUE) TO (1000001),
          PARTITION pgbench_accounts_2 FOR VALUES
            FROM (1000001) TO (2000001));
ALTER TABLE
Time: 3652.166 ms (00:03.652)
```

併合と分割の所要時間は上記のようになりました。

ALTER TABLE ... MERGE / SPLIT PARTITIONS の所要時間は、おおむね行を移動する所要時間ということが言えそうです。今回の試行では並行する pgbench の処理は OLTP タイプの短いトランザクションですので、ロック待ち時間は僅かです。2 秒、3.6 秒というのは、pgbench の VACUUM やインデックス作成を除いた初期データ投入の所要時間が 13 秒ほどですので、パーティション 10 個の内の 2 個分の行を移し替える程度の時間ということが言えます。分割の方が時間がかかるのは、行データを読んで、どちらに分別するか決める必要があるからと考えられます。

この所要時間だけトランザクションがロック待ちすることを容認できるなら、パーティションの並行・分割はオンラインで実行することも可能な操作と言うことができます。

4.2.4. グルーピングの拡張

GROUP BY に関する機能拡張として、GROUP BY ALL 構文が追加されました。また、GROUP BY で従来エラーになっていたケースが処理可能になりました。

◆ *GROUP BY ALL 構文*

従来、GROUP BY を使用する場合は、SELECT 句に記述した非集約列をすべて GROUP BY 句へ明示的に記述する必要がありました。PostgreSQL 19 で追加された GROUP BY ALL を利用すると、集約関数およびウィンドウ関数を含まない SELECT 句の項目を、自動的に GROUP BY 対象として扱えます。

以下、実行例です。

（テーブル作成）

```
db1=# CREATE TABLE t424 (region text, category text, price int,
```

```
                        quantity int, amount int);
（データ登録）
db1=# INSERT INTO t424 VALUES
         ('east', 'A', 10, 2, 100), ('east', 'A', 10, 2, 120),
         ('west', 'B', 20, 3, 200), ('west', 'B', 20, 3, 180),
         ('north', 'C', 15, 4, 140), ('north', 'C', 15, 4, 160);

（通常の GROUP BY）
db1=# SELECT region, category, price * quantity, sum(amount)
         FROM t424
         GROUP BY region, category, price * quantity ORDER BY region;
 region | category | ?column? | sum
--------+----------+----------+-----
 east   | A        |       20 | 220
 north  | C        |       60 | 300
 west   | B        |       60 | 380
(3 rows)

（GROUP BY ALL を使用）
db1=# SELECT region, category, price * quantity, sum(amount)
        FROM t424 GROUP BY ALL ORDER BY region;
 region | category | ?column? | sum
--------+----------+----------+-----
 east   | A        |       20 | 220
 north  | C        |       60 | 300
 west   | B        |       60 | 380
(3 rows)
```

このように、GROUP BY ALL は、SELECT 句の非集約項目を自動的に GROUP BY 対象へ追加する構文であり、従来の GROUP BY と同等の結果を取得できます。特に、SELECT 句へ式を多数記述する分析 SQL では、GROUP BY 句の記述量削減や SQL の可読性向上が期待できます。

◆ *GROUP BY 内部処理の改善*

従来は、GROUP BY を含むクエリのサブクエリ内で、外側クエリの GROUP BY 式を参照すると、エラーになる場合がありました。PostgreSQL 19 では GROUP BY 式の照合処理が改善され、このようなクエリも実

行可能になっています。

```
（18.4 の動作）
db1=# SELECT price * quantity,
               (SELECT sum(s2.amount) FROM t424 s2
                WHERE s2.price * s2.quantity = s1.price * s1.quantity)
         FROM t424 s1 GROUP BY price * quantity ORDER BY 1;
ERROR:  subquery uses ungrouped column "s1.price" from outer query

（19beta1 の動作）
db1=# SELECT price * quantity,
               (SELECT sum(s2.amount) FROM t424 s2
                WHERE s2.price * s2.quantity = s1.price * s1.quantity)
        FROM t424 s1 GROUP BY price * quantity ORDER BY 1;
 ?column? | sum
----------+-----
       20 | 220
       60 | 680
(2 rows)
```

上記の例では、結合対象のサブクエリ内に集約関数がある場合について、PostgreSQL 18 でエラーになっていたものが、PostgreSQL 19 で実行可能になっています。

4.2.5. COPY コマンドの拡張

PostgreSQL 19 では、COPY コマンドに関する複数の機能拡張が行われました。
COPY FROM に関する主な改善点は以下の通りです。

- COPY FROM の HEADER オプションで複数行スキップに対応
- ON_ERROR SET_NULL オプション追加

COPY TO に関する主な改善点は以下の通りです。

- JSON 形式でのエクスポート機能追加
- パーティションテーブルに対する COPY TO 対応

◆ COPY FROM の改善

PostgreSQL 19 から COPY FROM の HEADER オプションで複数行のヘッダスキップが可能になりまし

た。従来は HEADER オプションでは先頭 1 行のみスキップ可能でしたが、1 以上の任意の整数を指定できるようになりました。

以下の手順で動作を確認しました。

```
（CSV ファイルを用意）
$ cat > /tmp/t425a.csv <<EOS
header line 1
header line 2
1001,jiro,30
1002,hanako,25
EOS

（テーブル作成）
$ psql db1
db1=# CREATE TABLE t425a (id int, name text, age int);

（HEADER オプションに 2 を指定）
db1=# COPY t425a
        FROM '/tmp/t425a.csv' WITH (FORMAT csv, HEADER 2);
COPY 2

db1=# SELECT * FROM t425a;
  id  |  name  | age
------+--------+-----
 1001 | jiro   |  30
 1002 | hanako |  25
(2 rows)
```

PostgreSQL 19 では ON_ERROR SET_NULL オプションが追加され、COPY FROM 実行時にデータ型変換エラーが発生した場合、不正値を NULL として取り込めるようになりました。従来は COPY 全体がエラー終了していました。

以下の手順で動作を確認しました。

```
（CSV ファイルを作成）
$ cat > /tmp/t425a_err.csv <<EOS
```

```
1001,jiro,30
1002,hanako,abc
1003,saburo,25
EOS

（テーブルを初期化）
db1=# TRUNCATE t425a;

（ON_ERROR SET_NULL オプションを指定）
db1=# COPY t425a
         FROM '/tmp/t425a_err.csv'
         WITH (FORMAT csv, ON_ERROR set_null);
NOTICE:  in 1 row, columns were set to null due to data type incompatibility
COPY 3
db1=# SELECT * FROM t425a;
  id  |  name  | age
------+--------+-----
 1001 | jiro   |  30
 1002 | hanako |
 1003 | saburo |  25
(3 rows)
```

int 型であり整数値が期待されている 3 列目に「abc」を記載した行が、ON_ERROR set_null を指定した COPY FROM を使うことで、エラーを起こさずに読み込みできました。

◆ *COPY TO の改善*

COPY TO コマンドに JSON 形式でのエクスポート機能が追加されました。従来は JSON 形式でデータを出力するために、row_to_json() や json_agg() を利用した SQL を作成する必要がありましたが、PostgreSQL 19 では COPY TO のオプションとして JSON 形式を直接指定できるようになり、より簡単に JSON データを出力できるようになりました。

出力形式としては、1 行ごとに JSON オブジェクトを出力する NDJSON（Newline Delimited JSON）形式に加え、FORCE_ARRAY オプションを指定することで、結果全体を単一の JSON 配列として出力することも可能です。

以下に動作例を示します。

```
（NDJSON 形式で出力）
db1=# COPY t425a TO STDOUT WITH (FORMAT JSON);
{"id":1001,"name":"jiro","age":30}
{"id":1002,"name":"hanako","age":null}
{"id":1003,"name":"saburo","age":25}

（FORCE_ARRAY オプションを使用し、JSON 配列形式で出力）
db1=# COPY t425a TO STDOUT WITH (FORMAT JSON, FORCE_ARRAY);
[
 {"id":1001,"name":"jiro","age":30}
,{"id":1002,"name":"hanako","age":null}
,{"id":1003,"name":"saburo","age":25}
]
```

JSON 形式は現在 COPY TO のみでサポートされており、COPY FROM では利用できません。また、HEADER、DEFAULT、NULL、DELIMITER、FORCE QUOTE、FORCE NOT NULL、FORCE NULL などのテキスト形式および CSV 形式向けオプションとは互換性がないため、同時に指定することはできません。

なお、JSON 形式で出力した場合、SQL の NULL 値と JSON 型列に格納された JSON リテラル null はいずれも JSON の null として出力されるため、COPY TO の出力結果から両者を区別することはできません。

パーティションテーブルを直接 COPY TO の対象として指定できるようになりました。従来は、パーティションテーブルに対して直接 COPY TO を実行することはできず、COPY (SELECT ...) を利用する必要がありましたが、PostgreSQL 19 では COPY TO でパーティションテーブルを直接指定できるようになりました。これにより記述が簡潔になるだけでなく、クエリ実行および結果受け渡しのオーバーヘッドが削減されるため、従来の COPY (SELECT ...) TO よりも高い性能が期待できます。

以下に実行例を示します。

```
（パーティションテーブル作成）
db1=# CREATE TABLE t425b_part(id int, sales_date date, amount int)
         PARTITION BY RANGE (sales_date);
db1=# CREATE TABLE t425b_part_2025 PARTITION OF t425b_part
         FOR VALUES FROM ('2025-01-01') TO ('2026-01-01');
db1=# CREATE TABLE t425b_part_2026 PARTITION OF t425b_part
         FOR VALUES FROM ('2026-01-01') TO ('2027-01-01');

（データ登録）
```

```
db1=# INSERT INTO t425b_part VALUES
        (1,'2025-01-10',100), (2,'2025-06-15',200), (3,'2026-02-01',300);

（18.4 の動作）
db1=# COPY t425b_part TO STDOUT WITH (FORMAT csv);
ERROR:  cannot copy from partitioned table "t425b_part"
HINT:  Try the COPY (SELECT ...) TO variant.

db1=# COPY (SELECT * FROM t425b_part ORDER BY id)
         TO STDOUT WITH (FORMAT csv);
1,2025-01-10,100
2,2025-06-15,200
3,2026-02-01,300

（19beta1 の動作）
db1=# COPY t425b_part TO STDOUT WITH (FORMAT csv);
1,2025-01-10,100
2,2025-06-15,200
3,2026-02-01,300
```

PostgreSQL 19 ではパーティションテーブルを通常のテーブル同様に COPY TO で指定できることが確認できました。

4.2.6. 各種の追加関数

本節ではいくつかの追加されたアプリケーション用の SQL で利用可能な関数を取り上げます。

◆*jsonpath メソッド追加*

jsonpath 式で利用可能な文字列操作メソッドが追加されました。追加されたメソッドは以下の通りです。

- ltrim()　　文字列の左側を削る
- rtrim()　　文字列の右側を削る
- btrim()　　文字列の両側を削る
- lower()　　小文字に変換する
- upper()　　大文字に変換する
- initcap()　　単語ごと、頭文字を大文字に、続く文字を小文字にする
- replace()　　文字列を置換する

- split_part()　　文字列を分割して、その一つを取り出す

これらを使って jsonpath 式内で文字列加工を行えるようになったため、JSON データの検索や抽出条件をより簡潔に記述できるようになりました。

一例として以下のように使用できます。

```
（文字置換と大文字小文字変換を jsonpath 式の中で記述）
db1=# SELECT jsonb_path_query_array(
        '["PostgreSQL 19","JSON Path"]'::jsonb,
        '$[*].replace(" ", "_").upper()');
     jsonb_path_query_array
--------------------------------
 ["POSTGRESQL_19", "JSON_PATH"]
(1 row)
```

◆ *日付のランダム関数*

date 型、timestamp 型、および timestamptz 型を対象とした random(min, max)関数が追加されました。追加された関数は以下の 3 種類です。

- random(min date, max date)
- random(min timestamp, max timestamp)
- random(min timestamptz, max timestamptz)

以下のように、指定した範囲内（min ≦ 値 ≦ max）のランダムな日付・時刻値を返します。

```
db1=# SELECT random('2026-01-01'::date, '2026-12-31'::date) AS random_date,
             random('2026-01-01'::timestamp,
                    '2026-12-31'::timestamp) AS random_timestamp,
             random('2026-01-01 +09'::timestamptz,
                    '2026-12-31 +09'::timestamptz) AS random_timestamptz;
 random_date |      random_timestamp      |      random_timestamptz
-------------+----------------------------+-------------------------------
 2026-08-03  | 2026-10-11 14:03:45.361967 | 2026-02-02 09:19:18.223676+09
(1 row)
```

◆*error_on_null()関数*

引数が NULL の場合にエラーを発生させる error_on_null()関数が追加されました。入力値が NULL でなければそのまま返し、NULL の場合はエラーを発生させます。任意のデータ型を引数に受け取り、戻り値は同じデータ型になります。以下のように動作します。

（引数が NULL 以外の場合）

```
db1=# SELECT error_on_null(100);
 error_on_null
---------------
           100
(1 row)
```

（引数が NULL の場合）

```
db1=# SELECT error_on_null(NULL::text);
ERROR:  null value not allowed
```

この関数はサブクエリ結果を値として使うときに 0 行ではなく 1 行であることを保証したい場合などに便利です。0 行であると NULL になるため、以下のように使用できます。

（サブクエリ結果が 0 行で NULL になる場合をエラーにする使い方）

```
db1=# SELECT error_on_null((SELECT name FROM t425a WHERE id = 1001)) name;
 name
------
 jiro
(1 row)

db1=# SELECT error_on_null((SELECT name FROM t425a WHERE id = 2001)) name;
ERROR:  null value not allowed
```

4.3. 運用管理

4.3.1. Autovacuum スコア

autovacuum がデータベース内のどのテーブルを優先して VACUUM や ANALYZE を実行するかを決めるために、スコアリングシステムが導入されました。

従来は、VACUUM または ANALYZE が必要なテーブルが複数ある場合、基本的に pg_class システムテーブル上の並び順に従って処理されていました。これに対して PostgreSQL 19 では、各テーブルの状態からスコアを計算し、スコアの高いテーブルを優先して処理するようになりました。

このスコアリングシステムでは、VACUUM や ANALYZE が必要になる理由を 5 つの観点からそれぞれのスコアを算出します。そして、それらのスコアの最大値をそのテーブルの優先度として扱います。この判断に使われる値は pg_stat_autovacuum_scores ビューで確認できます。以下表にビューの、テーブルを特定する relid、schemaname 列以外の各列の意味を示します。

列名	意味
xid_score	トランザクション ID（XID）周回防止の観点から見たスコア
mxid_score	マルチトランザクション ID（MXID）周回防止の観点から見たスコア
vacuum_score	更新または削除された行数に基づく vacuum のスコア
vacuum_insert_score	挿入された行数に基づく vacuum のスコア
analyze_score	挿入、更新、削除された行数に基づく analyze のスコア
score	各スコアの最大値。autovacuum がテーブル一覧を並べ替えるときに使う値
do_vacuum	現在の情報に基づいて vacuum 対象になるか（真偽値）
do_analyze	現在の情報に基づいて analyze 対象になるか（真偽値）
for_wraparound	周回防止目的の vacuum 対象になるか（真偽値）

また、各観点のスコアには、以下の設定パラメータで重みを設定して DB 管理者が優先したいメンテナンス項目を調整可能です。デフォルトは全て 1.0 です。

設定パラメータ	意味
autovacuum_freeze_score_weight	xid_score の重み
autovacuum_multixact_freeze_score_weight	mxid_score の重み

設定パラメータ	意味
autovacuum_vacuum_score_weight	vacuum_score の重み
autovacuum_vacuum_insert_score_weight	vacuum_insert_score の重み
autovacuum_analyze_score_weight	analyze_score の重み

これらの重み付けパラメータは、postgresql.conf またはサーバ起動時のオプションで設定します。

たとえば、自動 vacuum で ANALYZE 処理を優先したい場合には、autovacuum_analyze_score_weight を増やして analyze_score の影響を強められます。すべての重みを 0.0 にすると、PostgreSQL 18 以前の挙動（pg_class システムテーブル上の並び順に基づく方式）に戻せます。

以下の通り、動作確認を行いました。

まず、テーブルを作成して 10000 行を登録し、VACUUM ANALYZE を実行して統計情報を更新しておきます。

（テスト用テーブル作成）

```
db1=# CREATE TABLE t431_vacuum (id int PRIMARY KEY, v text);
db1=# INSERT INTO t431_vacuum
        SELECT g, md5(g::text) FROM generate_series(1, 10000) g;
db1=# VACUUM ANALYZE t431_vacuum;
```

続いて、半数の行を更新します。

（行を更新して autovacuum 対象になりやすい状態を作る）

```
db1=# UPDATE t431_vacuum
        SET v = md5((id * 2)::text) WHERE id % 2 = 0;
UPDATE 5000
```

この状態で pg_stat_autovacuum_scores ビューを参照します。

（テーブルの autovacuum スコアを確認）

```
db1=# \x
db1=# SELECT relname,
             round(score::numeric, 3) AS score,
             round(vacuum_score::numeric, 3) AS vacuum_score,
             round(vacuum_insert_score::numeric, 3) AS vacuum_insert_score,
             round(analyze_score::numeric, 3) AS analyze_score,
             do_vacuum, do_analyze, for_wraparound
```

```
      FROM pg_stat_autovacuum_scores  WHERE relname = 't431_vacuum';
-[ RECORD 1 ]-------+------------
relname             | t431_vacuum
score               | 14.286
vacuum_score        | 2.439
vacuum_insert_score | 3.333
analyze_score       | 14.286
do_vacuum           | t
do_analyze          | t
for_wraparound      | f
```

テーブルの中からスコアの高いテーブルを取り出すには次のように問い合わせができます。

（特定スキーマ内でスコアの高いテーブルを取り出す例）

```
db1=# SELECT relname, round(score::numeric, 3) AS score,
        do_vacuum, do_analyze, for_wraparound
        FROM pg_stat_autovacuum_scores WHERE schemaname = 'public'
        ORDER BY score DESC LIMIT 5;
  relname   | score  | do_vacuum | do_analyze | for_wraparound
------------+--------+-----------+------------+----------------
 t412hex    | 29.851 | t         | t          | f
 t431vacuum | 14.286 | t         | t          | f
 t_follows  |  0.190 | f         | f          | f
 t_usr      |  0.098 | f         | f          | f
 t_radix7   |  0.000 | f         | f          | f
(5 rows)
```

更新された行数が autovacuum の閾値を超えると、vacuum_score や analyze_score が高くなり、do_vacuum、do_analyze が true になります。for_wraparound は XID または MXID の周回防止目的で VACUUM される場合に true になります。

pg_stat_autovacuum_scores ビューを使うと、autovacuum がどのテーブルを優先して処理するかを確認しやすくなります。特に、多数のテーブルがあり、どのテーブルが VACUUM や ANALYZE の対象になりやすいかを確認したい場合に有用です。

ただし、pg_stat_autovacuum_scores ビューの結果は、autovacuum worker が実際に処理対象一覧を作成した時点の情報と一致するとは限りません。このビューは参照した時点の情報をもとにスコアを計算します。そのため、実際にどのテーブルがどの順番で処理されるかを完全に保証するものではなく、現在の優先度を確

認するための目安として扱う必要があります。

PostgreSQL 19 の autovacuum スコアは、autovacuum の動作を直接高速化する機能ではありません。autovacuum が複数の候補テーブルを扱うときに、優先度の高いテーブルから処理しやすくなることと、その判断材料を pg_stat_autovacuum_scores ビューで確認できるようになる機能です。

運用上の活用方法としては、autovacuum が追いついていない場合の調査において、従来の pg_stat_all_tables や pg_stat_progress_vacuum に加えて、pg_stat_autovacuum_scores を参照して、どのテーブルが優先的に処理される状態にあるかを確認することが考えられます。

4.3.2. 各種モニタリングビューの追加

PostgreSQL の稼働状態を報告するシステムビューがいくつか追加され、また、既存ビューに列が追加されました。それらのうち、主なものを取り上げます。

◆pg_stat_lock

ロック種別ごとの統計情報を確認する pg_stat_lock ビューが追加されました。

pg_stat_lock は、ロック種別ごとの累積統計を確認するためのビューです。ロック種別（locktype）ごとのデータベースクラスタ全体の統計が 1 行ずつ格納されています。waits 列は deadlock_timeout 設定を超えたロック待ちの発生数、wait_time はそのときの待ち時間の総計（ミリ秒単位）です。fastpath_exceeded 列は、一部の頻出する種類のロック取得を省力化できる fastpath の仕組みが、取得しているロック数が多すぎて使えなかった場合の数です。

ある時点のロック状態を見る用途では pg_locks を使い、ロック待ちがどの種類でどの程度発生していたかを見る用途では pg_stat_lock を使う、という使い分けになります。

以下は確認用に pgbench を 1000 接続でしばらく実行した後の pg_stat_lock ビュー出力です。行ロックに相当する「transactionid」と「tuple」が多数カウントされていることがわかります。

```
db1=# SELECT locktype, waits, wait_time, fastpath_exceeded
        FROM pg_stat_lock ORDER BY waits DESC;
     locktype     | waits | wait_time | fastpath_exceeded
------------------+-------+-----------+-------------------
 transactionid    |  5947 |  12004366 |                 0
 tuple            |   810 |   1259821 |                 0
 frozenid         |     0 |         0 |                 0
 page             |     0 |         0 |                 0
 virtualxid       |     0 |         0 |                 0
 spectoken        |     0 |         0 |                 0
 object           |     0 |         0 |                 0
```

```
 userlock          |      0 |          0 |                   0
 advisory          |      0 |          0 |                   0
 relation          |      0 |          0 |                 368
 applytransaction  |      0 |          0 |                   0
 extend            |      0 |          0 |                   0
(12 rows)
```

◆*pg_stat_recovery*

リカバリ中のサーバの状態を確認する pg_stat_recovery ビューが追加されました。

このビューは、startup process のリカバリ状態を 1 行だけ返します。プライマリサーバのようにリカバリ中でないサーバでは行を返しません。

以下は同一ホスト上にストリーミングレプリケーションのスタンバイを作成し、スタンバイ側で pg_stat_recovery ビューを参照しています。

```
db1=# \x
db1=# SELECT * FROM pg_stat_recovery;
-[ RECORD 1 ]------------+------------------------------
promote_triggered        | f
last_replayed_read_lsn   | 0/1304E540
last_replayed_end_lsn    | 0/1304E5F8
replay_end_lsn           | 0/1304E5F8
recovery_last_xact_time  | 2026-06-02 16:34:32.364968+09
current_chunk_start_time | 2026-06-02 16:34:32.376657+09
pause_state              | not paused
```

列の意味は以下表の通りです。

列名	意味
promote_triggered	昇格が要求されたか
last_replayed_read_lsn	最後にリプレイに成功した WAL レコードの開始位置
last_replayed_end_lsn	最後にリプレイに成功した WAL レコードの終了位置
replay_end_lsn	現在リプレイ中の WAL レコード位置
recovery_last_xact_time	リカバリ中に最後に再生されたトランザクションの commit または abort 時刻
current_chunk_start_time	startup process が最新の受信済み WAL chunk に追いついたことを観測した時刻

列名	意味
pause_state	リカバリー時停止状態

pause_state には、「not paused」、「pause requested」、「paused」のいずれかが入ります。これにより、スタンバイ側でリカバリが進行中なのか、一時停止要求中なのか、実際に停止しているのかを区別できます。

◆pg_stat_replication_slots.mem_exceeded_count

pg_stat_replication_slots ビューに mem_exceeded_count 列が追加されました。

この列は、論理デコーディングで使用するメモリ量が logical_decoding_work_mem 設定値を超えた回数を示します。

以下では、確認用に logical_decoding_work_mem を小さく設定し、1 トランザクションで大量の行を INSERT した後、論理レプリケーションスロットから変更内容を読み出しました。これにより、論理デコード中のメモリ使用量が logical_decoding_work_mem を超え、mem_exceeded_count が増加したことを確認できます。

```
db1=# \x
db1=# SELECT * FROM pg_stat_replication_slots
        ORDER BY mem_exceeded_count DESC;
-[ RECORD 1 ]------+------------------------------
slot_name          | slot_mem_test
mem_exceeded_count | 59
spill_count        | 60
spill_bytes        | 3899927
stream_count       | 0
stream_bytes       | 0
stats_reset        | 2026-06-02 16:50:11.227041+09
```

mem_exceeded_count だけではなく、spill_count、spill_bytes、stream_count、stream_bytes も合わせて見ることで、論理デコーディング中にメモリ不足に近い状態が起きているのか、ディスク退避が多く発生しているのかを確認できます。これらの値は、logical_decoding_work_mem の調整を検討する材料になります。

◆pg_stat_all_*.stats_reset

いくつかの統計ビューで、統計情報の最終リセット時刻を示す stats_reset 列が参照できるようになりました。pg_stat_all_tables、pg_stat_all_indexes、pg_statio_all_sequences に stats_reset 列が追加され、対応す

る sys/user 系のビューにも追加されています。また、pg_stat_user_functions と pg_stat_database_conflicts にも stats_reset 列が追加されています。

以下では、確認用にテーブル等を作成し、統計情報をリセットした後、データ操作や関数呼び出しを行い、その後に統計ビューを参照しました。stats_reset 列が確認できています。

```
db1=# \x
db1=# SELECT schemaname, relname,
        seq_scan, idx_scan, n_tup_ins, n_tup_upd, n_tup_del, stats_reset
      FROM pg_stat_user_tables ORDER BY schemaname, relname;
-[ RECORD 1 ]------------------------------
schemaname  | public
relname     | t432_stat_reset
seq_scan    | 2
idx_scan    | 4
n_tup_ins   | 10000
n_tup_upd   | 100
n_tup_del   | 100
stats_reset | 2026-06-02 21:35:42.975207+09
  （後略）
```

stats_reset によって統計値がどの期間の累積値なのかを SQL だけで確認しやすくなります。たとえば idx_scan が少ないインデックスを調べる場合でも、統計が直近でリセットされたばかりであれば、単純に「使われていない」と判断するのは早計です。stats_reset を合わせて見ることで、統計値の評価期間を明確にできます。

従来はデータベース全体を共通してリセットした時点を知るには pg_stat_database ビューを確認する必要があり、また、オブジェクト毎のリセットも可能でしたが個々のリセット時点を参照する手段がありませんでした。

◆pg_stat_progress_vacuum.started_by / pg_stat_progress_analyze.started_by

pg_stat_progress_vacuum と pg_stat_progress_analyze に、処理の開始要因を示す started_by 列が追加されました。

pg_stat_progress_vacuum は実行中の VACUUM の進捗を、pg_stat_progress_analyze は実行中の ANALYZE の進捗を確認するビューです。PostgreSQL 19 では、これらの処理が手動で開始されたものなのか、自動処理として開始されたものなのかを判別できるようになりました。

以下では、確認用に手動の VACUUM と ANALYZE を実行して実行中に進捗ビューを参照しています。

```
db1=# \x
db1=# SELECT pid, datname, relid::regclass AS relname, phase, mode,
        started_by FROM pg_stat_progress_vacuum;
-[ RECORD 1 ]----------------------------
pid        | 2553
datname    | db1
relname    | t432_progress_vacuum_analyze
phase      | scanning heap
mode       | aggressive
started_by | manual
```

pg_stat_progress_vacuum.started_by の値は、「manual」、「autovacuum」、「autovacuum_wraparound」のいずれかです。「manual」は明示的な VACUUM コマンド、「autovacuum」は autovacuum worker、「autovacuum_wraparound」は XID または multixact ID の周回防止を目的とした autovacuum worker による実行を示します。

ANALYZE については、以下のように確認できます。

```
db1=# \x
db1=# SELECT pid, datname, relid::regclass AS relname, phase, started_by
        FROM pg_stat_progress_analyze;
-[ RECORD 1 ]----------------------------
pid        | 2568
datname    | db1
relname    | t432_progress_vacuum_analyze
phase      | acquiring sample rows
started_by | manual
```

pg_stat_progress_analyze.started_by の値は、「manual」または「autovacuum」です。「manual」は明示的な ANALYZE、または VACUUM の ANALYZE オプションによる実行を示し、「autovacuum」は autovacuum worker による実行を示します。

これにより、進捗ビューを見たときに、処理が管理者の操作によるものか、自動メンテナンスによるものかをすぐに切り分けられます。特に autovacuum_wraparound は周回防止を目的とした重要な VACUUM であるため、通常の autovacuum と区別して確認できる点は運用上有用です。

◆*pg_dsm_registry_allocations*

動的共有メモリの割り当て状況を確認する pg_dsm_registry_allocations ビューが追加されました。

動的共有メモリとは、DSM（dynamic shared memory）とも呼ばれ、サーバ起動時に固定的に確保される共有メモリとは別の、稼働中に確保される共有メモリ領域です。これを管理する共有メモリ上の内部データ構造が DSM レジストリです。

DSM の代表的な用途はパラレル問い合わせを実行するときのプロセス間の情報共有です。その他、サードパーティ拡張が DSM を使うことがあります。

pg_dsm_registry_allocations は、DSM レジストリの内容を表示するビューです。以下のように参照します。

```
db1=# SELECT name, type, size FROM pg_dsm_registry_allocations;
 name | type | size
------+------+------
(0 rows)
```

検証環境では、実行時に表示対象となる DSM の割り当てが存在しなかったため、0 行となりました。

列は以下の 3 つです。

列名	意味
name	DSM registry 上の割り当て名。
type	割り当ての種類。segment、area、hash のいずれか。
size	割り当てサイズ。単位はバイト。初期化に失敗した場合は NULL

DSM としてメモリがどれだけ使われているかをリアルタイムで把握するために活用できます。

4.3.3. ログ出力の拡張

ログ出力の対象とするレベル（重大度）の指定が出力元プロセスのタイプごとに行なえるようになりました。従来の log_min_messages は PostgreSQL サーバ全体に対して 1 つの重大度を指定する設定でした。PostgreSQL 19 からは従来通りの指定に加えて、backend:debug1 や autovacuum:debug1 のように、プロセスタイプを付けて指定できます。

書式は以下の通りです。

```
log_min_messages = '《既定の重大度》,《プロセスタイプ》:《重大度》, ...'
```

コロンを含まない重大度は、個別指定されていないプロセスタイプに対する既定値として使われます。たとえば、次の設定では、通常のログ出力は ERROR 以上に抑えつつ、walsender と autovacuum だけは

DEBUG1 以上を出力します。

```
log_min_messages = 'error, walsender:debug1, autovacuum:debug1'
```

プロセスタイプは、 archiver、autovacuum、backend、bgworker、bgwriter、checkpointer、checksums、ioworker、postmaster、slotsyncworker、startup、syslogger、walreceiver、walsender、walsummarizer、walwriter を指定できます。

重大度には、DEBUG5、DEBUG4、DEBUG3、DEBUG2、DEBUG1、INFO、NOTICE、WARNING、ERROR、LOG、FATAL、PANIC を指定できます。既定値は WARNING です。

以下の通り、動作確認を行います。

まず、バックエンドプロセスだけ DEBUG1 以上をログ出力する設定を行ないます。

（backend だけ DEBUG1 以上をログ出力する設定）

```
db1=# ALTER SYSTEM SET log_min_messages = 'panic, backend:debug1';
db1=# SELECT pg_reload_conf();
 pg_reload_conf
----------------
 t
(1 row)
```

ここでは、コロンを含まない panic が既定値です。そのため、個別指定していないプロセスタイプでは PANIC 以上だけがログ出力対象になります。一方で、backend:debug1 を指定しているため、通常の SQL 実行を担当するバックエンドプロセスでは DEBUG1 以上がログ出力対象になります。

この状態で、バックエンドプロセスから DEBUG メッセージを出力します。

（バックエンドプロセスから DEBUG メッセージを出力）

```
db1=# DO $$
        BEGIN  RAISE DEBUG 'log_min_messages backend debug test'; END $$;
```

サーバログを確認すると、DEBUG メッセージの出力が確認できます。

（サーバログの確認）

```
$ grep 'log_min_messages backend debug test' $PGDATA/log/*
2026-06-04 10:05:26.814 JST [12503] DEBUG:  log_min_messages backend debug
test
```

次に、バックエンドプロセスを個別指定せず、既定値だけを PANIC にします。

（既定値だけを PANIC に設定）

```
db1=# ALTER SYSTEM SET log_min_messages = 'panic';
db1=# SELECT pg_reload_conf();
 pg_reload_conf
----------------
 t
(1 row)
```

同じように DEBUG メッセージを出力します。

（バックエンドプロセスから DEBUG メッセージを出力）

```
db1=# DO $$
        BEGIN
          RAISE DEBUG 'log_min_messages backend debug suppressed test';
        END $$;
```

この場合、バックエンドプロセスに対する個別指定がないため、既定値の PANIC が適用されます。DEBUG メッセージはサーバログに出力されません。

（サーバログの確認）

```
$ grep 'log_min_messages backend debug suppressed test' $PGDATA/log/* ||
echo '該当ログ無し'
該当ログ無し
```

最後に、設定を元に戻しておきます。

（設定を元に戻す）

```
db1=# ALTER SYSTEM RESET log_min_messages;
db1=# SELECT pg_reload_conf();
 pg_reload_conf
----------------
 t
(1 row)
```

4.3.4. REPACK コマンド

PostgreSQL 19 では、VACUUM FULL と CLUSTER を統合した REPACK が追加されました。従来のコマンドは互換性のために残されています。

REPACK はテーブルから不要な領域を取り除いてファイルサイズを縮めるコマンドです。不要になったタプルが占有している領域を回収するために、対象テーブルの内容を新しいファイルへ書き直して、最後に古いファイル捨てます。通常の VACUUM と異なり、テーブル全体を書き直すことで、未使用領域も OS に返却できます。

構文は以下の通りです。

```
REPACK [ ( option [, ...] ) ]
       [ table_and_columns [ USING INDEX [ index_name ] ] ] ;
REPACK [ ( option [, ...] ) ] USING INDEX ;
```

指定できる主なオプションは以下です。

オプション	意味
VERBOSE	テーブルごとの処理状況を INFO レベルで出力
ANALYZE	REPACK 後に対象テーブルに ANALYZE を実行。単一の非パーティションテーブルを指定した場合に利用可能
CONCURRENTLY	REPACK 中も他トランザクションが対象テーブルを利用できるようにする

USING INDEX index_name を指定すると、REPACK は指定したインデックスの順序に従ってテーブル行を物理的に並べ替えます。さらに、指定したインデックスは対象テーブルのクラスタ化インデックスとして記録されます。この記録により、次回以降は REPACK table_name USING INDEX のようにインデックス名を省略して、同じインデックス順で再度並べ替えることができます。

通常の REPACK では、対象テーブルに ACCESS EXCLUSIVE ロックが取得されます。そのため、処理が完了するまで対象テーブルへの読み書きはブロックされます。

一方、CONCURRENTLY を指定した REPACK では、リパック中も対象テーブルへアクセスできます。内部的には、既存のテーブル内容を新しいテーブルファイルへ再編成します。その間に発生した INSERT、UPDATE、DELETE などのデータ変更は、論理レプリケーションでも使われる論理デコードの仕組みを使って新しいテーブルファイルへ反映します。その後、古いテーブル・インデックスファイルと新しいファイルを入れ替えます。この入れ替えの最終段階で対象テーブルに ACCESS EXCLUSIVE ロックを取得します。通常、このロックはファイルを入れ替えるための短時間で済みます。ただし、REPACK がロックを待っている間に対象テーブルへ多くの更新が行われた場合、それらの変更をロック保持中に反映する必要があるため、ロック

時間が長くなる可能性があります。

また、CONCURRENTLY には制限があります。UNLOGGED テーブル、パーティションテーブル、主キーもインデックスにもとづいたレプリカ識別情報も持たないテーブル、システムカタログ、TOAST テーブルには使用できません。また、トランザクションブロック内では実行できず、追加のレプリケーションスロットを作成できない場合にも使用できません。CONCURRENTLY 付きの REPACK はトランザクション分離において必ずしも安全でない点にも注意が必要です。CONCURRENTLY 付 REPACK 完了前に取得された古いスナップショットを使っている並行トランザクションから、REPACK 後の対象テーブルが正しく見えない場合があり得ます。特に、長時間実行されるトランザクションと同時に実行する場合には注意が必要です。

以下のように、不要領域を含むテーブルを作成して、REPACK によりテーブルサイズが縮小することを確認します。

（100 万行の確認用テーブルを作成）

```
db1=# CREATE TABLE t434_repack (id int PRIMARY KEY,  v text);
db1=# INSERT INTO t434_repack SELECT g, repeat(md5(g::text), 20)
        FROM generate_series(1, 1000000) g;
db1=# VACUUM ANALYZE t434_repack;
db1=# SELECT pg_size_pretty(pg_total_relation_size('t434_repack'))
        AS before_delete;
 before_delete
---------------
 673 MB
(1 row)
```

（DELETE で不要領域を作成、半分の行を削除する）

```
db1=# DELETE FROM t434_repack WHERE id % 2 = 0;
```

（VACUUM 後のテーブルサイズを確認）

```
db1=# VACUUM t434_repack;
db1=# SELECT pg_size_pretty(pg_total_relation_size('t434_repack'))
        AS after_vacuum;
 after_vacuum
--------------
 673 MB
(1 row)
```

```
（REPACK 実施）
db1=# REPACK (ANALYZE, VERBOSE) t434_repack;
INFO:  repacking "public.t434_repack" in physical order
INFO:  "public.t434_repack": found 0 removable, 500000 nonremovable row
versions in 83334 pages
DETAIL:  0 dead row versions cannot be removed yet.
CPU: user: 0.66 s, system: 0.53 s, elapsed: 19.61 s.
INFO:  analyzing "public.t434_repack"
INFO:  "t434_repack": scanned 30000 of 41667 pages, containing 359996 live
rows and 0 dead rows; 30000 rows in sample, 499998 estimated total rows
INFO:  finished analyzing table "db1.public.t434_repack"
avg read rate: 531.640 MB/s, avg write rate: 0.177 MB/s
buffer usage: 54 hits, 30010 reads, 10 dirtied
WAL usage: 32 records, 9 full page images, 54044 bytes, 51644 full page
image bytes, 0 buffers full
system usage: CPU: user: 0.20 s, system: 0.06 s, elapsed: 0.44 s
REPACK

（REPACK 後にテーブルサイズが縮小していることを確認）
db1=# SELECT pg_size_pretty(pg_total_relation_size('t434_repack'))
        AS after_repack;
 after_repack
--------------
 336 MB
(1 row)
```

上記の実行結果では、DELETE 後に通常の VACUUM を実行しても、テーブルサイズは大きくは縮小しません。一方で、REPACK の実行後には、テーブルが書き直されるため、pg_total_relation_size の値が小さくなります。

続いて、USING INDEX を指定した場合の動作を確認します。

```
（確認テーブルにインデックスを作成）
db1=# CREATE INDEX t434_repack_v_idx ON t434_repack (v);

（インデックスを使って REPACK 実行）
db1=# REPACK t434_repack USING INDEX t434_repack_v_idx;
```

```
db1=# SELECT c.relname, i.indisclustered FROM pg_index i
       JOIN pg_class c ON c.oid = i.indexrelid
       WHERE c.relname = 't434_repack_v_idx';
      relname      | indisclustered
-------------------+----------------
 t434_repack_v_idx | t
```

USING INDEX に指定したインデックスがクラスタ化用インデックスとして設定されるため、indisclustered が t になります。

REPACK の実行状況は、pg_stat_progress_repack ビューで確認できます。このビューには、REPACK を実行中のバックエンドごとに 1 行が表示されます。phase 列では、initializing、seq scanning heap、index scanning heap、sorting tuples、writing new heap、swapping relation files、rebuilding index、performing final cleanup などの処理段階を確認できます。CONCURRENTLY 実行時には、処理中に発生した DML を反映する catch-up 段階も報告されます。

```
db1=# \x
db1=# SELECT p.pid, p.datname, c.relname, command, phase,
        heap_blks_scanned, heap_blks_total, heap_tuples_inserted,
        heap_tuples_updated, heap_tuples_deleted, index_rebuild_count
        FROM pg_stat_progress_repack p
        LEFT JOIN pg_class c ON c.oid = p.relid;
-[ RECORD 1 ]--------+-----------------
pid                  | 14255
datname              | db1
relname              | t434_repack
command              | REPACK
phase                | seq scanning heap
heap_blks_scanned    | 18847
heap_blks_total      | 83334
heap_tuples_inserted | 0
heap_tuples_updated  | 0
heap_tuples_deleted  | 0
index_rebuild_count  | 2
```

以上のように、PostgreSQL 19 では、テーブルの不要領域回収やインデックス順の物理的な並べ替えを REPACK コマンドで実行できるようになりました。

通常の REPACK は VACUUM FULL や CLUSTER と同様にテーブルを書き直す処理ですが、CONCURRENTLY を指定すると、対象テーブルを利用できない時間を主にファイル入れ替え時に抑えられます。そのため、大きなテーブルを運用中に再編成したい場合に有効です。ただし、REPACK は 処理中に新しいテーブルファイルやインデックスファイルを作成するため、追加のディスク空き容量が必要です。さらに CONCURRENTLY では、処理中に発生した変更を一時的に保持する分の領域も使うことがあり得ます。そのため、保守時間を確保できる場合や対象テーブルが小さい場合には、通常の REPACK を選ぶ方が有利です。

4.3.5. バイナリ pg_dumpall

PostgreSQL 19 では、pg_dumpall コマンドが plain 形式以外の出力形式に対応しました。pg_dumpall は、データベースクラスタ内の全データベースをダンプするコマンドです。各データベースの内容に加えて、ロール、テーブルスペース、設定パラメータに対する権限など、データベースクラスタ全体で共有されるオブジェクトも出力します。設定ファイルや ALTER SYSTEM による設定変更は、通常の pg_dumpall 出力には含まれません。

従来の pg_dumpall は、クラスタ全体の内容を 1 つの SQL スクリプトファイルとして出力していました。PostgreSQL 19 では、-F または―format オプションにより、pg_dump と同様に custom 形式、directory 形式、tar 形式を指定できるようになりました。

本機能ではデータベースクラスタ全体のバックアップを pg_restore を使ってリストアできるようになりました。これにより、リストア対象のデータベースを除外する、グローバルオブジェクトの復元を抑止する、custom 形式または directory 形式で並列リストアを試す、といった選択ができるようになりました。

本検証では、plain 形式以外の出力形式を指定した場合に、バックアップとリストアの手順がどのように変わるかを確認します。次に、クラスタ全体を取得したアーカイブから、特定のデータベースだけを復元する手順を検証します。また、データベースを復元対象から外した場合でも、ロールなどのグローバルオブジェクトがどのように扱われるかを確認します。さらに、--no-globals を指定してグローバルオブジェクトを復元しない場合の動作を確認します。-C、--single-transaction、--jobs といった pg_restore 側の指定について、クラスタ全体のリストアで注意すべき制約を整理します。最後に、統計情報を含めてダンプした場合に、リストア用 SQL にどのような内容が含まれるかを確認します。また、非 plain 形式のアーカイブであっても、リストア時には SQL が実行されるため、pg_restore -f で事前に内容を確認できることを示します。

◆ *検証データの準備*

検証用に、所有者が異なる *2* つのデータベースを作成します。ここでは、*db435a* をロール *r435a* の所有、*db435b* をロール *r435b* の所有とします。

（検証用ロールとデータベースを作成）

```
$ psql -U postgres postgres
```

```
postgres=# CREATE ROLE r435a LOGIN;
postgres=# CREATE ROLE r435b LOGIN;
postgres=# CREATE DATABASE db435a OWNER r435a;
postgres=# CREATE DATABASE db435b OWNER r435b;
postgres=# \c db435a
db435a=# SET ROLE r435a;
db435a=> CREATE TABLE t435a (id int PRIMARY KEY, msg text);
db435a=> INSERT INTO t435a VALUES (1, 'data in db435a');
db435a=> RESET ROLE;
db435a=# \c db435b
db435b=# SET ROLE r435b;
db435b=> CREATE TABLE t435b (id int PRIMARY KEY, msg text);
db435b=> INSERT INTO t435b VALUES (1, 'data in db435b');
db435b=> \q
```

◆ *出力形式毎の違い*

custom 形式、directory 形式、tar 形式、plain 形式で pg_dumpall を実行し、作成されるファイル構成を確認します。

```
（custom 形式でダンプ）
$ pg_dumpall -F c -f dumpall_custom
$ find dumpall_custom -maxdepth 2 -print | sort
dumpall_custom
dumpall_custom/databases
dumpall_custom/databases/1.dmp
dumpall_custom/databases/24917.dmp
dumpall_custom/databases/24920.dmp
dumpall_custom/databases/24921.dmp
dumpall_custom/databases/5.dmp
dumpall_custom/map.dat
dumpall_custom/toc.glo
```

custom 形式では、databases ディレクトリ配下に、データベースごとの custom 形式アーカイブが作成されます。ファイル名にはデータベースの OID が使われます。

```
（directory 形式でダンプ）
$ pg_dumpall -F d -f dumpall_dirctory
$ find dumpall_dirctory -maxdepth 2 -print | sort
dumpall_dirctory
dumpall_dirctory/databases
dumpall_dirctory/databases/1
dumpall_dirctory/databases/24917
dumpall_dirctory/databases/24920
dumpall_dirctory/databases/24921
dumpall_dirctory/databases/5
dumpall_dirctory/map.dat
dumpall_dirctory/toc.glo
```

directory 形式では、databases ディレクトリ配下に、データベースごとのディレクトリアーカイブが作成されます。

```
（tar 形式でダンプ）
$ pg_dumpall -F t -f dumpall_tar
$ find dumpall_tar -maxdepth 2 -print | sort
dumpall_tar
dumpall_tar/databases
dumpall_tar/databases/1.tar
dumpall_tar/databases/24917.tar
dumpall_tar/databases/24920.tar
dumpall_tar/databases/24921.tar
dumpall_tar/databases/5.tar
dumpall_tar/map.dat
dumpall_tar/toc.glo
```

tar 形式では、databases ディレクトリ配下に、データベースごとの tar アーカイブが作成されます。

```
（plain 形式でダンプ）
$ rm -f dumpall.sql
$ pg_dumpall -F p -f dumpall.sql
$ file dumpall.sql
dumpall.sql: ASCII text
```

plain 形式では、従来通り、単一の SQL スクリプトファイルが作成されます。

以上の通り、plain 形式では、クラスタ全体を復元するための 1 つの SQL スクリプトとして出力されますが、非 plain 形式では、クラスタ全体の情報がグローバルオブジェクトとデータベースごとのアーカイブに分けて保存されます。

◆ *map.dat と toc.glo の確認*

非 plain 形式で作成されたディレクトリに格納されている map.dat は、リストア時に処理対象となるデータベースの一覧です。pg_restore は、pg_dumpall が作成した非 plain 形式アーカイブを復元するとき、まず toc.glo からグローバルオブジェクトを復元し、その後 map.dat に列挙されたデータベースを順に処理します。また、map.dat 内の行を#でコメントアウトすると、そのデータベースは復元対象から外れます。

（map.dat の確認）

```
$ grep db435 dumpall_custom/map.dat
24920 db435a
24921 db435b
```

toc.glo には、ロールやテーブルスペースなど、クラスタ全体で共有されるグローバルオブジェクトが格納されます。toc.glo は custom 形式のアーカイブなので、pg_restore -f を使うと SQL として内容を出力できます。

（toc.glo の確認）

```
$ pg_restore -f dumpall_globals.sql dumpall_custom/toc.glo
$ grep -E 'CREATE ROLE|CREATE TABLESPACE|GRANT' dumpall_globals.sql | head
CREATE ROLE postgres;
CREATE ROLE r435a;
CREATE ROLE r435b;
```

◆ *クラスタ全体のアーカイブから一部のデータベースだけをリストア*

クラスタ全体を取得したアーカイブから、db435a だけを復元します。まず、リストア先として、新しいデータベースクラスタを別ポートで起動します。

（リストア先クラスタの作成）

```
$ export PGDATA_RESTORE="$HOME/19_restore"
$ initdb --no-locale --encoding=UTF8 -D "$PGDATA_RESTORE" \
  -c logging_collector=on -c port=55435
```

```
The files belonging to this database system will be owned by user
"postgres".
...出力後略
$ pg_ctl -D "$PGDATA_RESTORE" start
```

非 plain 形式アーカイブをリストアする場合、pg_restore に「-C」または「--create」を指定します。また、「--exclude-database」でリストア対象から除外したいデータベースを指定します。db1、db435b を除外し、db435a だけをリストアします。-d postgres で指定したデータベースは、リストア処理を開始するための接続先として使われます。

```
（リストア）
$ pg_restore -p 55435 -d postgres -C \
  --exclude-database='template*' \
  --exclude-database='postgres' \
  --exclude-database='db1' \
  --exclude-database='db435b' \
  dumpall_custom
pg_restore: error: could not execute query: ERROR:  role "postgres" already
exists
Command was: CREATE ROLE postgres;
ALTER ROLE postgres WITH SUPERUSER INHERIT CREATEROLE CREATEDB LOGIN
REPLICATION BYPASSRLS;


pg_restore: warning: errors ignored on restore: 1
```

すでに存在している postgres ロールの CREATE ROLE に失敗しますが、このエラーは無視されます。pg_restore はグローバルオブジェクト（toc.glo の内容）をリストアする際に、内部的に exit_on_error = false に設定する設計になっています。

なお、同様の除外は、map.dat を編集する方法でも行なえます。map.dat 内の行は、先頭に#を付けてコメントアウトできます。コメントアウトされたデータベースは、pg_restore の処理対象から外れます。

```
（map.dat を編集して db435b の行をコメントアウトする例）
$ sed -i.bak '/db435b/s/^/# /' dumpall_custom/map.dat
```

リストア後にデータベース一覧を出力すると、db435a だけがリストアされ、db435b は作成されていないことが確認できます。

```
（リストアされたデータベースを確認）
$ psql -l -p 55435 -U postgres postgres
   Name    |  Owner   | ...
-----------+----------+-----
 db435a    | r435a    | ...
 postgres  | postgres | ...右側省略
 template0 | postgres | ...
           |          |
 template1 | postgres | ...
           |          |
(4 rows)

（テーブルの内容を確認）
$ psql -p 55435 -d db435a -c "SELECT * FROM t435a;"
 id |      msg
----+---------------
  1 | data in db435a
(1 row)
```

◆ *--exclude-database とグローバルオブジェクトの関係*

前の手順では、db435b を復元対象から外しましたが、db435b の所有者だった r435b も復元されています。これは、--exclude-database はデータベースのリストアを除外するオプションであり、グローバルオブジェクトのリストアを除外するオプションではないためです。したがって、db435b を除外しても、toc.glo に含まれるロールはリストアされます。

特定のデータベースだけを戻したい場合でも、ロールはクラスタ全体の情報として復元されるので、不要なロールまで作成したくない場合は、後述する--no-globals を使うか、グローバルオブジェクトの復元方法を別途考慮する必要があります。

```
（ロールの確認）
$ psql -p 55435 -d postgres -c \
 "SELECT rolname FROM pg_roles WHERE rolname LIKE 'r435%' ORDER BY rolname;"
 rolname
---------
 r435a
 r435b
```

```
(2 rows)
```

◆ *--no-globals 指定時の動作*

次に、グローバルオブジェクトをリストアしない場合の動作を確認します。pg_restore には--no-globals オプションがあり、非 plain 形式アーカイブからのリストア時に、グローバルオブジェクトを格納する toc.glo のリストアをスキップできます。なお、--no-globals オプション指定時に-C/--create オプションを省略すると、既にリストア先にあるデータベースだけがリストアされます。

まず、ロールを事前に作成しない状態で、--no-globals を指定して db435a をリストアします。

```
（リストア先クラスタの作成）
$ export PGDATA_RESTORE_NOGLOBALS="$HOME/19_restore_noglobals"
$ initdb --no-locale --encoding=UTF8 -D "$PGDATA_RESTORE_NOGLOBALS" \
  -c logging_collector=on -c port=55436
The files belonging to this database system will be owned by user
"postgres".
...出力後略
$ pg_ctl -D "$PGDATA_RESTORE_NOGLOBALS" start

（リストア）
$ pg_restore -p 55436 -d postgres -C --no-globals \
  --exclude-database='template*' \
  --exclude-database='postgres' \
  --exclude-database='db1' \
  --exclude-database='db435b' \
  dumpall_custom
pg_restore: error: could not execute query: ERROR:  role "r435a" does not exist
Command was: ALTER DATABASE db435a OWNER TO r435a;

pg_restore: error: could not execute query: ERROR:  role "r435a" does not exist
Command was: ALTER TABLE public.t435a OWNER TO r435a;

pg_restore: warning: errors ignored on database "db435a" restore: 2
pg_restore: warning: errors ignored on restore: 2
```

（リストアされたデータベースを確認）

```
$ psql -l -p 55436 -U postgres postgres
                                    List of databases
   Name    |  Owner   | ...
-----------+----------+-
 db435a    | postgres |
 postgres  | postgres | ...右側省略
 template0 | postgres |
 template1 | postgres |
           |          | ...
(4 rows)
```

この場合、db435a の所有者である r435a がリストア先クラスタに存在しないため、データベース所有者設定の段階でエラーになります。db435a 自体はリストアされ、所有者は pg_restore 実行時の接続ロールとなります。

--no-globals を使う場合は、必要なロールやテーブルスペースをリストア先に事前作成しておく必要があります。

このように、--no-globals を使うと、リストア先のロール管理を維持したまま、データベースだけをリストアできます。ただし、リストア対象のデータベースやオブジェクトが参照するロールは、事前に用意しておく必要があります。所有者のリストアが不要な場合は –no-owner、権限のリストアが不要な場合は –no-acl を組み合わせることで、リストア先ロールへの依存を減らせます。

◆ 非 plain 形式アーカイブの安全性確認

非 plain 形式のアーカイブは、SQL スクリプトそのものではありません。しかし、リストア時にはリストア先で SQL が実行されます。したがって、信頼できない環境で取得したダンプを、そのまま本番環境へリストアしてはいけません。ダンプをリストアすると、リストア先ではダンプ元のスーパーユーザが選んだ任意のコードが実行される可能性があります。また、部分ダンプや部分リストアであっても、この性質は変わりません。非 plain 形式のダンプは、pg_restore -f《出力ファイル名》で SQL として出力しできるので、リストアの前に実行される SQL に意図しない処理が含まれていないかを確認できます。

（非 plain 形式の全体ダンプを SQL として出力、データベース作成の -C 指定が必要）

```
$ pg_restore -C -f dumpall_check.sql dumpall_custom
$ vi dumpall_check.sql
--
-- PostgreSQL database dump
```

```
--

\restrict eZcoeZHz8x6Ks1nTrT4gQROQnFwgsmGsnlEJncClNfldF48Q6nNlcm6Xecb5uah

-- Dumped by pg_dump version 19beta1
...後略
```

4.4. レプリケーション

4.4.1. ストリーミングレプリケーションの拡張

PostgreSQL 19 では、ストリーミングレプリケーションとリカバリに関する機能として、指定した WAL 位置までの到達を待つ WAIT FOR コマンドと、停止時に WAL 送信完了を待つ時間を制限する設定パラメータ wal_sender_shutdown_timeout が追加されました。

本検証では、以下のようにプライマリ 1 台、スタンバイ 1 台の物理ストリーミングレプリケーション環境を作成し、これらの動作を確認しました。

```
（検証用のスタンバイ PostgreSQL を作成して、起動する）
$ pg_basebackup -D ./pg5433 -R -c fast
$ pg_ctl start -D ./pg5433 -o '-c port=5433'
```

◆ WAIT コマンド

非同期レプリケーションでは、プライマリで更新が完了した直後にスタンバイへ参照すると、その更新内容がまだスタンバイに反映されていない場合があります。PostgreSQL 19 では、このような場合に、アプリケーション側で取得した WAL の位置（LSN）までスタンバイが追いつくのを SQL で待てるようになりました。

WAIT FOR コマンドの基本構文は以下の通りです。

```
WAIT FOR LSN '《LSN》' [ WITH ( option [, ...] ) ]
```

オプション	意味
MODE 'standby_replay'	スタンバイで WAL が再生され、データベースへ反映されるまで待つ（デフォルト）
MODE 'standby_write'	スタンバイで WAL が書き込まれるまで待つ

オプション	意味
MODE 'standby_flush'	スタンバイで WAL がディスクへフラッシュされるまで待つ
MODE 'primary_flush'	プライマリで WAL がディスクへフラッシュされるまで待つ
TIMEOUT '時間'	指定時間を超えたら待機を終了する（デフォルトは 0 で LSN 到達まで待ち続ける
NO_THROW	タイムアウト、プライマリ昇格時にエラーではなく状態文字列を返す

以下の検証では、スタンバイ側で WAL 再生を一時停止してから、プライマリでテーブル作成とデータ投入を行いました。これにより、スタンバイ側では WAL の受信やフラッシュは進む一方で、データベースへの反映は止まる状態を作ります。この状態で WAIT FOR を実行すると、standby_write、standby_flush、standby_replay の違いを確認できます。

まず、スタンバイで WAL 再生を一時停止します。pg_wal_replay_pause() はリカバリの一時停止を要求する関数です。実際に一時停止したかどうかは pg_get_wal_replay_pause_state() で確認します。

（スタンバイで WAL 再生を停止）

```
$ psql -p 5433 db1
db1=# SELECT pg_wal_replay_pause();
 pg_wal_replay_pause
---------------------


(1 row)
db1=# SELECT pg_get_wal_replay_pause_state();
 pg_get_wal_replay_pause_state
-------------------------------
 paused
(1 row)
```

（プライマリで検証テーブルを作成して、WAL 位置を取得）

```
$ psql -p 5432 db1
db1=# CREATE TABLE t441_wait (id int PRIMARY KEY, v text);
db1=# INSERT INTO t441_wait VALUES (1, 'primary');
db1=# SELECT pg_current_wal_insert_lsn();
 pg_current_wal_insert_lsn
---------------------------
 0/040448A8
```

```
(1 row)
```

次に、スタンバイで WAL の受信位置と再生位置を確認します。pg_last_wal_receive_lsn()はストリーミングレプリケーションで受信してディスクへ同期済みの WAL 位置を返し、pg_last_wal_replay_lsn()は再生済みの WAL 位置を返します。

```
（スタンバイで受信位置と再生位置を確認）
$ psql -p 5433 db1
db1=# \x
db1=# SELECT pg_get_wal_replay_pause_state(),
        pg_last_wal_receive_lsn(),  pg_last_wal_replay_lsn();
-[ RECORD 1 ]-----------------+-----------
pg_get_wal_replay_pause_state | paused
pg_last_wal_receive_lsn       | 0/04055158
pg_last_wal_replay_lsn        | 0/04028EC0

（この時点ではテーブル t441_wait の作成は伝搬していない）
db1=# SELECT * FROM t441_wait;
ERROR:  relation "t441_wait" does not exist
```

この時点では WAL 再生を止めているため、pg_last_wal_receive_lsn（WAL を受け取りまで完了した位置）は進んでいても、pg_last_wal_replay_lsn（WAL 適用まで完了した位置）は追いついていません。そのため作成した t441_wait テーブルはスタンバイでは参照できない状態です。ここで WAIT FOR を様々なモードで実行してみます。pg_last_wal_receive_lsn で返った位置まで待機させる指定をします。

まず、standby_write を確認します。

```
（スタンバイで standby_write を確認）
db1=# WAIT FOR LSN '0/04055158'
        WITH (MODE 'standby_write', TIMEOUT '5s', NO_THROW);
-[ RECORD 1 ]---
status | success
```

続いて、standby_flush を確認します。

```
（スタンバイで standby_flush を確認）
db1=# WAIT FOR LSN '0/04055158'
        WITH (MODE 'standby_flush', TIMEOUT '5s', NO_THROW);
```

```
-[ RECORD 1 ]---
status | success
```

止めているのは replay（WAL の適用）ですから、ここまではいずれも即座に success が返りました。

デフォルトの standby_replay モードは、WAL がデータベースへ反映されるまで待ちます。現在は WAL 再生を止めているため、タイムアウトになります。

```
（スタンバイで standby_replay を確認 - 5 秒後にタイムアウト）
db1=# WAIT FOR LSN '0/04055158' WITH (TIMEOUT '5s', NO_THROW);
-[ RECORD 1 ]---
status | timeout
```

WAL の受信またはフラッシュが終わっていても、WAL の再生が止まっている場合には、standby_replay の条件は満たされないことが確認できます。

スタンバイで WAL 再生を再開し、もう一度 standby_replay を確認します。

```
（スタンバイで WAL を再開）
db1=# SELECT pg_wal_replay_resume();
-[ RECORD 1 ]--------+-
pg_wal_replay_resume |

db1=# SELECT pg_get_wal_replay_pause_state();
-[ RECORD 1 ]-----------------+-----------
pg_get_wal_replay_pause_state | not paused

（スタンバイで standby_replay を確認）
db1=# WAIT FOR LSN '0/04055158' WITH (TIMEOUT '5s', NO_THROW);
-[ RECORD 1 ]---
status | success

（プライマリで投入した行がスタンバイ参照できることを確認）
db1=# SELECT * FROM t441_wait;
-[ RECORD 1 ]
id | 1
v  | primary
```

WAL 再生を一時停止することで、standby_write、standby_flush、standby_replay の違いを確認できました。WAL を受信済みまたはフラッシュ済みであっても、再生が止まっていればテーブル作成や INSERT は参照結果に反映されません。参照用のスタンバイで「更新後の結果まで見える」ことを保証したい場合には、standby_replay を使って待つ必要があります。

なお、WAIT FOR はトップレベルの SQL コマンドとして実行する必要があります。関数、プロシージャ、DO ブロックの内部では実行できません。また、WAIT FOR は LSN の数値としての到達を待ちますが、タイムラインの妥当性までは判定しません。現在より手前のタイムラインの場合、何であれ到達済と判定してしまいます。

◆*wal_sender_shutdown_timeout*

PostgreSQL 19 では、wal_sender_shutdown_timeout パラメータが追加されました。このパラメータは、サーバ停止時に WAL 送信側が、受信側へ WAL を送り終えるまで待つ最大時間を指定するものです。

従来、レプリケーションを使用している場合、送信側サーバは停止処理中に、未送信の WAL が接続中の受信側へ転送されるまで待ち続けます。これは計画的なスイッチオーバでは有用ですが、受信側の反応が止まっている場合には停止処理が長くなります。低速なネットワーク、遠隔地のスタンバイ、またはロック待ちで止まっている論理レプリケーション適用処理があると、サーバ停止が長時間完了しないことがあります。wal_sender_shutdown_timeout を設定すると、指定時間を過ぎた時点で待機を打ち切り、停止処理を進められます。デフォルト値は-1 で、この場合はタイムアウトが無効、つまり停止時の待機を制限しない動作となっています。

前節で作ったストリーミングレプリケーションを使って、動作検証を行ないます。

wal_sender_shutdown_timeout に 3 秒を設定しました。

（プライマリで、postgresql.conf 設定を変更して、再起動）

```
$ vi $PGDATA/postgresql.conf
   wal_sender_shutdown_timeout = '3s'

$ pg_ctl restart
```

検証のために、スタンバイ PostgreSQL の wal receiver プロセスを停止させます。

（レプリケーション経路に帯域制限をかける）

```
$ ps x | grep walreceiver
   5646 ?        Ss     0:00 postgres: walreceiver streaming 2/38000060
   5654 pts/1    S+     0:00 grep --color=auto walreceiver
$ kill -STOP 5646
```

この状態で、プライマリ側に一定量の WAL を発生させます。

（プライマリで WAL を発生させる）

```
$ pgbench -s 10 -i db1
```

プライマリを停止し、停止完了までの時間を確認します。

（プライマリ PostgreSQL を停止）

```
$ time pg_ctl stop
waiting for server to shut down..... done
server stopped

real        0m3.276s
user        0m0.003s
sys         0m0.006s
```

（停止した wal receiver プロセスを再開しておく）

```
$ kill -CONT 5646
```

wal_sender_shutdown_timeout を 3 秒に設定しているため、WAL sender が WAL reciever に WAL を送信中でも、プライマリの停止処理は無制限には待ち続けません。本検証では、停止完了までの実時間が設定値付近になることを確認できました。

wal_sender_shutdown_timeout が設定されていない場合、おおよそ wal_sender_timeout （デフォルト 60s）の時間だけ待たされる動作になります。

wal_sender_shutdown_timeout によるサーバ停止の場合、ログには以下のメッセージが出力されます。

```
WARNING:  terminating walsender process due to replication shutdown timeout
DETAIL:  Walsender process might have been terminated before all WAL data was replicated to the receiver.
```

4.4.2. 論理レプリケーションの拡張

PostgreSQL 19 では、論理レプリケーションに関して、シーケンス値の同期、WAL レベルの実効値の自動変更、および関連コマンド構文の拡張が行なわれています。

◆ シーケンス対応

従来の論理レプリケーションでは、テーブルの行データは同期されても、シーケンス値は同期されませんでした。そのため、論理レプリケーションを冗長化やサーバ移行のために使用して、アプリケーションの接続先をサブスクライバ側へ切り替えてサブスクライバ側で新しい書き込みを受け付ける場合には、シーケンス値の調整を行う必要がありました。

PostgreSQL 19 では、シーケンスを論理レプリケーション対象に含めることができるようになりました。

ただし、シーケンスの nextval() の実行ごとに増分が同期されるわけではありません。CREATE SUBSCRIPTION 、 ALTER SUBSCRIPTION ... REFRESH PUBLICATION 、 ALTER SUBSCRIPTION ... REFRESH SEQUENCES のタイミングで、パブリッシャ側のシーケンス値がサブスクライバ側へ同期されます。また、レプリケーション対象のシーケンスを個別に指定することはできず、データベース内の全てのシーケンスを対象とするという指定だけが可能です。

以下に例を示します。

注文テーブルの主キーを IDENTITY 列として定義し、テーブルデータとシーケンス値の関係を確認します。同一パブリッシャを 55432 番ポート、サブスクライバを 55433 番ポートで起動した別クラスタとして扱います。

（検証用のデータベースクラスタとデータベースを作成）

```
$ initdb -D ./pg55432 --encoding=UTF8 -c port=55432 -c logging_collector=on
$ initdb -D ./pg55433 --encoding=UTF8 -c port=55433 -c logging_collector=on
$ pg_ctl start -D ./pg55432
$ pg_ctl start -D ./pg55433
$ createdb -p 55432 pubdb
$ createdb -p 55433 subdb
```

（パブリッシャ側で IDENTITY 列を持つテーブルを作成して、行を挿入）

```
$ psql -p 55432 pubdb
pubdb=# CREATE TABLE t442_orders (
          order_id bigint GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
          item text NOT NULL);

pubdb=# INSERT INTO t442_orders (item)
          VALUES ('book'), ('pen'), ('note') RETURNING order_id, item;
 order_id | item
----------+------
        1 | book
        2 | pen
```

```
        3 | note
(3 rows)

（作成されたシーケンスとシーケンス値を確認）
pubdb=# SELECT pg_get_serial_sequence('t442_orders', 'order_id');
     pg_get_serial_sequence
--------------------------------
 public.t442_orders_order_id_seq
(1 row)

pubdb=# SELECT * FROM pg_get_sequence_data('t442_orders_order_id_seq');
 last_value | is_called |  page_lsn
------------+-----------+------------
          3 | t         | 0/119F1D40
(1 row)

（全てのテーブルとシーケンスを含むパブリケーションを作成）
pubdb=# CREATE PUBLICATION pub442 FOR ALL TABLES, ALL SEQUENCES;
CREATE PUBLICATION
```

サブスクライバ側には、同じ定義のテーブルを事前に作成してから、サブスクリプションを作成します。論理レプリケーションはDDLを自動的には反映しないため、テーブルやシーケンスを含むオブジェクト定義は、別途そろえておく必要があります。

```
（サブスクライバ側で同じテーブルを作成）
$ psql -p 55433 subdb
subdb=# CREATE TABLE t442_orders (
          order_id bigint GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
          item text NOT NULL);

（サブスクリプションを作成）
subdb=# CREATE SUBSCRIPTION sub442 CONNECTION
          'port=55432 dbname=pubdb user=postgres' PUBLICATION pub442;
NOTICE:  created replication slot "sub442" on publisher
CREATE SUBSCRIPTION
```

（初期同期ができていることを確認）

```
subdb=# SELECT * FROM t442_orders;
 order_id | item
----------+------
        1 | book
        2 | pen
        3 | note
(3 rows)
```

（シーケンス値も同期している）

```
subdb=# SELECT last_value, is_called FROM t442_orders_order_id_seq;
 last_value | is_called
------------+-----------
          3 | t
(1 row)
```

この時点では、サブスクライバ側で次に採番すると 4 が使われるため、テーブルの最大 ID とシーケンス値の整合が取れています。

次に、パブリッシャ側でさらにデータを追加します。

（パブリッシャ側で行を 5 件追加）

```
$ psql -p 55432 pubdb
pubdb=# INSERT INTO t442_orders (item)
          SELECT 'after-sync-' || g FROM generate_series(1, 5) g
          RETURNING order_id, item;
 order_id |     item
----------+--------------
        4 | after-sync-1
        5 | after-sync-2
        6 | after-sync-3
        7 | after-sync-4
        8 | after-sync-5
(5 rows)
```

テーブルデータの変更はサブスクライバ側へ反映されますが、シーケンス値は 3 のままです。

（サブスクライバ側でテーブルの行は同期されている）

```
$ psql -p 55433 subdb
subdb=# SELECT * FROM t442_orders;
 order_id |     item
----------+--------------
        1 | book
        2 | pen
        3 | note
        4 | after-sync-1
        5 | after-sync-2
        6 | after-sync-3
        7 | after-sync-4
        8 | after-sync-5
(8 rows)
```

（サブスクライバ側のシーケンス値は 3 のまま）

```
subdb=# SELECT last_value, is_called FROM t442_orders_order_id_seq;
 last_value | is_called
------------+-----------
          3 | t
(1 row)
```

この状態でサブスクライバを新しい書き込み先として使い始め、INSERT を実行すると、サブスクライバ側のシーケンスが 4 を返そうとします。しかし、order_id = 4 の行はすでにパブリッシャから同期済みなので、ユニーク制約違反になります。

（サブスクライバ側で INSERT を実行）

```
subdb=# INSERT INTO t442_orders (item)
          VALUES ('subscriber-local-before-refresh')
          RETURNING order_id, item;
ERROR:  duplicate key value violates unique constraint "t442_orders_pkey"
DETAIL:  Key (order_id)=(4) already exists.
```

ここで、シーケンス値を明示的に再同期します。

（サブスクライバ側でシーケンスを同期するコマンドを実行）

```
subdb=# ALTER SUBSCRIPTION sub442 REFRESH SEQUENCES;
ALTER SUBSCRIPTION
```

再同期後、サブスクライバ側のシーケンス値は、パブリッシャ側の最新値に追従するので、サブスクライバ側で次に採番される値は 9 になりました。

（サブスクライバ側でシーケンス値を確認）

```
subdb=# SELECT last_value, is_called FROM t442_orders_order_id_seq;
 last_value | is_called
------------+-----------
          8 | t
(1 row)
```

（サブスクライバ側で INSERT を実行 - こんどは成功）

```
subdb=# INSERT INTO t442_orders (item)
           VALUES ('subscriber-local-after-refresh')
           RETURNING order_id, item;
 order_id |              item
----------+--------------------------------
        9 | subscriber-local-after-refresh
(1 row)
```

このようにシーケンスは明示的な指示で同期が行なわれるという動作になっています。

◆ *WAL レベル自動変更*

従来の論理レプリケーションでは、パブリッシャ側で設定パラメータ wal_level に logical を指定たうえでサービス起動しておく必要がありました。PostgreSQL 19 では、設定値としての wal_level が replica の場合でも、論理レプリケーションが必要になった時点で実効 WAL レベルを logical 相当に引き上げられるようになりました。

この機能は前節の検証手順で既に使用しています。initdb を行うときに wal_level = logical とする指定は与えておらず、wal_level のデフォルトは replica ですが、論理レプリケーションが実行可能でした。

新しい設定パラメータ effective_wal_level でその時点の実効 WAL レベルを確認できます。この時、設定パラメータ wal_level は引き続き設定ファイル上の値を示します。

以下のように動作確認できます。

（前節で作成した論理レプリケーション中のパブリケーション側データベースで確認）

```
$ psql -p 55432 pubdb
pubdb=# SHOW wal_level;
 wal_level
-----------
 replica
(1 row)

pubdb=# SHOW effective_wal_level;
 effective_wal_level
---------------------
 logical
(1 row)
```

続いて、論理レプリケーションを終了して、その後に確認してみます。

（サブスクリプションを終了して、レプリケーションスロットを削除）

```
$ psql -p 55433 subdb
subdb=# DROP SUBSCRIPTION sub442;
NOTICE:  dropped replication slot "sub442" on publisher
DROP SUBSCRIPTION
subdb=# \q

$ psql -p 55432 pubdb
pubdb=# SHOW effective_wal_level;
 effective_wal_level
---------------------
 replica
(1 row)
```

このように論理レプリケーションスロットが削除された後で確認すると、effective_wal_level の設定値が wal_level と同じ replica に戻りました。

本機能により、論理レプリケーションやロジカルデコーディングに基づく機能を一時的に使いたい場合に、サービス再起動を回避できるメリットが得られます。

◆ コマンド構文追加

PostgreSQL 19 では、論理レプリケーション関連のコマンド構文も拡張されました。

シーケンス同期のための構文が加わったことに加えて、全テーブル指定時の除外指定が加わり、サブスクリプション接続情報の指定方法が拡張でされました。

CREATE PUBLICATION ... FOR ALL TABLES で、対象から一部のテーブルを除外する EXCEPT 句も使えるようになりました。CREATE PUBLICATION および ALTER PUBLICATION で、ALL TABLES の指定に対して一部のテーブルを EXCEPT 句で除外できます。以下のように使用できます。

```
（テーブルを追加して、EXCEPT 句を使ったパブリケーションを作成）
pubdb=# CREATE TABLE t442a (id int PRIMARY KEY);
pubdb=# CREATE TABLE t442b (id int PRIMARY KEY);
pubdb=# CREATE TABLE t442c (id int PRIMARY KEY);
pubdb=# CREATE PUBLICATION pub442ac
          FOR ALL TABLES EXCEPT ( TABLE t442b, TABLE t442_orders);

（定義を確認：「All tables」が t で、「Except Tables」で除外テーブルが示されている）
pubdb=# \dRp+ pub442ac
                                                    Publication pub442ac
  Owner   | All tables | All sequences | Inserts | Updates | Deletes |
Truncates | Generated columns | Via root | Description
----------+------------+---------------+---------+---------+---------+
-----------+-------------------+----------+-------------
 postgres | t          | f             | t       | t       | t       |
 t         | none              | f        |
Except tables:
    "public.t442b"
    "public.t442_orders"
```

また、CREATE SUBSCRIPTION では、従来の接続文字列指定に替えて、外部サーバオブジェクトを使う「SERVER servername」という指定ができるようになりました。SERVER を使う場合、対象の外部サーバには接続情報を返す関数が登録されている必要があり、サブスクリプション所有者のユーザマッピングと、外部サーバに対する USAGE 権限も必要です。

以下のように使用します。

```
（postgres_fdw の外部サーバとユーザマッピングを作成）
$ psql -p 55433 subdb
```

```
subdb=# CREATE EXTENSION postgres_fdw;
subdb=# CREATE SERVER sv442 FOREIGN DATA WRAPPER postgres_fdw OPTIONS
          (host 'localhost', port '55432', dbname 'pubdb')
subdb=# CREATE USER MAPPING FOR postgres SERVER sv442 OPTIONS
          (user 'postgres');

（レプリケーション対象のテーブルを作成して、サブスクリプションを作成）
subdb=# CREATE TABLE t442a (id int PRIMARY KEY);
subdb=# CREATE TABLE t442c (id int PRIMARY KEY);
subdb=# CREATE SUBSCRIPTION sub442ac SERVER sv442 PUBLICATION pub442ac;
NOTICE:  created replication slot "sub442ac" on publisher
CREATE SUBSCRIPTION
```

接続情報をサブスクリプション定義に直接書くのではなく、外部サーバオブジェクトに寄せられるため、接続先情報の管理を分離できるメリットがあります。特に同じ接続先のサブスクリプションが複数あるときに有用です。

4.5. 拡張モジュール

本節では拡張モジュールに関する主要な拡張を取り上げます。実行プランを調整する新たな拡張モジュール pg_plan_advice、pg_stash_advice が追加されました。また、pg_stat_statements でいくつか改善が適用されました。

4.5.1. pg_plan_advice

プランナの判断を制御するための pg_plan_advice モジュールが追加されました。

プランナに「この SQL ではこの実行プランを使ってほしい」という指示を与えることができます。以下の通り、動作を確認しました。

```
（テストテーブル／テストデータの作成）
db1=# CREATE TABLE t451_1 (id int PRIMARY KEY, name text);
db1=# CREATE TABLE t451_2 (id int PRIMARY KEY,
        t1_id int NOT NULL, name text);
db1=# CREATE TABLE t451_3 (id int PRIMARY KEY,
        t1_id int NOT NULL, t2_id int NOT NULL, name text);
db1=# INSERT INTO t451_1 VALUES (1, 'A'), (2, 'B'), (3, 'C');
```

```
db1=# INSERT INTO t451_2 VALUES
        (101, 1, 'AA'), (102, 1, 'AB'), (103, 2, 'BB'), (104, 3, 'CC');
db1=# INSERT INTO t451_3 VALUES (1001, 1, 101, 'AAA'),
        (1002, 1, 102, 'ABA'), (1003, 2, 103, 'BBB'), (1004, 3, 104, 'CCC');
```

pg_plan_advice モジュールが読み込まれると、EXPLAIN で PLAN_ADVICE オプションが使えるようになります。このオプションを指定すると、実行プランの出力に「Generated Plan Advice:」として、プランアドバイス文字列が付加されます。

（モジュールを読み込んで、plan advice が付加された実行プランを出力）

```
db1=# LOAD 'pg_plan_advice';
db1=# explain (costs off, plan_advice)
        SELECT * FROM t451_2 t2 JOIN t451_1 t1 ON t2.t1_id = t1.id
          JOIN t451_3 t3 ON t2.t1_id = t3.t1_id AND t2.id = t3.t2_id;
                            QUERY PLAN
------------------------------------------------------------------------
 Nested Loop
   ->  Hash Join
         Hash Cond: ((t3.t1_id = t2.t1_id) AND (t3.t2_id = t2.id))
         ->  Seq Scan on t451_3 t3
         ->  Hash
               ->  Seq Scan on t451_2 t2
   ->  Index Scan using t451_1_pkey on t451_1 t1
         Index Cond: (id = t2.t1_id)
 Generated Plan Advice:
   JOIN_ORDER(t3 t2 t1)
   NESTED_LOOP_PLAIN(t1)
   HASH_JOIN(t2)
   SEQ_SCAN(t3 t2)
   INDEX_SCAN(t1 public.t451_1_pkey)
   NO_GATHER(t2 t1 t3)
(15 rows)
```

ここで出力されるアドバイス文字列は、今回プランナが選択した実行プランを選択させることを促す内容です。

設定パラメータ pg_plan_advice.advice にアドバイス文字列を設定することで、実行プラン作成にアドバイスを反映させることができます。複数指定する場合には空白文字で区切ります。いくつか例を示します。

（アドバイス文字列のひとつを指定する）

```
db1=# SET pg_plan_advice.advice TO 'JOIN_ORDER(t3 t2 t1)';
db1=# explain (costs off)
        SELECT * FROM t451_2 t2 JOIN t451_1 t1 ON t2.t1_id = t1.id
          JOIN t451_3 t3 ON t2.t1_id = t3.t1_id AND t2.id = t3.t2_id;
                              QUERY PLAN
-----------------------------------------------------------------------
 Nested Loop
   ->  Hash Join
         Hash Cond: ((t3.t1_id = t2.t1_id) AND (t3.t2_id = t2.id))
         ->  Seq Scan on t451_3 t3
         ->  Hash
               ->  Seq Scan on t451_2 t2
   ->  Index Scan using t451_1_pkey on t451_1 t1
         Index Cond: (id = t2.t1_id)
 Supplied Plan Advice:
   JOIN_ORDER(t3 t2 t1) /* matched */
(10 rows)
```

（結合の順番を変更して指定する）

```
db1=# SET pg_plan_advice.advice TO 'JOIN_ORDER(t2 t1 t3)';
db1=# explain (costs off)
        SELECT * FROM t451_2 t2 JOIN t451_1 t1 ON t2.t1_id = t1.id
          JOIN t451_3 t3 ON t2.t1_id = t3.t1_id AND t2.id = t3.t2_id;
                             QUERY PLAN
-------------------------------------------------------------------
 Merge Join
   Merge Cond: ((t2.t1_id = t3.t1_id) AND (t2.id = t3.t2_id))
   ->  Sort
         Sort Key: t2.t1_id, t2.id
         ->  Hash Join
               Hash Cond: (t2.t1_id = t1.id)
               ->  Seq Scan on t451_2 t2
```

```
               ->  Hash
                     ->  Seq Scan on t451_1 t1
   ->  Sort
         Sort Key: t3.t1_id, t3.t2_id
         ->  Seq Scan on t451_3 t3
 Supplied Plan Advice:
   JOIN_ORDER(t2 t1 t3) /* matched */
(14 rows)
```

（結合の順番と、ネステッドループの２つをスペース区切りで指定する）

```
db1=# SET pg_plan_advice.advice
        TO 'JOIN_ORDER(t2 t1 t3) NESTED_LOOP_PLAIN(t1)';
db1=# EXPLAIN (COSTS OFF)
        SELECT * FROM t451_2 t2 JOIN t451_1 t1 ON t2.t1_id = t1.id
          JOIN t451_3 t3 ON t2.t1_id = t3.t1_id AND t2.id = t3.t2_id;
                      QUERY PLAN
-------------------------------------------------------
 Merge Join
   Merge Cond: (t2.id = t3.t2_id)
   Join Filter: (t3.t1_id = t2.t1_id)
   ->  Nested Loop
         ->  Index Scan using t451_2_pkey on t451_2 t2
         ->  Index Scan using t451_1_pkey on t451_1 t1
               Index Cond: (id = t2.t1_id)
   ->  Sort
         Sort Key: t3.t2_id
         ->  Seq Scan on t451_3 t3
 Supplied Plan Advice:
   JOIN_ORDER(t2 t1 t3) /* matched */
   NESTED_LOOP_PLAIN(t1) /* matched */
(13 rows)
```

pg_plan_advice.advice に指定するアドバイス文字列の違いにより、同じ SQL 文で違う実行プランが選択されていることがわかります。また、「/* matched */ 」と出力されている箇所で、plan advice 文字列を制御に利用していることがわかります。

アドバイス文字列に指定できるタグ（JOIN_ORDER など）の一覧は、PostgreSQL ドキュメントに記載されています。スキャン方式、結合順序、結合方式、パーティションワイズ使用、セミ結合方式、パラレル問合せ関連、について指定が可能です。

4.5.2. pg_stash_advice

pg_stash_advice モジュールは、plan advice 文字列を動的共有メモリに保存し、自動的に適用できるようにするものです。利用するには postgresql.conf で設定 shared_preload_libraries に 「pg_stash_advice」を加えて、サービス再起動する必要があります。このとき「pg_plan_advice」の事前ロードは必ずしも必要ではありません。

以下に動作確認を行ないます。

```
（shared_preload_libraries を設定して、サービス再起動）
$ vi $PGDATA/postgresql.conf
   shared_preload_library = 'pg_stash_advice'
$ pg_ctl restart
```

pg_stash_advice モジュールは格納領域（スタッシュ）に SQL 文のクエリ ID（問合せ識別子）とアドバイス文字列の組み合わせを登録しておくことで、そのプランアドバイスを利用できるようになります。クエリ ID は SQL 文を 64bit 整数のハッシュ値にしたものです。

```
（使用したいデータベースで CREATE EXTENSION を行う）
db1=# CREATE EXTENSION pg_stash_advice;

（格納領域を my_stash1 という名前で作成）
db1=# SELECT pg_create_advice_stash('my_stash1');
 pg_create_advice_stash
------------------------

(1 row)

（登録する予定の SELECT 文のクエリ ID の確認）
db1=# explain (verbose)
        SELECT * FROM t451_2 t2 JOIN t451_1 t1 ON t2.t1_id = t1.id
          JOIN t451_3 t3 ON t2.t1_id = t3.t1_id AND t2.id = t3.t2_id;
                                    QUERY PLAN
-------------------------------------------------------------------------------
```

```
 Nested Loop  (cost=40.15..68.49 rows=6 width=120)
   Output: t2.id, t2.t1_id, t2.name, t1.id, t1.name, t3.id, t3.t1_id, t3.t2_id,
t3.name
   Inner Unique: true
   ->  Hash Join  (cost=40.00..67.23 rows=6 width=84)
         Output: t2.id, t2.t1_id, t2.name, t3.id, t3.t1_id, t3.t2_id, t3.name
         Inner Unique: true
         Hash Cond: ((t3.t1_id = t2.t1_id) AND (t3.t2_id = t2.id))
         ->  Seq Scan on public.t451_3 t3  (cost=0.00..21.30 rows=1130 width=44)
               Output: t3.id, t3.t1_id, t3.t2_id, t3.name
         ->  Hash  (cost=22.00..22.00 rows=1200 width=40)
               Output: t2.id, t2.t1_id, t2.name
               ->  Seq Scan on public.t451_2 t2  (cost=0.00..22.00 rows=1200
width=40)
                     Output: t2.id, t2.t1_id, t2.name
   ->  Index Scan using t451_1_pkey on public.t451_1 t1  (cost=0.15..0.21 rows=1
width=36)
         Output: t1.id, t1.name
         Index Cond: (t1.id = t2.t1_id)
 Query Identifier: 924161323295403897
(17 rows)

（格納領域 my_stash1 にクエリ ID とアドバイス文字列文字列を登録）
db1=# SELECT pg_set_stashed_advice('my_stash1',
        924161323295403897, 'JOIN_ORDER(t2 t1 t3)');
 pg_set_stashed_advice
-----------------------

(1 row)
```

次に、格納領域の内容を利用して、実行プランが選択されることを確認します。格納領域に登録を行ったセッションとは別のセッションでも利用可能です。

```
（格納領域 my_stash1 をプランアドバイスに利用することを指定）
db1=# SET pg_stash_advice.stash_name = 'my_stash1';
```

（格納領域 my_stash1 の内容を利用した実行プランの確認）

```
db1=# explain (verbose, costs off)
        SELECT * FROM t451_2 t2 JOIN t451_1 t1 ON t2.t1_id = t1.id
          JOIN t451_3 t3 ON t2.t1_id = t3.t1_id AND t2.id = t3.t2_id;
                                    QUERY PLAN
--------------------------------------------------------------------------------
 Merge Join
   Output: t2.id, t2.t1_id, t2.name, t1.id, t1.name, t3.id, t3.t1_id, t3.t2_id,
t3.name
   Merge Cond: ((t2.t1_id = t3.t1_id) AND (t2.id = t3.t2_id))
   ->  Sort
         Output: t2.id, t2.t1_id, t2.name, t1.id, t1.name
         Sort Key: t2.t1_id, t2.id
         ->  Hash Join
               Output: t2.id, t2.t1_id, t2.name, t1.id, t1.name
               Inner Unique: true
               Hash Cond: (t2.t1_id = t1.id)
               ->  Seq Scan on public.t451_2 t2
                     Output: t2.id, t2.t1_id, t2.name
               ->  Hash
                     Output: t1.id, t1.name
                     ->  Seq Scan on public.t451_1 t1
                           Output: t1.id, t1.name
   ->  Sort
         Output: t3.id, t3.t1_id, t3.t2_id, t3.name
         Sort Key: t3.t1_id, t3.t2_id
         ->  Seq Scan on public.t451_3 t3
               Output: t3.id, t3.t1_id, t3.t2_id, t3.name
 Query Identifier: 924161323295403897
 Supplied Plan Advice:
   JOIN_ORDER(t2 t1 t3) /* matched */
(24 rows)
```

EXPLAIN の出力の「/* matched */」と出力されている箇所で、アドバイス文字列が適用されたことがわかります。選択された実行プランもアドバイスに従って、t2 と t1 を結合して、その結果を t3 と結合する、とい

う順序になっています。

pg_stash_advice のアドバイス文字列格納領域はデフォルトでは「$PGDATA/pg_stash_advice.tsv」ファイルに書き出されて永続化されて、PostgreSQL 再起動後も維持されます。ただし、システムテーブル上ではないため、PostgreSQL の非正常停止に対して必ずしも安全ではありません。場合によっては、作り直しが必要となるはずです。

現在のアドバイス格納領域の内容は以下のように確認できます。

```
db1=# SELECT * FROM pg_get_advice_stash_contents('my_stash1');
  stash_name   |       query_id       |     advice_string
---------------+----------------------+-----------------------
 my_stash1     | 924161323295403897 | JOIN_ORDER(t2 t1 t3)
(1 row)
```

4.5.3. pg_stat_statements 改善

pg_stat_statements 拡張の改善点を見ていきます。

pg_stat_statements を利用するには postgresql.conf で「shared_preload_libraries = 'pg_stat_statements'」とモジュールの設定をして再起動する必要があります。

```
（shared_preload_libraries に pg_stat_statements を設定して、再起動）
$ vi $PGDATA/postgresql.conf
   shared_preload_libraries = 'pg_stat_statements'

$ pg_ctl restart
```

pg_stat_statements には 2 つの改善点があります。

1 点目は FETCH クエリの取得件数が異なるものをまとめて扱うようになりました。

以下のように動作確認を行ないました。

```
（pg_stat_statements 拡張を利用、最初に統計情報をリセットする）
db1=# CREATE EXTENSION pg_stat_statements;
db1=# SELECT pg_stat_statements_reset();
   pg_stat_statements_reset
-------------------------------
 2026-06-09 15:06:07.046167+09
(1 row)
```

```
（10 件の FETCH を行う）
db1=# BEGIN;
db1=*# DECLARE c CURSOR FOR SELECT * FROM generate_series(1,1000);
db1=*# FETCH 10 FROM c;
 generate_series
-----------------
               1
               2
               3
               : 出力後略

（続いて 100 件の FETCH を行う）
db1=*# FETCH 100 FROM c;
 generate_series
-----------------
              11
              12
              13
               : 出力後略

（さらに 1000 件の FETCH を行う）
db1=*# FETCH 1000 FROM c;
 generate_series
-----------------
             111
             112
             113
              :  出力中略
             998
             999
            1000
(890 行)

db1=*# COMMIT;
```

```
（ここで FETCH 文のステートメント統計を確認）
db1=# SELECT query, calls FROM pg_stat_statements WHERE query LIKE 'FETCH%';
      query      | calls
-----------------+-------
 FETCH $1 FROM c |     3
(1 row)
```

これは PostgreSQL 18.x で実行すると、下記のように取得件数ごとに別々に扱われていました。

```
（PostgreSQL 18.x の場合の出力）
       query       | calls
-------------------+-------
 FETCH 10 FROM c   |     1
 FETCH 1000 FROM c |     1
 FETCH 100 FROM c  |     1
(3 rows)
```

2 点目としては、実行した汎用プランとカスタムプランの数を確認できるようになりました。そのために pg_stat_statements ビューに新たな列 generic_plan_calls と custom_plan_calls が追加されました。

以下のように動作を確認しました。

```
（検証用テーブル作成とステートメント統計のリセット）
db1=# CREATE TABLE t453 (id integer PRIMARY KEY, name text);
db1=# INSERT INTO t453 SELECT i, i::text || 'name'
        FROM generate_series(1,10000) AS i;
db1=# ANALYZE t453;
db1=# SELECT pg_stat_statements_reset();
   pg_stat_statements_reset
-------------------------------
 2026-06-17 14:31:49.665609+09
(1 row)

（汎用プランが選択される可能性のある PREPARE/EXECUTE で SQL を実行する）
db1=# PREPARE q(int) AS SELECT COUNT(*) FROM t453 WHERE id = $1;
db1=# EXECUTE q(10) \watch count=15 1
  → \watch により 1 秒ごと 15 回実行する
```

```
Mon Jun 17 14:37:40 2026 (every 1s)

 count
-------
     1
(1 row)

Mon Jun 17 14:37:41 2026 (every 1s)

 count
-------
     1
(1 row)
    :  出力後略（１５回実行される）

（pg_stat_statements ビューで汎用プラン、カスタムプランの回数を確認）
db1=# \x
db1=# SELECT query, calls, generic_plan_calls, custom_plan_calls
        FROM pg_stat_statements WHERE query LIKE 'PREPARE%';
-[ RECORD 1 ]------+-----------------------------------------------------------
query              | PREPARE q(int) AS SELECT COUNT(*) FROM t453 WHERE id = $1
calls              | 15
generic_plan_calls | 10
custom_plan_calls  | 5
```

デフォルト設定の動作では、プリペアドステートメントの実行を繰り返すと、最初の 5 回はカスタムプランで実行して、そのとき汎用プランがカスタムプラン同等以上に低コストであったなら、以降は汎用プランが選択されます。上記の pg_stat_statements ビューで、カスタムプラン 5 回、汎用プラン 10 回となっていますので、その振る舞いが確認できていると言えます。

5. 非互換変更

PostgreSQL 18.x から 19.x へのアップグレードでは、いくつか非互換の変更点があります。本節では主な非互換の変更点を取り上げます。

5.1.1. 設定パラメータの変更

変更された設定パラメータの一覧を以下の表に示します。設定値が拡張されていても、デフォルト値が変わらず、従来の設定値も許容されている項目については、この一覧に含めていません。また、特に説明が必要な項目は、小節を設けて記載しています。

設定パラメータ名	説明
default_toast_compression	デフォルト値が pglz から lz4（利用可能時）に変更されました。
escape_string_warning	廃止されました。これからは常に off の動作になります。 古いサーバーとの互換性を維持するため、クライアントツールでは引き続き本設定パラメータを認識します。
io_workers	io_workers が廃止され、I/O ワーカ関連パラメータ体系の再設計により次のパラメータに分割されました。 io_min_workers、io_max_workers、 io_worker_idle_timeout、io_worker_launch_interval
jit	デフォルト値が on から off に変更されました。
log_lock_waits	デフォルト値が off から on に変更されました。
max_locks_per_transaction	デフォルト値が 64 から 128 に変更されました。
standard_conforming_strings	参照専用の項目になり、値は常に on となりました。 詳細を後述します。

◆ パスワード期限警告（password_expiration_warning_threshold）

パスワードの有効期限を警告するパラメータ password_expiration_warning_threshold が追加されました。デフォルト値は 7 日間です。

下記の実行例では、パスワード認証が必要となるように設定したうえで、翌日に期限切れとなるユーザを作成して、そのユーザでパスワードログインを実行しました。

```
$ vi $PGDATA/pg_hba.conf
    （以下のエントリを先頭に追加）
   local   db1      usr1                           scram-sha-256

$ pg_ctl reload
$ psql -U postgres -d db1
db1=# CREATE USER usr1 PASSWORD 'pass' VALID UNTIL '2026/06/05 00:00:00';
db1=# \q

$ date
Thu Jun  4 12:51:32 JST 2026
$ psql -U usr1 -d db1
Password for user usr1:
WARNING:  role password will expire soon
DETAIL:  The password for role "usr1" will expire in 11 hours.
```

「role password will expire soon」の WARNING メッセージと共に、ロール名とパスワード有効期間の残り時間の詳細メッセージが出力されました。

◆ *md5 パスワード認証で警告（md5_password_warnings）*

設定パラメータ md5_password_warnings が on のときの振る舞いとして、MD5 でパスワードを設定した時だけでなく MD5 を使用してログインした際にも警告が出力されるようになりました。MD5 パスワード格納は PostgreSQL 18 から非推奨となっていました。

MD5 パスワードを設定した時と、md5 認証で MD5 パスワードを使用してログインした時に出る警告メッセージは以下の通りです。

```
$ psql -U postgres -d db1
db1=# SET password_encryption = md5;
db1=# \password usr1
Enter new password for user "usr1":
Enter it again:
WARNING:  setting an MD5-encrypted password
DETAIL:  MD5 password support is deprecated and will be removed in a future
release of PostgreSQL.
HINT:  Refer to the PostgreSQL documentation for details about migrating to
```

```
another password type.
db1=# \q
$ vi $PGDATA/pg_hba.conf
    （前節で加えた先頭エントリを scram-sha-256 から md5 に書き換え）
   local   db1      usr1                           md5

$ pg_ctl reload
$ psql -U usr1 -d db1
Password for user usr1:
WARNING:  authenticated with an MD5-encrypted password
DETAIL:  MD5 password support is deprecated and will be removed in a future
release of PostgreSQL.
```

いずれも警告が出るのみであって、パスワード設定や接続自体は可能です。

◆ *standard_conforming_strings = off が廃止*

通常の文字列リテラルがバックスラッシュを含む場合に、バックスラッシュを単なる文字としてそのまま扱うか（設定値 :on）、バックスラッシュをエスケープシーケンスとして解釈するか（設定値 :off）を変更できるパラメータ standard_conforming_strings が off に設定できなくなりました。常に on として動作します。

psql で SET 文で設定すると以下のようにエラーが出ます。

```
db1=> SHOW standard_conforming_strings;
 standard_conforming_strings
-----------------------------
 on
(1 row)

db1=# set standard_conforming_strings TO off;
ERROR:  non-standard string literals are not supported
```

また postgresql.conf に standard_conforming_strings = on を設定して、PostgreSQL 起動を試みると、以下のようにエラーが出て起動に失敗します。

```
$ pg_ctl start
waiting for server to start....2026-06-11 16:15:31.820 JST [18005] LOG:
non-standard string literals are not supported
```

```
2026-06-11 16:15:31.820 JST [18005] FATAL:  configuration file
"/var/lib/pgsql/data19/postgresql.conf" contains errors
 stopped waiting
pg_ctl: could not start server
```

5.1.2. システムシステビューの変更

PostgreSQL 19 では、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_subscription_stats ビュー*

pg_stat_subscription_stats ビューの sync_error_count 列の列名が sync_table_error_count 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（１８まで）	変更後（１９から）
sync_error_count	（名前変更）sync_table_error_count
なし（１９で追加）	sync_seq_error_count
なし（１９で追加）	confl_update_deleted

5.1.3. その他の非互換変更点

各種の非互換の変更点について記載します。

◆ *radius 認証方式廃止*

PostgreSQL の RADIUS 認証は UDP 経由でのみ実装されていましたが、この方式には本質的に回避できないセキュリティ上の問題があるため、RADIUS のサポートが廃止されました。

◆ *改行を含むグローバルオブジェクト識別子の禁止*

セキュリティ上の問題を防止するため、データベース名、ロール名、およびテーブルスペース名に改行文字（CR / LF）を含めることが禁止されました。また、これらの文字を含む名前を使用しているクラスタからの pg_upgrade は実行できません。

改行文字を含めたオブジェクトの作成を試みると、以下のようなメッセージが表示されます。

```
（PostgreSQL 19 での動作）
db1=# CREATE DATABASE "t_database1;
t_database2";
ERROR:  database name "t_database1;
t_database2" contains a newline or carriage return character
```

◆ 文字エンコーディング MULE_INTERNAL 廃止

MULE_INTERNAL は実装が複雑で、利用実績もほとんどないため削除されました。MULE_INTERNAL を使用する既存のデータベースについては、別の文字エンコーディングへ移行するため、ダンプ/リストアによるデータ移行が必要となります。

◆ イベントタイプ BUFFERPIN が改名

待機イベントタイプ BUFFERPIN が BUFFER に名称変更されました。今後、バッファロック関連の待機イベントが同じ待機イベント分類に追加される予定であり、従来の名称では対象範囲を適切に表現できなくなるためです。

◆ CREATE SCHEMA 内でのオブジェクト作成順序

CREATE SCHEMA 実行時に作成されるオブジェクトについて、PostgreSQL が内部的に実施していた並び替え処理が廃止されました。従来はオブジェクト間の依存関係を考慮して作成順序を変更していましたが、この仕組みは完全ではありませんでした。今後は CREATE SCHEMA 文で指定された順序通りにオブジェクトが作成されます。なお、外部キーのみ例外として、最後に作成されます。

そのため、外部キーを除き CREATE SCHEMA 内の前方参照はサポートされなくなり、以下のようにエラーが発生するようになりました。

```
（PostgreSQL 19 での動作）
db1=# CREATE SCHEMA t_schema
         CREATE VIEW t_view AS
           SELECT * FROM t_table
         CREATE TABLE t_table (id integer);
ERROR:  relation "t_table" does not exist
LINE 3:     SELECT * FROM t_table
```

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	2026/6/30	初版作成