PostgreSQL 18 検証レポート

1.0 版
2025 年 8 月 15 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 18 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 18 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。
2025 年 5 月 8 日にリリースされた PostgreSQL 18 beta1 を使用して検証を行いました。その後、2025 年 7 月 17 日にリリースされた beta2 及びその後の更新に対応して、いくつか記載を修正しています。

2. 概要

PostgreSQL 18 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
- 非同期 I/O
- Btree インデックスのスキップスキャン
- プランナ改善（WHERE 句内に OR 条件がある場合の最適化、ほか）
- GIN インデックスの並列作成
- 大量のテーブルアクセス時のロック処理の改善
- 組み込み照合順序 pg_unicode_fast

メジャーバージョンアップ
- プランナ統計情報の移行
- チェック処理の並列化オプションの追加
- --swap オプションの追加

SQL 機能
- 仮想生成列の追加
- 更新系 DML の RETURNING 句へ OLD, NEW の追加
- UUID v7 関数の追加
- 非決定論的な照合順序利用時の LIKE 句のサポート
- CASEFOLD 関数の追加
- 各種制約の追加（WITHOUT OVERLAPS、ほか）
- ラージオブジェクトに対するデフォルト権限設定のサポート

セキュリティ機能
- TLS v1.3 暗号スイートの追加
- postgres_fdw、dblink への SCRAM 認証のパススルーオプションの追加

運用管理

- プランナ統計情報のダンプ／リストア
- EXPLAIN 文の強化
  ・ANALYZE オプション追加時の BUFFERS オプション自動追加
  ・インデックス参照回数の表示追加
  ・無効ノードの明示的な出力、ほか
- pg_stat_all_tables への VACUUM、ANALYZE 総実行時間の追加
- 接続単位での I/O・WAL 利用量の取得
- 論理レプリケーション時の書き込み衝突の情報詳細化

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 18 ドキュメント内のリリースノート（以下 URL）に記載されています。

https://www.postgresql.org/docs/18/release-18.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 18（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 9.x (x >= 5) (x86_64) 互換環境の仮想マシンを使用しました。

本検証は具体的な特定マシン上の性能の提示や大規模サーバにおける性能の検証は意図していません。性能を検証する場合も、旧バージョンや新機能を使わない場合との比較を行っています。

3.3. インストール

gcc、zlib-devel、readline-devel 、libicu-devel、openssl-devel、libzstd-devel、lz4-devel、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、以下のオプションにてソースコードのビルドを行いました。事前に postgres ユーザで読み書き可能な /usr/local/pgsql/18 ディレクトリを用意したうえで、postgres ユーザにて実行しました。

```
（以下、postgres ユーザで実行）
$ wget
https://ftp.postgresql.org/pub/source/v18beta1/postgresql-18beta1.tar.bz2
                        《実際は１行、リリース後は「v18.0/postgresql-18.0.tar.bz2」》
$ tar jxf postgresql-18beta1.tar.bz2
$ cd postgresql-18beta1
$ ./configure --prefix=/usr/local/pgsql/18 --enable-debug --with-openssl \
    --with-icu --with-zstd --with-lz4 --with-liburing --with-uuid=e2fs
$ make world-bin
$ make install-world-bin
```

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

環境変数を設定するファイルを書き出して、適用します。postgres ユーザで読み書き可能な /var/lib/pgsql ディレクトリがあるものとし、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 > ~/pg18.env <<'EOF'
VER=18
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
$ . ~/pg18.env
$ cat ~/pg18.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。

```
$ initdb --no-locale --encoding=UTF8
```

設定ファイルに最小限の設定を与えます。これによりログメッセージがファイルに蓄積されます。

```
$ cat >> $PGDATA/postgresql.conf << EOF
logging_collector = on
EOF
```

PostgreSQL を起動します。

```
$ pg_ctl start
```

検証用のデータベースを作成します。

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

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

```
$ psql -U postgres -d db1
psql (18beta1)
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/18/share/doc/html/
```

また、以下 URL にて PostgreSQL 18 のドキュメントが公開されています。いずれも英語となります。

```
https://www.postgresql.org/docs/18/
```

4.1. 性能向上

4.1.1. 非同期 I/O

PostgreSQL がファイル読み書きを行うときの非同期 I/O（Asynchronous I/O、AIO）の仕組みが導入されました。PostgreSQL 18 時点では実際に使われる個所は、共有バッファに載せるためにテーブル等のデータを読み込む処理部分です。

以下の設定パラメータが導入されました。

設定パラメータ	意味
io_max_combine_limit	稼働中に変更できる io_combine_limit の上限、デフォルト 128kB。
io_combine_limit	まとめて I/O 処理する最大サイズ。デフォルト 128kB。
io_method	sync（同期 I/O で実行）、 worker（ワーカープロセスで非同期 I/O 実行、デフォルト）、 io_uring（io_uring を使って非同期 I/O 実行）、から I/O 方式を選択。
io_max_concurrency	1 プロセスが同時実行する最大 I/O 数。 -1（デフォルト）は shared_buffers サイズから自動決定。
io_workers	I/O worker プロセスの数（1 ～ 32、デフォルト 3）。

非同期 I/O は worker モードでデフォルト有効です。稼働中 PostgreSQL のプロセスを確認すると以下のように 3 つの io worker プロセスが動作していることがわかります。sync と io_uring モードの場合、このような子プロセスは生じません。

```
$ ps x
    PID TTY      STAT TIME COMMAND
   1472 ?        Ss     0:00 /usr/local/pgsql/18/bin/postgres
   1473 ?        Ss     0:00 postgres: logger
   1474 ?        Ss     0:04 postgres: io worker 0
   1475 ?        Ss     0:04 postgres: io worker 2
   1476 ?        Ss     0:04 postgres: io worker 1
   1477 ?        Ss     0:00 postgres: checkpointer
（後略）
```

io_method = io_uring の設定を使用するには、Linux カーネルが対応していること、使用可能なようにカーネル設定が行なわれていることが必要です。io_uring は Red Hat Enterprise Linux であれば 9.3 以降でサポー

トされます。以下のように kernel.io_uring_disabled に 0 を設定すると、全てのプロセスで io_uring が利用可能になります。

```
（root ユーザで io_uring を使用可能にするカーネル設定を付与）
# sysctl -w kernel.io_uring_disabled=0
# cat >> /etc/sysctl.conf <<EOS
kernel.io_uring_disabled=0
EOS
```

非同期 I/O の効果を調べるため、いくつか性能検証を行ないました。

◆pgbench

shared_buffers と io_method の設定を変えて、標準シナリオの pgbench を繰り返し実行しました。

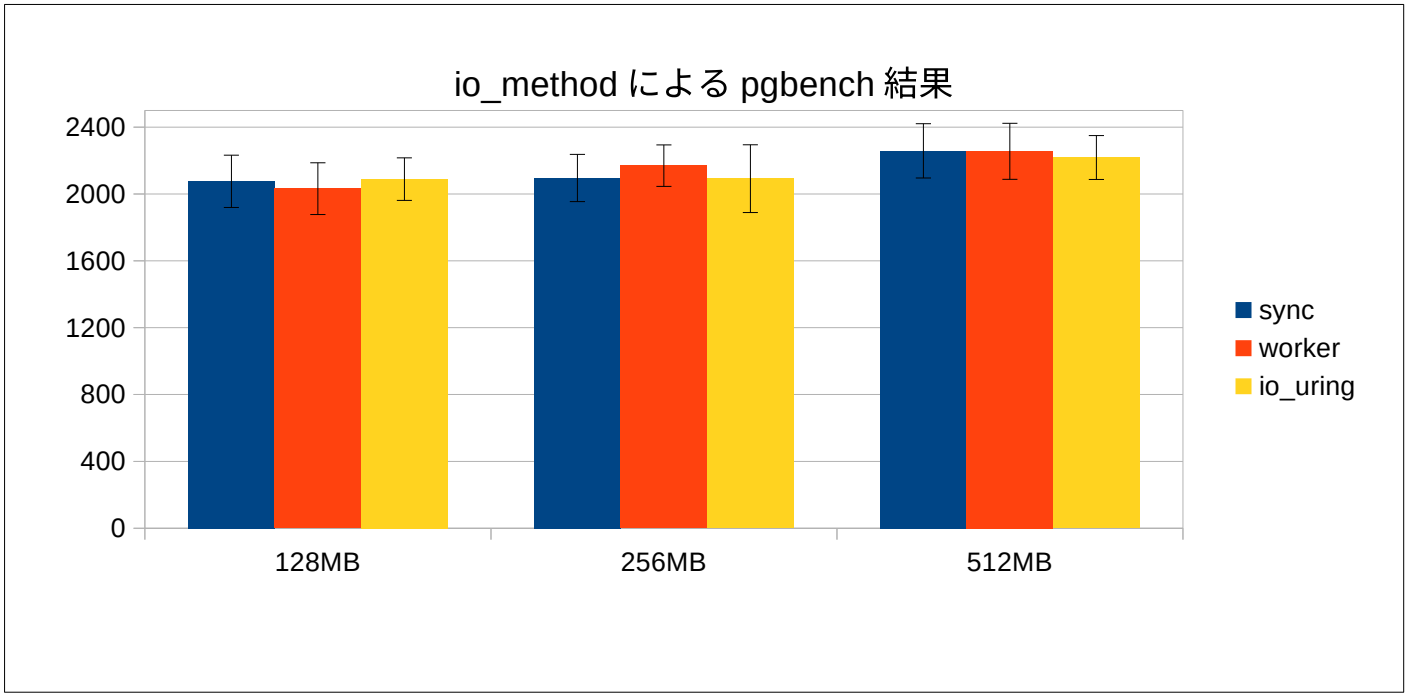
データ初期化を毎回行い、pgbench 実行前に Linux のカーネルキャッシュを捨てる命令を実行しています。

カーネルキャッシュを捨てる操作を行うスクリプトを作成し、それを postgres ユーザから実行できるように visudo コマンドで sudo 可能なユーザ・コマンドを加える手順を行なっています。このスクリプトはこの後でも使用します。

```
（root で実行）
# visudo
  （以下エントリを追加）
  postgres    ALL=(ALL)    NOPASSWD: /var/lib/pgsql/dropcache.sh

# su - postgres

（キャッシュを無効化するスクリプト）
$ cat > /var/lib/pgsql/dropcache.sh <<EOS
#!/bin/sh
echo 3 > /proc/sys/vm/drop_caches
EOS

$ chmod +x /var/lib/pgsql/dropcache.sh

（データサイズ 450MB にて標準シナリオ pgbench を実行）
$ pg_ctl restart
$ pgbench -i -s 30 --fillfactor=90 db1
$ sudo /var/lib/pgsql/dropcache.sh
```

```
$ pgbench -n -c 64 -T 180 db1
```

以下の結果が得られました。誤差線は標準偏差を上下に伸ばしたものです。



shared_buffers の大きさがいずれであっても、io_method による差異がほとんどあらわれませんでした。

これは本ワークロードの負荷に占める、非同期 I/O が効果を発揮する個所の割合が小さいものであるからと考えられます。WAL 書き込みや、バッファ書き出し、また、ファイルからの少ないブロック数の読み込みにおいては非同期 I/O は働きません。また、読み込むデータがカーネルキャッシュに載ってしまえば、以降は効果は見えなくなります。

◆ *CREATE TABLE AS SELECT*

続いて、CREATE TABLE … AS SELECT …（CTAS と呼ばれます）で大きなテーブルを全読み込みして書き出す処理について性能比較を実施しました。以下の手順を io_method を変えて繰り返し実行します。shared_buffers は 512MB としています。

（4GB 程度になるテストテーブルを作成）

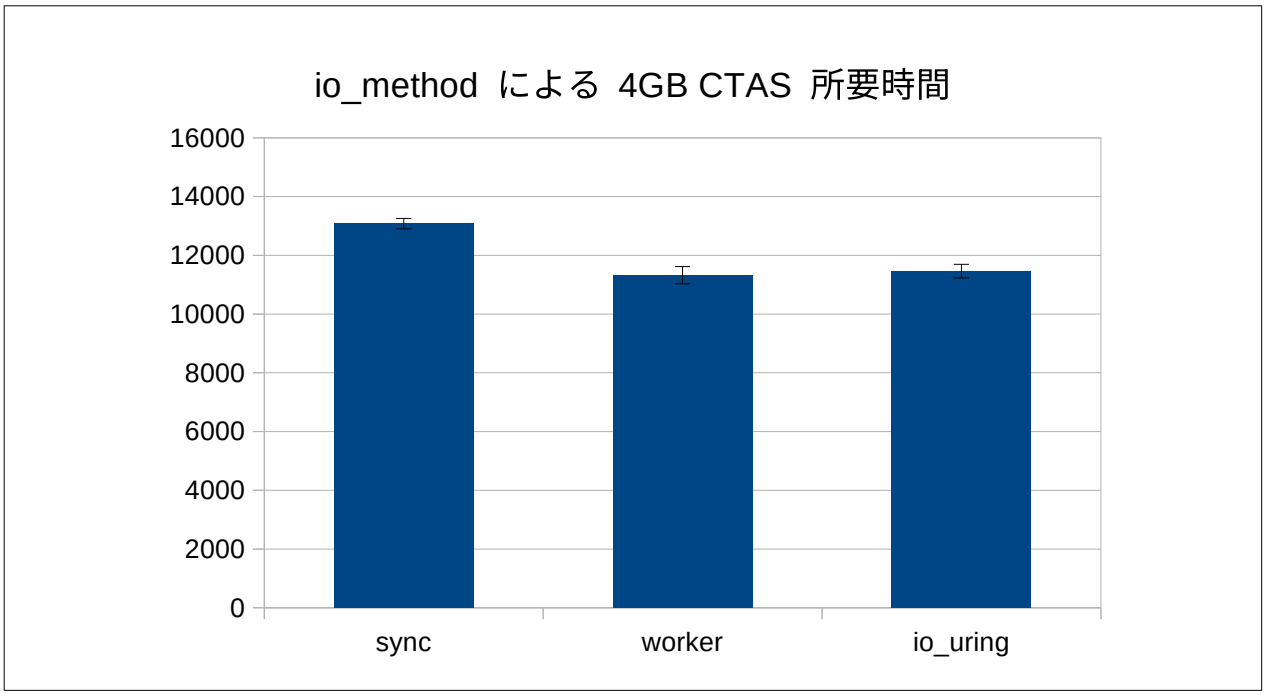
```
$ psql -d db1 <<'EOS'
DROP TABLE IF EXISTS t411;
CREATE UNLOGGED TABLE t411 (
  id int PRIMARY KEY, c1 text, c2 text, c3 text, c4 text, c5 text, c6 text,
  c7 text, c8 text, c9 text, ts1 timestamp);
```

```
INSERT INTO t411 SELECT g, repeat(md5('1' || g), 100),
  repeat(md5('2' || g), 100), repeat(md5('3' || g), 100),
  repeat(md5('4' || g), 100), repeat(md5('5' || g), 100),
  repeat(md5('6' || g), 100), repeat(md5('7' || g), 100),
  repeat(md5('8' || g), 100), repeat(md5('9' || g), 100), now()
  FROM generate_series(1, 5000000) g;
EOS
```

（カーネルキャッシュ無効化、PostgreSQL 再起動をしてから、一時テーブルに CTAS を実行）

```
$ sudo /var/lib/pgsql/dropcache.sh
$ pg_ctl restart
$ psql -d db1 <<'EOS'
    \timing on
    CREATE TEMP TABLE tt411 AS SELECT * FROM t411;
EOS
Timing is on.
SELECT 5000000
Time: 11853.050 ms (00:11.853)
```

次の結果が得られました。誤差線は標準偏差を上下に伸ばしたものです。



sync が最も遅く、worker と io_uring がそれぞれ 15%ほど所要時間が短縮されています。ばらつきが小さく、差異は確実なものと言えます。大きなブロックを一括して読み込むときに非同期 I/O が効果を発揮した結果と考えられます。前述の通り、非同期 I/O は書き出し部分には効きませんので、差異は読み込み速度の違いです。

本検証環境ではローカルのストレージを使用しています。ストレージ装置がネットワーク越しに在って、処理能力は高いのだけれどレスポンスは遅い、といった状況ですと、非同期 I/O によって処理単位ごとに応答を待つ必要がなくなって、性能改善により大きく貢献することが見込まれます。

◆ *非同期 IO のモニタビュー*

非同期 I/O の動作状況を確認できるシステムビューも追加されています。

以下は、前節の CREATE TABLE AS SELECT の試験を実行中にシステムビュー pg_aios を参照した結果です。非同期に実行されている複数の I/O 処理がそれぞれの行で報告されます。

（非同期 I/O 動作中の pg_aios ビュー出力 - io_uring の場合）

```
db1=# \x
db1=# SELECT * FROM pg_aios;
-[ RECORD 1 ]---+-------------------------------------------------
pid             | 13792
io_id           | 259
io_generation   | 11160
state           | SUBMITTED
operation       | readv
off             | 778174464
length          | 131072
target          | smgr
handle_data_len | 16
raw_result      |
result          | UNKNOWN
target_desc     | blocks 357136..357151 in file "base/16384/23183"
f_sync          | f
f_localmem      | f
f_buffered      | t
-[ RECORD 2 ]---+-------------------------------------------------
pid             | 13792
io_id           | 260
```

```
io_generation    | 7066
state            | COMPLETED_SHARED
operation        | readv
off              | 778305536
length           | 131072
target           | smgr
handle_data_len  | 16
raw_result       | 131072
result           | OK
target_desc      | blocks 357152..357167 in file "base/16384/23183"
f_sync           | f
f_localmem       | f
f_buffered       | t
```

target_desc 列を見ると、同じファイル（テーブル等に対応したファイル）の 357136 番から 357151 番までのブロックと、357152 番から 357167 番までのブロックが並行して処理されていることが分かります。state 列でそれぞれの処理段階が報告されます。このビュー出力は io_method が worker や io_uring の場合だけでなく、sync の場合にも出力されます。ただし、その場合には、1 つのバックエンドからの SQL 実行に対応するエントリは 1 つだけになります。sync モードのときでも、巨大なファイル読み込みを伴う問い合わせ実行が遅いときに、今どのファイルのどこを読んでいるのかを確認するためにこのビューを活用できます。

4.1.2. Btree インデックスのスキップスキャン

マルチカラムの Btree インデックスを Index Scan する場合にスキップスキャンと呼ばれるスキャン方法に対応しました。スキップスキャンとは、インデックスからある範囲の値を取り出すときに、範囲の端から端までのリーフページを見るのではなく、不要な部分を飛ばして読むことで負荷を減らす方式です。インデックスの先頭項目の値の種類が少なくて、条件がインデックスの 2 番目以降の列にある場合に、特に有効です。

以下のように、PostgreSQL17.5 と比較して動作検証を行いました。

（サンプルデータを作成 - id0 列の値は 5 種類、id1 列と id2 列の値は 1000 種類）

```
db1=# CREATE TABLE t412 (id0 text, id1 int, id2 int, v text,
          PRIMARY KEY (id0, id1, id2));
db1=# INSERT INTO t412 SELECT
          substr(md5((g%5)::text), 1, 10), g / 1000, g % 1000, g::text
          FROM generate_series(0, 1000 * 1000 - 1) g;
```

```
db1=# VACUUM ANALYZE t412;

（PostgreSQL 17.5 の場合）
db1=# SET enable_seqscan TO off; -- 17.5 では通常 Seq Scan になるため
db1=# explain (analyze, buffers) SELECT * FROM t412
         WHERE id1 IN (100, 500, 800) AND id2 BETWEEN 100 AND 105;
                                                  QUERY PLAN
------------------------------------------------------------------------------
 Index Scan using t412_pkey on t412
  (cost=0.42..32418.92 rows=19 width=25)
  (actual time=3.890..102.620 rows=18 loops=1)
   Index Cond: ((id1 = ANY ('{100,500,800}'::integer[])) AND (id2 >= 100)
AND (id2 <= 105))
   Buffers: shared hit=4944
 Planning Time: 0.142 ms
 Execution Time: 102.649 ms
(5 rows)

（PostgreSQL 18beta1 の場合）
db1=# explain (analyze, buffers) SELECT * FROM t412
         WHERE id1 IN (100, 500, 800) AND id2 BETWEEN 100 AND 105;
                                                  QUERY PLAN
------------------------------------------------------------------------------
 Index Scan using t412_pkey on t412
  (cost=0.42..150.48 rows=18 width=25)
  (actual time=0.062..0.232 rows=18.00 loops=1)
   Index Cond: ((id1 = ANY ('{100,500,800}'::integer[])) AND (id2 >= 100)
AND (id2 <= 105))
   Index Searches: 21              ← 21 回インデックススキャンが発生
   Buffers: shared hit=78          ← 参照バッファ数が 4944 から大幅な減少
 Planning Time: 0.161 ms
 Execution Time: 0.261 ms        ← 実行時間が 102ms からの大幅な短縮
(6 rows)
```

3 つの列を含む主キーインデックスを持つテーブルで、2 番目と 3 番目の列で条件を指定して検索を行

なっています。このとき 1 番目の列は文字列型でサイズが大きいものの値の種類は 5 つだけです。このようにすると、PostgreSQL 18 でスキップスキャンが行なわれ、高速に実行できました。上記の出力例は両バージョンとも何回か繰り返してバッファヒットする状態での結果ですが、Execution Time が 102ms から 0.26ms と桁違いに短縮されています。

「Index Searches:《整数値》」はインデックスの使用回数を報告するもので、スキップスキャンの場合、数が多くなります。この数値の回数だけ、インデックス先頭列の様々な指定パターンについて、インデックス後方列の条件に基づくインデックス範囲検索を行なったことを意味しています。

スキップスキャンがない以前の PostgreSQL でも、以下のように記述するチューニングは可能です。

（PostgreSQL 17.5 での実行結果）

```
db1=# explain (analyze, buffers)
 SELECT * FROM t412 WHERE id0 = 'a87ff679a2' AND id1 IN (100, 500, 800)
   AND id2 BETWEEN 100 AND 105
 UNION ALL
  《中略 - 全ての id0 列の値について列記する（あらかじめ値が分かっている前提）》
 UNION ALL
 SELECT * FROM t412 WHERE id0 = 'eccbc87e4b' AND id1 IN (100, 500, 800)
   AND id2 BETWEEN 100 AND 105;
                                        QUERY PLAN
-------------------------------------------------------------------------------
 Append
  (cost=0.42..145.49 rows=20 width=25)
  (actual time=0.032..0.129 rows=18 loops=1)
   Buffers: shared hit=60
   ->  Index Scan using t412_pkey on t412
         (cost=0.42..29.08 rows=4 width=25)
         (actual time=0.031..0.047 rows=3 loops=1)
          Index Cond: ((id0 = 'a87ff679a2'::text) AND (id1 = ANY
('{100,500,800}'::integer[])) AND (id2 >= 100) AND (id2 <= 105))
          Buffers: shared hit=12
  （中略）
   ->  Index Scan using t412_pkey on t412 t412_4
         (cost=0.42..29.08 rows=4 width=25)
         (actual time=0.008..0.018 rows=3 loops=1)
          Index Cond: ((id0 = 'eccbc87e4b'::text) AND (id1 = ANY
('{100,500,800}'::integer[])) AND (id2 >= 100) AND (id2 <= 105))
```

```
         Buffers: shared hit=12
 Planning Time: 0.309 ms
 Execution Time: 0.163 ms
(19 rows)
```

スキップスキャンは、これと似たことを自動で行なってくれる機能と言えます。

4.1.3. プランナ改善

PostgreSQL 18 ではプランナに多数の改良が加わっています。本節では主なものを取り上げます。

なお、本節で取り上げているもの以外にも、VALUE 句を配列演算に変換する最適化や、多数のパーティションがあるときのウィンドウ集約の最適化など、いくつかのプランナ改善が行なわれています。

◆ *自己結合の除去*

以下のように、処理不要の自己結合をプランナの段階で除去できるようになりました。

（テスト用テーブル作成）

```
db1=# CREATE TABLE t413a (x int PRIMARY KEY);
db1=# INSERT INTO t413a SELECT generate_series(1, 1000) g;
db1=# VACUUM ANALYZE t413a;
```

（17.5 の動作）

```
db1=# explain SELECT * FROM t413a m, t413a n WHERE m.x = n.x;
                                QUERY PLAN
-------------------------------------------------------------------------
 Hash Join  (cost=27.50..45.14 rows=1000 width=8)
   Hash Cond: (m.x = n.x)
   ->  Seq Scan on t413a m  (cost=0.00..15.00 rows=1000 width=4)
   ->  Hash  (cost=15.00..15.00 rows=1000 width=4)
         ->  Seq Scan on t413a n  (cost=0.00..15.00 rows=1000 width=4)
(5 rows)
```

（18beta2 の動作 - 通常の結合のほか、WHERE x IN (..) でも不要な処理が除去される）

```
db1=# explain SELECT * FROM t413a m, t413a n WHERE m.x = n.x;
                          QUERY PLAN
```

```
------------------------------------------------------------
 Seq Scan on t413a n  (cost=0.00..15.00 rows=1000 width=8)
(1 row)

db1=# explain SELECT * FROM t413a WHERE x IN (SELECT s.x FROM t413a s);
                           QUERY PLAN
------------------------------------------------------------
 Seq Scan on t413a s  (cost=0.00..15.00 rows=1000 width=4)
(1 row)
```

◆ *OR 句を配列に転換*

以下のように、同じ列に対する OR 条件が続いている場合に、「x = ANY (《配列》)」という条件に書き換えできるようになりました。

```
（17.5 の動作）
db1=# explain SELECT * FROM t413a
        WHERE x = 1 OR x = 2 OR x = 3 OR x = 5 OR x = 8;
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on t413a  (cost=21.42..26.80 rows=5 width=4)
   Recheck Cond: ((x = 1) OR (x = 2) OR (x = 3) OR (x = 5) OR (x = 8))
   ->  BitmapOr  (cost=21.42..21.42 rows=5 width=0)
       ->  Bitmap Index Scan on t413a_pkey  (cost=0.00..4.28 rows=1 width=0)
               Index Cond: (x = 1)
       ->  Bitmap Index Scan on t413a_pkey  (cost=0.00..4.28 rows=1 width=0)
               Index Cond: (x = 2)
       ->  Bitmap Index Scan on t413a_pkey  (cost=0.00..4.28 rows=1 width=0)
               Index Cond: (x = 3)
       ->  Bitmap Index Scan on t413a_pkey  (cost=0.00..4.28 rows=1 width=0)
               Index Cond: (x = 5)
       ->  Bitmap Index Scan on t413a_pkey  (cost=0.00..4.28 rows=1 width=0)
               Index Cond: (x = 8)
(13 rows)
```

（18beta2 の動作）

```
db1=# explain SELECT * FROM t413a
        WHERE x = 1 OR x = 2 OR x = 3 OR x = 5 OR x = 8;
                                  QUERY PLAN
------------------------------------------------------------------------------
 Index Only Scan using t413a_pkey on t413a  (cost=0.28..8.63 rows=5 width=4)
   Index Cond: (x = ANY ('{1,2,3,5,8}'::integer[]))
(2 rows)
```

式ごとに Bitmap Index Scan を行う従来の実行プランと比べて、1 回の Index Scan または Index Only Scan で済ませられるので、大幅な性能向上が期待できます。

◆ *EXCEPT、INTERSECT の改善*

集合演算を行う EXCEPT、INTERSECT の実行プラン作成が改善されました。以下に例を示します。

（テスト用テーブルを追加作成）

```
db1=# CREATE TABLE t413b (x int PRIMARY KEY);
db1=# INSERT INTO t413b SELECT generate_series(101, 900) g;
db1=# VACUUM ANALYZE t413b;
```

（17.5 の動作）

```
db1=# explain (SELECT x FROM t413a ORDER BY x)
               EXCEPT (SELECT x FROM t413b ORDER BY x);
                                      QUERY PLAN
------------------------------------------------------------------------------
 HashSetOp Except  (cost=0.28..99.05 rows=1000 width=8)
   ->  Append  (cost=0.28..94.55 rows=1800 width=8)
       -> Subquery Scan on "*SELECT* 1"  (cost=0.28..45.27 rows=1000 width=8)
               ->  Index Only Scan using t413a_pkey on t413a
                (cost=0.28..35.27 rows=1000 width=4)
       -> Subquery Scan on "*SELECT* 2"  (cost=0.28..40.27 rows=800 width=8)
               ->  Index Only Scan using t413b_pkey on t413b
                (cost=0.28..32.27 rows=800 width=4)
(6 rows)
```

（18beta2 の動作）

```
db1=# explain (SELECT x FROM t413a ORDER BY x)
               EXCEPT (SELECT x FROM t413b ORDER BY x);
                                      QUERY PLAN
------------------------------------------------------------------------------
 SetOp Except  (cost=0.55..74.55 rows=1000 width=4)
   ->  Index Only Scan using t413a_pkey on t413a
       (cost=0.28..35.27 rows=1000 width=4)
   ->  Index Only Scan using t413b_pkey on t413b
       (cost=0.28..32.27 rows=800 width=4)
(3 rows)
```

本例では、1 列だけの 2 つのテーブルをソートして EXCEPT 集合処理を行なっています。テーブル t413a の行のうち、テーブル t413b にもある行を除いた結果を出力するという問い合わせです。

バージョン 17 の実行プランでは、Subquery Scan プラン要素が挟まっていますが、バージョン 18 では省略されています。これによりエグゼキュータ要素間でデータを引き渡す処理が節約されます。

さらに、バージョン 17 の実行プランで HashSetOp が選択されているところが、バージョン 18 では SetOp となっています。HashSetOp はメモリ上にハッシュデータ構造を作って処理する方式ですが、これは下位プラン要素から渡されてくるデータがインデックス順にスキャンされることでソート済であることを活かしていません。SetOp は下位プラン要素がソート済みであることが前提の集合処理方式ですので、本ケースではこちらを選択する方が理にかなっています。バージョン 18 では SetOp か HashSetOp かを適切に判断するようになりました。

4.1.4. GIN インデックスの並列作成

PostgreSQL 18 から、GIN インデックスをパラレル作成できるようになりました。PostgreSQL 17.x までは Btree と BRIN のインデックスのみがパラレル作成に対応していました。

以下の通り動作確認を行ないました。「/tmp」以下に PostgreSQL 18beta1 のソースコードが postgres ユーザにアクセス可能な権限設定で配置されているものとします。

（サンプルテーブル・データの作成）

```
db1=# CREATE TABLE t414 (id serial PRIMARY KEY, file text, body text);
db1=# INSERT INTO t414 (file, body)
        SELECT ls file, regexp_split_to_table(pg_read_file(
          '/tmp/postgresql-18beta1/doc/src/sgml/' || ls), '<para>') body
```

```
          FROM pg_ls_dir('/tmp/postgresql-18beta1/doc/src/sgml') ls
          WHERE ls ~ '\.sgml$' ORDER BY ls;
INSERT 0 18583

db1=# INSERT INTO t414 (file, body)
        SELECT ls file, regexp_split_to_table(pg_read_file(
          '/tmp/postgresql-18beta1/doc/src/sgml/ref/' || ls), '<para>') body
          FROM pg_ls_dir('/tmp/postgresql-18beta1/doc/src/sgml/ref') ls
          WHERE ls ~ '\.sgml$' ORDER BY ls;
INSERT 0 6165

db1=# VACUUM ANALYZE t414;
VACUUM
```

テキスト全文検索の GIN インデックスを作るため、PostgreSQL ドキュメントの SGML ファイルの内容をパラグラフ（<para>タグ）ごと 1 行として、テーブルに格納しました。この後、インデックス作成の所要時間を比較するときにバッファに載っている状態で揃えるため、VACUUM ANALYZE も実行しておきます。

続いて GIN インデックスを作ります。パラレル作成が働いているかを確認するため、DEBUG1 レベルのメッセージが出力されるようにしています。

（インデックス作成）

```
db1=# SET client_min_messages TO debug1;
db1=# \timing on
db1=# CREATE INDEX ON t414 USING gin (to_tsvector('english', body));
DEBUG:  building index "t414_to_tsvector_idx" on table "t414" with request
for 1 parallel workers     → ワーカが 1 つ加わり、2 並列で作成が行なわれている
CREATE INDEX
Time: 514.446 ms
```

（パラレルを無効にしてインデックス作成）

```
db1=# SET max_parallel_maintenance_workers TO 0;    → この設定でパラレル無効化
db1=# DROP INDEX t414_to_tsvector_idx;
db1=# CREATE INDEX ON t414 USING gin (to_tsvector('english', body));
DEBUG:  building index "t414_to_tsvector_idx" on table "t414" serially
CREATE INDEX
Time: 1079.166 ms (00:01.079)
```

パラレル作成は直列作成（serially）と比べて、約半分の時間でインデックス作成が完了しました。GIN インデックスは作成に時間がかかることの多いインデックスですので、GIN インデックスの並列作成対応はメリットの大きい機能追加と言えます。

なお、本節テーマの主題ではありませんが、本インデックスの使用例も示しておきます。

```
db1=# SELECT id, file FROM t414 WHERE to_tsvector('english', body)
          @@ to_tsquery('english', 'parallel & index & create');
  id   |       file
-------+------------------
  2239 | config.sgml
  8670 | indexam.sgml
  8785 | indices.sgml
 15262 | release-18.sgml
 20152 | create_index.sgml
  《後略》
```

4.1.5. 大量のテーブルアクセス時のロック処理の改善

PostgreSQL18 では多数のテーブルにアクセスする問い合わせのロック処理について性能改善が行なわれています。

PostgreSQL ではテーブル等を参照するだけで最も弱い ACCESS SHARE テーブルロックが発生します。このような弱いロック（ACCESS SHARE、ROW SHARE、ROW EXCLUSIVE の各モード）を簡易的に高速に行う仕組みがこれまでも用意されていましたが、1 つのセッションにつき固定的に 16 個の要素（テーブルやインデックス）しか扱えませんでした。そのため、多数のテーブルにアクセスする問い合わせは、ロック処理が負担になっていました。典型的には大きなパーティションテーブルへの問い合わせが該当します。

PostgreSQL 18 から、この上限数が固定的なものではなくなり、設定 max_locks_per_transaction で与えられるようになりました。以下の手順で効果を確認しました。

```
（pgbench でスケール 100、パーティション 100 のテスト用テーブルを作成）
$ pgbench -i --partitions=100 --scale=100 --partition-method=hash db1
    《出力省略》

（作成されたパーティションテーブル pgbench_accounts の定義を書き換え）
$ psql db1 <<EOS
ALTER TABLE pgbench_accounts ADD aid2 int;
```

```
UPDATE pgbench_accounts SET aid2 = aid;
ALTER TABLE pgbench_accounts RENAME aid TO aid1;
ALTER TABLE pgbench_accounts RENAME aid2 TO aid;
CREATE INDEX ON pgbench_accounts (aid);
VACUUM FULL pgbench_accounts;
EOS
```

→ この書き換えにより、 aid 列が非パーティションキーになり、「WHERE aid = 1」といった検索条件で全パーティションを調べる動作になる

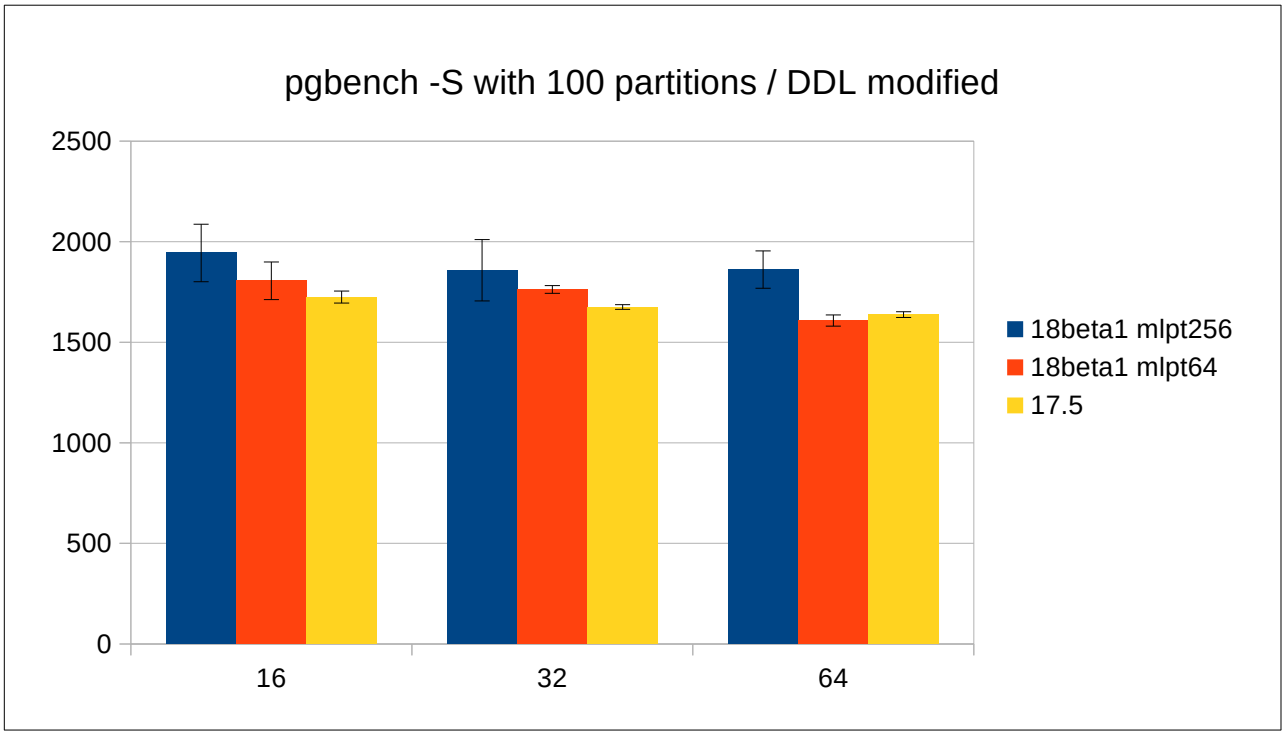
（参照シナリオの pgbench で性能計測）

```
$ pgbench --select-only -c 32 -T 60 db1
pgbench (18beta1)
starting vacuum...end.
transaction type: <builtin: select only>
```

《出力中略》

```
latency average = 18.347 ms
initial connection time = 48.586 ms
tps = 1744.142981 (without initial connection time)
```

この pgbench を PostgreSQL 18 で max_locks_per_transaction をデフォルトの 64 の場合、256 に増やした場合、PostgreSQL 17.5 の場合、で実行した結果が以下です。標準偏差を誤差線として付加しています。



おおむね、17.5 よりも 18beta1 が勝り、18beta1 の中でも max_locks_per_transaction 設定値を 256 に増やした場合が勝る、という結果が確認できました。

4.1.6. 組み込み照合順序 *pg_unicode_fast*

新たな組み込みの UTF8 専用の照合順序 pg_unicode_fast（ロケール名 PG_UNICODE_FAST）が追加されました。これは、ソート順については単純なコードポイント順で動作しますが、大文字小文字については ASCII 範囲の英字以外も認識して動作します。大文字小文字の対応を認識していることは、lower 関数、upper 関数、initcap 関数、および、文字列パターンマッチの関数や演算子の動作に反映されます。

以下の通り動作確認をしました。

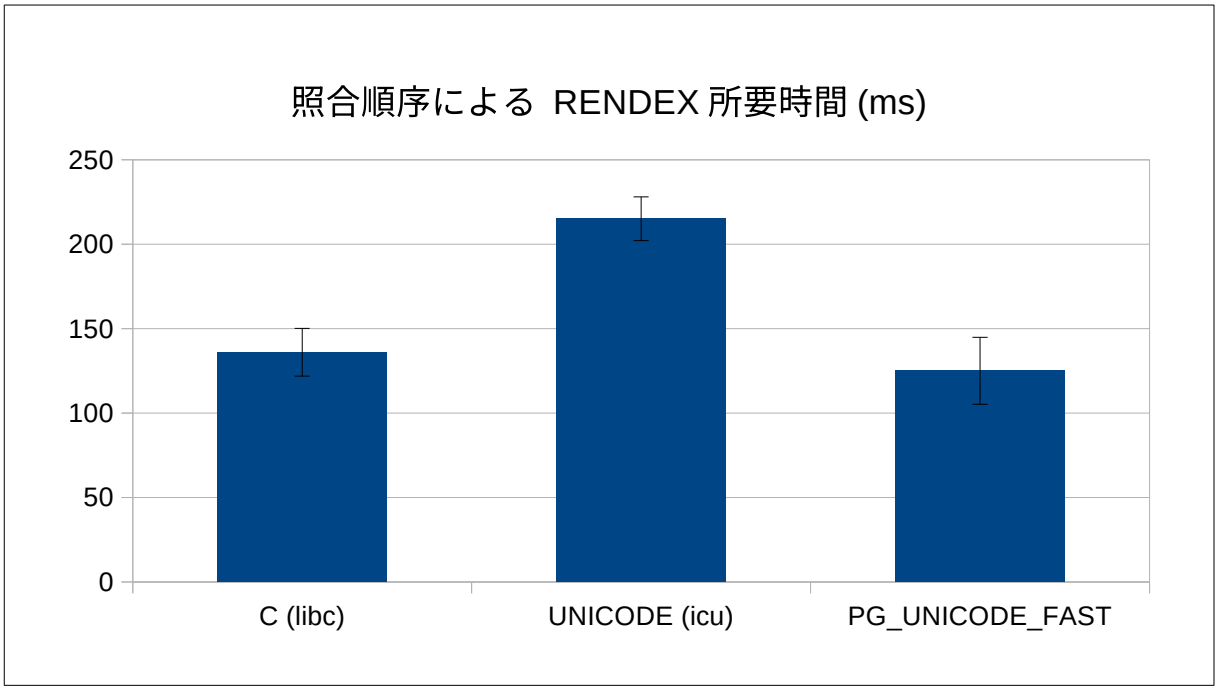
```
（PG_UNICODE_FAST ロケールのデータベースを作成）
$ createdb -T template0 --builtin-locale=PG_UNICODE_FAST \
    --locale-provider=builtin --encoding UTF8 fastunicode

（比較用に und ロケールの icu 照合順序のデータベースも作成）
$ createdb -T template0 --icu-locale=und --locale-provider=icu \
    --encoding UTF8 icuunicode

$ psql -d icuunicode \
       -c "SELECT * FROM (VALUES ('ア'), ('さ'), ('か')) v(a) ORDER BY a;"
 a
----
 ア
 か
 さ                ← icu の unicode 照合順序（und ロケール）なら辞書順に対応
(3 rows)

$ psql -d fastunicode \
       -c "SELECT * FROM (VALUES ('ア'), ('さ'), ('か')) v(a) ORDER BY a;"
 a
----
 か
 さ
 ア                ← PG_UNICODE_FAST ではソートはコードポイント順になる
```

```
(3 rows)

$ psql -d fastunicode -c "SELECT lower('Σ');"   ← ASCII 外の大文字小文字を認識
 lower
-------
 σ
(1 row)

$ psql -d db1 -c "SELECT lower('Σ');"   ← db1 は C 照合順序なので変換されない
 lower
-------
 Σ
(1 row)
```

続いて、PG_UNICODE_FAST を使うことでどのくらいオーバーヘッドを減らせるかを、以下のように 文字列データ型の値を持つテーブルの REINDEX 所要時間で確認しました。

```
$ psql -d fastunicode
fastunicode=# CREATE TABLE t416 (code text PRIMARY KEY, name text UNIQUE);
fastunicode=# INSERT INTO t416 SELECT md5(g::text),
                'name' || g FROM generate_series(1, 100000) g;
fastunicode=# \timing on
fastunicode=# REINDEX TABLE t416;
REINDEX
Time: 126.891 ms
```

結果は下記グラフのようになりました。誤差線は標準偏差を上下に延ばしています。
PG_UNICODE_FAST は C ロケールと変わらない所要時間で済んでいることが分かりました。



4.2. メジャーバージョンアップ

PostgreSQL18 では、PostgreSQL のメジャーバージョンアップを行うコマンド pg_upgrade にいくつか機能追加があります。
なお、チェックサムが有効なインスタンスと無効なインスタンスの間で pg_upgrade を使用してのアップグレードを行うことはできません。PostgreSQL18 ではチェックサムがデフォルトで有効になっているので、注意して pg_upgrade をお使いください。

4.2.1. プランナ統計情報の移行

pg_upgrade でメジャーバージョンアップする際にプランナ統計情報を移行できるようになりました。これにより、アップグレード後すぐに効率的なクエリプランが利用可能になり、運用効率が向上します。
以前までは、統計情報は移行されず、pg_upgrade 後に ANALYZE を実行して統計情報を生成する必要がありました。統計情報を移行しない場合は、--no-statistics オプションを指定します。

ただし、CREATE STATISTICS で作成された拡張統計情報や拡張機能によって追加された統計情報など、一部の統計情報は移行されない場合があります。

移行されなかった統計情報を生成するために、vacuumdb の新しいオプション --missing-stats-only オプションを使用して、プランナ統計情報がないテーブルに対して最小限のプランナ統計情報を迅速に生成してください。このオプションは、--analyze-only または --analyze-in-stages と組み合わせてのみ使用できます。

以下、統計情報が移行されること、CREATE STATISTICS で作成した拡張統計情報が移行されないことを

検証した実行例です。

```
（PG17 でサンプルテーブル、データ作成 ）
db1=# CREATE TABLE t_stats1 (a int, b int);
db1=# CREATE INDEX ON t_stats1(b);

（データは 100 万件、b = 1 は 1000 件 ）
db1=# INSERT INTO t_stats1
           SELECT i, CASE WHEN i % 1000 = 0 THEN 1 ELSE 0 END
           FROM generate_series(1, 1000000) i;

（ANALYZE 前は Bitmap Heap Scan が選択される）
db1=# EXPLAIN SELECT * FROM t_stats1 WHERE b = 1;
                                QUERY PLAN
-------------------------------------------------------------------------
 Bitmap Heap Scan on t_stats1  (cost=55.17..4753.05 rows=5000 width=8)
   Recheck Cond: (b = 1)
   ->  Bitmap Index Scan on t_stats1_b_idx  (cost=0.00..53.92 rows=5000 width=0)
         Index Cond: (b = 1)
(4 rows)

db1=# ANALYZE t_stats1;

（ANALYZE 後は Index Scan が選択される）
db1=# EXPLAIN SELECT * FROM t_stats1 WHERE b = 1;
                                QUERY PLAN
-------------------------------------------------------------------------
 Index Scan using t_stats1_b_idx on t_stats1 (cost=0.42..30.38 rows=700 width=8)
   Index Cond: (b = 1)
(2 rows)

（CREATE STATISTICS で明示的に統計情報を作成する）
db1=# CREATE TABLE t_stats2 (c int, d int);
db1=# INSERT INTO t_stats2 SELECT i / 100, i / 500
           FROM generate_series(1,1000000) i;
```

（プランナは 2 つの WHERE 条件を独立なものとみなし、実際より小さな行数見積もる）

```
db1=# EXPLAIN ANALYZE SELECT * FROM t_stats2 WHERE (c = 1) AND (d = 0);
                                QUERY PLAN
-------------------------------------------------------------------------
 Gather  (cost=1000.00..11677.81 rows=25 width=8)
         (actual time=1.096..30.656 rows=100 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   ->  Parallel Seq Scan on t_stats2  (cost=0.00..10675.31 rows=10 width=8)
                                (actual time=7.515..16.107 rows=33 loops=3)
         Filter: ((c = 1) AND (d = 0))
         Rows Removed by Filter: 333300
 Planning Time: 0.147 ms
 Execution Time: 30.824 ms
(8 rows)
```

（関数的依存統計を作成し、プランナが WHERE 条件が冗長であることを認識するようにする）

```
db1=# CREATE STATISTICS s1 (dependencies) ON c, d FROM t_stats2;
db1=# ANALYZE t_stats2;
```

（行数の見積もりが rows=100 になる）

```
db1=# EXPLAIN ANALYZE SELECT * FROM t_stats2 WHERE (c = 1) AND (d = 0);
                                QUERY PLAN
-------------------------------------------------------------------------
 Gather  (cost=1000.00..11685.00 rows=100 width=8)
         (actual time=0.275..59.176 rows=100 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   ->  Parallel Seq Scan on t_stats2  (cost=0.00..10675.00 rows=42 width=8)
                               (actual time=28.964..46.456 rows=33 loops=3)
         Filter: ((c = 1) AND (d = 0))
         Rows Removed by Filter: 333300
 Planning Time: 0.249 ms
 Execution Time: 59.203 ms
(8 rows)
```

（PG18 の pg_upgrade を使用しアップグレードする）

```
$ /usr/local/pgsql/18/bin/pg_upgrade \
  -d /var/lib/pgsql/17/data \
  -D /var/lib/pgsql/18/data \
  -b /usr/pgsql-17/bin \
  -B /usr/local/pgsql/18/bin
```

（PG18 の db1 に接続する）

```
$ /usr/local/pgsql/18/bin/pg_ctl start -D /var/lib/pgsql/18/data
$ /usr/local/pgsql/18/bin/psql -p 5433 db1
psql (18beta1)
db1=# \d
          List of relations
 Schema |   Name    | Type  |  Owner
--------+-----------+-------+----------
 public | t_stats1 | table | postgres
 public | t_stats2 | table | postgres
(2 rows)
```

（統計情報が移行され、ANALYZE しなくても Index Scan が選択される）

```
db1=# EXPLAIN SELECT * FROM t_stats1 WHERE b = 1;
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using t_stats1_b_idx on t_stats1 (cost=0.42..30.38 rows=700 width=8)
   Index Cond: (b = 1)
(2 rows)
```

（CREATE STATISTICS で作成した統計情報は移行されないため、見積行数が実際より小さい）

```
db1=# EXPLAIN ANALYZE SELECT * FROM t_stats2 WHERE (c = 1) AND (d = 0);
                                QUERY PLAN
-------------------------------------------------------------------------------
 Gather  (cost=1000.00..11677.81 rows=25 width=8)
         (actual time=5.327..101.616 rows=100.00 loops=1)
   Workers Planned: 2
```

```
   Workers Launched: 2
   Buffers: shared read=4425
   ->  Parallel Seq Scan on t_stats2  (cost=0.00..10675.31 rows=10 width=8)
                              (actual time=61.949..93.950 rows=33.33 loops=3)
         Filter: ((c = 1) AND (d = 0))
         Rows Removed by Filter: 333300
         Buffers: shared read=4425
 Planning:
   Buffers: shared hit=9 read=4 dirtied=1
 Planning Time: 0.295 ms
 Execution Time: 101.636 ms
(12 rows)
```

以下、上記で移行されなかった CREATE STATISTICS で作成した統計情報を vacuumdb の新しいオプション --missing-stats-only を使用して生成します。

```
$ /usr/local/pgsql/18/bin/vacuumdb --all \
    --analyze-in-stages --missing-stats-only -p 5433
$ /usr/local/pgsql/18/bin/psql -p 5433 db1

（統計情報が生成され、行数の見積もりが rows=100 になる）
db1=# EXPLAIN ANALYZE SELECT * FROM t_stats2 WHERE (c = 1) AND (d = 0);
                                QUERY PLAN
-------------------------------------------------------------------------------
 Gather  (cost=1000.00..11685.00 rows=100 width=8)
         (actual time=1.975..41.391 rows=100.00 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   Buffers: shared hit=1102 read=3323
   ->  Parallel Seq Scan on t_stats2  (cost=0.00..10675.00 rows=42 width=8)
                              (actual time=20.966..34.062 rows=33.33 loops=3)
         Filter: ((c = 1) AND (d = 0))
         Rows Removed by Filter: 333300
         Buffers: shared hit=1102 read=3323
 Planning:
   Buffers: shared hit=40
```

```
 Planning Time: 0.631 ms
 Execution Time: 41.461 ms
(12 rows)
```

以下、統計情報を移行しない場合、--no-statistics オプションを指定した場合の実行例です。

```
（PG18 クラスタを初期化し、再度 --no-statistics を指定して pg_upgrade を実行）
$ /usr/local/pgsql/18/bin/pg_upgrade \
    -d /var/lib/pgsql/17/data \
    -D /var/lib/pgsql/18/data \
    -b /usr/pgsql-17/bin \
    -B /usr/local/pgsql/18/bin \
    --no-statistics

（PG18 の db1 に接続）
$ /usr/local/pgsql/18/bin/pg_ctl start -D /var/lib/pgsql/18/data
$ /usr/local/pgsql/18/bin/psql -p 5433 db1

（統計情報が移行されないため、Bitmap Heap Scan が選択される）
db1=# EXPLAIN SELECT * FROM t_stats1 WHERE b = 1;
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on t_stats1  (cost=55.17..4753.05 rows=5000 width=8)
   Recheck Cond: (b = 1)
   ->  Bitmap Index Scan on t_stats1_b_idx  (cost=0.00..53.92 rows=5000
width=0)
         Index Cond: (b = 1)
(4 rows)
```

4.2.2. チェック処理の並列化オプションの追加

pg_upgrade の処理の中には、クラスタ内のすべてのデータベースに接続し、各データベースで同じクエリを実行するという繰り返しのチェック処理が多数存在します。これらの処理は順次実行されるため、データベースの数が多い場合はこれらの処理に時間がかかります。

今回のリリースでは、--jobs オプションで並列実行数を指定することで、複数のデータベースを同時に処

理できるようになりました。設定の目安としては、マシンの CPU コア数を使うのが適しています。

PostgreSQL 17.x にも--jobs オプションはありますが、ファイルのコピーやリンク、一部のスキーマのダンプ／リストア処理などで使用されていましたが、各データベースごとに実行するタスクでは使われていませんでした。

以下、--jobs に 4 を指定した場合と、--jobs を使用しない場合の性能比較をおこないました。検証では、データベース 100 個、テーブルスペース 100 個存在する PostgreSQL 17.5 のクラスタを作成し、pg_upgrade を使用して PostgreSQL 18 に バージョンアップを行います。

（テーブルスペースディレクトリ作成スクリプト）

```
$ cat > /var/lib/pgsql/create_ts_dir.sh <<'EOS'
#!/bin/bash
TBLSPC_BASE=/var/lib/pgsql/pg_tblspc
mkdir -p $TBLSPC_BASE

for i in $(seq 1 100); do
  mkdir -p $TBLSPC_BASE/ts$i
  chmod 700 $TBLSPC_BASE/ts$i
done
EOS

$ chmod +x create_ts_dir.sh.sh
$ ./create_ts_dir.sh.sh
```

（テーブルスペース、データベース作成スクリプト）

```
$ cat > /var/lib/pgsql/create_db_ts.sh <<'EOS'
#!/bin/bash
PORT=5433
PG_BIN=/usr/pgsql-17/bin
TBLSPC_BASE=/var/lib/pgsql/pg_tblspc

for i in $(seq 1 100); do
  $PG_BIN/psql -p $PORT -d postgres -c \
    "CREATE TABLESPACE ts$i LOCATION '$TBLSPC_BASE/ts$i';"
done

for i in $(seq 1 100); do
```

```
  $PG_BIN/createdb -p $PORT --tablespace=ts$i db$i
done
EOS

$ chmod +x  create_db_ts.sh
$ ./create_db_ts.sh
```

（PG17 の db1 に接続）

```
$ /usr/pgsql-17/bin/psql -db1 -p 5433
```

（データベース数、テーブルスペース数確認）

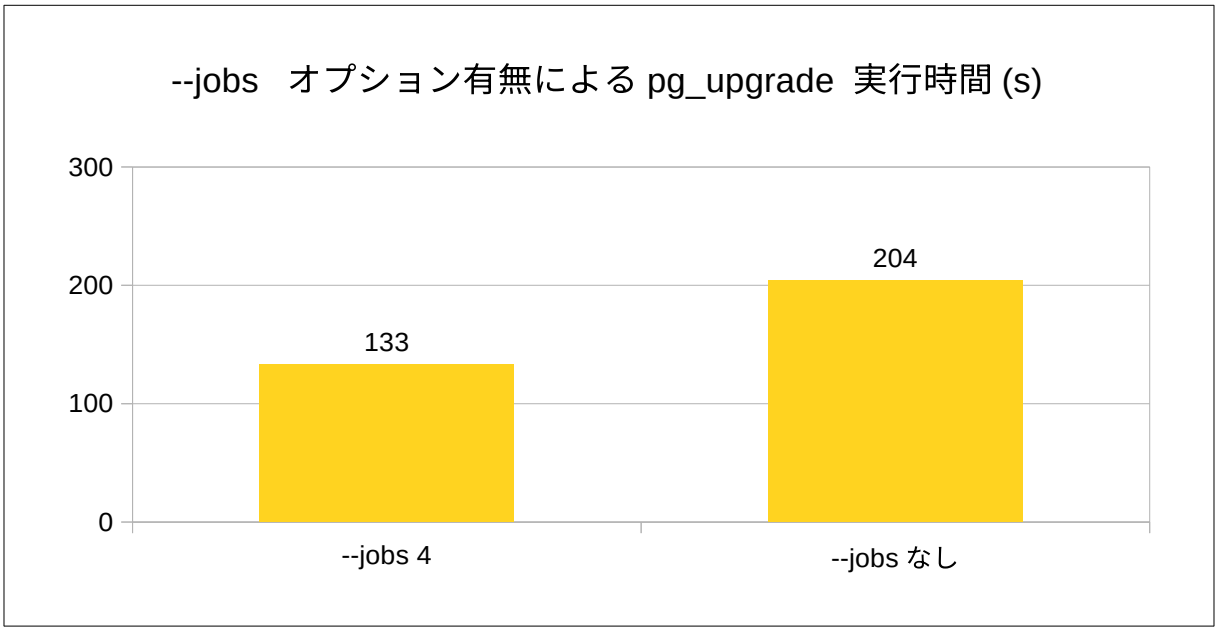
```
db1=# SELECT COUNT(*) FROM pg_tablespace;
 count
-------
 102
(1 row)

db1=# SELECT COUNT(*) FROM pg_database;
 count
-------
 103
(1 row)
```

（pg_upgrade --jobs の処理時間を取得するスクリプト）

```
$ cat > /var/lib/pgsql/PG18_parallel.sh <<'EOS'
#!/bin/bash
START=$(date +%s)

/usr/local/pgsql/18/bin/pg_upgrade \
  -d /var/lib/pgsql/17/data \
  -D /var/lib/pgsql/18/data \
  -b /usr/pgsql-17/bin \
  -B /usr/local/pgsql/18/bin \
  --jobs 4
```

```
END=$(date +%s)
ELAPSED=$((END - START))
echo "Elapsed time: ${ELAPSED} seconds ($((ELAPSED/60))m $((ELAPSED%60))s)"
EOS

$ chmod +x PG18_parallel.sh

（PG17 停止し、pg_upgrade -jobs 8 スクリプト実行）
$ /usr/pgsql-17/bin/pg_ctl stop -D /var/lib/pgsql/17/data
$ ./PG18_parallel.sh
Elapsed time: 133 seconds (2m 13s)
```

以上の手順で--jobs オプションを使用しない場合でも pg_upgrade を実行しました。以下、実行時間(s)の比較グラフです。環境にもよりますが、--jobs オプションを付けた方が速く終了していることが分かります。



4.2.3. --swap オプションの追加

pg_upgrade に新たに --swap オプションが導入されました。旧クラスタのデータディレクトリを新クラスタに移動し、カタログファイルを新しいクラスタ用に生成されたファイルに置き換える仕組みです。ディレクトリ全体を入れ替えるので、大量に DB オブジェクトがある環境では従来の--copy や--link、--clone オプ

ションを使うよりも高速です。

ただし、このモードでは古いクラスタに多くのガベージファイルが作成されるため、--sync-method=syncfs が使用されている場合はファイル同期ステップが長くなる可能性があります。swap モードでは –sync-method=fsync（デフォルト）を使用することが推奨されます。これは --sync-method=fsync とすることで、pg_upgrade は initdb コマンドを--sync-only と--no-sync-data-files オプションを付けて実行するためです。

また、他にも、このモードを使用すると古いクラスタが破壊的に変更されている可能性があり旧クラスタへの切り戻しが困難になるほか、古いクラスタと新しいクラスタが同じファイルシステムに存在する必要がある、PostgreSQL 10 以降のバージョンのみ対応、といった注意点があります。

以下で従来のモード copy、link、clone と新しいモード swap の性能比較をおこないました。検証では、PostgreSQL 17.5 から PostgreSQL 18 に upgrade を使用してバージョンアップを行います。

（PG17 でテーブル 30000 個 を作成、各テーブル 1000 行）

```
$ cat > /var/lib/pgsql/create_tbl_30000.sh <<'EOS'
#!/bin/bash
for i in $(seq 1 30000); do
  psql -d db1 -p 5433 -q -c "
    CREATE TABLE table_$i (id SERIAL PRIMARY KEY, val INT);
    INSERT INTO table_$i (val)
      SELECT g % 100 FROM generate_series(1,1000) AS g;"
done
EOS

$ chmod +x create_tbl_30000.sh

$ ./create_tbl_30000.sh
```

（PG17 の db1 に接続）

```
$ /usr/pgsql-17/bin/psql -db1 -p 5433
```

（データベースサイズ、テーブル数確認）

```
db1=# SELECT pg_size_pretty(pg_database_size('db1'));
 pg_size_pretty
----------------
 3550 MB
(1 row)
```

```
db1=# SELECT COUNT(*) AS total_tables
FROM information_schema.tables
WHERE table_schema NOT IN ('information_schema','pg_catalog');
 total_tables
--------------
  30000
(1 row)
```

（pg_upgrade の処理時間を取得するスクリプト）

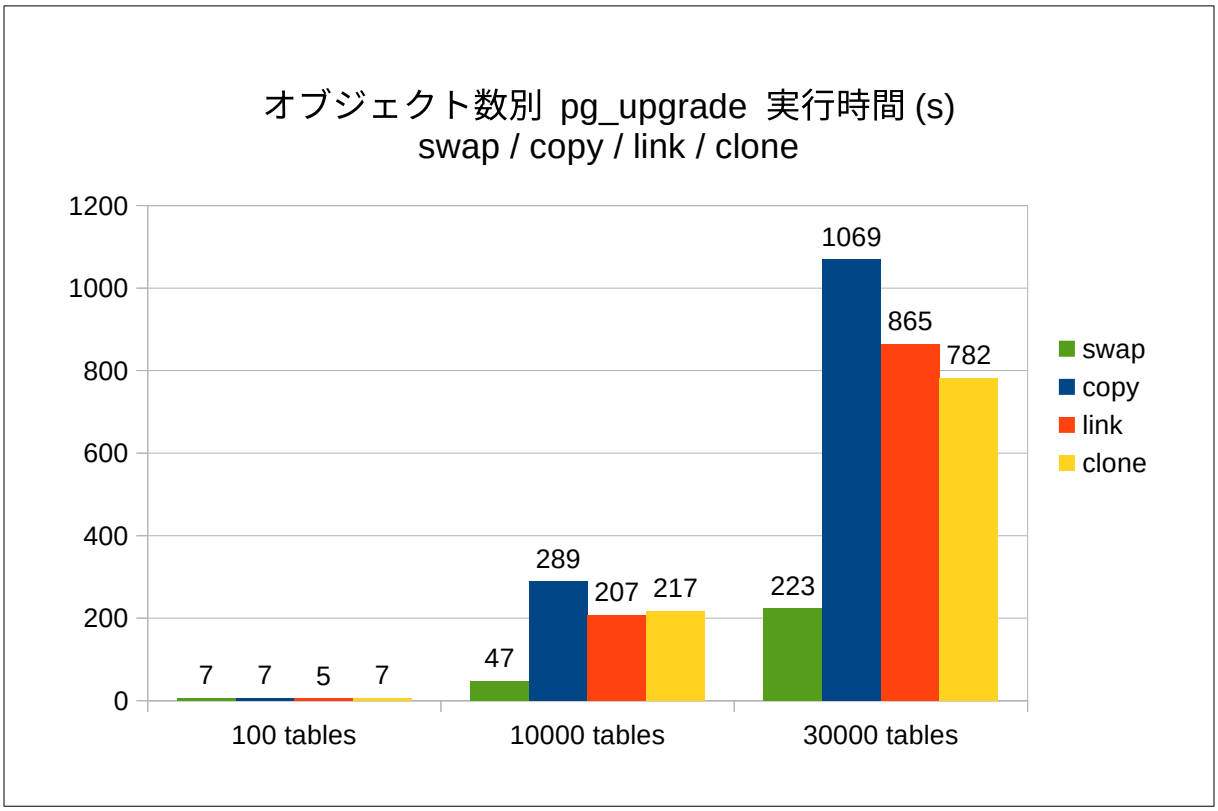
```
$ cat > /var/lib/pgsql/swap_30000.sh <<'EOS'
#!/bin/bash
START=$(date +%s)

/usr/local/pgsql/18/bin/pg_upgrade \
  -d /var/lib/pgsql/17/data \
  -D /var/lib/pgsql/18/data \
  -b /usr/pgsql-17/bin \
  -B /usr/local/pgsql/18/bin \
  --swap

END=$(date +%s)
ELAPSED=$((END - START))
echo "Elapsed time: ${ELAPSED} seconds ($((ELAPSED/60))m $((ELAPSED%60))s)"
EOS

$ chmod +x swap_30000.sh
```

（PG17 停止し、pg_upgrade --swap スクリプト実行）

```
$ /usr/pgsql-17/bin/pg_ctl stop -D /var/lib/pgsql/17/data
$ ./swap_30000.sh
Elapsed time: 223 seconds (3m 43s)
```

以上の手順でテーブル数 100、10,000、30,000 個の環境で swap、copy、link、clone モードを使用し

pg_upgrade を実行しました。以下、実行時間(s)の比較グラフです。



テーブル数が 100 個の時点では各モード間で実行時間に差はありませんが、テーブル数が増加するにつれて、swap と copy／link／clone モード間の差が著しく大きくなっていることが分かります。

4.3. SQL 機能

PostgreSQL 18 では、いくつか新たな SQL の構文や機能が加わっています。

4.3.1. 仮想生成列の追加

列の値をクエリ実行時にその場で計算する仮想生成列が導入されました。これは、従来の生成列とは異なり、計算結果をディスクに保存せず、必要なときにだけ値を算出する仕組みで、ディスクスペースの節約や更新時の負荷を削減することができます。仮想生成列ではセキュリティ上のリスクがあるため、ユーザーが定義した関数や型を利用することはできません。

仮想生成列は、PostgreSQL 18 以降、生成列のデフォルトの動作として採用されています。CREATE TABLE 文の GENERATED ALWAYS に VIRTUAL を指定するか、キーワードを省略すると、仮想生

成列が作成されます。従来の生成列を作成するには、GENERATED ALWAYS に STORED を指定してください。

（仮想生成列を含むテーブルを作成）

```
db1=# CREATE TABLE t_virt(id int PRIMARY KEY, v1 int,
        v2 int GENERATED ALWAYS AS (v1 + 1) VIRTUAL,
        v3 int GENERATED ALWAYS AS (v1 - 1) STORED);
db1=# \d t_virt
                            Table "public.t_virt"
 Column |  Type   | Collation | Nullable |               Default
--------+---------+-----------+----------+-------------------------------------
 id     | integer |           | not null |
 v1     | integer |           |          |
 v2     | integer |           |          | generated always as (v1 + 1)
 v3     | integer |           |          | generated always as (v1 - 1) stored
Indexes:
    "t_virt_pkey" PRIMARY KEY, btree (id)

db1=# INSERT INTO t_virt (id, v1) VALUES (1, 10), (2, 20), (3, 30);
db1=# SELECT * FROM t_virt ;
 id | v1 | v2 | v3
----+----+----+----
  1 | 10 | 11 |  9
  2 | 20 | 21 | 19
  3 | 30 | 31 | 29
(3 rows)
```

オーバーフロー発生時の挙動に関しても、生成列とは異なります。仮想生成列で算出された値が列定義の上限を超える場合、INSERT 文や UPDATE 文では検証されず、検索時にエラーが発生します。

（生成列、仮想生成列を含むテーブルを作成）

```
db1=# CREATE TABLE t_virt_of (id int PRIMARY KEY,
  v1 varchar(3), v2 varchar(3),
  v3 varchar(3) GENERATED ALWAYS AS ('A' || v1) STORED,
  v4 varchar(3) GENERATED ALWAYS AS ('A' || v2) VIRTUAL);
```

```
（生成列の場合、INSERT した時にエラー）
db1=# INSERT INTO t_virt_of (id, v1, v2) VALUES (1, 'BBB','CC');
ERROR:  value too long for type character varying(3)

（仮想生成列の場合、SELECT した時にエラー）
db1=# INSERT INTO t_virt_of (id, v1, v2) VALUES (2, 'BB','CCC');
INSERT 0 1
db1=# SELECT * FROM t_virt_of;
ERROR:  value too long for type character varying(3)

（格納された行(id=2)や列（v4）を抽出しなければエラーはなく抽出可能）
db1=# INSERT INTO t_virt_of (id, v1, v2) VALUES (3, 'BB','CC');
db1=# SELECT * FROM t_virt_of WHERE id = 3;
 id | v1 | v2 | v3  | v4
----+----+----+-----+-----
  3 | BB | CC | ABB | ACC
(1 row)
db1=# SELECT id, v1, v2, v3 FROM t_virt_of;
 id | v1 | v2  | v3
----+----+-----+-----
  2 | BB | CCC | ABB
  3 | BB | CC  | ABB
(2 rows)
```

4.3.2. 更新系 *DML* の *RETURNING* 句へ *OLD*, *NEW* の追加

前バージョンまでの RETURNING 句では、クエリ実行の前後どちらの時点の値を返すかは、SQL 文の種類により決まっていました。例えば、INSERT 文と UPDATE 文ではそれぞれ挿入後・更新後の値、DELETE 文では削除された値、MERGE 文の場合は内部で実行されたクエリに応じた適切な値が返されていました。

今回の新しい構文では、"old" と "new" というエイリアスが導入され、これにより RETURNING 句が新旧どちらの値を返すかを指定できるようになりました。

具体的には、従来に加えて以下の動作などが可能となっています。

- UPDATE 文で更新される前の値を返す(old)
- ON CONFLICT 句を含む INSERT 文で、挿入前(DO UPDATE により更新される前)の値を返す(old)
- DELETE 文の動作が RULE によって UPDATE （削除フラグ列を更新する等）に変更された場合、

更新後の値を返す(new)

また、エイリアスは RETURNING 句内で WITH を用いることで別名に変更することも可能です。
MERGE 文については、バージョン 17.x と同じ動作になります。

◆ *UPDATE 文と INSERT 文*

UPDATE 文では、デフォルトもしくは"new"を指定すると、更新後の値を返します。"old”を指定すると更新前の値を返します。

INSERT 文では、デフォルトもしくは"new”を指定すると、挿入後の値が返されます。”old”を指定すると、通常の場合は挿入前に値が存在しないため NULL を返しますが、ON CONFLICT DO UPDATE 句を用いた場合は、UPDATE による更新前の値を返します。

以下に実行例を示します。

（テーブルの作成）

```
db1=# CREATE TABLE cities (id SERIAL PRIMARY KEY,
                           city_name TEXT UNIQUE, weather TEXT);
```

（通常の INSERT の実行例）

```
db1=# INSERT INTO cities (city_name, weather) VALUES ('Tokyo', 'sunny')
      RETURNING old.*, new.*;
 id | city_name | weather | id | city_name  | weather
----+-----------+---------+----+------------+---------
    |           |         |  1 | Tokyo      | sunny
(1 row)
```

→ old の指定で NULL が返される

（UPDATE の実行例 - WITH を用いてエイリアスを別名に変更している）

```
db1=# UPDATE cities SET city_name = 'Paris' WHERE id = 1
      RETURNING WITH (old AS o, new AS n)
      o.city_name AS old, n.city_name AS new;
  old  |  new
-------+-------
 Tokyo | Paris
(1 row)
```

（old で NULL 以外が返る INSERT の実行例）

```
db1=# INSERT INTO cities (city_name, weather) VALUES ('Paris', 'snowy')
      ON CONFLICT (city_name) DO UPDATE SET weather = EXCLUDED.weather
      RETURNING old.*, new.*;
 id | city_name  | weather | id | city_name  | weather
----+------------+---------+----+------------+---------
  1 | Paris      | sunny   |  1 | Paris      | snowy
(1 row)
```

→ 既存の値と同値の挿入が UNIQUE 制約に反したため行われず、
代わりに ON CONFLICT DO UPDATE により weather カラムの値が更新された。
old は更新前の値を、new は更新後の値を返している

◆*DELETE 文*

DELETE 文では、デフォルトもしくは "old" を指定すると、削除前の値が返されます。"new" を指定すると、通常の場合は削除後に値が存在しないため NULL を返しますが、RULE AS ON DELETE DO INSTEAD UPDATE の条件下では、UPDATE による更新後の値を返します。

以下に実行例を示します。

（通常の DELETE の実行例）

```
db1=# DELETE FROM cities WHERE city_name = 'Paris' RETURNING old.*, new.*;
 id | city_name  | weather | id | city_name | weather
----+------------+---------+----+-----------+---------
  1 | Paris      | snowy   |    |           |
(1 row)
```

→ 行削除後は値が存在しないため、new の指定で NULL が返される

（new で NULL 以外の値が返る DELETE の実行例 － 削除フラグを示すカラムをテーブルに追加）

```
db1=# ALTER TABLE cities ADD COLUMN delete_flag BOOLEAN DEFAULT FALSE;
ALTER TABLE
```

（削除を行わず、代わり削除フラグを更新させる RULE を追加）

```
db1=# CREATE RULE update_instead_of_delete AS ON DELETE TO cities
        DO INSTEAD UPDATE cities SET delete_flag = TRUE WHERE id = OLD.id
        RETURNING *;
```

```
（削除対象の行を挿入）
db1=# INSERT INTO cities (city_name, weather) VALUES ('London', 'cloudy');

（DELETE 文実行）
db1=# DELETE FROM cities WHERE city_name = 'London' RETURNING old.*, new.*;
 id | city_name | weather | delete_flag | id | city_name | weather | delete_flag
----+-----------+---------+-------------+----+-----------+---------+------------
  3 | London    | cloudy  | f           |  3 | London    | cloudy  | t
(1 row)

DELETE 0
  → 削除件数は 0 となり、delete_flag 更新前の値が old、更新後の値が new として返される
```

4.3.3. UUID v7 関数の追加

PostgreSQL18 では、新しく UUID バージョン 7 を生成する関数 uuidv7() が追加されました。

これは RFC 9562 (https://tex2e.github.io/rfc-translater/html/rfc9562.html) に基づいた形式で、生成時刻順にソート可能な UUID を提供します。データベースでは UUID を主キーなどに利用するケースがありますが、UUID バージョン 4 のようなランダムな ID が生成されると、ソートやインデックスのデータ局所性が失われ、性能上の問題が発生することがありました。この点 UUID バージョン 7 では UNIX タイムスタンプを利用して値が生成されるため、この問題を回避できます。

また uuidv7() の追加に伴い、以下 2 点の変更もあります。

- uuid_extract_timestamp() および uuid_extract_version() が、 UUID バージョン 7 をサポートするようになりました。
- UUID バージョン 4 を明示的に生成する関数 uuidv4() が追加されました。

uuidv4() は従来の gen_random_uuid() のエイリアス関数であり、 uuidv7() と命名の一貫性を保つための追加です。PostgreSQL18 では、uuidv7() と uuidv4() がネイティブ関数となります。その他のバージョンの UUID 生成関数は、拡張機能「uuid-ossp」を有効にすることで利用可能です。

以下に使用例を示します。

```
（uuidv7()の単純な使用）
db1=# SELECT uuidv7();
                uuidv7
--------------------------------------
```

```
 0197afc6-7728-712e-afb2-89602fcd0118
(1 row)

（uuid_extract_timestamp()で UUID からタイムスタンプを抽出）
db1=# SELECT uuid_extract_timestamp('0197afc6-7728-712e-afb2-89602fcd0118');
   uuid_extract_timestamp
----------------------------
 2025-06-27 14:05:08.904+09
(1 row)

（uuid_extract_version()で UUID のバージョンを抽出）
db1=# SELECT uuid_extract_version('0197afc6-7728-712e-afb2-89602fcd0118');
 uuid_extract_version
----------------------
                    7
(1 row)
```

uuidv7() で生成される UUID は時系列順にソート可能であることも確認します。

```
（サンプルとなる UUID を生成）
db1=# SELECT i, uuidv7() FROM generate_series(1, 3) i;
          i |                uuidv7
------------+--------------------------------------
          1 | 0197b01c-0f4e-740a-9a50-83018f24a35f
          2 | 0197b01c-0f4e-747f-b47f-ab6b49a02ccf
          3 | 0197b01c-0f4e-7491-9409-09cffc0fb314
(3 rows)

（テーブルに上記の UUID をでたらめな順序で挿入）
db1=# CREATE TEMP TABLE t_uuidv7(seq int, id uuid);
CREATE TABLE
db1=# INSERT INTO t_uuidv7 VALUES
       (2,'0197b01c-0f4e-747f-b47f-ab6b49a02ccf'),
       (3,'0197b01c-0f4e-7491-9409-09cffc0fb314'),
       (1,'0197b01c-0f4e-740a-9a50-83018f24a35f');
INSERT 0 3
```

```
（UUIDv7 でソートすると生成時刻順になる）
db1=# SELECT * FROM t_uuidv7 ORDER BY id;
 seq |                  id
-----+--------------------------------------
   1 | 0197b01c-0f4e-740a-9a50-83018f24a35f
   2 | 0197b01c-0f4e-747f-b47f-ab6b49a02ccf
   3 | 0197b01c-0f4e-7491-9409-09cffc0fb314
(3 rows)
```

4.3.4. *LIKE* 句の非決定論的な照合順序のサポート

これまでサポートされていなかった非決定論的な照合順序（nondeterministic collation）における LIKE 句の利用が可能になりました。この実装は SQL 標準に準拠したものとなっています。

照合順序（collation）は、文字列の並び順や比較方法を決定するルールです。決定論的な照合順序（deterministic collation）では、2 つの文字列のバイト列が完全に一致する場合にのみ文字列を同じと判断します。一方、非決定論的な照合順序（nondeterministic collation）では、バイト列が異なっていても、言語的・文化的なルールや視覚的な等価性に基づいて照合できるルールです。等価と判断する例として、例えば大文字/小文字、アクセント記号の有無(例: A, À)などがあります。

PostgreSQL17.x までは、大文字小文字の区別を無視するような照合順序（例：ICU 経由の und-u-ks-level2）を設定しても、LIKE クエリを実行すると未サポートであるためエラーとなっていました。

PostgreSQL 18 では、非決定論的な照合順序で LIKE 句が利用可能となり、検索の自由度が向上しました。ただし、こうした非決定論的な照合順序においても、大文字小文字の違いやアクセント記号の有無を実際に無視するかどうかは、選択した照合順序のプロバイダー(例えば libc や ICU 等)の実装次第で結果が変わる場合があります。下記検証は、あくまで provider = icu かつ、デフォルトオプションでの結果である旨(ICU 照合の構成は co や ka など複数パラメータに依存する可能性があります)にご留意ください。

```
（ICU 照合順序レベル 2※の作成（大文字小文字を無視）
db1=# CREATE COLLATION case_insensitive
       (provider = icu, locale = 'und-u-ks-level2', deterministic = false);
CREATE COLLATION

（テスト用テーブルの作成）
```

※ ICU 照合順序レベル：等価性を決定する際の感度。level が高くなると感度が上がる。
https://www.postgresql.org/docs/devel/collation.html#ICU-COLLATION-LEVELS

```
db1=# CREATE TABLE test_like (val TEXT COLLATE case_insensitive);
CREATE TABLE

（テスト用データの挿入）
db1=# INSERT INTO test_like (val) VALUES
  ('abc'),
  ('ABC'),
  ('n'),
  ('ñ');
INSERT 0 4

（テストデータの状況を参照して確認）
db1=# SELECT * FROM test_like;
  val
--------
 abc
 ABC
 n
 ñ
(4 rows)

（大文字小文字の違いを無視するかどうかの検証）
db1=# SELECT val FROM test_like WHERE val LIKE 'abc';
 val
-----
 abc
 ABC
(2 rows)    → 大文字小文字の違いが無視されている

（アクセント記号の有無を無視するかの検証）
db1=# SELECT val FROM test_like WHERE val LIKE 'n';
 val
-----
  n
(1 rows)
```

```
db1=# SELECT val FROM test_like WHERE val LIKE 'ñ';
 val
-----
  ñ
(1 row)    → level2 ではアクセント記号を無視しない

（level1 に感度を下げた照合順序を作成）
db1=# CREATE COLLATION ignore_accent
       (provider = icu, locale = 'und-u-ks-level1', deterministic = false);
CREATE COLLATION

（アクセント記号の有無での等価性の検証）
db1=# SELECT 'ñ' = 'n' COLLATE "ignore_accent";
 ?column?
----------
 t
(1 row)    → level1 ではアクセント記号の有無に依らず等価となる

（明示的に level1 の照合順序を選び、再度アクセント記号の有無での等価性の検証）
db1=# SELECT val FROM test_like WHERE val LIKE 'n' COLLATE "ignore_accent";
 val
-----
  n
  ñ
(2 rows)    → アクセント記号の有無は無視される
```

なお、ILIKE（大小文字を無視した LIKE）は、非決定論的な照合順序との組み合わせでの仕様が曖昧なため、今回の変更の対象外となっています。

```
（ILIKE 未対応の検証 - エラーが発生する）
db1=# SELECT val FROM test_like WHERE val ILIKE 'resume';
ERROR:  nondeterministic collations are not supported for ILIKE
```

4.3.5. CASEFOLD 関数の追加

PostgreSQL18 から Unicode のケースフォールディングを行う関数 CASEFOLD() が利用可能になりました。

ケースフォールディングは、大文字・小文字を区別せずに文字列を比較するために、言語に依存せずに文字列を標準的な形式に変換する方法です。これまで多くのユーザーはその目的で LOWER() 関数を使用してきましたが、以下のような場合問題がありました。

- 2 つ以上の大小文字の形が存在する場合。例えば、"Σ" (U+03A3) は、"σ" (U+03C3) または "ς" (U+03C2) の 2 通り小文字がある
- 一部のロケールで予期しない挙動を招く可能性のある場合。例えば、"İ" (U+0130) (大文字の I の上にドットの付いた文字)の小文字が未定義の場合
- Unicode に新しい小文字が導入され、LOWER() の挙動が変更された場合

このような問題を解決するために、CASEFOLD() 関数が導入されました。特に、国際化されたアプリケーションなどの多言語環境の照合であれば、CASEFOLD() 関数を使うことで、LOWER() 関数では対応できなかった表記揺れや特殊文字の違いを吸収できます。また、Unicode のケースフォールディングの仕様は、Unicode バージョン間で比較的安定しているため、CASEFOLD() 関数による変換結果もまた LOWER() に比べ安定性が高いものとなっています。

なお、仕様上いくつかの留意するポイントがあります。

- 関数の利用に当たっては、ICU ベースの照合順序 (und-x-icu など) が必須
- サーバサイドエンコーディングが UTF8 の場合にのみ使用可能
- 引数と戻り値それぞれで、文字列の長さや Unicode 正規化形式が変わる場合がある

以下ではコーナーケースとなる入力に対して、CASEFOLD()関数と LOWER()関数の出力の比較をします。

（CASEFOLD()関数と LOWER()関数の比較、また文字列長が変化することの確認）

```
db1=# SELECT U&'\00DF' AS original,
   LENGTH(U&'\00DF') AS o_length,
   LOWER(U&'\00DF' COLLATE "und-x-icu") AS lower,
   LENGTH(LOWER(U&'\00DF')) AS l_length,
   CASEFOLD(U&'\00DF' COLLATE "und-x-icu") AS casefold,
   LENGTH(CASEFOLD(U&'\00DF' COLLATE "und-x-icu")) AS c_length;
 original | o_length | lower | l_length | casefold | c_length
----------+----------+-------+----------+----------+----------
 ß        |        1 | ß     |        1 | ss       |        2
(1 row)
```

```
("Σ"を例に２つ以上の大小文字の形（case）が存在する場合の挙動を確認
 "Σ"の小文字は文脈依存であり、文中では"σ"文末では"ς"となる）
db1=# SELECT U&'\03A3\03A3' AS original,
        LOWER(U&'\03A3\03A3' COLLATE "und-x-icu"),
        CASEFOLD(U&'\03A3\03A3' COLLATE "und-x-icu");
 original | lower | casefold
----------+-------+----------
 ΣΣ       | σς    | σσ
(1 row)    → 大小文字の形（case）が複数ある場合もケースフォールディングにより統一される
```

4.3.6. 各種制約の拡張

PostgreSQL 18 では制約についていくつか機能追加が行なわれています。
以下に主なものを取り上げます。

◆ *WITHOUT OVERLAPS 指定の主キー・ユニークキー*

主キー制約やユニーク制約に WITHOUT OVERLAPS オプションを指定して、排他制約と組み合わせることができるようになりました。
以下に使用例を示します。

```
（排他制約で使用する btree_gist 拡張を導入）
db1=# CREATE EXTENSION btree_gist;

（「同 uid で range が重ならない」条件の主キーを持つ、スケジュールテーブルを作成）
db1=# CREATE TABLE t_schedule (uid int, range tsrange, description text,
        PRIMARY KEY (uid, range WITHOUT OVERLAPS));

db1=# INSERT INTO t_schedule VALUES
        (101, '[2025-07-10 9:00,2025-07-10 11:00)', '打合せ'),
        (101, '[2025-07-10 15:00,2025-07-10 18:00)', '外出'),
        (102, '[2025-07-11 9:00,2025-07-11 13:00)', '午前休');

db1=# SELECT * FROM t_schedule;
```

```
 uid |                        range                         | description
-----+------------------------------------------------------+-------------
 101 | ["2025-07-10 09:00:00","2025-07-10 11:00:00") | 打合せ
 101 | ["2025-07-10 15:00:00","2025-07-10 18:00:00") | 外出
 102 | ["2025-07-11 09:00:00","2025-07-11 13:00:00") | 午前休
(3 rows)

（同 uid で重なりのあるエントリの INSERT を試みると、制約違反エラーになる）
db1=# INSERT INTO t_schedule VALUES
       (102, '[2025-07-11 10:00,2025-07-11 11:00)', '打合せ');
ERROR:  conflicting key value violates exclusion constraint "t_schedule_pkey"
DETAIL:  Key (uid, range)=(102, ["2025-07-11 10:00:00","2025-07-11 11:00:00"))
conflicts with existing key (uid, range)=(102, ["2025-07-11 09:00:00","2025-07-
11 13:00:00")).
```

排他制約を伴う主キーが機能していることが確認できました。ユニーク制約でも同様に使用できます。

主キー制約またはユニーク制約で WITHOUT OVERLAPS オプションを付けられるのは、一致条件で比較される通常の列が手前にある場合に限られます。排他制約のみで主キー制約、ユニーク制約を構成することはできません。

このような排他制約は PostgreSQL 9.2 以降から利用可能で、以下のように書くこともできます。

```
（以前からある排他制約を使った書き方）
db1=# CREATE TABLE t_schedule2
        (uid int NOT NULL, range tsrange NOT NULL, description text,
         EXCLUDE USING gist (uid WITH =, range WITH &&));
```

主キー制約、ユニーク制約として表現できるようになったという点が、今回の機能追加と言えます。

◆ *ENFORCED / NOT ENFORCED 指定*

チェック制約と外部キー制約に ENFORCED / NOT ENFORCED の指定ができるようになりました。これは SQL 標準で規定されている構文です。

NOT ENFORCED の制約は、制約が強制されません。振る舞いとしては制約を定義していないときと同じになります。似た機能に NOT VALID の指定がありますが、こちらは既にある値の検査を即座には行なわないということであって、新たに追加・更新される値は制約条件で検査されます。

実行例を以下に示します。

（チェック制約に NOT ENFORCED を指定する）

```
db1=# CREATE TABLE t436a (id int PRIMARY KEY, CHECK (id > 0) NOT ENFORCED);
db1=# INSERT INTO t436a VALUES (-1), (0), (1);
db1=# SELECT * FROM t436a;
 id
----
 -1      ← 制約が検査されていないことが分かる
  0
  1
(3 rows)
```

（外部キー制約に NOT ENFORCED を指定する）

```
db1=# CREATE TABLE t436b (id int PRIMARY KEY);
db1=# ALTER TABLE t436a ADD CONSTRAINT fk_t436b
        FOREIGN KEY (id) REFERENCES t436b (id) NOT ENFORCED;
db1=# ALTER TABLE t436a ALTER CONSTRAINT "fk_t436b" ENFORCED;
ERROR:  insert or update on table "t436a" violates foreign key constraint
"fk_t436b"
DETAIL:  Key (id)=(-1) is not present in table "t436b".
```

→ ENFORCED に変更すると制約が検査される ／ 現時点では t436b が空なので失敗

```
db1=# INSERT INTO t436b VALUES (-1), (0), (1);
db1=# ALTER TABLE t436a ALTER CONSTRAINT "fk_t436b" ENFORCED;
```

→ 今度は外部キー制約を満たすので成功する

（psql の \d で NOT ENFORCED 指定があれば表示される、デフォルトは ENFORCED）

```
db1=# \d t436a
                Table "public.t436a"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 id     | integer |           | not null |
Indexes:
    "t436a_pkey" PRIMARY KEY, btree (id)
Check constraints:
    "t436a_id_check" CHECK (id > 0) NOT ENFORCED
```

```
Foreign-key constraints:
    "fk_t436b" FOREIGN KEY (id) REFERENCES t436b(id)
```

実用用途としては、NOT ENFORCED で定義しておいて、条件を満たす行データが揃ってから ENFORCED に変えるという使い方が考えられます。

ただし、「ALTER TABLE .. ALTER CONSTRAINT ...」で ENFORCED 指定の切り替えができるのは、今のところ外部キー制約だけです。チェック制約については、いったん制約を削除して、ENFORCED 指定を変更して再度追加しなければなりません。

◆ *NOT NULL 制約の拡張*

NOT NULL 制約についていくつか機能追加が行なわれました。

第一に NOT NULL 制約に名前が付くようになりました。以下の手順で動作を確認します。

（テーブルを作成して定義を確認）

```
db1=# CREATE TABLE t436p (id int PRIMARY KEY, v text NOT NULL);
db1=# \d+ t436p
                                          Table "public.t436p"
 Column |  Type   | Collation | Nullable | Default | Storage  | Compression |
 Stats target | Description
--------+---------+-----------+----------+---------+----------+-------------
+--------------+-------------
 id     | integer |           | not null |         | plain    |             |
           |
 v      | text    |           | not null |         | extended |             |
           |
Indexes:
    "t436p_pkey" PRIMARY KEY, btree (id)
Not-null constraints:
    "t436p_id_not_null" NOT NULL "id"
    "t436p_v_not_null" NOT NULL "v"
Access method: heap
```

（pg_constraint システムテーブルに NOT NULL 制約に対応するエントリがある）

```
db1=# SELECT oid, conname, contype FROM pg_constraint
        WHERE conrelid = 't436p'::regclass;
```

```
  oid   |       conname       | contype
-------+---------------------+---------
 31582 | t436p_id_not_null | n
 31587 | t436p_pkey        | p
 31583 | t436p_v_not_null  | n
(3 rows)
```

加えて、NOT NULL 制約を NOT VALID で作成して、データの検査を先送りすることが可能になりました。
以下の通り、動作を確認しました。

```
（テーブルに NOT VALID の  NOT NULL 制約を付加）
db1=# CREATE TABLE t436n (id int);
db1=# INSERT INTO t436n VALUES (1), (2), (3);
db1=# ALTER TABLE t436n ADD CONSTRAINT nn_id NOT NULL id NOT VALID;
db1=# SELECT oid, conname, contype, convalidated FROM pg_constraint
        WHERE conrelid = 't436n'::regclass;
  oid   | conname | contype | convalidated
-------+---------+---------+--------------
 31608 | nn_id   | n       | f
(1 row)

（制約名を指定して VALIDATE を実行）
db1=# ALTER TABLE t436n VALIDATE CONSTRAINT nn_id;
db1=# SELECT oid, conname, contype, convalidated FROM pg_constraint
        WHERE conrelid = 't436n'::regclass;
  oid   | conname | contype | convalidated
-------+---------+---------+--------------
 31608 | nn_id   | n       | t
(1 row)
```

さらに継承親テーブルの NOT NULL 制約が継承対象かどうかを指定できるようになりました。
以下のように ALTER TABLE … ALTER CONSTRAINT … で変更します。

```
（v 列の NOT NULL 制約は継承対象外と指定して、子テーブルを作成）
db1=# ALTER TABLE t436p ALTER CONSTRAINT t436p_v_not_null NO INHERIT;
db1=# CREATE TABLE t436c1 () INHERITS (t436p);
```

```
db1=# \d+ t436c1
                                                  Table "public.t436c1"
 Column |  Type   | Collation | Nullable | Default | Storage  | Compression
 | Stats target | Description
--------+---------+-----------+----------+---------+----------+------------
-+--------------+-------------
 id     | integer |           | not null |         | plain    |
 |              |
 v      | text    |           |          |         | extended |
 |              |
Not-null constraints:
    "t436p_id_not_null" NOT NULL "id" (inherited)
Inherits: t436p                    → 子テーブルの v 列には NOT NULL が付かない
Access method: heap
```

4.3.7. ラージオブジェクトに対するデフォルト権限設定のサポート

ALTER DEFAULT PRIVILEGES 文でラージオブジェクトのデフォルト権限を設定できるようになりました。これまでも GRANT 文、REVOKE 文でラージオブジェクトに対するロール（ユーザ）ごとの参照権限、更新権限を設定することができましたが、ラージオブジェクト 1 個ごとの設定であるため、一定の権限設定を加える場合でもクライアントプログラムでラージオブジェクトを作成するごとに GRANT を実行する必要がありました。本機能追加でこれを自動化できます。

以下に実行例を示します。

ユーザー foo と bar が定義されているものとして、foo で作ったラージオブジェクトに bar に対しても参照権限を与えるようにしています。

```
（postgres ユーザでデフォルト権限設定を追加して、pg_default_acl ビューで確認）
$ psql -U postgres -d db1
db1=# ALTER DEFAULT PRIVILEGES FOR USER foo
        GRANT SELECT ON LARGE OBJECTS TO bar;
ALTER DEFAULT PRIVILEGES
db1=# SELECT * FROM pg_default_acl;
  oid  | defaclrole | defaclnamespace | defaclobjtype |       defaclacl
-------+------------+-----------------+---------------+-----------------------
 30645 |      30446 |               0 | L             | {foo=rw/foo,bar=r/foo}
(1 row)
```

```
db1=# \q

（foo ユーザで bash コマンドの実行バイナリをラージオブジェクトとして登録して、権限確認）
$ psql -U foo -d db1
db1=> \lo_import '/bin/bash' bash
lo_import 30647
db1=>  \lo_list+
                    Large objects
  ID   | Owner | Access privileges | Description
-------+-------+-------------------+-------------
 30647 | foo   | foo=rw/foo       +| bash
       |       | bar=r/foo         |
(1 row)            ↑ bar ユーザに参照権限が付与されている

db1=> \q

（bar ユーザでデータベースからラージオブジェクトを取り出せることを確認）
$ psql -U bar -d db1
db1=> \lo_export 30647 '/tmp/bash'
lo_export
db1=> \q
$ ls -l /tmp/bash
-rw-r--r--. 1 postgres postgres 1389024 Jul  9 11:42 /tmp/bash
```

以上のようにラージオブジェクトへのデフォルト権限付与が機能していることが確認できました。

4.4. セキュリティ機能

4.4.1. TLS v1.3 暗号スイート設定

SSL 関連の設定パラメータで下記の追加・変更が加わりました。

設定パラメータ	説明
ssl_tls13_ciphers	TLS v1.3 による SSL 接続で使用する暗号化スイートを指定するパラメータ。 PostgreSQL 18 から追加。デフォルトは ''（空文字列）で、使用している SSL/TLS ライブラリのデフォルトが使われる。 openssl などの SSL/TLS ライブラリの SSL_CTX_set_ciphersuites() 関数に渡せる文字列を指定できる。
ssl_groups	SSL 接続における ECDH 鍵交換での曲線を指定するパラメータ。 ssl_ecdh_curve から名称変更（旧称も使用可能）。 デフォルト値が 'prime256v1' から 'X25519:prime256v1' に変更。 openssl などの SSL/TLS ライブラリの SSL_CTX_set1_groups_list() 関数に渡せる文字列を指定できる。

PostgreSQL では以前から TLS v1.3 プロトコルによる SSL 接続に対応していましたが、サーバ側で使用可能な暗号化スイートを設定する方法がありませんでした。これが本バージョンで 設定パラメータ ssl_tls13_ciphers として加わりました。

なお、以前からあった設定パラメータ ssl_ciphers は引き続き使用可能です。こちらは、TLS v1.2 以下のバージョンのプロトコルでの SSL 接続のときに適用されます。

4.4.2. postgres_fdw、dblink で SCRAM 認証パススルーオプション追加

リモートの PostgreSQL サーバ上のテーブルにアクセスする機能である postgres_fdw と dblink で、新たなユーザ認証方式「SCRAM 認証パススルー」が利用可能になりました。これは、クライアント側の PostgreSQL サーバで接続時に使った scram-sha-256 方式の認証情報をそのままリモート PostgreSQL サーバへの接続認証に使うというものです。これにより、ユーザマッピングオブジェクトの定義に平文のパスワード文字列を持たせることを回避できます。

使用条件は以下の通りです。

- クライアント側、リモート側とも、scram-sha-256 認証を使用している
- クライアント側とリモート側の格納された認証情報（パスワード文字列ではなく、pg_authid テーブルの rolepassword 列の値そのもの）が一致している
- クライアント側は PostgreSQL 18 以上のバージョン

以下のように動作確認を行ないました。

（pg_hba.conf にエントリ追加）

```
$ vi $PGDATA/pg_hba.conf
```

```
  → 先頭に以下エントリを追加する
# TYPE    DATABASE    USER        ADDRESS                 METHOD
host       db1         foo        127.0.0.1/32           scram-sha-256

（リモートサーバ用に PostgreSQL サーバを複製して、5433 ポートで起動）
$ pg_basebackup -h localhost -D ${PGDATA}r -c fast
$ pg_ctl start -D ${PGDATA}r -o '-c port=5433'
$ pg_ctl reload -D $PGDATA
  → 元あった方の PostgreSQL はリロードして pg_hba.conf 変更を反映

（両サーバでユーザ foo を作成、パスワードを設定、認証情報を一致させる）
$ psql -p 5432 db1
db1=# CREATE USER foo PASSWORD 'pass';
db1=# SELECT rolpassword FROM pg_authid WHERE rolname = 'foo';
                                          rolpassword
------------------------------------------------------------------------------
 SCRAM-
SHA-256$4096:MXBDm4NWwh4Q45iPEuujjg==$P1szXKsLJzUzm4lZp1jsshSeK73QtwiFQ8U4Rv
U+oHk=:RuqLICsNXeLooVi2Yj4w9gSzVU1hOwSWm2fAjTKcKio=
(1 row)
db1=# \q
$ psql -p 5433 db1
db1=# CREATE USER foo;
db1=# ALTER ROLE foo PASSWORD 'SCRAM-
SHA-256$4096:MXBDm4NWwh4Q45iPEuujjg==$P1szXKsLJzUzm4lZp1jsshSeK73QtwiFQ8U4Rv
U+oHk=:RuqLICsNXeLooVi2Yj4w9gSzVU1hOwSWm2fAjTKcKio=';
                              → 実際は 1 行、手前の SELECT 結果の文字列を転記する
db1=# \q

（サンプルの外部テーブルを作成）
$ psql -p 5433 db1
db1=# CREATE TABLE t442 (id int, v text);
db1=# GRANT ALL ON t442 TO foo;
db1=# INSERT INTO t442 VALUES (100, 'a table on 5433');
db1=# \q
```

```
$ psql -p 5432 db1
db1=# CREATE EXTENSION postgres_fdw;
db1=# CREATE SERVER pg5433 FOREIGN DATA WRAPPER postgres_fdw
        OPTIONS (host '127.0.0.1', port '5433', dbname 'db1',
          use_scram_passthrough 'true');
db1=# CREATE USER MAPPING FOR foo SERVER pg5433 OPTIONS (user 'foo');
db1=# CREATE FOREIGN TABLE ft442 (id int, v text) SERVER pg5433
        OPTIONS (schema_name 'public', table_name 't442');
db1=# GRANT ALL ON ft442 TO foo;
db1=# \q

（外部テーブルへのアクセスを試す）
$ psql -p 5432 -U foo -h 127.0.0.1 db1
Password for user foo:    → 「pass」と入力
db1=> SELECT * FROM ft442;
 id  |        v
-----+-----------------
 100 | a table on 5433
(1 row)
db1=> \q

（クライアント側の PostgreSQL に接続するときに SCRAM を使っていないと失敗する）
$ psql -p 5432 -U foo -h localhost db1
   → trust 認証で接続される
db1=> SELECT * FROM ft442;
ERROR:  password or GSSAPI delegated credentials required
DETAIL:  Non-superusers must delegate GSSAPI credentials, provide a
password, or enable SCRAM pass-through in user mapping.
```

以上のように USER MAPPING オブジェクトにパスワードを持たせていなくとも、外部テーブルアクセスに成功しました。また、クライアント側への接続で scram-sha-256 認証方式を使っていない場合には失敗することも確認できました。

dblink を使う場合には以下のようになります。

```
（クライアント側に dblink 拡張を導入して、必要な定義を作成）
$ psql -p 5432 db1
```

```
db1=# CREATE EXTENSION dblink;
db1=# CREATE SERVER pg5433dblink FOREIGN DATA WRAPPER dblink_fdw
        OPTIONS (host '127.0.0.1', dbname 'db1', port '5433',
          use_scram_passthrough 'true');
db1=# CREATE USER MAPPING FOR foo SERVER pg5433dblink OPTIONS (user 'foo');
db1=# GRANT ALL ON FOREIGN SERVER pg5433dblink TO foo;
db1=# \q

（dblink でリモートサーバへのアクセスを確認）
$ psql -p 5432 -U foo -h 127.0.0.1 db1
Password for user foo:    → 「pass」と入力
db1=> SELECT dblink_connect('pg5433dblink');
 dblink_connect
----------------
 OK
(1 row)

db1=> SELECT * FROM dblink('SELECT id, v FROM t442') AS t(id int, v text);
 id  |        v
-----+-----------------
 100 | a table on 5433
(1 row)
```

以上のように dblink を使っても同様に SCRAM 認証パススルーでのリモートテーブルへのアクセスができました。dblink で SCRAM 認証パススルーを行うには、接続文字列を使う方法ではなく、dblink_fdw でサーバを定義する方式を使う必要があります。

4.5. 運用管理

4.5.1. ダンプ／リストアの拡張

PostgreSQL18 ではダンプ・リストアにもいくつか機能追加があります。

◆ *プランナ統計情報のダンプ／リストア*

テーブルのプランナ統計情報をダンプ、リストアできるようになりました。これまで、ある時点のプランナ統計情報を保存しておいて復旧するには pg_dbms_stats のようなサードパーティ拡張モジュールを使う必要がありましたが、PostgreSQL 本体機能として利用可能になりました。

以下に使用例を示します。

```
（サンプルテーブル作成 - c1 は 0 と 1 だけ、c2 が 0 から 9999 の値となる）
db1=# CREATE TABLE t451 (id int PRIMARY KEY, c1 int, c2 int);
db1=# CREATE INDEX ON t451 (c1);
db1=# CREATE INDEX ON t451 (c2);
db1=# INSERT INTO t451 SELECT g, g % 2, g % 10000
        FROM generate_series(1, 100000) g;
db1=# VACUUM ANALYZE t451;

（c2 列の方が値の種類が多いので、先に c2 を調べる実行プランになる）
db1=# explain SELECT * FROM t451 WHERE c1 = 1 AND c2 = 1;
                                QUERY PLAN
---------------------------------------------------------------------------
 Bitmap Heap Scan on t451  (cost=4.37..40.44 rows=5 width=12)
   Recheck Cond: (c2 = 1)
   Filter: (c1 = 1)
   ->  Bitmap Index Scan on t451_c2_idx  (cost=0.00..4.37 rows=10 width=0)
         Index Cond: (c2 = 1)
(5 rows)
db1=# \q

（テーブル t451 について、プランナ統計情報のみのダンプを取得する。
  オプション指定をしなければ、定義、データ、プランナ統計の全てが含まれる）
$ pg_dump -d db1 -t t451 --statistics-only -f t451_stat.sql

（プランナ統計のダンプは、プランナ統計を登録する関数実行の形をとる）
$ cat t451_stat.sql
--
-- PostgreSQL database dump
  （中略）
SELECT * FROM pg_catalog.pg_restore_relation_stats(
```

```
        'version', '180000'::integer,
        'schemaname', 'public',
        'relname', 't451',
        'relpages', '541'::integer,
        'reltuples', '100000'::real,
        'relallvisible', '541'::integer,
        'relallfrozen', '0'::integer
);
SELECT * FROM pg_catalog.pg_restore_attribute_stats(
        'version', '180000'::integer,
        'schemaname', 'public',
        'relname', 't451',
        'attname', 'c1',
        'inherited', 'f'::boolean,
        'null_frac', '0'::real,
        'avg_width', '4'::integer,
        'n_distinct', '2'::real,
        'most_common_vals', '{0,1}'::text,
        'most_common_freqs', '{0.5017,0.4983}'::real[],
        'correlation', '0.5027289'::real
);
```

（後略）

（c1 列と c2 列の値を入れ替えると、c1 を先に調べる実行プランに変わる）

```
$ psql -d db1
db1=# UPDATE t451 SET c1 = c2, c2 = c1;
db1=# VACUUM ANALYZE t451;
db1=# explain SELECT * FROM t451 WHERE c1 = 1 AND c2 = 1;
                                QUERY PLAN
-------------------------------------------------------------------------
 Bitmap Heap Scan on t451  (cost=4.49..42.60 rows=5 width=12)
   Recheck Cond: (c1 = 1)
   Filter: (c2 = 1)
   ->  Bitmap Index Scan on t451_c1_idx  (cost=0.00..4.49 rows=10 width=0)
         Index Cond: (c1 = 1)
```

```
(5 rows)
db1=# \q

（ここでプランナ統計情報をリストアすると、元の実行プランが選択される）
[postgres@rocky9 ~]$ psql -d db1 -f t451_stat.sql
  （出力省略）

$ psql db1
db1=# explain SELECT * FROM t451 WHERE c1 = 1 AND c2 = 1;
                                QUERY PLAN
---------------------------------------------------------------------------
 Bitmap Heap Scan on t451  (cost=4.73..149.01 rows=20 width=12)
   Recheck Cond: (c2 = 1)
   Filter: (c1 = 1)
   ->  Bitmap Index Scan on t451_c2_idx  (cost=0.00..4.72 rows=40 width=0)
         Index Cond: (c2 = 1)
(5 rows)
```

以上のようにプランナ統計情報のダンプとリストアができることが確認できました。

メリットとしては、ダンプによるバックアップからリストアした後に ANALYZE を実行する必要が無くなるということがあります。

それに加えて、SQL 実行でプランの変動を避けたい場合に、対象テーブルについてプランナ統計情報だけのダンプを取得しておいて、それを SQL 実行前にリストアするという使い方も考えられます。ただし、PostgreSQL はプラン選択にあたってはプランナ統計以外の情報（例えばテーブルファイルのサイズ）も参考にしますので、この手法では実行プランを固定できない場合もあります。

◆ *バイナリ形式の pg_dumpall*

> 注意
>
> バイナリ形式の pg_dumpall はコミット ce9a6244b5b4 により 2025 年 7 月 30 日に取り下げられたため、beta3・正式版では-F オプションは利用できない見込みです。

これまで pg_dumpall コマンドは plain 形式（テキスト形式）の出力しかできませんでしたが、本バージョンで各種形式の出力に対応しました。-F (--format) オプションで custom、directory、tar、plain という pg_dump コマンドで使用可能であった各型式を指定できます。

以下に各型式での実行と出力結果を示します。

```
（custom 形式でダンプ - ディレクトリで出力される）
$ pg_dumpall -F c -f ./dumpall.c
$ ls -F ./dumpall.c/
databases/  global.dat  map.dat
$ ls -F ./dumpall.c/databases/
1.dmp  16384.dmp  5.dmp                    ← データベースごとに dmp ファイルができる

（directory 形式でダンプ - ディレクトリで出力される）
$ pg_dumpall -F d -f ./dumpall.d
$ ls -F ./dumpall.d/
databases/  global.dat  map.dat
$ ls -F ./dumpall.d/databases/
1/  16384/  5/                             ← データベースごとにディレクトリができる

（tar 形式でダンプ - ディレクトリで出力される）
$ pg_dumpall -F t -f ./dumpall.t
$ ls -F ./dumpall.t/
databases/  global.dat  map.dat
$ ls -F ./dumpall.t/databases/
1.tar  16384.tar  5.tar                    ← データベースごとに tar ファイルができる

（plain 形式でダンプ - これは従来と同じ動作で一つのテキストファイルが出力される）
$ pg_dumpall -F p -f ./dumpall.p
$ ls -F ./dumpall.p
./dumpall.p
```

plain 以外の形式では、pg_dump コマンドの各形式のデータベースごとのダンプファイルが含まれるディレクトリとして出力されました。

続いて、リストアを試しました。ここでは tar 形式の出力をリストアしています。

```
$ pg_ctl stop
サーバー停止処理の完了を待っています....完了
サーバーは停止しました
$ cp $PGDATA/postgresql.conf .
```

```
$ rm -rf $PGDATA/*
$ initdb --no-locale -E UTF8
  （出力省略）
$ mv ./postgresql.conf $PGDATA/
$ pg_ctl start
  （出力省略）
$ pg_restore -U postgres -d postgres -C dumpall.t
$
```

pg_dumpall のバイナリ出力は、pg_restore でリストアできます。リストアするファイルとしてディレクトリを指定します。このとき必ず -C (--create) オプションを指定しなければなりません。

また、pg_dumpall 出力のリストアに対応したことに合わせて、pg_restore では-g (--globals-only) というオプションも追加されました。ダンプデータのうちグローバルデータのみをリストアします。

4.5.2. EXPLAIN 文の強化

実行計画を出力するコマンドである EXPLAIN 文が強化されています。本項では主なものを取り上げます。

なお、本項で取り上げているもの以外にも、Material ノード・Window Aggregate ノードが利用したメモリ・ディスクの量を出力する変更、loops が 1 より大きい場合に小数点以下 2 桁まで rows を出力する変更など、いくつかの改善が行なわれ、EXPLAIN 文から確認できる情報が増えています。

◆ *ANALYZE オプション追加時の BUFFERS オプション自動追加*

ANALYZE オプションを指定した場合、BUFFERS オプションも有効化されました。

ANALYZE オプションを指定すると、実際にクエリを実行した上で実行計画を出力します。また BUFFERS オプションを指定すると、共有バッファのヒット数などバッファの利用状況に関する情報が確認できます。

実行計画を分析する場合、ANALYZE オプションに加えて、バッファの利用状況も把握することは有効なことが多いため、今回の変更が加わりました。

```
（サンプルデータを作成 - 10 万件テーブル t452）
db1=# CREATE TABLE t452 (id int, v text, PRIMARY KEY (id));
db1=# INSERT INTO t452 SELECT g, g::text FROM generate_series(0, 100000 - 1)
g;
db1=# VACUUM ANALYZE t452;
```

```
（PostgreSQL 17.5 の結果）
db1=# explain analyze SELECT * FROM t452;
                                QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on t452  (cost=0.00..1540.00 rows=100000 width=9)
                   (actual time=0.015..14.469 rows=100000 loops=1)
 Planning Time: 0.670 ms
 Execution Time: 25.514 ms
(3 rows)

（18beta1 の結果）
db1=# explain analyze SELECT * FROM t452;
                                QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on t452  (cost=0.00..1540.00 rows=100000 width=9)
                   (actual time=0.019..15.071 rows=100000.00 loops=1)
   Buffers: shared hit=540
 Planning Time: 0.082 ms
 Execution Time: 27.608 ms
(4 rows)
```

◆ *インデックス参照回数の表示追加*

ANALYZE オプションを指定した場合、クエリがインデックスを参照した回数が表示されるようになりました。

この情報は、インデックスにアクセスする回数が実行するまでわからないようなケース、例えばクエリに IN 句がありその要素とマッチするインデックススキャンをする場合、あるいはインデックスのスキップスキャンが実行される場合、特に有用です。

以下では、IN 句がありその要素とマッチするインデックススキャンをする場合について例示します。インデックスのスキップスキャンでの EXPLAIN 文の例は、4.1.2. を参照してください。

```
(PostgreSQL 17.5 の結果)
db1=# explain analyze SELECT * FROM t452 WHERE id IN (100, 101, 102);
                                QUERY PLAN
-------------------------------------------------------------------------
 Index Scan using t452_pkey on t452  (cost=0.29..16.93 rows=3 width=9)
```

```
                              (actual time=0.017..0.019 rows=3 loops=1)
   Index Cond: (id = ANY ('{100,101,102}'::integer[]))
 Planning Time: 0.240 ms
 Execution Time: 0.040 ms
(4 rows)
```

（18beta2 の結果）

```
db1=# explain analyze SELECT * FROM t452 WHERE id IN (100, 101, 102);
                                QUERY PLAN
-------------------------------------------------------------------------
 Index Scan using t452_pkey on t452  (cost=0.29..16.93 rows=3 width=9)
                           (actual time=0.043..0.046 rows=3.00 loops=1)
   Index Cond: (id = ANY ('{100,101,102}'::integer[]))
   Index Searches: 1
   Buffers: shared hit=3
 Planning Time: 0.129 ms
 Execution Time: 0.067 ms
(6 rows)
```

（18beta2 の結果 - 複数回数インデックスを参照するよう IN 句の条件を変更）

```
db1=# explain analyze SELECT * FROM t452 WHERE id IN (100, 1000, 10000);
                                QUERY PLAN
-------------------------------------------------------------------------
 Index Scan using t452_pkey on t452  (cost=0.29..16.93 rows=3 width=9)
                           (actual time=0.034..0.053 rows=3.00 loops=1)
   Index Cond: (id = ANY ('{100,1000,10000}'::integer[]))
   Index Searches: 3            ← インデックス参照回数が増加
   Buffers: shared hit=9
 Planning Time: 0.118 ms
 Execution Time: 0.074 ms
(6 rows)
```

◆ *無効ノードの明示的な出力*

ANALYZE オプションを指定した場合、enable_seqscan など GUC パラメータを off にして無効化されたプランノードが存在すると、無効化されたノードであることを出力するようになりました。

PostgreSQL 17 以前は、無効化したプランノードは大きなコスト値(100 億)を加算することで、そのプランノードを利用するプランを選択しないよう調整していましたが、複雑なクエリなどの場合、他のプランも 100 億を超えるコスト値となり、ユーザの意図と異なり GUC パラメータで off にしたプランノードを利用するプランが選択されることがありました。

この問題に対応するため、PostgreSQL 18 から無効化したプランはコスト値ではなく、無効化したプランノード数を別途カウントし、プラン選択時に考慮するようになりました。これによりコストの大きさに関わらず、無効化したプランが選択されないよう調整できるようになりました。

以下例では、無効化したプランを選択せざるをえないクエリを実行し、EXPLAIN 文の出力を確認しています。

(PostgreSQL 17.5 の結果)

```
db1=# SET enable_seqscan = off;
db1=# explain analyze SELECT * FROM t452 WHERE v = '100';
```

↑ インデックスが利用できない条件のクエリ。
無効化したもののシーケンシャルスキャンを選択せざるをえない

```
                                QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on t452 (cost=10000000000.00..10000001790.00 rows=1 width=9)
                  (actual time=0.170..13.134 rows=1 loops=1)
```

↑ コストが 100 億以上となっており、無効化したプランノードである
シーケンシャルスキャンを選択していることがわかる

```
   Filter: (v = '100'::text)
   Rows Removed by Filter: 99999
 Planning Time: 7.835 ms
 Execution Time: 13.924 ms
(5 rows)
```

(18beta2 の結果)

```
db1=# explain analyze SELECT * FROM t452 WHERE v = '100';
                                QUERY PLAN
-------------------------------------------------------------------------
Seq Scan on t452  (cost=0.00..1790.00 rows=1 width=9) (actual
time=0.041..12.827 rows=1.00 loops=1)
   Disabled: true          ← シーケンシャルスキャンが無効化されていることがわかる
   Filter: (v = '100'::text)
```

```
   Rows Removed by Filter: 99999
   Buffers: shared hit=540
 Planning Time: 0.108 ms
 Execution Time: 12.856 ms
(7 rows)
```

4.5.3. *VACUUM*、*ANALYZE* に関するモニタリング項目の拡張

VACUUM、ANALYZE に関するモニタリングがいくつか拡張されました。

◆*pg_stat_all_tables 等にVACUUM、ANALYZE 総所要時間の列が追加*

実行時統計情報を参照するビュー pg_stat_all_tables（および pg_stat_user_tables、pg_stat_sys_tables）に、VACUUM、ANALYZE の総使用時間をミリ秒単位で報告する以下の列が追加されました。

- total_vacuum_time　　VACUUM 総使用時間 (ms)
- total_autovacuum_time　　ANALYZE 総使用時間 (ms)
- total_analyze_time　　自動 VACUUM 総使用時間 (ms)
- total_autoanalyze_time　　自動 ANALYZE 総使用時間 (ms)

これらは track_counts = on （デフォルト有効）で収集される項目に含まれます。

以前からあった、vacuum_count、autovacuum_count、analyze_count、autoanalyze_count 列の値と組み合わせて、以下例のように 1 回の平均所要時間を調べる応用ができます。

```
db1=# SELECT schemaname, relname,
        total_autovacuum_time/autovacuum_count avgtime FROM
          pg_stat_user_tables WHERE autovacuum_count != 0 ORDER BY 3 DESC;
 schemaname |     relname      | avgtime
------------+------------------+---------
 public     | pgbench_accounts |      95
 public     | pgbench_history  |   25.75
 public     | pgbench_branches |   20.25
 public     | pgbench_tellers  |       1
(4 rows)
```

◆ *VACUUM、ANALYZE 休止時間の報告*

自動および手動の VACUUM 設定に基づいて休止を挟みながら実行されます。この休止時間を報告できるようになりました。新たな設定パラメータ track_cost_delay_timing（デフォルト off）を on（有効）にすると情報取得可能になります。

以下の通り動作を確認しました。

```
$ vi $PGDATA/postgresql.conf
  → 以下の設定を加える
    track_cost_delay_timing = on

$ pg_ctl reload
$ psql db1
db1=# SET vacuum_cost_delay TO '1ms';
SET
db1=# VACUUM (VERBOSE) pgbench_accounts;
INFO:  vacuuming "db1.public.pgbench_accounts"
INFO:  finished vacuuming "db1.public.pgbench_accounts": index scans: 0
pages: 0 removed, 16397 remain, 509 scanned (3.10% of total), 0 eagerly
scanned
tuples: 80 removed, 999908 remain, 0 are dead but not yet removable
removable cutoff: 112981865, which was 0 XIDs old when operation ended
frozen: 0 pages from table (0.00% of total) had 0 tuples frozen
visibility map: 3 pages set all-visible, 0 pages set all-frozen (0 were all-
visible)
index scan bypassed: 120 pages from table (0.73% of total) have 120 dead
item identifiers
delay time: 90.503 ms
avg read rate: 25.516 MB/s, avg write rate: 0.213 MB/s
buffer usage: 597 hits, 360 reads, 3 dirtied
WAL usage: 86 records, 87 full page images, 682274 bytes, 0 buffers full
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.11 s
VACUUM

db1=# ANALYZE (VERBOSE) pgbench_accounts;
INFO:  analyzing "public.pgbench_accounts"
INFO:  "pgbench_accounts": scanned 16397 of 16397 pages, containing 1000000
```

```
live rows and 120 dead rows; 30000 rows in sample, 1000000 estimated total
rows
INFO:  finished analyzing table "db1.public.pgbench_accounts"
delay time: 562.870 ms
avg read rate: 144.869 MB/s, avg write rate: 0.000 MB/s
buffer usage: 3280 hits, 13314 reads, 0 dirtied
WAL usage: 22 records, 0 full page images, 2839 bytes, 0 buffers full
system usage: CPU: user: 0.07 s, system: 0.03 s, elapsed: 0.71 s
ANALYZE
```

VACUUM と ANALYZE に VERBOSE オプションを与えたときの出力に delay time の報告が加わっていることが確認できました。この delay time は、log_autovacuum_min_duration 設定に基づく、自動 VACUUM プロセスの動作報告ログメッセージでも同様に報告されます。

また、実行中の VACUUM・ANALYZE 処理の進捗を報告するシステムビュー pg_stat_progress_vacuum および pg_stat_progress_analyze にも delay_time 列が追加されました。

以下は、 psql の \watch を使って、進捗を観測した結果です。

（バックグラウンドで VACUUM を実行しながら、以下を実行する）

```
db1=# \x on
db1=# SELECT * FROM pg_stat_progress_vacuum \watch 1
(0 rows)

Wed Jul 16 12:18:22 2025 (every 1s)

-[ RECORD 1 ]--------+--------------
pid                  | 2866
datid                | 30586
datname              | db1
relid                | 31752
phase                | scanning heap
heap_blks_total      | 16403
heap_blks_scanned    | 521
heap_blks_vacuumed   | 0
index_vacuum_count   | 0
max_dead_tuple_bytes | 67108864
dead_tuple_bytes     | 24576
```

```
num_dead_item_ids     | 27
indexes_total         | 0
indexes_processed     | 0
delay_time            | 553.331642

Wed Jul 16 12:18:23 2025 (every 1s)

-[ RECORD 1 ]--------+--------------
pid                   | 2866
datid                 | 30586
    《中略》
indexes_processed     | 0
delay_time            | 1502.127233

Wed Jul 16 12:18:24 2025 (every 1s)
    《後略》
```

delay_time にその時点までの休止時間の合計が報告されています。

4.5.4. プロセスごとの I/O 量、WAL 量の統計取得

プロセスごとにストレージ I/O 量、WAL 書き出し量の統計を取得できるようになりました。
そのためのインタフェースとして、以下の関数が追加されました。

関数名	説明
pg_stat_get_backend_io(pid)	指定のバックエンドプロセスについて、オブジェクト種別ごと、コンテキストごとに、ストレージ I/O 統計情報を返す。統計情報の項目は、pg_stat_io ビューと同じ。
pg_stat_get_backend_wal(pid)	指定のバックエンドプロセスについて、WAL 出力に関する統計情報を返す。統計情報の項目は PostgreSQL18 の pg_stat_wal ビューと同じ（※１）。
pg_stat_reset_backend_stats(pid)	指定のバックエンドプロセスについて、プロセスごとの統計情報をリセットする。

※１ :PostgreSQL 18 から pg_stat_wal ビューから 4 つの列 wal_write、wal_sync、wal_write_time、wal_sync_time が除去されています。同じ情報が pg_stat_io ビューから取得できます。

新たな関数はいずれも pid（プロセス ID）を引数にとります。現在実行中でない（すなわち存在していない）プロセスの統計情報は取得できません。以下の例のように pg_stat_activity ビューと組み合わせて使用するのが良いでしょう。本動作確認手順では統計情報を取る対象の持続的なバックエンドプロセスを作るため、最初に pgbench をバックグランドで実行しています。

```
（pgbench を同時接続数 3 で 10 分間バックグラウンドで実行）
$ pgbench -i db1
    《出力省略》
$ pgbench -T 600 -c 3 db1 > /dev/null 2>1 &

（プロセスごとの統計情報を参照 - ストレージ I/O の統計）
$ psql db1
db1=# SELECT a.pid, s.* FROM pg_stat_activity a,
        LATERAL pg_stat_get_backend_io(pid) s ORDER BY write_bytes DESC;
 pid  |   backend_type    |    object     |  context  | reads | read_bytes |
 read_time | writes | write_bytes | write_time | writebacks | writeback_time |
 extends | extend_bytes | extend_time | hits  | evictions | reuses | fsyncs |
 fsync_time | stats_reset
------+-------------------+---------------+-----------+-------+------------
+-----------+--------+-------------+------------+------------+----------------
+---------+--------------+-------------+-------+-----------+--------+--------
+------------+-------------
 1884 | client backend    | wal           | normal    |     0 |          0 |
         0 |   2374 |    20881408 |          0 |            |                |
         |              |             |       |           |        |   2374 |
          0 |
 1886 | client backend    | wal           | normal    |     0 |          0 |
         0 |   2314 |    20291584 |          0 |            |                |
         |              |             |       |           |        |   2313 |
          0 |
 1885 | client backend    | wal           | normal    |     0 |          0 |
         0 |   2262 |    19791872 |          0 |            |                |
         |              |             |       |           |        |   2262 |
          0 |
 1580 | walwriter         | wal           | normal    |       |            |
```

```
           |     78 |     4481024 |           0 |             |                 |
         |               |             |        |           |         |     77 |
          0 |
1884 | client backend    | relation      | vacuum    |     0 |           0 |
         0 |      0 |           0 |           0 |          0 |               0 |
       0 |             0 |           0 |     0 |          0 |      0 |         |
           |
```

《後略》

→「～_time」列が空欄なのは track_io_timing と track_wal_io_timing が デフォルトの off のままであるため

（プロセスごとの統計情報を参照 - WAL 出力の統計）

```
db1=# SELECT a.pid, a.backend_type, w.* FROM pg_stat_activity a,
        LATERAL pg_stat_get_backend_wal(pid) w;
 pid  |         backend_type         | wal_records | wal_fpi | wal_bytes |
wal_buffers_full | stats_reset
------+------------------------------+-------------+---------+-----------+
------------------+-------------
 1884 | client backend               |       30629 |       0 |   2449456 |
                0 |
 1885 | client backend               |       29624 |       0 |   2369002 |
                0 |
 1886 | client backend               |       29879 |       0 |   2380334 |
                0 |
 1888 | client backend               |           0 |       0 |         0 |
                0 |
 1581 | autovacuum launcher          |             |         |           |
                  |
```

《後略》

（特定プロセスだけ統計情報をリセットしてみる）

```
db1=# SELECT pg_stat_reset_backend_stats(1885);
 pg_stat_reset_backend_stats
-----------------------------
```

```
(1 row)

db1=# SELECT a.pid, a.backend_type, w.* FROM pg_stat_activity a,
        LATERAL pg_stat_get_backend_wal(pid) w;
 pid  |         backend_type          | wal_records | wal_fpi | wal_bytes |
 wal_buffers_full |          stats_reset
------+-------------------------------+-------------+---------+-----------+
------------------+-------------------------------
 1884 | client backend                |       48530 |       0 |   3885932 |
                0 |
 1885 | client backend                |        1783 |       0 |    141448 |
                0 | 2025-07-15 09:40:04.85011+09    ↑ リセットで値が減っている
 1886 | client backend                |       47503 |       0 |   3793560 |
                0 |
  《後略》
```

プロセスごとのストレージ I/O と WAL の統計情報が取得できました。また、特定プロセスだけプロセスごとの統計情報をリセットすることができました。

postgresql.conf で synchronous_commit = off と設定して、「pg_ctl reload」で反映したうえで、同じように pgbench と統計情報を参照する問い合わせをすると、以下のように、pg_stat_get_backend_io の出力で WAL 出力のストレージ I/O が各バックエンドプロセスではなく、専ら walwriter プロセスに集中している結果が得られます。

```
db1=# SELECT a.pid, s.* FROM pg_stat_activity a,
        LATERAL pg_stat_get_backend_io(pid) s ORDER BY write_bytes DESC;
 pid  |   backend_type    |    object     |  context  | reads | read_bytes |
 read_time | writes | write_bytes | write_time | writebacks | writeback_time |
 extends | extend_bytes | extend_time |  hits  | evictions | reuses | fsyncs |
 fsync_time | stats_reset
------+-------------------+---------------+-----------+-------+------------+
----------+--------+-------------+------------+------------+----------------+
---------+--------------+-------------+--------+-----------+--------+--------+
------------+-------------
 1580 | walwriter         | wal           | normal    |       |            |
           |    171 |    16138240 |          0 |          |                |
         |              |             |        |           |        |    168 |
```

```
          0 |
2251 | client backend     | relation       | bulkwrite |      0 |           0 |
         0 |       0 |            0 |           0 |        0 |                0 |
       0 |             0 |            0 |       0 | 0 |       0 |          |
《後略》
```

このように、設定による挙動の変化を確認するために、新たな統計情報を活用できます。

4.5.5. 論理レプリケーションでコンフリクト詳細報告

論理（ロジカル）レプリケーションでコンフリクトが発生したときに、より詳細な報告がログと実行時統計情報に記録されるようになりました。コンフリクトは以下表に示す種別に分類されて、コンフリクト種別ごとに統計が蓄積されます。

コンフリクト種別	説明
insert_exists	パブリケーション側で INSERT した行が、サブスクリプション側のテーブルに既に在った場合。すなわち、サブスクリプション側で主キー・ユニークキーの制約違反が生じた場合。
update_origin_differs	パブリケーション側で UPDATE した行と、サブスクリプション側で UPDATE した行のオリジン（作成／更新元のサーバ）が異なっていた場合。すなわち、パブリケーション側と異なる行であったものをレプリケーションで上書きした場合。 報告されますがエラーにはなりません。
update_exists	パブリケーション側で UPDATE して、サブスクリプション側でも UPDATE を試みたら主キー・ユニークキーの制約違反が生じた場合。
update_missing	パブリケーション側で UPDATE して、サブスクリプション側でも UPDATE を試みたら該当行が無かった場合。 報告されますがエラーにはなりません。
delete_origin_differs	パブリケーション側で DELETE した行と、サブスクリプション側で DELETE した行のオリジン（作成／更新元のサーバ）が異なっていた場合。すなわち、パブリケーション側と異なる行であったものをレプリケーションで削除した場合。 報告されますがエラーにはなりません。

コンフリクト種別	説明
delete_missing	パブリケーション側で DELETE して、サブスクリプション側でも DELETE を試みたら該当行が無かった場合。 報告されますがエラーにはなりません。
multiple_unique_conflicts	パブリケーション側での DML 実行に対して、サブスクリプション側でのレプリケーション適用で複数の主キー制約またはユニーク制約に同時に違反が発生した場合。

これらのコンフリクトについてタイムスタンプを含む詳細な報告を得るには、track_commit_timestamp 設定が有効であることが必要です。特に update_origin_differs と delete_origin_differs については、track_commit_timestamp が無効だと検知すらされません。

以下の手順で、動作確認を行ないました。

```
（論理レプリケーションを構築）
$ psql db1
db1=# CREATE TABLE t455 (id int PRIMARY KEY, v text);
db1=# INSERT INTO t455 SELECT g, 'data' || g FROM generate_series(1, 100) g;
db1=# \q     -- 主キー列 id を持つ 100 行のテーブルを作成

$ vi $PGDATA/postgresql.conf
  → 以下の設定を付与
    wal_level = logical    #replica
    track_commit_timestamp = on    #off

$ pg_ctl restart                          ← 設定変更を反映するため PostgreSQL 再起動
$ pg_basebackup -D ${PGDATA}x -c fast     ← データベースクラスタディレクトリを複製
$ pg_ctl start  -D ${PGDATA}x -o '-c port=5433'    ← PostgreSQL をもう 1 つ起動
$ psql db1
db1=# CREATE PUBLICATION pub1 FOR TABLE t455;
db1=# \q
$ psql -p 5433 db1
db1=# CREATE SUBSCRIPTION sub1 CONNECTION 'port=5432 dbname=db1'
         PUBLICATION pub1 WITH (copy_data=off);
db1=# \q
```

ここまででテーブル t455 の論理レプリケーションが構成できました。続いて、コンフリクトを発生させていきます。

```
（id=101 の行をサブスクリプション側で先に作成して、コンフリクトを起こす）
$ psql -p 5433 db1
db1=# INSERT INTO t455 VALUES (101, 'X');
db1=# \q
$ psql db1
db1=# INSERT INTO t455 VALUES (101, 'Y');
db1=# \q

（サブスクリプション側 PostgreSQL のログファイルを確認）
$ cat ${PGDATA}x/log/* | tail
2025-07-03 09:51:42.077 JST [1537] ERROR:  conflict detected on relation
"public.t455": conflict=insert_exists    ←コンフリクト種別
2025-07-03 09:51:42.077 JST [1537] DETAIL:  Key already exists in unique
index "t455_pkey", modified locally in transaction 111972803 at 2025-07-03
09:50:52.060684+09.    ← 競合する行の作成トランザクション ID や作成日時を含む報告
       Key (id)=(101); existing local tuple (101, X); remote tuple (101, Y).
2025-07-03 09:51:42.077 JST [1537] CONTEXT:  processing remote data for
replication origin "pg_38779" during message type "INSERT" for replication
target relation "public.t455" in transaction 111972799, finished at
42/D5018EA0    ← オリジン名、処理種別、テーブル、トランザクション ID、LSN を報告
2025-07-03 09:51:42.077 JST [1462] LOG:  background worker "logical
replication apply worker" (PID 1537) exited with exit code 1

（サブスクリプション側で pg_stat_subscription_stats ビューを確認）
$ psql -p 5433 db1
db1=# \x
db1=# SELECT * FROM pg_stat_subscription_stats;
-[ RECORD 1 ]--------------------+------
subid                            | 38779
subname                          | sub1
apply_error_count                | 15
sync_error_count                 | 0      ↓ 以下、新たに追加された列 ↓
confl_insert_exists              | 14     ← insert_exists 発生がカウントされている
```

```
confl_update_origin_differs      | 0
confl_update_exists              | 0
confl_update_missing             | 0
confl_delete_origin_differs      | 0
confl_delete_missing             | 0
confl_multiple_unique_conflicts | 0
stats_reset                      |

（サブスクリプション側で原因の行を削除して、コンフリクトを解消させる）
db1=# DELETE FROM t455 WHERE id = 101;
```

論理レプリケーションのコンフリクト発生で、サブスクリプション側により詳細なログメッセージが出ていることが確認できました。また、 pg_stat_subscription_stats ビューにコンフリクト種別ごとの発生統計のための列が追加されて、発生させたコンフリクトの統計がカウントされていることが確認できました。

ログメッセージでレプリケーションオリジンの名前「pg_38779」が報告されています。これはシステムテーブル pg_subscription の oid 列の値が 38779 の行を調べることでサブスクリプション名に紐づけできます。

17.x までは検出できなかった種類のコンフリクトについても動作確認を行ないました。

```
（サブスクリプション側に書かれたデータをレプリケーションで上書きさせる）
$ psql -p 5433 db1
db1=# UPDATE t455 SET v = 'AAA' WHERE id = 101;
db1=# \q
$ psql db1
db1=# UPDATE t455 SET v = 'aaa' WHERE id = 101;
db1=# \q

$ cat ${PGDATA}x/log/* | tail
2025-07-03 10:19:49.543 JST [1557] LOG:  conflict detected on relation
"public.t455": conflict=update_origin_differs  ←コンフリクト種別
2025-07-03 10:19:49.543 JST [1557] DETAIL:  Updating the row that was
modified locally in transaction 111972829 at 2025-07-03 10:19:39.889196+09.
      Existing local tuple (101, AAA); remote tuple (101, aaa); replica
identity (id)=(101).   ← 上書きされる行の内容がログメッセージに残る
2025-07-03 10:19:49.543 JST [1557] CONTEXT:  processing remote data for
replication origin "pg_38779" during message type "UPDATE" for replication
```

```
target relation "public.t455" in transaction 111972804, finished at
42/D5049EE0
  《後略》

（サブスクリプション側で対象行を先に削除してから、パブリッシャで削除する）
$ psql -p 5433 db1
db1=# DELETE FROM t455 WHERE id = 101;
db1=# \q
$ psql db1
db1=# DELETE FROM t455 WHERE id = 101;
db1=# \q

$ cat ${PGDATA}x/log/* | tail
2025-07-03 13:47:50.792 JST [2183] LOG:  conflict detected on relation
"public.t455": conflict=delete_missing  ←コンフリクト種別
2025-07-03 13:47:50.792 JST [2183] DETAIL:  Could not find the row to be
deleted.
        Replica identity (id)=(101).
2025-07-03 13:47:50.792 JST [2183] CONTEXT:  processing remote data for
replication origin "pg_38779" during message type "DELETE" for replication
target relation "public.t455" in transaction 111972806, finished at
42/D504EA38
  《後略》
```

これらのケースでも詳細なログメッセージが出力されました。これらのログレベルは ERROR ではなく LOG であって、特に対処を行なわなくとも、レプリケーションはそのまま継続されます。

17.x までは、このような場合には上書きされても何も検知されませんでした。PostgreSQL 18 からは、論理レプリケーションについて何らかデータ異常が起きている可能性を、これらのログメッセージによって運用者が認識することができます。

5. 非互換変更

PostgreSQL 18 にはいくつか非互換の変更点があります。本章ではそのうち主要なものを説明します。

5.1. COPY の EOF マーカーの動作変更

COPY コマンドおよび psql の \copy コマンドでデータをテーブルに読み込むときに、EOF マーカー（ファイル終端子）を意味する「\.」（フォントによっては「¥.」、ASCII コードで 5c 2e）の扱いが変更されました。

PostgreSQL 18 から、CSV 形式の入力ファイルを読むときに「\.」を EOF マーカーとしては扱わなくなりました。CSV データをファイルではなく STDIN から読み込む場合、また、CSV 形式ではなく TEXT 形式の場合には、引き続き EOF マーカーとして扱われます。

以下のように動作の違いを確認しました。

（最後に EOF マーカー行のある CSV ファイルを作成）

```
$ cat t51.csv
aaa
bbb
ccc
\.
```

（18beta1 でテーブルに CSV 形式として読み込みすると 4 行が読み込まれてしまう）

```
$ psql db1
db1=# CREATE TABLE t51 (dat text);
db1=# \copy t51 FROM t51.csv WITH (FORMAT CSV)
COPY 4
db1=# SELECT * FROM t51;
 dat
-----
 aaa
 bbb
 ccc
 \.
(4 rows)
```

```
（17.5 で同じことを行うと以下の結果 - 最初の 3 行だけ読み込まれる）
db1=# \copy t51 FROM t51.csv WITH (FORMAT CSV)
COPY 3
db1=# SELECT * FROM t51;
 dat
-----
 aaa
 bbb
 ccc
(3 rows)
```

加えて、PostgreSQL 18 からは、入力元がファイルであれ STDIN であれ、また、CSV 形式であれ TEXT 形式であれ、「\.」は必ず単独行として書かなければエラーが出るようになりました。これまでもそのようにドキュメント記載されていましたが、一部で許容されていました。

```
（17.5 の奇妙な動作 - 最初の 2 行が読み込まれる結果になる）
db1=# \copy t51 FROM STDIN
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself, or an EOF signal.
>> ddd
>> eee\.
>> fff
>> \.
COPY 2    ← 18.x バージョンではエラーになる
```

5.2. タイムゾーン省略形の動作変更

日付時刻データ型の入力を読み込むにあたりタイムゾーン省略形（時間帯省略形、「JST」や「GMT」など）の扱いが変更されました。これまでは、使用可能なタイムゾーン省略形は timezone_abbreviations 設定と、その設定値に紐づく share/timezonesets/ 以下にある時間帯省略形ファイル によって規定されていました。これからは、IANA 時間帯データベース（share/timezone/ 以下のファイルで定義される）に基づくタイムゾーン省略形も追加して処理されるようになります。同じ省略形であれば IANA 時間帯データベースに基づく定義が優先されます。

17.x 以前のこれまでの振る舞いでは、以下のような問題がありました。

```
（17.5 で実行）
db1=# SET timezone TO 'Asia/Tokyo';
db1=# SET datestyle to 'Postgres';     ← タイムゾーン省略形を含む出力形式にする
db1=# SELECT '1000-01-01'::timestamptz::text;
             text
------------------------------
 Wed Jan 01 00:00:00 1000 LMT  ← 西暦 1000 年には JST は無いので LMT になる
(1 row)

db1=# SELECT '1000-01-01'::timestamptz::text::timestamptz;
ERROR:  invalid input syntax for type timestamp with time zone: "Wed Jan 01
00:00:00 1000 LMT"  ← 再度 timestamptz 型にキャストすると LMT を扱えずエラーになる

（18beta2 で実行した場合、ここで以下のように動作して、エラーにならない）
db1=# SELECT '1000-01-01'::timestamptz::text::timestamptz;
         timestamptz
------------------------------
 Wed Jan 01 00:00:00 1000 LMT
(1 row)
```

本手順では、出力にタイムゾーン省略形が含まれる出力形式「Postgres」を指定して、タイムゾーン指定の無い古い年代のタイムスタンプを与えています。タイムゾーン省略形が地域と年代に応じて IANA 時間帯データベースに基づいて選択されます。西暦 1000 年は JST（日本標準時）が有効な期間ではないため、 LMT（地方平時）が選択されました。ところがこれは、時間帯省略形ファイルに基づく入力処理では扱えないため、エラーが発生しました。

現在の timezone 設定値において有効なタイムゾーン省略形は、 timezone_abbreviations ビューで確認することができます。

```
（18beta2 で実行）
db1=# SET timezone TO 'Asia/Tokyo';
db1=# SELECT * FROM pg_timezone_abbrevs WHERE abbrev = 'LMT';
 abbrev | utc_offset | is_dst
--------+------------+--------
 LMT    | 09:18:59   | f
(1 row)
```

```
（timezone 設定値によって LMT の utc_offset は変化する）
db1=# SET timezone TO 'Europe/London';
db1=# SELECT * FROM pg_timezone_abbrevs WHERE abbrev = 'LMT';
 abbrev | utc_offset | is_dst
--------+------------+--------
 LMT    | -00:01:15  | f
(1 row)

（17.5 で実行すると 0 行となる）
db1=# SELECT * FROM pg_timezone_abbrevs WHERE abbrev = 'LMT';
 abbrev | utc_offset | is_dst
--------+------------+--------
(0 rows)
```

5.3. パーティション／継承テーブルに対する *VACUUM/ANALYZE*

パーティションテーブルおよび継承テーブルに対する VACUUM、ANALYZE の振る舞いが変更されました。また、VACUUM、ANALYZE の構文に ONLY オプションが加わりました。

◆ *パーティションテーブルに対する変更*

autovacuum はパーティションテーブル（親テーブル）に対して自動 ANALYZE を行ないません。そのため、コマンドによる ANALYZE を行うことになりますが、これまでの動作では、そのときに必ず属するパーティション（子テーブル）に対しても ANALYZE 処理が行なわれました。パーティション数が多い場合には余計な負荷になるため、好ましい動作ではありません。

PostgreSQL18 から、新たな ONLY オプションを指定することで、パーティションテーブル単独で ANALYZE を実行できるようになりました。

パーティションテーブルに対する VACUUM にも ONLY を指定できるようになりましたが、これは以下のように何もしないという動作になります。

```
（パーティションテーブルに ONLY 指定で VACUUM を実行）
db1=# CREATE TABLE t53 (id int primary key, v text) PARTITION BY RANGE (id);
db1=# CREATE TABLE t53p1 PARTITION OF t53 FOR VALUES FROM (0) TO (100);
db1=# CREATE TABLE t53p2 PARTITION OF t53 FOR VALUES FROM (100) TO (200);
db1=# INSERT INTO t53 SELECT g, md5(g::text) FROM generate_series(0, 199) g;
```

```
db1=# VACUUM (VERBOSE) ONLY t53;
WARNING:  VACUUM ONLY of partitioned table "t53" has no effect
```

ONLY 指定をしないパーティションテーブルに対する VACUUM は、これまで通り、子テーブル全てが処理対象となります。

◆ *継承テーブルに対する変更*

これまで継承親テーブルに対する VACUUM と ANALYZE は、子テーブルに対しては処理が行なわれませんでした。

PostgreSQL18 から、VACUUM、ANALYZE とも、デフォルトで継承子テーブルに対しても処理が行なわれるようになります。新たなオプション ONLY を指定すると、従来通りの指定した親テーブルだけに対する処理になります。

5.4. pg_backend_memory_contexts ビューの定義変更

現在のセッションに対応したバックエンドプロセスのメモリ使用状況をメモリコンテキストの一覧として報告させるシステムビュー pg_backend_memory_contexts の列定義が以下の通り変更されました。

列名	変更内容
parent	廃止されました。 これまでは上位メモリコンテキストの名前（name 列）が示されました。
level	値の意味が変わりました。 これまでは最上位コンテキストを 0 としていましたが、PostgreSQL 18 からは、最上位コンテキストが 1 になりました。下位メモリコンテキストに行くに従い 1 ずつ増えるのは同じです。
path	追加されました。 整数の配列で、最上位メモリコンテキストから（左端）、当レコードのメモリコンテキスト（右端）まで、親子関係にあるコンテキストを列挙します。

以下の PostgreSQL 18beta1 での出力例を示します。

```
db1=# \x on
db1=# SELECT * FROM pg_backend_memory_contexts;
```

```
-[ RECORD 1 ]-+-----------------------------------------------
name          | TopMemoryContext
ident         |
type          | AllocSet
level         | 1
path          | {1}
total_bytes   | 99456
total_nblocks | 5
free_bytes    | 5824
free_chunks   | 12
used_bytes    | 93632
-[ RECORD 2 ]-+-----------------------------------------------
name          | Record information cache
ident         |
type          | AllocSet
level         | 2
path          | {1,2}
total_bytes   | 8192
total_nblocks | 1
free_bytes    | 1640
free_chunks   | 0
used_bytes    | 6552
-[ RECORD 3 ]-+-----------------------------------------------
name          | Btree proof lookup cache
ident         |
type          | AllocSet
level         | 2
path          | {1,3}
total_bytes   | 8192
  : 《後略》
```

path 列の配列の数値は WHERE 句指定しないときのビュー出力の行出力順です。

WHERE 句を伴う問い合わせで path 列で示される行を特定するには、以下のようにウィンドウ関数の row_number() OVER() を組み合わせて、出力結果に行番号を含める方法が考えられます。

```
db1=# SELECT * FROM (SELECT row_number() OVER (), * FROM
        pg_backend_memory_contexts) s WHERE level = 1 OR level = 6;
-[ RECORD 1 ]-+------------------------------
row_number    | 1
name          | TopMemoryContext
ident         |
type          | AllocSet
level         | 1
path          | {1}
total_bytes   | 99456
total_nblocks | 5
free_bytes    | 5304
free_chunks   | 21
used_bytes    | 94152
-[ RECORD 2 ]-+------------------------------
row_number    | 147
name          | pg_get_backend_memory_contexts
ident         |
type          | AllocSet
level         | 6
path          | {1,20,30,136,141,147}
total_bytes   | 16384
total_nblocks | 2
free_bytes    | 5664
free_chunks   | 3
used_bytes    | 10720
```

5.5. *UNLOGGED* パーティションテーブル禁止

UNLOGGED のパーティションテーブルが禁止されました。

これまで定義可能でしたが、意味のある機能を提供していませんでした。具体的には、ALTER TABLE … SET LOGGED/UNLOGGED コマンドを行なっても効果が無い、UNLOGGED のパーティションテーブルを作っても、パーティション（子テーブル）は自動では UNLOGGED にならない、といった振る舞いでした。

PostgreSQL 18 では、UNLOGGED のパーティションテーブルを定義したり、パーティションテーブルを UNLOGGED に替えるときに、以下のようにエラーを出して失敗するようになりました。また、禁止であるこ

とがドキュメントに明記されました。

```
（5.3. 節で作ったパーティションテーブルの UNLOGGED 化を試みる — エラーになる）
db1=# ALTER TABLE t53 SET UNLOGGED;
ERROR:  ALTER action SET UNLOGGED cannot be performed on relation "t53"
DETAIL:  This operation is not supported for partitioned tables.
```

このため、以前の UNLOGGED パーティションテーブル定義を含むダンプファイルをリストアしたり、以前のバージョンの pgbench で --unlogged オプションと --partitions=N オプションを一緒に使う場合には、非互換のために失敗します。

なお、個々のパーティション（子テーブル）を UNLOGGED にすることは、これまで通り可能です。

5.6. *md5* パスワード認証が非推奨に

PostgreSQL 18 から md5 パスワード認証が非推奨になりました。

パスワード暗号化を md5 方式にしてユーザにパスワードを設定しようとすると、以下のように警告が出力されます。

```
db1=# SET password_encryption TO md5;
db1=# CREATE USER bar PASSWORD 'pass';
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.
CREATE ROLE
```

今のところ警告が出る以外には、md5 パスワード認証の使用に制限はありません。

警告メッセージは、新たに追加された設定パラメータ md5_password_warnings （デフォルト on）を off に設定することで抑止できます。

5.7. *RULE* 権限の完全廃止

GRANT/REVOKE 文で指定する RULE 権限が本バージョンで完全に廃止されました。

RULE 権限は PostgreSQL 8.2 で廃止されましたが、17.x まで以下のような実行が可能でした（実行して

も何ら効果はありません）。PostgreSQL 18 からはエラーになります。

```
db1=# GRANT RULE ON t51 TO PUBLIC;
GRANT
```

テーブルに RULE を付与できるのは、引き続き、所有者とスーパーユーザのみです。

5.8. 外部キー制約列の照合順序

外部キー制約の参照元・参照先の列の照合順序について、以下いずれかを満たすことが必須となりました。

- 両方とも決定論的照合順序を使っている（同一の照合順序でなくても良い）
- 両方とも同一の非決定論的照合順序を使っている

以下にエラー例を示します。

```
（非決定論的照合順序を 2 つ作成 - 2 つの文字を同じとみなすルール付き照合順序）
db1=# CREATE COLLATION col_my_rule1 (provider = icu, locale = 'und',
        deterministic = false, rules = '& 尭 = 堯');
db1=# CREATE COLLATION col_my_rule2 (provider = icu, locale = 'und',
        deterministic = false, rules = '& 嶌 = 鷥');

（外部キー制約でつながる 2 つのテーブルを作成）
db1=# CREATE TABLE t5mst (mstid text COLLATE col_my_rule1 PRIMARY KEY);
db1=# CREATE TABLE t5tx (txid int PRIMARY KEY,
        mstid text COLLATE col_my_rule2 REFERENCES t5mst(mstid));
ERROR:  foreign key constraint "t5tx_id_fkey" cannot be implemented
DETAIL:  Key columns "mstid" of the referencing table and "mstid" of the
referenced table have incompatible collations: "col_my_rule2" and
"col_my_rule1".  If either collation is nondeterministic, then both
collations have to be the same.
    → 照合順序に互換性が無いと言われてエラーになる
```

このような定義がある場合、旧バージョンからのダンプをリストアする場合や、pg_upgrade で失敗します。定義を修正したうえで、PostgreSQL 18 に移行する必要があります。

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	2025/8/15	初期公開バージョン