PostgreSQL14 検証レポート

1.2 版
2022 年 3 月 3 日

SRA OSS, Inc. 日本支社
〒170-0022 東京都豊島区南池袋 2-32-8
Tel. 03-5979-2701 Fax. 03-5979-2702
http://www.sraoss.co.jp/

目次

1. はじめに

本文書は PostgreSQL 14 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 14 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。
2021 年 5 月 20 日にリリースされた PostgreSQL 14 beta1 を使用して検証を行い、その後、2021 年 6 月 24 日にリリースされた 14 beta2 での変更内容を反映して、本文書を作成しています。

2. 概要

PostgreSQL 14 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：

- Btree インデックス肥大化抑止
- 式の拡張統計情報
- LZ4 によるデータ圧縮
- postgres_fdw の拡張
- ロジカルレプリケーション性能改善

SQL 新機能：

- マルチ範囲型
- 入れ子オブジェクトの添え字構文
- SEARCH 句／CYCLE 句
- GROUP BY 句の DISTINCT
- date_bin 関数
- ストアドプロシージャの OUT 引数

運用管理：

- アイドルセッションの検出と強制切断
- システムロール pg_read_all_data／pg_write_all_data
- 実行時統計ビューの追加と拡張
- pg_amcheck
- pg_rewind 機能追加

この他にも、機能追加や変更が多数あります。全ての変更点の一覧については PostgreSQL 14 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/14/release-14.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 14（ベータ版を含む）は以下 URL のページからダウンロード可能です。ソースコード 、Windows むけバイナリのインストーラ、RPM yum リポジトリが用意されています。

https://www.postgresql.org/download

3.2. 検証環境

検証環境として、仮想化基盤上の CentOS 7.5 (x86_64) の仮想マシンを使用しました。

本検証は具体的な特定マシン上の性能の提示や大規模サーバにおける性能の検証は意図していません。性能を検証する場合も、旧バージョンや新機能を使わない場合との比較を行い、相対的な違いを示すことを目的としました。

3.3. インストール

gcc、zlib-devel、readline-devel、lz4-devel の各パッケージがあらかじめインストールされている状態で、以下のオプションにてソースコードのビルドを行いました。事前に postgres ユーザで読み書き可能な /usr/local/pgsql/14 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/v14beta2/postgresql-
14beta2.tar.bz2          ※実際は１行

$ tar jxf postgresql-14beta2.tar.bz2
$ cd postgresql-14beta2
$ ./configure --prefix=/usr/local/pgsql/14 --enable-debug --with-lz4
$ make world
$ make install-world
```

環境変数を設定するファイルを書き出して、適用します。

```
$ cat > ~/pg14.env <<'EOF'
VER=14
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/14
EOF
$ . ~/pg14.env
```

データベースクラスタディレクトリを /var/lib/pgsql/data ディレクトリ以下に作成します。もし、このディレクトリが無ければ、root ユーザで作成して、postgres ユーザ所有者としておきます。

```
# mkdir -p /var/lib/pgsql/data
# chown postgres.postgres /var/lib/pgsql/data
```

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 (14beta2)
Type "help" for help.

db1=#
```

4. 主要な追加機能

主要な追加機能、性能向上について動作確認をしていきます。また、併せて機能の簡単な説明もします。

各追加機能の詳細な説明は同梱されるマニュアルに記載されています。本インストール手順を行った場合、以下の場所（インストール先の share/doc/html）に HTML のマニュアルが生成されます。

```
/usr/local/pgsql/14/share/doc/html/
```

また、以下 URL にて PostgreSQL 14 のドキュメントが公開されています。いずれも英語となります。

```
http://www.postgresql.org/docs/14/static/
```

4.1. 性能向上のための新機能

4.1.1. Btree インデックスの肥大化抑止

今回の改良では、UPDATE で更新されたインデックスに、削除可能を示すヒントを与える様になり、そのヒント情報からインデックスリーフの削除が可能になりました。インデックス内で不要になったリーフ情報を削除することでページ分割の必要性を減らし、インデックスの肥大化を避けることができます。

MVCC の仕組みでは行を UPDATE すると、内部的にバージョンの異なる複数の行が生成されます。HOT (heap only tuple) の仕組みが適用できればインデックスへの更新は行われませんが、HOT が適用できなければインデックスに同じインデックス値を持つ行情報が追加されます。このようなインデックスに行情報を追加する際に、古い行バージョンに対してヒント情報を付与するようになりました。

◆ ***検証***

以下のようにテーブル定義とデータ登録を行い、pgbench を使ってデータ更新を行います。

更新する列（abalance）と、更新とは無関係の列（filler）に対するインデックスを作成して、更新とは無関係の列のインデックス（idx_a_filler）について、そのファイルサイズの変化を確認しました。

（検証用のテーブル定義を作成）

```
db1=# CREATE TABLE pgbench_accounts (
        aid bigint, bid bigint, abalance bigint,
        filler text DEFAULT md5(random()::text));
db1=# CREATE UNIQUE INDEX idx_a_aid ON pgbench_accounts(aid);
db1=# CREATE INDEX idx_a_abalance ON pgbench_accounts(abalance);
```

```
db1=# CREATE INDEX idx_a_filler ON pgbench_accounts(filler);

（10 万件のデータを投入）
db1=# INSERT INTO pgbench_accounts
        SELECT id, 0, 0 FROM generate_series(1, 100000) as id;
```

以下の SQL でインデックス pgb_a_filler のサイズを確認しました。

```
db1=# SELECT pg_size_pretty(pg_relation_size('idx_a_filler')) AS size;
```

以下の pgbench 用の更新 SQL スクリプトを用意して、

```
$ cat > update.sql <<EOS
\set aid random(1, 100000 * :scale)
\set delta random(-5000, 5000)
UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;
EOS
```

以下のように pgbench で 10 並列 × 10 万回 の実行をしました。

```
$ pgbench -n -c 10 -t 100000 -f update.sql db1
```

ベンチマークの最中に autovacuum が実行されて不要領域の回収が発生すれば、インデックスページも再利用が行われるため、肥大化した時のサイズは小さくなります。そのため、autovacuum = off の設定を与えた状態でも実施しました。いずれの場合も、pgbench 実行前に インデックス idx_a_filler を REINDEX INDEX コマンドで初期化しています。

バージョン 13 とバージョン 14 で同じ処理を行った時のインデックスサイズは以下表の通りです。

状況	PG13	PG14
REINDEX 直後	5.6 MB	5.6 MB
pgbench 実行後（ autovacuum=on ）	11 MB	11 MB
pgbench 実行後（ autovacuum=off ）	22 MB	11 MB

autovacuum=on にした場合は、バージョン 13 と 14 で差があらわれませんでした。ここでは平均値を端数を省略した MB 単位で記載していますが、実際には結果にバラつきがありました。これは自動 VACUUM の実

行頻度によってインデックスの肥大化する程度が変わるためです。

autovacuum=off にした場合、バージョン 14 の優越性があらわれました。バージョン 14 では VACUUM をしていなくても肥大化が発生せず、autovacuum=on にしていた時と同じ程度のサイズ膨張に留まっています。

◆ ***考察***

インデックスの値に影響を与えない更新では HOT の仕組みが適用できれば、インデックスには変更が発生しません。しかし列の値の変化で、インデックス値が変化する場合は HOT が使えません。
その場合は、変更した列に影響しないインデックスも、同じ論理インデックス値に、複数の行情報が追加されます。

テーブルに複数のインデックスが定義されていて、そのうちのいずれかのインデックス値に影響する列が更新されると、その他のインデックスも更新の対象となり、この様な状況が発生しやすくなります。

UPDATE 以外に INSERT, DELETE も並行して行われた場合など、実際の状況により影響は変わりますが、複数のインデックスを定義している環境では、効果があると考えられます。

4.1.2. 拡張統計を式に適用

CREATE STATISTICS は PostgreSQL 10 で導入された仕組みです。複数列の値の関係性を使って、行数の見積りをより正確に行います。これまでは単純な列の値しか指定できませんでした。

PostgreSQL 14 では拡張統計情報を定義する際に、式が使えるようになりました。

以下の通り、動作確認を実施しました。

（検証用のテーブルとデータを作成、ANALYZE も実行）

```
db1=# CREATE TABLE t_exstat (a int);
db1=# INSERT INTO t_exstat SELECT generate_series(1, 1000);
db1=# ANALYZE t_exstat;
```

（拡張統計情報がない場合： WHERE 句に式を記述）

```
db1=# EXPLAIN (ANALYZE, TIMING OFF)
        SELECT * FROM t_exstat WHERE mod(a, 10) = 0 AND mod(a, 20) = 0;
                          QUERY PLAN
-------------------------------------------------------------------------------
 Seq Scan on t_exstat
     (cost=0.00..25.00 rows=1 width=4) (actual rows=50 loops=1)
   Filter: ((mod(a, 10) = 0) AND (mod(a, 20) = 0))
   Rows Removed by Filter: 950
```

```
..後略

（拡張統計情報がない場合： GROUP BY で式を記述）
=# EXPLAIN (ANALYZE, TIMING OFF)
     SELECT 1 FROM t_exstat GROUP BY mod(a,10), mod(a,20);
                                QUERY PLAN
--------------------------------------------------------------------------
 HashAggregate  (cost=25.00..40.00 rows=1000 width=12) (actual rows=20 loops=1)
   Group Key: mod(a, 10), mod(a, 20)
..後略
```

本例では拡張統計情報がない場合、予測行数が実際の件数とかけ離れています。WHERE や GROUP BY の条件に該当する式を拡張統計情報に使うと、より正確な行数見積りが行われます。

```
（式に対する拡張統計情報を作成する）
db1=# CREATE STATISTICS s ON mod(a,10), mod(a,20) FROM t_exstat;
db1=# ANALYZE t_exstat;

（前と同じクエリ）
db1=# EXPLAIN (ANALYZE, TIMING OFF)
        SELECT * FROM t_exstat WHERE mod(a,10) = 0 AND mod(a,20) = 0;
                           QUERY PLAN
--------------------------------------------------------------------------
 Seq Scan on t_exstat
    (cost=0.00..25.00 rows=50 width=4) (actual rows=50 loops=1)
   Filter: ((mod(a, 10) = 0) AND (mod(a, 20) = 0))
   Rows Removed by Filter: 950
..後略

db1=# EXPLAIN (ANALYZE, TIMING OFF)
        SELECT 1 FROM t_exstat GROUP BY mod(a,10), mod(a,20);
                                QUERY PLAN
--------------------------------------------------------------------------
 HashAggregate  (cost=25.00..25.30 rows=20 width=12) (actual rows=20
loops=1)
   Group Key: mod(a, 10), mod(a, 20)
```

```
..後略
```

式を使った拡張統計情報は pg_stats_ext_exprs ビューで参照できます。
また、psql の\dX で拡張統計が表示できるようになりました。以下のように表示されます。

```
db1=# \dX
                            List of extended statistics
 Schema | Name |               Definition               | Ndistinct |
 Dependencies |    MCV
--------+------+----------------------------------------+-----------+
--------------+---------
 public | s    | mod(a, 10), mod(a, 20) FROM t_exstat | defined   |
   defined    | defined
(1 row)
```

4.1.3. LZ4 による列の TOAST 圧縮

TOAST データの圧縮に LZ4 圧縮アルゴリズムが使えるようになりました。

圧縮方式は列ごとに設定できます。CREATE TABLE や ALTER TABLE で、 pglz（従来の圧縮方式）か lz4 を設定できます。 CREATE TABLE で圧縮方式を指定しなかった場合は、新しい GUC パラメータ default_toast_compression の設定が使われます。デフォルトは従来方式の pglz です。

LZ4 を利用するためには PostgreSQL をビルドする際に lz4 ライブラリを用意して、configure オプションに `--with-lz4` を指定する必要があります。`--with-lz4` が有効になっていない実行バイナリでも、データベースクラスタ自体は互換性があるので起動できますが、lz4 圧縮された行データをアクセスするときにエラーになります。

以下に lz4 を使用する指定の例を示します。

```
（デフォルトは従来の圧縮方式）
db1=# SHOW default_toast_compression;
 default_toast_compression
---------------------------
 pglz
(1 行)

（lz4 圧縮を指定した列を含むテーブルを作成し、\d+ で確認）
db1=# CREATE TABLE t_compress_test (
```

```
        id int PRIMARY KEY, filename TEXT, content TEXT COMPRESSION lz4);

db1=# \d+ t_compress_test
                                        Table "public.t_compress_test"
  Column  |  Type   | Collation | Nullable | Default | Storage  | Compression |
 Stats target | Description
----------+---------+-----------+----------+---------+----------+-------------+
-------------+-------------
 id       | integer |           | not null |         | plain    |             |
             |
 filename | text    |           |          |         | extended |             |
             |
 content  | text    |           |          |         | extended | lz4         |
             |
Indexes:
    "t_compress_test_pkey" PRIMARY KEY, btree (id)
Access method: heap
```

以下のようなALTER TABLEで列の圧縮方式を変更しても、既存のデータは変換されません。列の値が更新されれば、新しい圧縮方式で保存されます。

```
ALTER TABLE 《テーブル名》ALTER COLUMN 《列名》 SET COMPRESSION lz4;
```

単純な列値の代入では最適化処理が働き、圧縮データのままコピーが行われるケースもあります。その様な場合も圧縮方式は変化しません。以下に、そのような場合の例を示します。

```
（pglz圧縮のテーブルとlz4圧縮のテーブルを作成して、データ投入）
db1=# CREATE TABLE t_src (id INT, content TEXT COMPRESSION pglz);
db1=# CREATE TABLE t_dest (id INT, content TEXT COMPRESSION lz4);
db1=# INSERT INTO t_src VALUES (1, 'text 1' || repeat(' long', 1000));
db1=# INSERT INTO t_dest VALUES (2, 'text 2' || repeat(' long', 1000));
db1=# INSERT INTO t_dest VALUES (3, 'text 3' || repeat(' long', 1000));

（各行の content 列の圧縮方式を確認）
db1=# SELECT id, pg_column_compression(content) FROM t_src;
 id | pg_column_compression
```

```
----+-----------------------
  1 | pglz
(1 行)

db1=# SELECT id, pg_column_compression(content) FROM t_dest;
 id | pg_column_compression
----+-----------------------
  2 | lz4
  3 | lz4
(2 行)

（サブクエリを使った UPDATE 文で列値を更新する）
db1=# UPDATE t_dest SET content = (
        SELECT content FROM t_src WHERE id = 1) WHERE id = 3;

=# SELECT id, pg_column_compression(content) FROM t_dest;
 id | pg_column_compression
----+-----------------------
  2 | lz4
  3 | pglz                          ← t_src の pglz のままコピーされる
(2 行)
```

psql の \d+ で圧縮方式が確認できます。以下の psql 設定で、従来通りの出力形式にすることも可能です 。psql の接続先サーバがバージョン 13 以下の場合も同じ出力になります。

```
db1=# \set HIDE_TOAST_COMPRESSION on
```

関連して、pg_dump コマンドに --no-toast-compression オプションが追加されました。本オプションを指定すると、ダンプ情報に圧縮方式を出力しません。

◆ *pglz と lz4 の性能比較*

実際に格納されているデータの圧縮方式は pg_column_compression()関数で確認できます 。pg_column_size() 関数を使うと圧縮されたデータサイズが確認できます。

pglz と lz4 を比較した場合、アルゴリズムの知られている特徴から、データの圧縮率は pglz の方が良く、圧縮展開の処理速度は lz4 が速いと予想されます。

実験として英文のテキストデータ（SGML ドキュメント）を投入して、記録上のサイズを比較しました。

```
（テスト用テーブル作成）
db1=# CREATE TABLE t_rate_test (
        id int primary key,
        c_pglz text COMPRESSION pglz, c_lz4 text COMPRESSION lz4);

（c_pglz, c_lz4 に同じテキストデータを投入：
   PostgreSQL 付属ドキュメントの sgml ファイルの 1 ファイル を 1 データとして 160 行投入
   投入手順は省略）

（pglz と lz4 の圧縮率を比較）
db1=# SELECT
        (s_pglz::double precision / s_plain)::numeric(5,2) AS rate_pglz,
        (s_lz4::double precision /  s_plain)::numeric(5,2) AS rate_lz4
        FROM (SELECT sum(length(c_pglz)) AS s_plain,
                     sum(pg_column_size(c_pglz)) AS s_pglz,
                     sum(pg_column_size(c_lz4)) AS s_lz4 FROM t_rate_test
              ) AS t_sum;
 rate_pglz | rate_lz4
-----------+----------
      0.31 |     0.34
(1 行)
```

オリジナルの文字数（＝バイト数、英文であるため）に対していずれも 3 割程度のサイズに圧縮されており、pglz の方がより小さく圧縮されている結果となりました。

圧縮データの参照速度の比較として、pgbench を利用し以下の SELECT クエリを繰り返し実行した時の所要時間を計測しました。

```
SELECT sum(length(content)) FROM t_decompress;
```

対象テーブルには、先ほど同様に PostgreSQL ドキュメントの sgml ファイルのデータを格納しています 。shared_buffers のサイズ内に収まるサイズであって、バッファ上に載せてからテストを行いました。また SQL の最終結果は長さを集計するだけでデータ転送処理の影響も少ない条件としました。

実際のアプリケーションから利用する場合は、データ内容や実行させるクエリ内容によって異なるので、

あくまでも参考程度ですが、lz4 の方が若干速い結果になりました。

```
$ time pgbench -n -c 1 -t 100 -f select_sum_pglz.sql
...
real    0m12.664s

$ time pgbench -n -c 1 -t 100 -f select_sum_lz4.sql
...
real    0m11.199s
```

文書テキストデータで実験した結果、それほど大きな差異はあらわれませんでしたが、サイズと処理速度について、傾向としては想定通りの結果が得られました。

4.1.4. postgres_fdw の拡張

postgres_fdw が拡張されて以下の動作が可能となりました。

- 外部テーブルの TRUNCATE
- 外部テーブルの非同期スキャン
- 外部テーブルへの一括挿入

外部データラッパ（Foreign data wrapper、FDW）の機能としてこれらが追加され、外部テーブルラッパの具体的実装の一つである postgres_fdw でも対応されました。

TRUNCATE はデフォルトで利用可能になります。外部サーバまたは外部テーブルのオプションとして 、truncatable オプションが追加されていて、デフォルトは true（有効）です。外部テーブルに対して TRUNCATE コマンドが実行できて、リモート先でも TRUNCATE が実行されます。

非同期スキャンを利用するには外部サーバまたは外部テーブルの定義で async_capable オプションの指定が必要になります。デフォルトは false（無効）です。非同期スキャンを有効にすることで、実行プランにおける Append 処理の下位でのスキャンを並列に実行することが可能になります。これは外部テーブルによるパーティションテーブルに対する問い合わせでよく現れる実行プラン形状です。

一括挿入を利用するには、外部サーバまたは外部テーブルの定義で batch_size オプションを指定します。デフォルト値は 1 で一括処理を行わないことを意味します。 INSERT INTO … SELECT 構文などで複数行をインサートする場合、従来は 1 行ずつインサート文が送られていたため、大量の行を挿入すると処理が非常に遅くなっていました。この動作を改善することができます。

以下にこれら機能を実際に動作させる例を示します。

```
（postgres_fdw の動作確認用にデータベース dbr とテーブル t_fdwtest を作成）
$ createdb dbr
$ psql dbr
dbr=# CREATE TABLE t_fdwtest (id int, val text);
dbr=# \q

（最初に postgres_fdw 拡張を導入）
$ psql db1
db1=# CREATE EXTENSION postgres_fdw;

（非同期スキャンを有効にするオプション指定で外部サーバ作成）
db1=# CREATE SERVER fs_test FOREIGN DATA WRAPPER postgres_fdw
        OPTIONS (host 'localhost', dbname 'dbr', async_capable 'true');

（外部テーブル利用にはユーザマッピング定義が必要）
db1=# CREATE USER MAPPING FOR postgres SERVER fs_test
        OPTIONS (user 'postgres');

（一括挿入のオプションを指定した外部テーブル作成）
db1=# CREATE FOREIGN TABLE ft_fdwtest_a (id int, val text)
        SERVER fs_test OPTIONS (schema_name 'public',
        table_name 't_fdwtest', batch_size '1');
db1=# CREATE FOREIGN TABLE ft_fdwtest_b (id int, val text)
        SERVER fs_test OPTIONS (schema_name 'public',
        table_name 't_fdwtest', batch_size '10');

（非同期スキャンを有効にした効果を確認）
db1=# EXPLAIN (COSTS off)
        SELECT * FROM ft_fdwtest_a UNION ALL SELECT * FROM ft_fdwtest_b;
              QUERY PLAN
----------------------------------------
 Append
   ->  Async Foreign Scan on ft_fdwtest_a
   ->  Async Foreign Scan on ft_fdwtest_b
(3 rows)          ※ 2 つの Async Foreign Scan は同時並行に実行されます
```

```
（TRUNCATE も実行可能）
db1=# TRUNCATE ft_fdwtest_a, ft_fdwtest_b;
        ※ 従来は「... is not a table」という ERROR になりました
```

一括挿入動作の違いは、リモートサーバ側で postgresql.conf に log_statement='mod' を設定して、リモート側で実行された更新 SQL をログ記録することで確認できます。ログ記録を取るように設定したうえで以下の手順を実行しました。

```
（複数行データ挿入の SQL 実行）
db1=# INSERT INTO ft_fdwtest_a
        SELECT g, md5(g::text) FROM generate_series(1, 100) g;
db1=# INSERT INTO ft_fdwtest_b
        SELECT g, md5(g::text) FROM generate_series(1, 100) g;
```

ログ出力を確認します。batch_size=1 の場合、INSERT 文が繰り返されています。

```
（ログ出力： 従来または batch_size=1 の場合 ／ タイムスタンプを省略した抜粋）
LOG:  execute pgsql_fdw_prep_1: INSERT INTO public.t_fdwtest(id, val) VALUES
($1, $2)
DETAIL:  parameters: $1 = '1', $2 = ...
LOG:  execute pgsql_fdw_prep_1: INSERT INTO public.t_fdwtest(id, val) VALUES
($1, $2)
DETAIL:  parameters: $1 = '2', $2 = ...
LOG:  execute pgsql_fdw_prep_1: INSERT INTO public.t_fdwtest(id, val) VALUES
($1, $2)
DETAIL:  parameters: $1 = '3', $2 = ...
```

batch_size=10 とした場合、一括挿入によりインサート文に複数のパラメータが列挙されて送られます。インサート文の実行回数が減り、処理が高速化されます。

```
（ログ出力： batch_size=10 の場合 ／ タイムスタンプを省略した抜粋）
LOG:  execute pgsql_fdw_prep_1: INSERT INTO public.t_fdwtest(id, val) VALUES
($1, $2), ($3, $4), ($5, $6), ($7, $8), ($9, $10), ($11, $12), ($13, $14),
($15, $16), ($17, $18), ($19, $20)
DETAIL:  parameters: $1 = '1', $2 = ..., $3 = '2', $4 = ..., $5 = '3',$6
= ..., $7 = '4', $8 = ..., $9 = '5', $10 = ..., $11 = '6', $12 = ..., ...
```

```
LOG:  execute pgsql_fdw_prep_1: INSERT INTO public.t_fdwtest (id, val)
VALUES ($1, $2), ($3, $4), ($5, $6), ($7, $8), ($9, $10), ($11, $12), ($13,
$14), ($15, $16), ($17, $18), ($19, $20)
DETAIL:  parameters: $1 = '11', $2 = ..., $3 = '12', $4 = ..., $5 = '13',$6
= ..., $7 = '14', $8 = ..., $9 = '15', $10 = ..., $11 = '16', $12 = ..., ...
```

他にも postgres_fdw に関する改良が行われています。

外部サーバとの接続処理が改善されており、外部サーバとの接続が切れていたら自動的に再接続を行うようになりました。

以前から、外部サーバとの接続は維持される仕組みになっていました。しかし、外部サーバ側が再起動した場合など、接続が切れる場合があります。従来は、外部サーバとの接続が切れた状態で、外部テーブルを参照するクエリを実行するとクライアントにエラーが返ってきました。クライアント側で再度 SQL を実行すれば、外部サーバとの再接続が行われて、処理は行われますが、そのためのエラー対応が必要となっていました。今回の改良により、外部サーバとの接続が切れていてもエラーにならず、自動的に再接続して処理が行われます。

関連して接続状態を維持するかを外部サーバのオプション keep_connections （デフォルト on）で設定できるようになりました。keep_connections を off にすれば、トランザクションが終了した時点で、外部サーバとの接続を切ります。これにより外部サーバ側のコネクション数の消費を減らすことができます。

postgres_fdw に関する接続状態を管理する以下の関数も追加されています。

関数	説明
postgres_fdw_get_connections()	現在の接続状態を確認する。
postgres_fdw_disconnect('外部サーバ名')	外部サーバとの接続を切る
postgres_fdw_disconnect_all()	全ての外部サーバとの接続を切る

postgres_fdw_disconnect、postgres_fdw_disconnect_all は、keep_connections が on であるときに、使い終った接続を早期に解放したい時に有用です。

4.1.5. ロジカルレプリケーションの改善

PostgreSQL 14 ではロジカルレプリケーションについて様々な改善が適用されています。本節では、主要な改善点についての検証を報告します。

検証のために、以下のように PostgreSQL インスタンスを同ホストのポート 5433 に立てて、データベース db2 を作り、これをロジカルレプリケーションの宛先として構成しました。

```
（postgresql.conf を修正して wal_level を logical にする）
$ vi $PGDATA/postgresql.conf
   wal_level = logical
$ pg_ctl restart

（5433 ポートに別の PostgreSQL を立てて、データベース db2 を作成）
$ pg_basebackup -D "${PGDATA}_sub"
$ pg_ctl start -D "${PGDATA}_sub" -o '-c port=5433'
$ createdb -p 5433 db2

（テスト用テーブルを両データベースに作成）
$ psql db1
db1=# CREATE TABLE t_log (id bigint primary key, ts timestamp, mes text);
db1=# \q
$ psql -p 5433 db2
db2=# CREATE TABLE t_log (id bigint primary key, ts timestamp, mes text);
db2=# \q
```

◆ ***ストリーム送信***

これまでロジカルレプリケーションの変更の送信はコミットを契機に行われていました。そのため、大きな変更内容を伴うトランザクションでは、以下のようにパブリケーション側のレプリケーションスロットのディレクトリに spill ファイルが書き出されました。

```
（ロジカルレプリケーションを構成）
db1=# CREATE PUBLICATION pub1 FOR TABLE t_log;

db2=# CREATE SUBSCRIPTION sub1 CONNECTION
       'port=5432 user=postgres dbname=db1' PUBLICATION pub1;

（一つのトランザクション内で大量データ投入）
db1=# BEGIN;
db1=*# INSERT INTO t_log SELECT g, now(), md5(g::text)
         FROM generate_series(1, 500000) as g;
db1=*# ^Z
[1]+  停止                 psql -p 5432 db1
```

```
（spill ファイルが生成されている）
$ ls -l data/13.3/pg_replslot/sub1/
合計 57632
-rw-------. 1 postgres postgres      184  7月 15 14:14 state
-rw-------. 1 postgres postgres 14588545  7月 15 14:15 xid-496-lsn-0-10000000.spill
-rw-------. 1 postgres postgres  8090790  7月 15 14:14 xid-496-lsn-0-D000000.spill
-rw-------. 1 postgres postgres 18100400  7月 15 14:14 xid-496-lsn-0-E000000.spill
-rw-------. 1 postgres postgres 16098330  7月 15 14:14 xid-496-lsn-0-F000000.spill

$ fg %1
db1=# ROLLBACK;
```

そのうえで、トランザクションがコミットされた時点で初めて、サブスクリプション側に送出されました。spill ファイルを使う動作はしばしば同期の遅延の原因となっていました。

本バージョンから、サブスクリプションのオプションに streaming = on を指定することで、コミットを待たずに順次にデータを送り出す、ストリーム送信を選択することができます。

```
（サブスクリプションを streaming = on で再作成）
db2=# DROP SUBSCRIPTION sub1;
db2=# CREATE SUBSCRIPTION sub1 CONNECTION
       'port=5432 user=postgres dbname=db1' PUBLICATION pub1
        WITH (streaming = on);

（前回同様に 1 トランザクション中に大量 INSERT）
db1=# BEGIN;
db1=*# INSERT INTO t_log SELECT g, now(), md5(g::text)
         FROM generate_series(1, 500000) as g;
db1=*# ^Z
[2]+  停止                 psql -p 5432 db1

（spill ファイルは出力されない、サブスクリプション側に changes ファイルができる）
$ ls -l $PGDATA/pg_replslot/sub1/
合計 4
-rw-------. 1 postgres postgres 200  7月 15 14:52 state
```

```
$ ls -l "${PGDATA}_sub"/base/pgsql_tmp/*/
合計 33216
-rw-------. 1 postgres postgres 34012187  7月 15 18:35 24627-2689.changes.0

$ fg %2
db1=# ROLLBACK;
```

上記のように同様のデータ投入のレプリケーションにおいて、 パブリケーション側に spill ファイルが生成されない動作が確認できました。コミット前の変更データはサブスクリプション側の pgsql_tmp ディレクトリ下のサブディレクトリ内にある changes ファイルとして書き出されています。

◆ *初期同期時の更新を複数トランザクション化*

PostgreSQL14 から、ロジカルレプリケーションの初期同期の振る舞いが変わりました。

これまでは、テーブルデータの初期コピーと、初期コピーしている間に生じた更新の適用が、単一のトランザクションとして扱われていました。このため、初期コピー中に生じた更新の適用が何らか理由で失敗すると、初期コピー処理についてもやり直しが必要となりました。また、初期コピーに時間を要した結果、その間に実行された更新処理も多く溜まってくると、初期同期処理が更に長い単一トランザクションとなり、長期トランザクションに伴う様々な問題を引き起こします。

これからは、初期コピーが終わった時点でサブスクリプション側のトランザクションが完了し、この時点でデータが参照可能になります。その後に、初期コピー中の更新が順次適用されます。

以下のように動作確認を行いました。

```
（t_log テーブルに大量行がある状態にします）
db1=# TRUNCATE t_log;
db1=# INSERT INTO t_log SELECT g, now(), md5(g::text)
        FROM generate_series(1, 500000) as g;

（サブスクリプションを作り直します）
db2=# DROP SUBSCRIPTION sub1;
db2=# CREATE SUBSCRIPTION sub1 CONNECTION
      'port=5432 user=postgres dbname=db1' PUBLICATION pub1;

（パブリケーション側でその後すぐに pgbench を使って先頭 3 行を 100 回更新）
$ cat > update.sql <<EOS
 UPDATE t_log SET mes = 'updated ' || now() WHERE id <= 3;
```

```
EOS
$ pgbench -p 5432 -f update.sql -n -c 1 -t 100 db1

（サブスクリプション側で繰り返しテーブルを参照）
db2=# SELECT * FROM t_log WHERE id <= 3;
 id | ts | mes
----+----+-----
(0 rows)

db2=# \watch 1              ← \watch は直前コマンドを指定秒数毎に繰り返し実行します
 {中略}
 id |             ts             |               mes
----+----------------------------+----------------------------------
  1 | 2021-07-15 16:29:42.031395 | c4ca4238a0b923820dcc509a6f75849b
  2 | 2021-07-15 16:29:42.031395 | c81e728d9d4c2f636f067f89cc14862c
  3 | 2021-07-15 16:29:42.031395 | eccbc87e4b5ce2fe28308fd9f2a7baf3
(3 rows)
 {中略}
 id |             ts             |                 mes
----+----------------------------+-------------------------------------
  1 | 2021-07-15 16:29:42.031395 | updated 2021-07-15 16:30:36.807872+09
  2 | 2021-07-15 16:29:42.031395 | updated 2021-07-15 16:30:36.807872+09
  3 | 2021-07-15 16:29:42.031395 | updated 2021-07-15 16:30:36.807872+09
(3 rows)
```

初期同期が完了した時点からデータが見える挙動になっていることが確認できました。PostgreSQL 13 では、初期同期の間に行われた UPDATE も一つのトランザクションとしてサブスクリプション側に適用されますので、サブスクリプション側で UPDATE 前の初期データが参照されることはありません。この確認手順は、タイミングに依存しますので、うまく初期データを参照できないかもしれません。

◆ ***バイナリ転送モード***

ロジカルレプリケーションのデータ転送でバイナリ転送を選択できるようになりました。 CREATE PUBLICATION のオプションで binary=on を指定するとバイナリ転送モードになります。バイナリ転送であると効率的に処理できるデータ型としては、timestamp 型や bytea 型があります。

以下のように、テスト用のテーブルに bytea 型に変えたうえで、転送の所要時間をサブスクリプション側のトリガを使って記録する試験を行いました。

```
（レプリケーションを解除、テーブルを空にして、列を bytea 型に変更、
  さらに所要時間記録用に timestamp 型の列 ts2 を追加します）
db2=# DROP SUBSCRIPTION sub1;
db2=# TRUNCATE t_log;
db2=# ALTER TABLE t_log ALTER mes TYPE bytea USING mes::bytea;
db2=# ALTER TABLE t_log ADD ts2 timestamp;

db1=# DROP PUBLICATION pub1;
db1=# TRUNCATE t_log;
db1=# ALTER TABLE t_log ALTER mes TYPE bytea USING mes::bytea;
db1=# ALTER TABLE t_log ADD ts2 timestamp;
```

更に以下のようにサブスクリプション側の t_log テーブルにトリガを設置します。ロジカルレプリケーションでは文単位トリガに対応していないため、行単位トリガを使用します。しかし、全行にタイムスタンプを記録すると処理負荷が高いため、ここでは 10000 行に 1 行の割合で記録するものとします。記録された最も遅いタイムスタンプをもってレプリケーションが完了した時刻の代替とします。

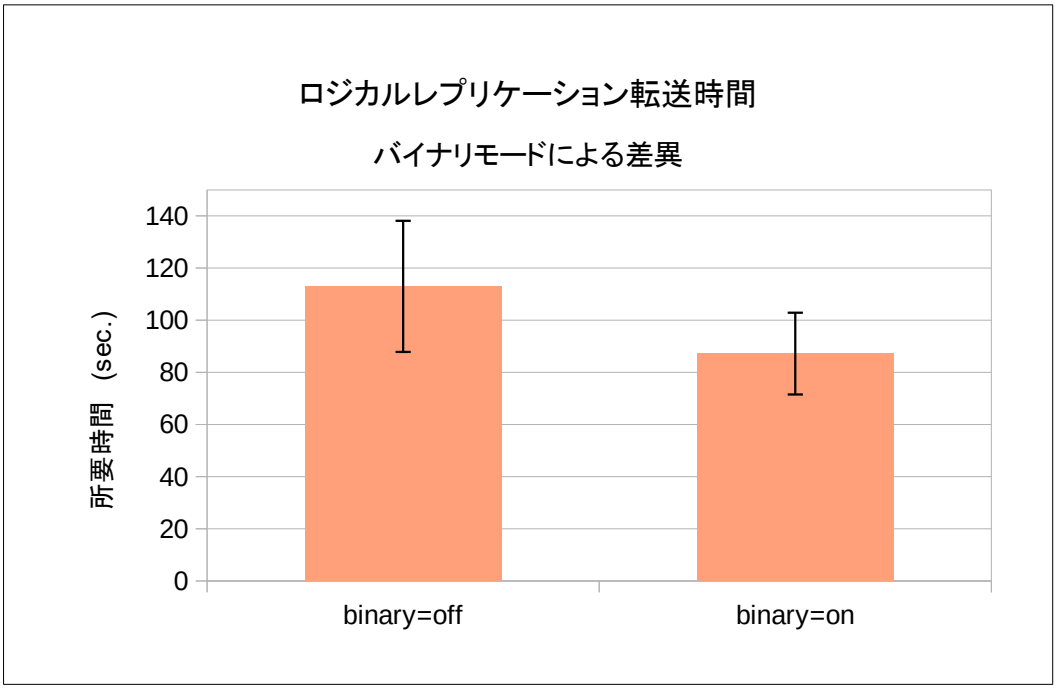
```
（追加した ts2 列に現在時刻を記入するトリガ関数を作成します）
db2=# CREATE FUNCTION trgf_t_log() RETURNS trigger LANGUAGE plpgsql AS
       $$ BEGIN NEW.ts2 := clock_timestamp(); RETURN NEW; END; $$;

（id が 10000 で割り切れるときだけ動作するようにトリガを設定します）
db2=# CREATE TRIGGER trg_t_log BEFORE INSERT ON t_log FOR EACH ROW
        WHEN (NEW.id % 10000 = 0) EXECUTE FUNCTION trgf_t_log();

（レプリケーションを受けるテーブルでトリガを動作させるには以下も必要です）
db2=# ALTER TABLE t_log ENABLE REPLICA TRIGGER trg_t_log;
```

そのうえで、以下のコマンドで 50 万行のロジカルレプリケーションを行って、所要時間を計測しました。

```
（レプリケーションを開始した後、50 万行を投入、binary = on の場合）
$ psql -p 5432 db1 -c "CREATE PUBLICATION pub1 FOR TABLE t_log"
  psql -p 5433 db2 <<EOS
  CREATE SUBSCRIPTION sub1 CONNECTION 'port=5432 user=postgres dbname=db1'
    PUBLICATION pub1 WITH (binary = on, streaming = on);
EOS
  psql -p 5432 db1 <<EOS
  INSERT INTO t_log SELECT g, CURRENT_TIMESTAMP, md5(g::text)::bytea
    FROM generate_series(1, 500000) as g;
EOS

（投入時タイムスタンプとトリガによるタイムスタンプの差分から所要時間を計測）
$ psql -p 5433 db2 <<EOS
    SELECT max(ts), max(ts2), max(ts2) - max(ts) AS duration FROM t_log
      WHERE id % 10000 = 0;
EOS
             max              |             max              |     duration
------------------------------+------------------------------+-----------------
 2021-07-20 16:36:43.258337 | 2021-07-20 16:38:26.140801 | 00:01:42.882464
(1 row)

（再度の計測をするにはレプリケーションを解除して、データを削除する）
$ psql -p 5433 db2 -c 'DROP SUBSCRIPTION sub1;'
  psql -p 5433 db2 -c 'TRUNCATE t_log;'
  psql -p 5432 db1 -c 'DROP PUBLICATION pub1;'
  psql -p 5432 db1 -c 'TRUNCATE t_log;'
```

binary=on と binary=off（デフォルト）との場合で実施した結果を以下グラフに示します。所要時間にはっきりと差異があらわれました。I 字型の細い線は標準偏差です。binary=on 指定により速度が 3 割程度向上しました。

なお、データの初期コピー、すなわち、CREATE PUBLICATION、CREATE SUBSCRIPTION を行った時点で初めから存在していた行データのレプリケーションにおいては、バイナリ指定で所要時間に違いは観測できませんでした。



4.1.6. パラレルクエリ対応の拡張

PostgreSQL 14 ではパラレルクエリについて、いくつか拡張されています。本節ではこれらについて動作確認を行います。

◆ *パラレルシーケンシャルスキャンの改善*

パラレルシーケンシャルスキャンが改善されました。これまでパラレルワーカが 1 ブロックごとに処理していたものを連続した複数ブロックを処理するように変更されました。これはストレージアクセスの効率化を意図しています。

以下のように 100 万行のデータに並列処理を強制する設定を与えたうえでインデックスの効かない検索を実行しました。

（テスト用のテーブル作成とデータ投入）

```
db1=# CREATE TABLE t_message (id int PRIMARY KEY, mes text);
db1=# INSERT INTO t_message
        SELECT g, md5(g::text) FROM generate_series(1, 1000000) g;
```

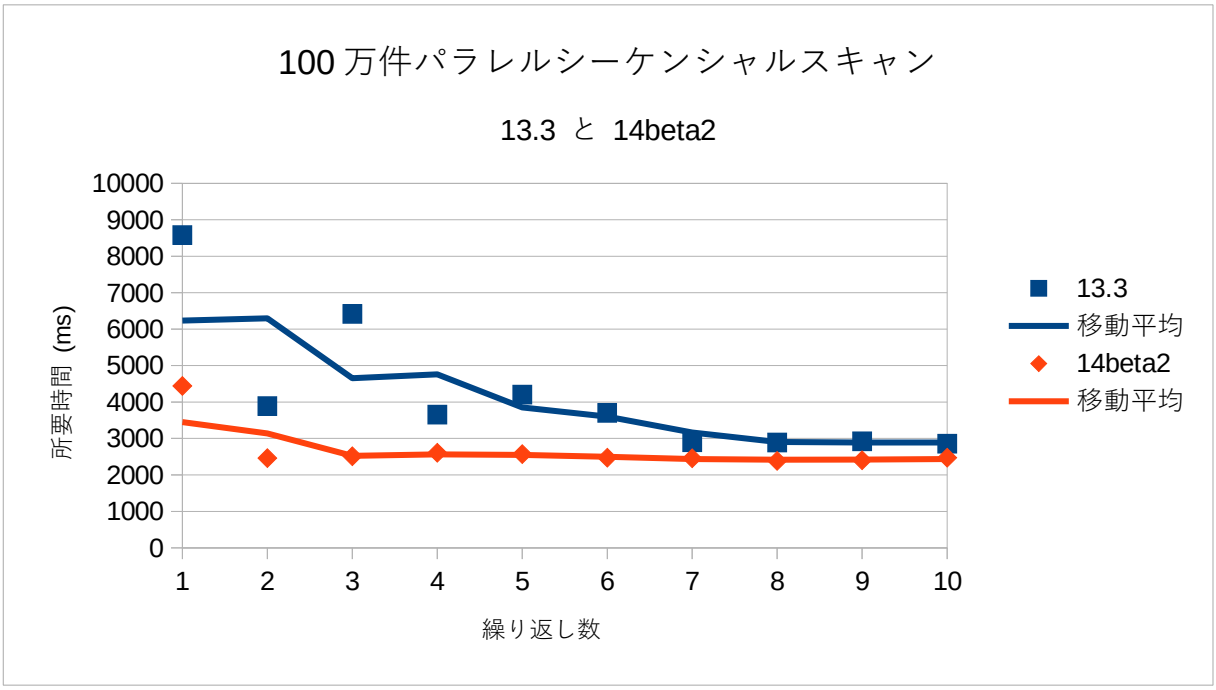
（強制で並列処理を使わせる設定）

```
db1=# SET force_parallel_mode TO true;
```

```
（EXPLAIN ANALYZE で並列動作を確認しつつ、所要時間を採取）
db1=# EXPLAIN (ANALYZE) SELECT * FROM t_message WHERE mes ~ 'abcde';
                                  QUERY PLAN
------------------------------------------------------------------------------
 Gather  (cost=1000.00..14770.18 rows=104 width=37)
         (actual time=72.185..4423.834 rows=29 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   ->  Parallel Seq Scan on t_message
         (cost=0.00..13759.78 rows=43 width=37)
         (actual time=89.611..2953.304 rows=10 loops=3)
       Filter: (mes ~ 'abcde'::text)
       Rows Removed by Filter: 333324
 Planning Time: 49.900 ms
 Execution Time: 4440.519 ms
(8 rows)

（これを 10 回実行）
db1=# EXPLAIN (ANALYZE) SELECT * FROM t_message WHERE mes ~ 'abcde';
...後略
```

同様に PostgreSQL 13.3 でも実施して結果を比較した結果を以下グラフに示します。プロットは測定値、折れ線は移動平均を示しています。

データ投入は最初の 1 回だけで、その後、同じ SELECT 文の実行を 10 回繰り返します。回数が進むとバッファにヒットすることが増えて所要時間が減っていきますが、14 バージョンの方が一貫して所要時間が短く、特にバッファヒットが悪い最初の時点で 2 倍ほどの性能差になっています。ストレージ I/O アクセスが生じる際に高速化するという、本改善の意図通りの結果になっていると言えます。



◆ *PL/pgSQL の「RETURN QUERY」対応*

手続き言語 PL/pgSQL における「RETURN QUERY 《問い合わせ文》」および「RETURN QUERY EXECUTE《問い合わせ文の文字列》」にて、並列処理が使われるようになりました。これまでは必ずカーソルを使って実行されていたため、並列処理にはなりませんでした。

以下の手順で、前項の検証で使用した t_message テーブルを検索する PL/pgSQL 関数を使って動作を確認しました。

```
（auto_explain でログメッセージから実行プランを確認できるようにする）
$ vi $PGDATA/postgresql.conf
  （以下を設定）
shared_preload_libraries = 'auto_explain'
auto_explain.log_min_duration = 0
auto_explain.log_nested_statements = on
（再起動して反映）
$ pg_ctl restart

（PL/pgSQL 関数を作成し、並列実行プランを選択させる設定を付与）
$ psql db1
db1=# CREATE FUNCTION f_message_pat(pat text) RETURNS SETOF text
        LANGUAGE plpgsql AS $$
```

```
          BEGIN RETURN QUERY SELECT mes FROM t_message WHERE mes ~ pat;
          END; $$;
db1=# ALTER FUNCTION f_message_pat(text) SET force_parallel_mode TO on;

（実行して、auto_explainによるログから並列実行を確認）
db1=# SELECT * FROM f_message_pat('abcde');
        f_message_pat
----------------------------------
 b1db5ee8a6bccccb7e72c7951abcde46
 aa37c29c8617abcde038d5e0ec298828
..後略

db1=# \q
$ cat $PGDATA/log/* | tail -100
..前略
2021-07-25 13:50:13.147 JST [17896] LOG:  duration: 2666.636 ms  plan:
       Query Text: SELECT mes FROM t_message WHERE mes ~ pat
       Gather  (cost=1000.00..14552.33 rows=100 width=33)
         Workers Planned: 2
         ->  Parallel Seq Scan on t_message  (cost=0.00..13542.33 rows=42
width=33)
               Filter: (mes ~ 'abcde'::text)
2021-07-25 13:50:13.147 JST [17896] CONTEXT:  SQL statement "SELECT mes FROM
t_message WHERE mes ~ pat"
       PL/pgSQL function f_message_pat(text) line 1 at RETURN QUERY
```

PL/pgSQL 内の RETURN QUERY の実行で並列処理が行われていることが確認できました。同じ手順を PostgreSQL 13 で実行すると、並列の実行プランにはなりません。

◆ *REFRESH MATERIALIZED VIEW 対応*

REFRESH MATERIALIZED VIEW コマンドで、並列問い合わせ実行に対応しました。

以下のように、前項の検証で作成した関数と auto_explain の設定を使って、マテリアライズドビューの動作を確認しました。

```
（並列実行プランを選択させる設定を含んだ関数を使ってマテリアライズドビューを作成）
db1=# CREATE MATERIALIZED VIEW mv_f_message AS
        SELECT * FROM f_message_pat('abcde');

（リフレッシュして auto_explain からその時の実行プランを確認）
db1=# REFRESH MATERIALIZED VIEW mv_f_message;
db1=# \q
$ cat $PGDATA/log/* | tail -100
..前略
2021-07-25 14:44:37.039 JST [18099] LOG:  duration: 2617.210 ms  plan:
       Query Text: SELECT mes FROM t_message WHERE mes ~ pat
       Gather  (cost=1000.00..14552.33 rows=100 width=33)
         Workers Planned: 2
         ->  Parallel Seq Scan on t_message  (cost=0.00..13542.33 rows=42
width=33)
               Filter: (mes ~ 'abcde'::text)
2021-07-25 14:44:37.039 JST [18099] CONTEXT:  SQL statement "SELECT mes FROM
t_message WHERE mes ~ pat"
       PL/pgSQL function f_message_pat(text) line 1 at RETURN QUERY
2021-07-25 14:44:37.040 JST [18099] LOG:  duration: 2652.755 ms  plan:
       Query Text: REFRESH MATERIALIZED VIEW mv_f_message;
       Function Scan on f_message_pat  (cost=0.25..10.25 rows=1000 width=32)
```

上記のように auto_explain によるログメッセージに報告される実行プランから、REFRESH MATERIALIZED VIEW コマンドから起動される問い合わせが並列実行されていることが確認できました。

4.1.7. 多セッションにおけるオーバーヘッドの軽減

PostgreSQL14 は、特に CPU 数が多くセッション数が多いシステムにおいて MVCC 可視化スナップショットの計算速度が改善されています。MVCC 可視化スナップショットの計算はテーブルのデータを参照する全ての処理に関わってきますので、CPU がボトルネックとなっているときの性能改善が期待できます。

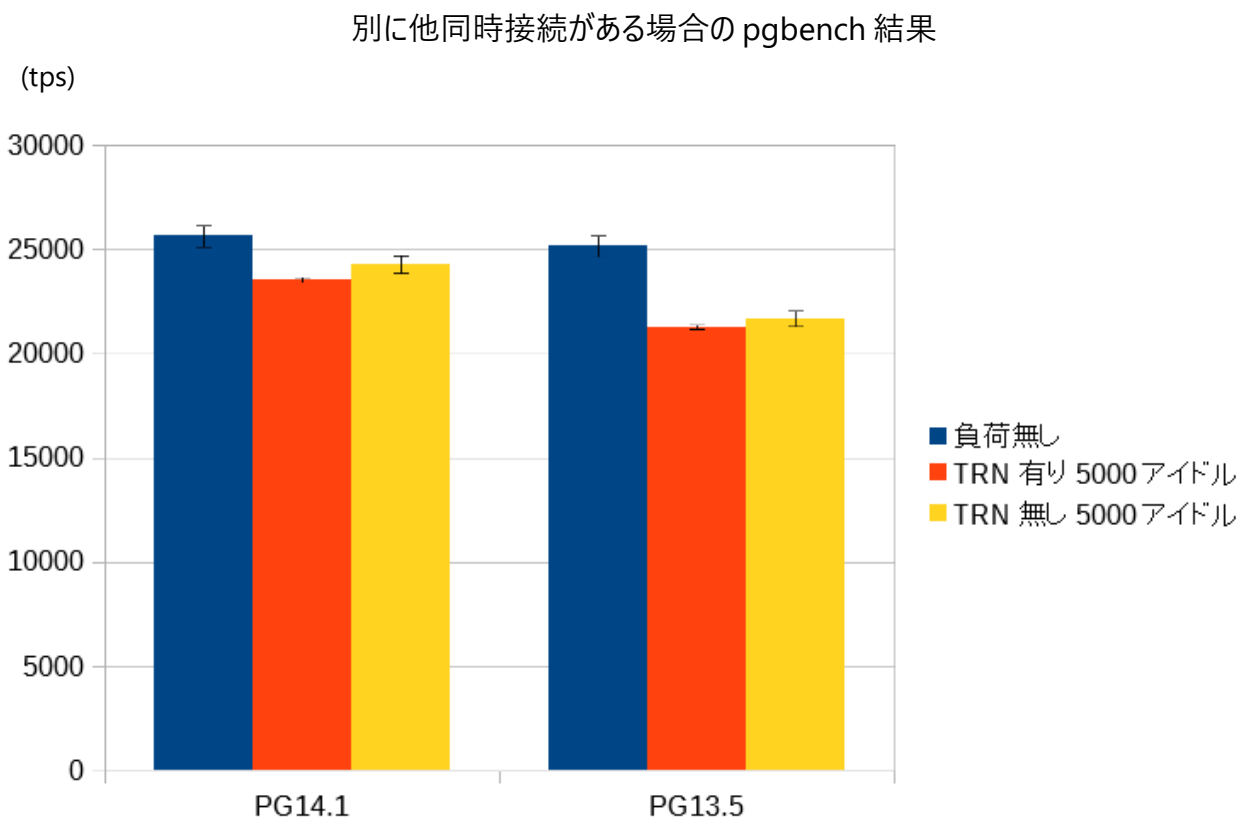
本検証では、PG14.1 と PG13.5 を使って、多セッションが存在する時の性能を比較する試験を行ないました。下記検証では、「追加の同時接続無し」、「追加で 5000 接続のトランザクション実行中のアイドルセッションあり」、「追加で 5000 接続のトランザクション未開始のアイドルセッションあり」のそれぞれの場合について pgbench に標準付属の参照ベンチマークを使って、PG14.1 と PG13.5 とで性能比較します。

以下にコマンド手順を示します。

```
（pgbench の各テーブルをスケールファクター 10 で初期化）
$ pgbench -i -s 10 db1
  （出力省略）

（追加セッション無しの pgbench 実行）
$ /usr/pgsql-13/bin/pgbench -n -c 10 -S -T 30 db1
transaction type: <builtin: select only>
scaling factor: 10
  （中略）
number of transactions actually processed: 738404
latency average = 0.406 ms
tps = 25418.871458 (including connections establishing)
tps = 25420.709976 (excluding connections establishing)

（トランザクション中アイドル状態を作るスクリプトをバックグラウンドで走らせて実行する）
$ cat ~/bench/pgb_sleep.sql
BEGIN;
SELECT * FROM pgbench_accounts LIMIT 1;
\sleep 300s
END;
$ /usr/pgsql-13/bin/pgbench -n -c 5000 -f ~/bench/pgb_sleep.sql db1 \
  &> sleep.log < /dev/null &
[1] 9237
$ /usr/pgsql-13/bin/pgbench -n -c 10 -S -T 30 db1
  （出力省略）

（アイドル状態のセッションを作るスクリプトをバックグラウンドで走らせて実行する）
$ cat ~/bench/pgb_sleep_NOTRN.sql
SELECT * FROM pgbench_accounts LIMIT 1;
\sleep 300s
$ /usr/pgsql-13/bin/pgbench -n -c 5000 -f ~/bench/pgb_sleep_NOTRN.sql db1 \
  &> sleep.log < /dev/null &
[1] 14630
$ /usr/pgsql-13/bin/pgbench -n -c 10 -S -T 30 db1
```

（出力省略）

上記の手順を PostgreSQL14.1 サーバと PostgreSQL13.5 サーバに対して、各条件で複数回実行しました。以下グラフに結果を示します。棒グラフの上にある線は標準偏差を示しています。



追加の同時接続無し（グラフでは「負荷無し」と記載）に対して、5000 個のアイドル状態の同時接続がある場合、5000 個のトランザクション実行中でアイドル状態の同時接続がある場合のいずれも、PostgreSQL14 の方が性能低下が小さい結果が確認できました。

なお、本改良の開発者からもベンチマーク結果が報告されていて、1000 以上のセッションが存在する時にオーバーヘッド軽減による性能が向上がはっきりあらわれています。詳細は以下ページを参照してください。

https://www.citusdata.com/blog/2020/10/25/improving-postgres-connection-scalability-snapshots/

4.2. データ型と SQL の新機能

4.2.1. マルチ範囲型

本バージョンからマルチ範囲型のデータ型がサポートされました。これらは従来の範囲型に類似しますが、重複しない複数の範囲を含むことができます。以下表に示すデータ型が組込みデータ型として提供されます。これらのデータ型の値を構成するためのデータ型名と同名のコンストラクタ関数も提供されます。

データ型	説明
int4multirange	integer 型のマルチ範囲型
int8multirange	bigint 型のマルチ範囲型
nummultirange	numeric 型のマルチ範囲型
tsmultirange	timestamp 型のマルチ範囲型
tstzmultirange	timestamp with time zone 型のマルチ範囲型
datemultirange	date 型のマルチ範囲型

以下のように実行を確認しました。

（マルチ範囲型の動作確認）

```
db1=# SELECT datemultirange(
        daterange('2021-07-01', '2021-07-02', '[]'),
        daterange('2021-07-05', '2021-07-09', '[]'),
        daterange('2021-07-12', '2021-07-16', '[]'),
        daterange('2021-07-19', '2021-07-21', '[]'),
        daterange('2021-07-26', '2021-07-30', '[]'));

                                  datemultirange
-------------------------------------------------------------------------------
 {[2021-07-01,2021-07-03),[2021-07-05,2021-07-10),[2021-07-12,2021-07-17),
[2021-07-19,2021-07-22),[2021-07-26,2021-07-31)}
(1 row)
```

上記の例は 2021 年 7 月の平日を一つのマルチ範囲型データで表現しています。マルチ範囲型のコンスト

ラクタ関数は、複数の範囲型を引数に取って動作します。また、マルチ範囲型の表現文字列は、範囲型の表現文字列を {} で括って、カンマ区切りでつなげたものとなります。

以下にマルチ範囲型を使った問い合わせ例を示します。

（指定した期間「7/16 から 7/17」が平日のみで構成されているかを判定）

```
db1=# SELECT datemultirange(
        daterange('2021-07-01', '2021-07-02', '[]'),
        daterange('2021-07-05', '2021-07-09', '[]'),
        daterange('2021-07-12', '2021-07-16', '[]'),
        daterange('2021-07-19', '2021-07-21', '[]'),
        daterange('2021-07-26', '2021-07-30', '[]') )
        @> daterange('2021-07-16', '2021-07-17', '[]');
 ?column?
----------
 f                                    ※ 範囲型で指定した期間の一部は休日に掛かっている
(1 row)
```

マルチ範囲型の列は、範囲型の列と同様に GiST インデックスを付加することができます。

（検証用に tsmultirange 型の列を含むテーブルを作成）

```
db1=# CREATE TABLE t_schedule
        (id int primary key, dmr tsmultirange, description text);
```

（ダミーデータを投入、1 日と翌週同じ曜日の 1 日から成るタイムスタンプのマルチ範囲を生成）

```
db1=# INSERT INTO t_schedule SELECT g, tsmultirange(
        tsrange(rd, rd + '1day', '[)'),
        tsrange(rd + '7day', rd + '8day','[)')),  'schedule ' || g
        FROM (SELECT g,  '2021-04-01'::timestamp +
          (floor(random()*365) || 'day')::interval rd
          FROM generate_series(1, 10000) g) v;

db1=# SELECT * FROM t_schedule LIMIT 5;
 id |                                        dmr
                                  | description
----+-----------------------------------------------------------------------
-----------------------------------+-------------
```

```
  1 | {["2022-03-23 00:00:00","2022-03-24 00:00:00"),["2022-03-30
00:00:00","2022-03-31 00:00:00")} | schedule 1
  2 | {["2021-04-20 00:00:00","2021-04-21 00:00:00"),["2021-04-27
00:00:00","2021-04-28 00:00:00")} | schedule 2
  3 | {["2021-04-17 00:00:00","2021-04-18 00:00:00"),["2021-04-24
00:00:00","2021-04-25 00:00:00")} | schedule 3
  4 | {["2022-01-24 00:00:00","2022-01-25 00:00:00"),["2022-01-31
00:00:00","2022-02-01 00:00:00")} | schedule 4
  5 | {["2021-05-27 00:00:00","2021-05-28 00:00:00"),["2021-06-03
00:00:00","2021-06-04 00:00:00")} | schedule 5
(5 rows)

（GiST インデックスを作成）
db1=# CREATE INDEX ON t_schedule USING gist (dmr);

（インデックスを使った実行プランが選択される）
db1=# explain SELECT * FROM t_schedule
        WHERE dmr && tsrange('2021-07-01', '2021-07-03', '[]');
                                QUERY PLAN
---------------------------------------------------------------------------
 Bitmap Heap Scan on t_schedule  (cost=10.16..137.41 rows=260 width=70)
   Recheck Cond:
   (dmr && '["2021-07-01 00:00:00","2021-07-03 00:00:00"]'::tsrange)
   ->  Bitmap Index Scan on t_schedule_dmr_idx
        (cost=0.00..10.10 rows=260 width=0)
          Index Cond:
          (dmr && '["2021-07-01 00:00:00","2021-07-03 00:00:00"]'::tsrange)
(4 rows)
```

10000 行のデータを持つテーブルのマルチ範囲型の列に対して、&&演算子で指定の範囲型が重なりを持つ行を取り出す問い合わせで、作成した GiST インデックスが使われることが確認できました。

4.2.2. 入れ子オブジェクトへの添え字構文

添え字構文が拡張モジュールとして新たに追加したデータ型においても利用可能になりました。組込み

（および標準付属拡張）のデータ型としては jsonb データ型、hstore データ型が添え字に対応しました。これまでは添え字を使った記述方法は配列データ型に限定されていました。

jsonb 型、hstore 型について以下の通り、動作を確認しました。

（jsonb 型の動作確認）

```
db1=# SELECT j FROM
       (VALUES ('{"1":{"A":100, "B":200}, "2":{"C":300}}'::jsonb)) AS v(j);
                    j
-----------------------------------------------
 {"1": {"A": 100, "B": 200}, "2": {"C": 300}}
(1 row)

db1=# SELECT j['1']['A'] FROM
       (VALUES ('{"1":{"A":100, "B":200}, "2":{"C":300}}'::jsonb)) AS v(j);
  j
-----
 100
(1 row)
```

（hstore 型の動作確認）

```
db1=# CREATE EXTENSION hstore;
db1=# SELECT * FROM (VALUES ('A=>100, B=>200'::hstore)) AS v(h);
          h
------------------------
 "A"=>"100", "B"=>"200"
(1 row)

db1=# SELECT h['B'] FROM (VALUES ('A=>100, B=>200'::hstore)) AS v(h);
  h
-----
 200
(1 row)
```

添え字を使ったアクセスが可能であることが確認できました。

なお、json 型については添え字の対応はありません。以下のようにエラーとなりました。

```
db1=# SELECT j['A'] FROM (VALUES ('{"A":100, "B":200}'::json)) AS v(j);
ERROR:  cannot subscript type json because it does not support subscripting
```

添え字に対応したデータ型を作るための API としては、CREATE TYPE コマンドに「SUBSCRIPT =《添え字処理関数》」というオプションが追加されています。

4.2.3. SEARCH 句／CYCLE 句

本バージョンから SQL 標準の SEARCH 句、CYCLE 句がサポートされました。

PostgreSQL では従来から CTE（Common Table Expressions、共通テーブル式）構文に対応していて、再帰的な問い合わせが可能でした。SEARCH 句、CYCLE 句は、再帰的な問い合わせに対して、探索順序の指定と循環参照の検出の機能を提供します。

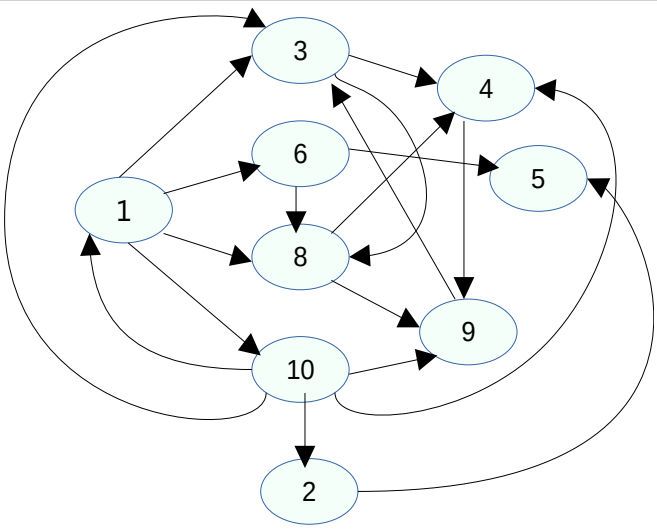
以下のサンプルテーブルで動作確認を行いました。何らか SNS サービスで、どのユーザがどのユーザを「フォロー」しているかを管理するテーブルを想定したものです。

```
（SNS のユーザをあらわすテーブル - ただし以下の検証手順では使用しません）
db1=# CREATE TABLE t_sns_user(id int primary key, name text);

（ユーザ 1 がユーザ 2 をフォローしていることを表すテーブル）
db1=# CREATE TABLE t_sns_follow(id1 int, id2 int, PRIMARY KEY (id1, id2));

（10 番までのユーザ間について、ランダムにフォロー関係のデータを最大 30 件登録）
db1=# INSERT INTO t_sns_follow SELECT ceil(random()*10), ceil(random()*10)
       FROM generate_series(1, 30) g ON CONFLICT DO NOTHING;

（自己フォローは除く）
db1=# DELETE FROM t_sns_follow WHERE id1 = id2;
```

まず、従来から利用可能な再帰問い合わせを確認します。

```
（id=1 からフォロー関係を辿る問い合わせ）
db1=# WITH RECURSIVE search_tree (id2, chain) AS (
      SELECT id2, ARRAY[id1,id2] FROM t_sns_follow f WHERE f.id1 = 1
      UNION ALL
      SELECT f.id2, st.chain || ARRAY[f.id2] FROM t_sns_follow f
        JOIN search_tree st ON (f.id1 = st.id2)
```

```
      ) SELECT chain AS follow_chain FROM search_tree LIMIT 20;
 follow_chain
--------------
 {1,10}
 {1,6}
 {1,8}
 {1,3}
 {1,3,4}
 {1,3,8}
 {1,6,5}
 {1,6,8}
 {1,8,4}
 {1,8,9}
 {1,10,1}          ← 1に戻っている
 {1,10,2}
 {1,10,3}
 {1,10,4}
 {1,10,9}
 {1,10,1,3}        ← 循環探索が生じている
 {1,10,1,6}
 {1,10,1,8}
 {1,10,1,10}
 {1,10,2,5}
(20 rows)
```



参考図：フォロー関係グラフ（一部のみ）

この問い合わせは、id=1 のユーザを起点に、フォローしているユーザ、そのユーザがフォローしているユーザ、というように再帰的に調べてフォローの連鎖を配列で返します。本問い合わせでは、循環探索が生じているため、LIMIT 句を付加しないと無限に出力が生じることになり問い合わせが終わりません。

値の意味を把握する助けとして、上記参考図で問い合わせ結果に現れている部分についてフォロー関係のグラフを示します。

◆ *SEARCH 句の使用*

本問い合わせに SEARCH 句を適用してみます。

SEARCH 句を使用すると、再帰問い合わせ結果を深さ優先探索の順、あるいは、幅優先探索の順で取り出すことができます。指定の列名でソート用に利用可能なデータを持った列が追加されるので、それをソートを

行う外側の問い合わせで指定します。

以下のように幅優先探索と深さ優先探索を実行しました。

（幅優先探索で結果を取り出す問い合わせ）

```
db1=# SELECT chain follow_chain, search_seq FROM (
        WITH RECURSIVE search_tree (id2, chain) AS (
          SELECT id2, ARRAY[id1,id2] FROM t_sns_follow f WHERE f.id1 = 1
          UNION ALL
          SELECT f.id2, st.chain || ARRAY[f.id2] FROM t_sns_follow f
            JOIN search_tree st ON (f.id1 = st.id2)
          ) SEARCH BREADTH FIRST BY id2 SET search_seq
        SELECT chain, search_seq FROM search_tree LIMIT 5000) v
        ORDER BY search_seq LIMIT 20;
 follow_chain | search_seq      ← search_seq は付加されたソート用の列
--------------+------------
 {1,3}        | (0,3)
 {1,6}        | (0,6)
 {1,8}        | (0,8)
 {1,10}       | (0,10)
 {1,10,1}     | (1,1)
 {1,10,2}     | (1,2)
 {1,10,3}     | (1,3)
 {1,3,4}      | (1,4)
 {1,10,4}     | (1,4)
 {1,8,4}      | (1,4)
 {1,6,5}      | (1,5)
 {1,6,8}      | (1,8)
 {1,3,8}      | (1,8)
 {1,10,9}     | (1,9)
 {1,8,9}      | (1,9)
 {1,8,9,3}    | (2,3)
 {1,10,1,3}   | (2,3)
 {1,10,9,3}   | (2,3)
 {1,6,5,4}    | (2,4)
 {1,10,3,4}   | (2,4)
(20 rows)
```

幅優先を指定した結果、要素数が少ないものが先に出現する順で結果が返っていることがわかります。指定したソート用列の search_seq は階層数と最後の値を要素に持つレコード型になっていることが分かります。再帰問い合わせ部分は LIMIT 句を付けないと無限行が返るものであるため、ここでは LIMIT 5000（適当な大きい行数）を付加して副問い合わせとして、その外側でソートを行っています。

（深さ優先探索で結果を取り出す問い合わせ）

```
db1=# SELECT chain follow_chain, search_seq FROM (
        WITH RECURSIVE search_tree (id2, chain) AS (
          SELECT id2, ARRAY[id1,id2] FROM t_sns_follow f WHERE f.id1 = 1
          UNION ALL
          SELECT f.id2, st.chain || ARRAY[f.id2] FROM t_sns_follow f
            JOIN search_tree st ON (f.id1 = st.id2)
          ) SEARCH DEPTH FIRST BY id2 SET search_seq
        SELECT chain, search_seq FROM search_tree LIMIT 5000) v
        ORDER BY search_seq LIMIT 20;
     follow_chain        |                  search_seq
-------------------------+-----------------------------------------------
 {1,3}                   | {(3)}
 {1,3,4}                 | {(3),(4)}
 {1,3,4,9}               | {(3),(4),(9)}
 {1,3,4,9,3}             | {(3),(4),(9),(3)}
 {1,3,4,9,3,4}           | {(3),(4),(9),(3),(4)}
 {1,3,4,9,3,4,9}         | {(3),(4),(9),(3),(4),(9)}
 {1,3,4,9,3,4,9,3}       | {(3),(4),(9),(3),(4),(9),(3)}
 {1,3,4,9,3,4,9,3,4}     | {(3),(4),(9),(3),(4),(9),(3),(4)}
 {1,3,4,9,3,4,9,3,4,9}   | {(3),(4),(9),(3),(4),(9),(3),(4),(9)}
 {1,3,4,9,3,4,9,3,4,9,3} | {(3),(4),(9),(3),(4),(9),(3),(4),(9),(3)}
 {1,3,4,9,3,4,9,3,4,9,3,4} | {(3),(4),(9),(3),(4),(9),(3),(4),(9),(3),(4)}
 {1,3,4,9,3,4,9,3,4,9,3,8} | {(3),(4),(9),(3),(4),(9),(3),(4),(9),(3),(8)}
 {1,3,4,9,3,4,9,3,8}     | {(3),(4),(9),(3),(4),(9),(3),(8)}
 {1,3,4,9,3,4,9,3,8,4}   | {(3),(4),(9),(3),(4),(9),(3),(8),(4)}
 {1,3,4,9,3,4,9,3,8,4,9} | {(3),(4),(9),(3),(4),(9),(3),(8),(4),(9)}
 {1,3,4,9,3,4,9,3,8,4,9,3} | {(3),(4),(9),(3),(4),(9),(3),(8),(4),(9),(3)}
 {1,3,4,9,3,4,9,3,8,9}   | {(3),(4),(9),(3),(4),(9),(3),(8),(9)}
```

```
 {1,3,4,9,3,4,9,3,8,9,3}    | {(3),(4),(9),(3),(4),(9),(3),(8),(9),(3)}
 {1,3,4,9,3,4,9,3,8,9,3,4}  | {(3),(4),(9),(3),(4),(9),(3),(8),(9),(3),(4)}
 {1,3,4,9,3,4,9,3,8,9,3,8}  | {(3),(4),(9),(3),(4),(9),(3),(8),(9),(3),(8)}
(20 rows)
```

深さ優先探索の場合には、循環探索の繰り返しを含め、フォローの連鎖を最後まで調べるのが優先される順で出現する結果になっていることがわかります。指定したソート用列の search_seq は探索経路の各要素に持つレコード型の配列になっています。内側の再帰問い合わせ部分に LIMIT 句を付けて有限行数の結果になるようにしているのは、幅優先探索の例と同様です。

深さ優先探索の場合、再帰問い合わせで生成した follow_chain 列の値とソート用列（search_seq）の値がほぼ同じですので、結果取得用にもソート用列を使いたいと思うかもしれません。しかしながら、レコード型の配列は使い勝手が悪い点を考慮する必要があります。上記の実行結果の search_seq の値から、先頭の数値 3 を取り出すには、以下の式が必要になります。

```
(row_to_json(search_seq[1])->>'id2')::int
```

◆ *CYCLE 句の使用*

SEARCH 句を使用する例では、再帰問い合わせをするときに循環探索があると無限の結果行が生じることになり、問い合わせを記述するうえで扱いにくくなりました。CYCLE 句を使用して、循環を検出して探索を打ち切る動作にすることができます。

以下の問い合わせで動作を確認しました。

（深さ優先探索で CYCLE 句を使用）

```
db1=# SELECT chain follow_chain, cyclemark, cyclepath FROM (
        WITH RECURSIVE search_tree (id2, chain) AS (
          SELECT id2, ARRAY[id1,id2] FROM t_sns_follow f WHERE f.id1 = 1
          UNION ALL
          SELECT f.id2, st.chain || ARRAY[f.id2] FROM t_sns_follow f
            JOIN search_tree st ON (f.id1 = st.id2)
          ) SEARCH DEPTH FIRST BY id2 SET search_seq
            CYCLE id2 SET cyclemark USING cyclepath
        SELECT chain, search_seq, cyclemark, cyclepath FROM search_tree) v
        ORDER BY search_seq;
  follow_chain    | cyclemark |           cyclepath
------------------+-----------+-------------------------------
 {1,3}            | f         | {(3)}
```

```
 {1,3,4}             | f          | {(3),(4)}
 {1,3,4,9}           | f          | {(3),(4),(9)}
 {1,3,4,9,3}         | t          | {(3),(4),(9),(3)}
 {1,3,8}             | f          | {(3),(8)}
 {1,3,8,4}           | f          | {(3),(8),(4)}
 {1,3,8,4,9}         | f          | {(3),(8),(4),(9)}
 {1,3,8,4,9,3}       | t          | {(3),(8),(4),(9),(3)}
 {1,3,8,9}           | f          | {(3),(8),(9)}
 {1,3,8,9,3}         | t          | {(3),(8),(9),(3)}
 {1,6}               | f          | {(6)}
 {1,6,5}             | f          | {(6),(5)}
 {1,6,5,4}           | f          | {(6),(5),(4)}
 {1,6,5,4,9}         | f          | {(6),(5),(4),(9)}
 {1,6,5,4,9,3}       | f          | {(6),(5),(4),(9),(3)}
 {1,6,5,4,9,3,4}     | t          | {(6),(5),(4),(9),(3),(4)}
 {1,6,5,4,9,3,8}     | f          | {(6),(5),(4),(9),(3),(8)}
 {1,6,5,4,9,3,8,4}   | t          | {(6),(5),(4),(9),(3),(8),(4)}
 {1,6,5,4,9,3,8,9}   | t          | {(6),(5),(4),(9),(3),(8),(9)}
 {1,6,5,10}          | f          | {(6),(5),(10)}
   ※以下の出力を省略
(175 rows)
```

上記の問い合わせでは LIMIT 句を使っていません。CYCLE 句によって循環探索が打ち切られるため 、LIMIT 句がなくとも無限の出力結果にはなりません。問い合わせ結果では、既出の番号に遭遇したところまではあらわれますが、それ以降は出力されません。具体例をあげますと、{1,3,4,9,3,4,9,3,4,9,...} という繰り返しは {1,3,8,4,9,3} までで打ち切られています。

CYCLE 句では指定の列名で、循環があるかを示すフラグ列（上記問い合わせでは cyclemark）と、循環を調べるときに使う値の経路を示す列（上記問い合わせでは cyclepath）が付加されます。cyclemark は循環を検出した行を結果に含めるかの WHERE 句の条件に利用できます。

幅優先探索でも CYCLE 句を適用可能です。

（幅優先探索で CYCLE 句を使用）

```
db1=# SELECT chain follow_chain, search_seq, cyclemark, cyclepath FROM (
        WITH RECURSIVE search_tree (id2, chain) AS (
          SELECT id2, ARRAY[id1,id2] FROM t_sns_follow f WHERE f.id1 = 1
          UNION ALL
```

```
           SELECT f.id2, st.chain || ARRAY[f.id2] FROM t_sns_follow f
             JOIN search_tree st ON (f.id1 = st.id2)
           ) SEARCH BREADTH FIRST BY id2 SET search_seq
             CYCLE id2 SET cyclemark TO true DEFAULT false USING cyclepath
         SELECT chain, search_seq, cyclemark, cyclepath FROM search_tree) v
         ORDER BY search_seq;
     follow_chain      | search_seq | cyclemark |            cyclepath
-----------------------+------------+-----------+-------------------------
 {1,3}                 | (0,3)      | f         | {(3)}
 {1,6}                 | (0,6)      | f         | {(6)}
 {1,8}                 | (0,8)      | f         | {(8)}
 {1,10}                | (0,10)     | f         | {(10)}
 {1,10,1}              | (1,1)      | f         | {(10),(1)}
 {1,10,2}              | (1,2)      | f         | {(10),(2)}
   ※以下の出力を省略
(175 rows)
```

幅優先探索の問い合わせでも、CYCLE 句を使うことで LIMIT 句を外すことができました。また、幅優先探索では、search_seq と cyclepath には異なる値が格納されていることがわかります。

4.2.4. GROUP BY 句の *DISTINCT* キーワード

本バージョンから ORDER BY 句に DISTINCT キーワードが指定可能になりました。

複数のグループ化セットを一緒に指定すると、グループ化セットの最終的なリストに重複が含まれてしまう場合があります。一般的には、ROLLUP 句や CUBE 句を複数指定したとき、重複したグループの組み合わせが生成されてしまいます。ORDER BY 句に DISTINCT キーワードを記述することで、重複するセットを削除することができます。

以下のように動作を確認しました。

```
（DISTINCT キーワードを使用しなかった場合）
db1=# WITH goods(id, product, color, size, stock) AS (
        VALUES (1, 'フリース', '赤', 'L', 150),
               (2, 'フリース', '青', 'S', 120),
               (3, 'フリース', '白', 'L', 200),
               (4, 'パーカー', '白', 'M', 300),
               (5, 'パーカー', '白', 'L', 250),
```

```
                (6, 'パーカー', '赤', 'S', 100))
      SELECT product, color, size, sum(stock) AS stock_sum FROM goods
        GROUP BY ROLLUP(1, 2), ROLLUP(1, 3) ORDER BY product, color, size;
 product  | color | size | stock_sum
----------+-------+------+-----------
パーカー | 白    | L    |       250
パーカー | 白    | M    |       300
パーカー | 白    |      |       550
パーカー | 白    |      |       550
パーカー | 赤    | S    |       100
パーカー | 赤    |      |       100
パーカー | 赤    |      |       100
パーカー |       | L    |       250
パーカー |       | L    |       250
パーカー |       | M    |       300
パーカー |       | M    |       300
パーカー |       | S    |       100
パーカー |       | S    |       100
パーカー |       |      |       650
パーカー |       |      |       650
パーカー |       |      |       650
フリース | 白    | L    |       200
フリース | 白    |      |       200
フリース | 白    |      |       200
フリース | 赤    | L    |       150
フリース | 赤    |      |       150
フリース | 赤    |      |       150
フリース | 青    | S    |       120
フリース | 青    |      |       120
フリース | 青    |      |       120
フリース |       | L    |       350
フリース |       | L    |       350
フリース |       | S    |       120
フリース |       | S    |       120
フリース |       |      |       470
```

※DISTINCT を指定しない場合
GROUP BY ALL と解釈されて全て出力

※ ROLLUP 句を複数指定したことで、
重複した行が多数出力されている。

```
 フリース |      |      |       470
 フリース |      |      |       470
          |      |      |      1120
(33 rows)
```

（GROUP BY にDISTINCT を指定した場合）

```
db1=# WITH goods(id, product, color, size, stock) AS (
        VALUES (1, 'フリース', '赤', 'L', 150),
               (2, 'フリース', '青', 'S', 120),
               (3, 'フリース', '白', 'L', 200),
               (4, 'パーカー', '白', 'M', 300),
               (5, 'パーカー', '白', 'L', 250),
               (6, 'パーカー', '赤', 'S', 100))
      SELECT product, color, size, sum(stock) AS stock_sum
        FROM goods GROUP BY DISTINCT ROLLUP(1, 2), ROLLUP(1, 3)
          ORDER BY product, color, size;
 product  | color | size | stock_sum
----------+-------+------+-----------
 パーカー | 白    | L    |       250
 パーカー | 白    | M    |       300
 パーカー | 白    |      |       550
 パーカー | 赤    | S    |       100
 パーカー | 赤    |      |       100
 パーカー |       | L    |       250
 パーカー |       | M    |       300
 パーカー |       | S    |       100
 パーカー |       |      |       650
 フリース | 白    | L    |       200
 フリース | 白    |      |       200
 フリース | 赤    | L    |       150
 フリース | 赤    |      |       150
 フリース | 青    | S    |       120
 フリース | 青    |      |       120
 フリース |       | L    |       350
 フリース |       | S    |       120
```

```
 フリース |       |       |        470
          |       |       |       1120
(19 rows)
```

DISTINCT キーワードを加えることで重複していた行は出力されなくなる動作が確認できました。

4.2.5. date_bin 関数

新たに data_bin 関数が追加されました。これは任意のタイムスタンプを指定された間隔の最も近い先頭の時間に揃えるためのシステム関数です。例えば、指定されたタイムスタンプが 1 時間のうち 20 分毎の区切りのどこに含まれるかを判定することに利用できます。

関数の書式は以下の通りです。

```
date_bin(stride, source, origin)        戻り値 timestamp 型
```

引数	意味
stride	指定する間隔（interval 型）
source	指定するタイムスタンプ（timestamp または timestamp with time zone 型）
origin	起点となるタイムスタンプ（timestamp または timestamp with time zone 型）

以下のように実行を確認しました。

```
（date_bin 関数の実行例）
db1=# SELECT date_bin('20 minutes', '2021-06-04 17:35:27',
         '2001-01-01 00:00:00');
     date_bin
---------------------
 2021-06-04 17:20:00
(1 row)
     ※ 2001-01-01 00:00 を起点とした 20 分区切りの先頭タイムスタンプが返る。
        最も近い 20 分区切りタイムスタンプ（本例なら 17:40:00）ではないことに注意。

db1=# SELECT TIMESTAMP '2021-06-24 19:35' -
         date_bin('29.53 days', '2021-06-24 19:35', '2021-01-13 14:00');
    ?column?
```

```
-----------------
 14 days 13:59:27
(1 row)
```

※ 新月の日時（2021 年 1 月 13 日 14 時）を起点に、ある時点（2021 年 6 月 24 日 19:35）の月齢を調べる応用例。14.5 日ほどなので満月に近いことがわかる。

類似の関数 date_trunc が以前から存在していましたが、週、日、時、分、秒など、決まった単位でのタイムスタンプ値の切り捨てしかできませんでした。

4.2.6. ストアドプロシージャの *OUT* パラメータ

本バージョンからストアドプロシージャで OUT パラメータが指定可能になりました。PostgreSQL 13.x までのバージョンでは IN パラメータと INOUT パラメータだけがサポートされていました。

プロシージャの OUT パラメータはプロシージャのシグネチャ（呼び出し情報）に含まれており、呼び出し時には仮パラメータを指定して使用します。

以下のように実行を確認しました。

（IN、INOUT、OUT パラメータを持つ プロシージャを定義する）

```
db1=# CREATE PROCEDURE plus(IN int, INOUT f1 int, OUT f2 text) LANGUAGE SQL
        AS $$ SELECT $1 + $2, 'result: ' || CAST($1 + $2 AS text) $$;
```

（psql での実行）

```
db1=# CALL plus(10, 20, null);
```

※OUT 引数には何を渡しても無視されます

```
 f1 |      f2
----+-------------
 30 | result: 30
(1 row)
```

次に JDBC におけるエスケープコール構文における使い方も確認しました。以下のテストコードを用意して実行しました。

// テストコード TestOutParam.java

```
import java.util.Properties;
import java.sql.*;

public class TestOutParam {
```

```
  public static void main(String[] args) throws SQLException {
    String url = "jdbc:postgresql://localhost:5432/db1";

    // escapeSyntaxCallMode に callIfNoReturn または call を指定して接続します。
    Properties props = new Properties();
    props.setProperty("escapeSyntaxCallMode", "callIfNoReturn");
    Connection con = DriverManager.getConnection(url, props);

    // psql 上の実行と同様に OUT 引数も含めた三引数で記述します。
    CallableStatement proc = con.prepareCall("{call plus(?, ?, ?)}");

    // IN 引数と INOUT 引数にパラメータを指定します。
    proc.setInt(1, 10);
    proc.setInt(2, 20);

    // 第二引数、第三引数について OUT パラメータであることを指定します。
    proc.registerOutParameter(2, Types.INTEGER);
    proc.registerOutParameter(3, Types.VARCHAR);

    proc.execute();
    int f1 = proc.getInt(2);
    String f2 = proc.getString(3);
    proc.close();
    System.out.printf("f1: %d\n", f1);
    System.out.printf("f2: \"%s\"\n", f2);
  }
}
```

（実行結果）

```
$ javac TestOutParam.java
$ java -classpath .:postgresql-42.2.22.jar TestOutParam
f1: 30
f2: "result: 30"
```

上記サンプルコードの書き方で INOUT、OUT パラメータ共に値が返却されることが確認できました。

4.3. 管理のための新機能

4.3.1. idle_session_timeout／client_connection_check_interval

クライアントとの TCP/IP 接続（ホスト接続）の制御を行う新たな設定パラメータ idle_session_timeout および client_connection_check_interval が追加されました。

◆ *idle_session_timeout*

idle_session_timeout は、クライアントの次問い合わせを待機している状態にあるセッションを強制終了させるタイムアウト時間を設定します。ただし、トランザクション内にあるセッションは対象外です。単位を指定しない場合はミリ秒として見なされます。0（デフォルト値）を指定した場合、タイムアウトは無効になります。

以下の手順で idle_session_timeout を設定したときのタイムアウト動作を確認しました。

```
（セッションタイムアウトを 5 秒に設定）
db1=# SET idle_session_timeout = 5000;
SET

（5 秒以上経過後）

db1=# SELECT current_user;
FATAL:  terminating connection due to idle-session timeout
server closed the connection unexpectedly
       This probably means the server terminated abnormally
       before or while processing the request.
The connection to the server was lost. Attempting reset: Succeeded.
```

◆ *client_connection_check_interval*

問い合わせ実行中にクライアントの接続検査を行う間隔を設定します。接続検査はソケットをポーリングすることによって行われ、クライアントが切断していた場合、長時間実行されていた問い合わせを速やかに中止させることができます。単位を指定しない場合はミリ秒として見なされます。0（デフォルト値）を指定した場合、接続検査は行われません。

psql で時間を要する問い合わせを実行して、問い合わせ実行が終わる前に psql 実行プロセスを強制終

了することで、client_connection_check_interval 設定の挙動を確認しました。このとき、サーバログから挙動を確認するため、postgresql.conf 以下の設定を与えています。

```
log_connections = on
log_disconnections = on
log_statement = 'all'
```

以下に手順を示します。
まず、本設定が無効である場合です。

```
（client_connection_check_interval = 0 の場合）
$ psql -h localhost -U postgres -d db1        ※ホスト接続をします
db1=# SHOW client_connection_check_interval;
 client_connection_check_interval
----------------------------------
 0
(1 row)

db1=# SELECT pg_sleep(300);
（上記問い合わせの開始後すぐに別ターミナルから kill -9 《PID》を実行して、
 psql 実行プロセスを強制終了）
```

```
（ログ出力）
2021-06-02 11:32:22.283 JST [4379] LOG:  connection received: host=127.0.0.1
port=52786
2021-06-02 11:32:22.283 JST [4379] LOG:  connection authorized:
user=postgres database=db1 application_name=psql
2021-06-02 11:32:25.865 JST [4379] LOG:  statement: SHOW
client_connection_check_interval;
2021-06-02 11:32:51.209 JST [4379] LOG:  statement: SELECT pg_sleep(300);
                        ※SELECT 実行直後に kill -9 を実行してクライアント切断。
2021-06-02 11:37:51.218 JST [4379] LOG:  could not send data to client:
Broken pipe
2021-06-02 11:37:51.218 JST [4379] FATAL:  connection to client lost
                        ※クライアントの切断を検出するのに数分掛かっている。
2021-06-02 11:37:51.218 JST [4379] LOG:  disconnection: session time:
```

```
0:05:28.935 user=postgres database=db1 host=127.0.0.1 port=52786
```

続いて、本設定を使った場合です。

```
（client_connection_check_interval = 5000 の場合）
$ psql -h localhost -U postgres -d db1        ※ホスト接続をします
db1=# SHOW client_connection_check_interval;
 client_connection_check_interval
----------------------------------
 5s
(1 row)

db1=# SELECT pg_sleep(300);
（上記問い合わせの開始後すぐに別ターミナルから kill -9 《PID》を実行して、
 psql 実行プロセスを強制終了）
```

```
（ログ出力）
...
2021-06-02 11:41:52.169 JST [4548] LOG:  connection received: host=127.0.0.1
port=52788
2021-06-02 11:41:52.169 JST [4548] LOG:  connection authorized:
user=postgres database=db1 application_name=psql
2021-06-02 11:42:05.004 JST [4548] LOG:  statement: SHOW
client_connection_check_interval;
2021-06-02 11:42:39.408 JST [4548] LOG:  statement: SELECT pg_sleep(300);
                        ※SELECT 実行直後に kill -9 を実行してクライアント切断。
2021-06-02 11:43:04.411 JST [4548] FATAL:  connection to client lost
                        ※クライアントの切断が数秒のうちに検出されている。
2021-06-02 11:43:04.411 JST [4548] STATEMENT:  SELECT pg_sleep(300);
2021-06-02 11:43:04.411 JST [4548] LOG:  disconnection: session time:
0:01:12.242 user=postgres database=db1 host=127.0.0.1 port=52788
...
```

client_connection_check_interval 設定により、問い合わせ実行中のクライアント切断を速やかに検出することとができました。

4.3.2. 新システムロール

pg_database_owner、pg_read_all_data、pg_write_all_data

PostgreSQL 14 では、以下の 3 つの定義済みシステムロールが追加されました。いずれもあらかじめ作成されているのみならず、特別な機能を持っています。

ロール名	説明
pg_database_owner	暗黙に現在のデータベース所有者をメンバーとしているロール。
pg_read_all_data	全スキーマの全てのテーブル、ビュー、シーケンスを参照できるロール。 スキーマの USAGE 権限とオブジェクトの SELECT 権限に相当。
pg_write_all_data	全スキーマの全てのテーブル、ビュー、シーケンスを更新できるロール。 スキーマの USAGE 権限とオブジェクトの INSERT、UPDATE、DELETE 権限に相当。

以下の動作確認を行いました。

◆ ***pg_database_owner の動作確認***

```
（各オブジェクト所有者を pg_database_owner として、定義 SQL を記述）
$ cat > app_ddl.sql <<EOS
CREATE SCHEMA app AUTHORIZATION pg_database_owner;
CREATE TABLE app.t1 (id int PRIMARY KEY, txt text, ts timestamp);
ALTER TABLE app.t1 OWNER TO pg_database_owner;
CREATE SEQUENCE app.seq1;
ALTER SEQUENCE app.seq1 OWNER TO pg_database_owner;
EOS

（新たなユーザとそのユーザを所有者とするデータベースを作成して、定義 SQL をロード）
$ psql -U postgres
postgres=# CREATE USER appuser1;
postgres=# ALTER USER appuser1 SET search_path TO 'app';
postgres=# CREATE DATABASE appdb1 OWNER appuser1;
postgres=# \c appdb1
appdb1=# \i app_ddl.sql
```

```
（データベース所有者ユーザが各オブジェクトの所有者同様にアクセス可能になる）
appdb1=# \c appdb1 appuser1
You are now connected to database "appdb1" as user "appuser1".
appdb1=> \d
            List of relations
 Schema | Name |   Type   |       Owner
--------+------+----------+-------------------
 app    | seq1 | sequence | pg_database_owner
 app    | t1   | table    | pg_database_owner
(2 rows)

appdb1=> INSERT INTO t1 VALUES (nextval('seq1'), 'TXT', CURRENT_TIMESTAMP);
INSERT 0 1
appdb1=> SELECT * FROM t1;
 id | txt |             ts
----+-----+----------------------------
  1 | TXT | 2021-06-11 11:22:02.823711
(1 row)
```

上記の使い方により、各オブジェクト所有者を pg_database_owner とすることで、同じ定義 SQL を、所有者の異なるいくつかのデータベースに適用するにあたり、シンプルな記述が可能になりました。この書き方であれば同じ定義 SQL をどのデータベースに投入しても、データベース所有者が各オブジェクトの所有者になります。

◆ *pg_read_all_data と pg_write_all_data の動作確認*

全データが読める pg_read_all_data ロールのメンバーであるユーザ reader1、pg_write_all_data ロールのメンバーであるユーザ writer1 を作成して、これらロールの挙動を確認しました。前節で作成した、データベースと各オブジェクトの所有者が appuser1 である appdb1 データベースにアクセスを試みます。

```
（検証用のユーザを作成）
$ psql -d db1 -U postgres
db1=# CREATE USER reader1;
db1=# GRANT pg_read_all_data TO reader1;
db1=# CREATE USER writer1;
```

```
db1=# GRANT pg_write_all_data TO writer1;

（appdb1 に reader1 で接続して動作確認）
db1=# \c appdb1 reader1
You are now connected to database "appdb1" as user "reader1".
appdb1=> SELECT * FROM app.t1;
 id | txt |             ts
----+-----+----------------------------
  1 | TXT | 2021-06-11 11:22:02.823711
(1 row)
appdb1=> INSERT INTO app.t1 VALUES (nextval('app.seq1'), 'TXT2', now());
ERROR:  permission denied for table t1    ← 書き込み（参照以外）はできない

（appdb1 に writer1 で接続して動作確認）
appdb1=> \c appdb1 writer1
You are now connected to database "appdb1" as user "writer1".
appdb1=> INSERT INTO app.t1 VALUES (nextval('app.seq1'), 'TXT2', now());
INSERT 0 1
appdb1=> SELECT * FROM app.t1;
ERROR:  permission denied for table t1    ← 逆に参照はできない
```

pg_read_all_data ロールや pg_write_all_data ロールのメンバーであれば、何ら権限付与していないオブジェクトについて、それぞれ参照と書き込みが可能となっている動作が確認できました。読み書き共に可能とするためには両ロールのメンバーである必要があります。

4.3.3. 実行時統計ビューの拡張 (pg_stat_progress_copy, pg_stat_wal, pg_stat_replication_slots)

PostgreSQL 14 では、以下表に示す 3 つの実行時統計ビューの追加と拡張がされました。

実行時統計ビュー名	説明
pg_stat_progress_copy	COPY の進捗を表示します。バックエンドごと 1 行が現れます。
pg_stat_wal	WAL の生成と書き込みに関する統計が 1 行データで表示します。
pg_stat_replication_slots	ロジカルレプリケーションスロットの使用状況に関する統計を表示します。スロットごとの 1 行が現れます。

◆ *pg_stat_progress_copy ビュー*

追加された pg_stat_progress_copy ビューの定義を以下に示します。

列名	データ型	説明
pid	integer	バックエンドのプロセス ID
datid	oid	接続先データベースの OID
datname	name	接続先データベース名
relid	oid	対象テーブルの OID、クエリからのコピーでは 0
command	text	実行中コマンド（COPY FROM または COPY TO）
type	text	入出力型：FILE、PROGRAM、PIPE、CALLBACK（論理レプリケーションの初期テーブル同期）のいずれか
bytes_processed	bigint	処理されたバイト数
bytes_total	bigint	ソースファイルのサイズ（不明なら 0）
tuples_processed	bigint	既に処理された行数
tuples_excluded	bigint	WHERE 句で除外された行数

これらのビューの動作確認をおこないました。

COPY 実行時の動作を確認するため、検証用のデータベースとテーブルを作成し、/tmp ディレクトリにデータの書き出し先および読み取り元として使用するデータファイルを作成しました。以下に手順を示します。

（検証用のデータベースを作成し、t_copy テーブルとデータを作成）

```
$ createdb copydb
$ psql copydb
copydb=# CREATE TABLE t_copy (id int primary key, ts timestamp, mes text);
           INSERT INTO t_log SELECT g, now(), md5(g::text)
```

```
             FROM generate_series(1, 1500000) as g;

（別ターミナルで pg_stat_progress_copy ビューを 1 秒ごとに表示をくり返すよう準備）
$ psql copydb
copydb=# SELECT * FROM pg_stat_progress_copy;
 pid | datid | datname | relid | command | type | bytes_processed |
bytes_total | tuples_processed | tuples_excluded
-----+-------+---------+-------+---------+------+-----------------
+-------------+------------------+-----------------
(0 rows)
copydb=# \x
copydb=# \watch 1
..後略

（元のターミナルでテーブルからファイルへの COPY TO を実行）
copydb=# COPY t_log TO '/tmp/t_copy.dat';

（\watch 1 実行中のターミナルでは COPY TO の実行状況が表示される）
2021 年 07 月 27 日 10 時 15 分 33 秒 (every 1s)
-[ RECORD 1 ]----+---------   ← 実行中のコピーがある場合のみ行が表示される
pid              | 3987
datid            | 16406
datname          | copydb
relid            | 16407
command          | COPY TO
type             | FILE
bytes_processed  | 37164345
bytes_total      | 0          ← COPY TO の時は 0 となる（全体サイズが不明なため）
tuples_processed | 556350
tuples_excluded  | 0

(0 rows)  ← 実行中のコピーが終了すると 0 行になる

（元のターミナルでファイルからテーブルへの COPY FROM を実行）
copydb=# TRUNCATE TABLE t_copy;
```

```
copydb=# COPY t_copy FROM '/tmp/t_copy.dat';

（COPY 中は pg_stat_progress_copy ビューに進捗状況が表示される）
2021 年 07 月 27 日 10 時 24 分 12 秒 (every 1s)
-[ RECORD 1 ]----+----------    ← 実行中のコピーがある場合のみ行が表示される
pid              | 3987
datid            | 16406
datname          | copydb
relid            | 16407
command          | COPY FROM
type             | FILE
bytes_processed  | 4194304
bytes_total      | 100888896    ← COPY FROM では全体サイズが表示される
tuples_processed | 62999
tuples_excluded  | 0

..中略
2021 年 07 月 27 日 10 時 24 分 16 秒 (every 1s)
-[ RECORD 1 ]----+----------
pid              | 3987
datid            | 16406
datname          | copydb
relid            | 16407
command          | COPY FROM
type             | FILE
bytes_processed  | 90243072
bytes_total      | 100888896
tuples_processed | 1342646
tuples_excluded  | 0

(0 rows)    ← 実行中のコピーが終了すると行が無くなる
```

以上のように pg_stat_progress_copy ビューで COPY の進捗が確認できました。

◆ *pg_stat_wal ビュー*

追加された pg_stat_wal ビューの定義を以下表に示します。

列名	データ型	説明
wal_records	bigint	生成された WAL レコードの総数
wal_fpi	bigint	生成された WAL フルページイメージの総数
wal_bytes	numeric	生成された WAL のバイト単位での合計量
wal_buffers_full	bigint	WAL バッファがいっぱいになってディスクに書き込まれた回数
wal_write	bigint	XLogWrite 要求で WAL バッファがディスクに書き出された回数
wal_sync	bigint	issue_xlog_fsync 要求で WAL ファイルがディスクに同期された回数
wal_write_time	double precision	XLogWrite 要求によるディスク書き込みの時間総計（ミリ秒）
wal_sync_time	double precision	issue_xlog_fsync 要求によるディスクの時間総計（ミリ秒）
stats_reset	timestamp with time zone	統計情報が最後にリセットされた時点

検証のために、新規に PostgreSQL インスタンスを作成して起動直後の pg_stat_wal ビューを表示しました。初期状態は、全ての値が 0 となっています。

（検証用の PostgreSQL インスタンスを作成、起動）

```
$ initdb --no-locale -encoding=UTF8 -D /var/lib/pgsql/data/statwaldb
$ pg_ctl start -D /var/lib/pgsql/data/statwaldb -o '-c port=5434'
$ psql -p 5434
postgres=# \x
postgres=# SELECT * FROM pg_stat_wal;
-[ RECORD 1 ]----+------------------------------  ← DB 作成直後で WAL 生成無し
wal_records      | 0
wal_fpi          | 0
wal_bytes        | 0
wal_buffers_full | 0
wal_write        | 0
wal_sync         | 0
wal_write_time   | 0
```

```
wal_sync_time     | 0
stats_reset       | 2021-07-26 22:17:30.282104+09
```

続いて別のターミナルから、pgbench を実行して書き込み処理を発生させて、再度 pg_stat_wal ビューを表示すると、WAL 書き出しに関する統計値が確認できました。

```
（別のターミナルで pgbench の初期データ作成を実行）
$ pgbench -i -p 5434


（再度 pg_stat_wal ビューを表示すると各列の数値が増加）
postgres=# SELECT * FROM pg_stat_wal;
-[ RECORD 1 ]----+------------------------------  ← pgbench 初期化の WAL 生成量
wal_records      | 4006
wal_fpi          | 322
wal_bytes        | 13075616
wal_buffers_full | 800
wal_write        | 820
wal_sync         | 18
wal_write_time   | 0
wal_sync_time    | 0
stats_reset      | 2021-07-26 22:17:30.282104+09
```

◆ *pg_stat_replication_slots ビュー*

pg_stat_replication_slots ビューの定義を以下に示します。

列名	データ型	説明
slot_name	text	レプリケーションスロット名
spill_txns	bigint	ロジカルデコーディングでディスク書き出しをしたトランザクション数
spill_count	bigint	ロジカルデコーディングでディスク書き出しをした回数
spill_bytes	bigint	ロジカルデコーディングでディスク書き出しをしたバイト数
stream_txns	bigint	ロジカルデコーディングでストリーミングされたトランザクション数。
stream_count	bigint	ロジカルデコーディングでストリーミングされた回数

列名	データ型	説明
stream_bytes	bigint	ロジカルデコーディングでストリーミングされたバイト数
total_txns	bigint	ロジカルデコーディングで送出されたトランザクション総数
total_bytes	bigint	ロジカルデコーディングで送出された総バイト数
stats_reset	timestamp with time zone	統計情報が最後にリセットされた時点

「4.1.5 ロジカルレプリケーションの改善 」の検証を実施して、その時の pg_stat_replication_slots ビューの情報表示を確認しました。

パブリケーション側で 1 トランザクション内に大量データ投入を行うと、以下のように、書き出された spill ファイル出力関連の情報が pg_stat_replication_slots ビューの spill_txn、spill_counts、spill_bytes 列にあらわれました。

```
（パブリケーション側で pg_stat_replication_slots ビューを 1 秒ごとに表示）
db1=# SELECT * FROM pg_stat_replication_slots;
db1=# \watch 1

（pg_stat_replication_slots ビュー: spill ファイル出力中の観測）
2021 年 07 月 27 日 15 時 49 分 07 秒 (every 1s)
-[ RECORD 1 ]+---------
slot_name    | sub1
spill_txns   | 1                ←
spill_count  | 1                ←
spill_bytes  | 67109019         ← これらの値が増加
stream_txns  | 0
stream_count | 0
stream_bytes | 0
total_txns   | 0
total_bytes  | 0
stats_reset  |
(1 row)
```

続いてストリーミングが有効にした場合を確認しました。streaming = on の設定でサブスクリプションを作り直して、再度パブリケーション側テーブルに 1 トランザクションで大量データ投入すると、以下の内容が

観測されました。

```
（pg_stat_replication_slots ビュー： streaming=on で大量データ投入中の観測）
2021 年 07 月 27 日 16 時 21 分 50 秒 (every 1s)
-[ RECORD 1 ]+----------
slot_name    | sub1
spill_txns   | 0
spill_count  | 0
spill_bytes  | 0
stream_txns  | 1                ←
stream_count | 1                ←
stream_bytes | 67109019         ← これらの値が増加
total_txns   | 1                ←
total_bytes  | 67109019         ←
stats_reset  |
```

以上の通り、ロジカルレプリケーションの処理内容に応じた統計情報が表示されることが確認できました。

4.3.4. pg_amcheck コマンド

PostgreSQL 14 から新たに pg_amcheck コマンドが追加されました。これは、テーブルやインデックスの構造の論理的な一貫性を検査します。pg_amcheck の機能は contrib として提供される amcheck 拡張の関数によって実現されています。したがって、pg_amcheck を使うには amcheck の導入が必要で、pg_amcheck は amcheck 拡張を使いやすくして利便性を高めたものと言えます。

amcheck も PostgreSQL 14 で機能追加されており、B-Tree インデックスのみならず、標準的に使われる heapam を用いたテーブルの検査もできるようになっています。pg_amcheck でもテーブルの検査ができます。

以下に使用例を示します。

```
（amcheckdb データベースを作成、amcheck 拡張を導入後、 pgbench でテーブル作成）
$ createdb amcheckdb
$ psql amcheckdb -c 'CREATE EXTENSION amcheck;'
$ pgbench -i amcheckdb

（引数にデータベースを指定して、データベース全体を検査）
$ pg_amcheck amcheckdb
$                          ← エラーが無ければ何もでない
```

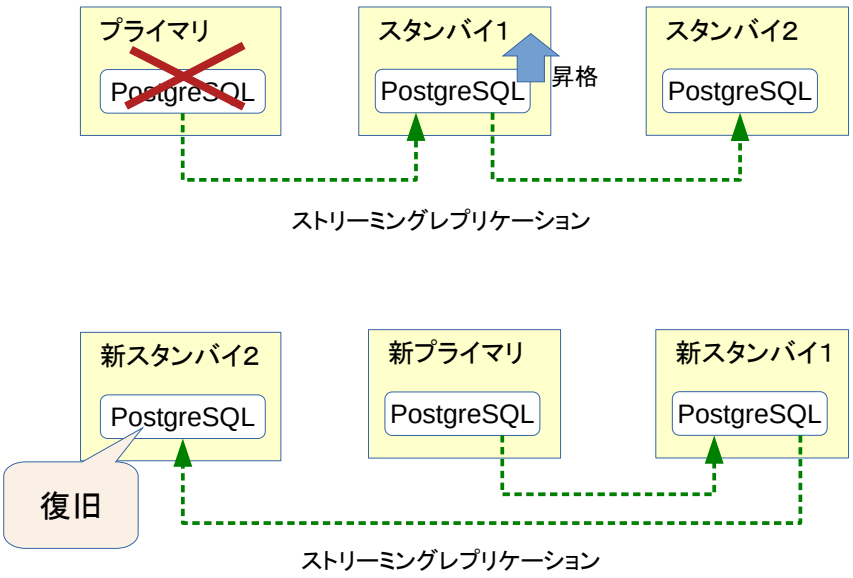
```
（--progress オプションで進捗状況表示）
$ pg_amcheck --progress amcheckdb
266/266 relations (100%) 2855/2855 pages (100%)

（--all オプションで全データベース対象）
$ pg_amcheck --all
pg_amcheck: warning: skipping database "postgres": amcheck is not installed
pg_amcheck: warning: skipping database "template1": amcheck is not installed
          ※ amcheck 未導入のデータベースについては警告を出してスキップします

（-d でデータベース指定、-t でテーブル指定、-v で冗長出力指定）
$ pg_amcheck -d amcheckdb -s public -t pgbench_accounts -v
pg_amcheck: including database "amcheckdb"
pg_amcheck: in database "amcheckdb": using amcheck version "1.3" in schema "public"
pg_amcheck: checking heap table "amcheckdb"."public"."pgbench_accounts"
pg_amcheck: checking btree index "amcheckdb"."public"."pgbench_accounts_pkey"
```

4.3.5. pg_rewind でスタンバイをソースサーバとして利用

pg_rewind のソースとしてスタンバイサーバを利用した巻き戻しが可能となりました。以前はソースサーバはプライマリでなければなりませんでした。これはソースサーバに一時テーブルを作成する動作があったためで、これを不要とする改修が行われました。

スタンバイサーバをソースとして pg_rewind を使う状況としては、3 台のカスケードレプリケーション（プライマリ → スタンバイ 1 → スタンバイ 2）でプライマリがダウンしてスタンバイ 1 が新プライマリに昇格した後、再び 3 台のカスケードレプリケーションにしたい場合が考えられます。下記の図に示します。

ダウンした旧プライマリサーバ上で、新スタンバイ 1（旧スタンバイ 2）をソースとした pg_rewind の実行により、3 台のカスケードレプリケーション構成が再び構成できます。



上記の 3 台のカスケードレプリケーション構成を想定した検証を、同一サーバ内でポート番号を変えた 3 つの PostgreSQL インスタンスを動作させて、以下の通り実施しました。

（プライマリサーバを作成）

```
$ initdb --no-locale --encoding=UTF8 -D /var/lib/pgsql/data/sv1
$ cat >> /var/lib/pgsql/data/sv1/postgresql.conf <<EOF
logging_collector = on
port = 5432
wal_log_hints = on
archive_mode = always
archive_command = 'cp %p /var/lib/pgsql/data/arc/%f'
EOF
$ mkdir /var/lib/pgsql/data/arc1
$ pg_ctl start -D /var/lib/pgsql/data/sv1
```

（スタンバイ 1 サーバを作成）

```
$ pg_basebackup -D /var/lib/pgsql/data/sv2 -R -p 5432 -h localhost
$ cat >> /var/lib/pgsql/data/sv2/postgresql.conf <<EOF
port = 5433
archive_command = 'cp %p /var/lib/pgsql/data/arc2/%f'
restore_command = 'cp /var/lib/pgsql/data/arc1/%f %p'
EOF
```

```
$ mkdir -p /var/lib/pgsql/data/arc2
$ pg_ctl start -D /var/lib/pgsql/data/sv2

（スタンバイ２サーバを作成）
$ pg_basebackup -D /var/lib/pgsql/data/sv3 -R -p 5433 -h localhost
$ cat >> /var/lib/pgsql/data/sv3/postgresql.conf <<EOF
port = 5434
archive_command = 'cp %p /var/lib/pgsql/data/arc3/%f'
restore_command = 'cp /var/lib/pgsql/data/arc2/%f %p'
EOF
$ mkdir -p /var/lib/pgsql/data/arc3
$ pg_ctl start -D /var/lib/pgsql/data/sv3

（プライマリに pgbench でデータ生成、データ更新を行う）
$ createdb pgbench
$ pgbench -i pgbench
$ pgbench -T 30 pgbench

（プライマリを非正常停止、スタンバイ１を昇格 - 障害時のフェイルオーバを模した操作）
$ kill -9 $(head -1 /var/lib/pgsql/data/sv1/postmaster.pid)
$ pg_ctl promote -D /var/lib/pgsql/data/sv2

（新プライマリに対して、引き続き更新処理を実行）
$ pgbench -p 5433 -T 30 pgbench

（旧プライマリで新スタンバイ１をソースに pg_rewind を実行）
$ pg_rewind -P -R --source-server='host=localhost, port=5434' \
   --target-pgdata=/var/lib/pgsql/data/sv1
pg_rewind: connected to server
pg_rewind: executing "/usr/local/pgsql/14/bin/postgres" for target server to
complete crash recovery
2021-07-30 21:00:43.879 JST [31675] LOG:  database system was interrupted;
last known up at 2021-07-30 20:46:40 JST
2021-07-30 21:00:43.937 JST [31675] LOG:  database system was not properly
shut down; automatic recovery in progress
```

```
2021-07-30 21:00:43.949 JST [31675] LOG:  redo starts at 0/30008D0
2021-07-30 21:00:44.407 JST [31675] LOG:  invalid record length at
0/3D57918: wanted 24, got 0
2021-07-30 21:00:44.407 JST [31675] LOG:  redo done at 0/3D578E0 system
usage: CPU: user: 0.07 s, system: 0.06 s, elapsed: 0.45 s
```

※ 非正常停止であったため、ここまでで自動クラッシュリカバリが働いた

```
PostgreSQL stand-alone backend 14beta2
backend> pg_rewind: servers diverged at WAL location 0/3D57918 on timeline 1
pg_rewind: rewinding from last common checkpoint at 0/3000908 on timeline 1
pg_rewind: reading source file list
pg_rewind: reading target file list
pg_rewind: reading WAL in target
pg_rewind: need to copy 68 MB (total source directory size is 96 MB)
70466/70466 kB (100%) copied
pg_rewind: creating backup label and updating control file
pg_rewind: syncing target data directory
pg_rewind: Done!
```

※ スタンバイをソースとして pg_rewind が処理に成功している

（新スタンバイサーバ２を設定変更して起動）

```
$ cat >> /var/lib/pgsql/data/sv1/postgresql.conf <<EOF
port = 5432
archive_command = 'cp %p /var/lib/pgsql/data/arc1/%f'
restore_command = 'cp /var/lib/pgsql/data/arc3/%f %p'
EOF
$ pg_ctl start -D /var/lib/pgsql/data/sv1
```

以上の手順で 3 台のカスケードレプリケーションが復旧しました。pg_rewind を使って、旧プライマリのデータディレクトリから、スタンバイサーバをターゲットとして、末端の新たなスタンバイサーバを構成することができました。

5. 重要な非互換変更

本章では PostgreSQL 14 の PostgreSQL 13.x に対する互換性の無い変更点のうち、重要と思われるものを取り上げます。

5.1. クライアント証明書認証の変更

pg_hba.conf の認証オプションで clientcert=1 という書き方（意味は clientcert=verify-ca と同じ）が廃止されました。clientcert を指定する場合には、verify-ca または verify-full を指定しなければなりません。

また、これは機能追加であって非互換ではありませんが、pg_hba.conf の認証オプションとして clientname オプションが追加されました。CN または DN を指定することができます。hostssl 接続であって clientcert=verify-ca とする場合、あるいは、cert 認証方式を使う場合に、接続ユーザ名と証明書内のどの項目とを比較するかを指定します。従来は必ず CN（Common Name）が使われていましたが、clientname=DN を指定することで、DN（Distinguished Name）を使うことができるようになりました。

5.2. V2 プロトコル廃止

PostgreSQL は 7.4 バージョン（2003 年リリース）からプロトコルバージョン 3.0 が使われるようになりましたが、それ以前のプロトコル 2.0 も引き続き受け付けていました。PostgreSQL 14 からプロトコル 2.0 がサポートされなくなります。

openjdk 1.8.0 と JDBC ドライバによる接続で以下のように protocolVersion=2 を指定した場合についてテストしました。

```
// テストコード jdbctest1.java
import java.sql.*;
public class jdbctest1 {
  public static void main(String[] args) {
    String url = "jdbc:postgresql://127.0.0.1:5432/db1?protocolVersion=2";
    Connection conn;
    try {
      conn = DriverManager.getConnection(url, "postgres", "pass");
      System.out.println("Connection OK");
      conn.close();
    } catch (Exception e) {
      System.err.println("jdbctest1: ERROR: " + e);
    }
  }
}
```

その結果、以下の接続エラーが生じる動作が確認できました。

```
$ javac -classpath ./postgresql-42.2.20.jar jdbctest1.java
$ java -Djdbc.drivers=org.postgresql.Driver \
       -classpath .:./postgresql-42.2.20.jar jdbctest1
jdbctest1: ERROR: org.postgresql.util.PSQLException: A connection could not
be made using the requested protocol 2.
```

protocolVersion=3 を指定するか、protocolVersion を指定しなければ、バージョン 3.0 プロトコルとなって、本エラーは発生しません。

今日、プロトコル 2.0 しか使えないデータベースクライアントは、極めて古いものを除き、ほとんど無いと考えられますが、サーバ側プリペアドステートメントの利用を回避するために敢えてプロトコル 2.0 を選択している場合があるかもしれません。

5.3. 後置単項演算子の廃止

本バージョンで後置単項演算子が廃止されました。PostgreSQL 13.x において標準で存在する後置単項演算子は、階乗を意味する「!」だけで、この演算子も廃止されました。なお、階乗の演算子には前置型の「!!」も存在しましたが、こちらも一緒に廃止されています。代替に factorial 関数を使用することができます。

ユーザ定義の後置単項演算子がある場合、バージョン 14 むけには仕様を変更する必要があります。

```
（PostgreSQL 14 で後置単項演算子の定義に失敗する例）
db1=# CREATE FUNCTION plusplus(int) RETURNS int LANGUAGE sql AS
      $$ SELECT $1 + 1 $$;
db1=# CREATE OPERATOR ~++ (FUNCTION = plusplus, LEFTARG = int);
ERROR:  operator right argument type must be specified
DETAIL:  Postfix operators are not supported.
```

5.4. EXTRACT の動作変更

EXTRACT 関数の戻り値の数値のデータ型が、以下に示すように double precision 型から numeric 型に変更されました。アプリケーション内の問い合わせで EXTRACT を含む式がある場合に影響確認が必要と考えられます。

```
db1=# SELECT pg_typeof(EXTRACT(DAY FROM now()));
 pg_typeof
-----------
```

```
 numeric
(1 row)
```

また、date 型に対して、日付には含まれない「時」「分」「秒」などを取り出そうとするときに 0 が返るのではなく、以下のようにエラーを出すように変わりました。

```
db1=# SELECT EXTRACT(MINUTE FROM now()::date);
ERROR:  date units "minute" not supported
```

5.5. *pg_standby* コマンドの廃止

contrib として提供されていた pg_standby コマンドが廃止されました。これは、restore_command 設定パラメータ内に本コマンドを使って WAL ファイル単位のレプリケーションを実現するものです。PostgreSQL がストリーミングレプリケーションに対応していなかった時点から、本コマンドによってスタンバイサーバの作成が可能でした（問い合わせ実行はできないのでウォームスタンバイと呼ばれました）。

9.0 以降バージョンであれば、pg_standby を使用しなくとも、WAL ファイル単位のレプリケーションは可能です。PostgreSQL 14 では以下の手順で構成できます。

```
（WAL アーカイブ用ディレクトリ、スタンバイ用データディレクトリを作成）
$ mkdir "${PGDATA}_arc"
$ mkdir "${PGDATA}_standby"
$ chmod 700 "${PGDATA}_standby

（ベースバックアップでスタンバイの初期データを生成）
$ pg_basebackup -D "${PGDATA}_standby" -R -c fast

（WAL ファイルコピーによる同期を設定して、スタンバイを起動）
$ cat > ${PGDATA}_standby/postgresql.auto.conf <<EOS
restore_command = 'cp ${PGDATA}_arc/%f %p'
port = 5433
EOS
$ pg_ctl start -D "${PGDATA}_standby"

（プライマリ側で WAL アーカイブを出力させる）
$ psql -U postgres << EOS
ALTER SYSTEM SET archive_mode TO on;
```

```
ALTER SYSTEM SET archive_command TO 'cp %p ${PGDATA}_arc/%f';
EOS
$ pg_ctl restart

（pgbench で書き込みをして、pg_switch_wal で WAL アーカイブを促します）
$ pgbench -i db1 &> /dev/null
$ psql -c 'SELECT pg_switch_wal()'
 pg_switch_wal
---------------
 0/5C7E958
(1 row)

（ps コマンドでスタンバイ側が次の WAL ファイルを待っている状態が確認できます）
$ ps xw
  PID TTY      STAT   TIME COMMAND
32462 ?        Ss     0:00 /usr/local/pgsql/14/bin/postgres
32475 ?        Ss     0:00 /usr/local/pgsql/14/bin/postgres -D
/var/lib/pgsql/data/14_standby
32477 ?        Ss     0:00 postgres: startup waiting for
000000010000000000000006
32640 ?        Ss     0:00 postgres: archiver last was
000000010000000000000005
   ※その他の出力を省略
```

上記は同ホスト内での実行ですが、 cp コマンドを scp コマンドに置き換えたり、NFS マウントしたディレクトリからコピーさせたりすることで、リモートサーバ間のレプリケーションにすることができます。

◆ *pg_standby の昇格モード*

pg_standby ではスタンバイ PostgreSQL を昇格させるのに smart モード（現時点の WAL アーカイブを全て適用して昇格）、fast モード（すぐに昇格）を選択できました。

上記の restore_command 設定パラメータを使った方法では、スタンバイを昇格させる操作を行うと 、pg_standby における smart モードに相当する動作になります。fast モードに相当する振る舞いをさせるには、スタンバイに recovery_target = 'immediate' が設定されている状態で昇格する必要があります。

以下は fast モード相当の昇格手順の検証です。

```
（スタンバイに recovery_target 設定を追加、ただし反映は再起動後）
$ psql -p 5433 -U postgres << EOS
ALTER SYSTEM SET recovery_target TO 'immediate';
EOS

（スタンバイ PostgreSQL を停止）
$ pg_ctl stop -D "${PGDATA}_standby"

（スタンバイ停止中にプライマリに大量の書き込みを行い、最後にテーブル作成を行う）
$ pgbench -i db1 &> /dev/null
$ pgbench -i db1 &> /dev/null
$ pgbench -i db1 &> /dev/null
$ psql -c 'CREATE TABLE t_new(id int)' db1

（standby.signal ファイルを削除して、スタンバイを昇格起動する）
$ rm "${PGDATA}_standby/standby.signal"
$ pg_ctl start -D "${PGDATA}_standby"

（プライマリで最後に行ったテーブル作成は反映されていない）
$ psql -p 5433 -c '\d t_new' db1
Did not find any relation named "t_new".
```

WAL ファイルとしては存在しているけれども未適用の変更について、それらを反映せずに即座に昇格させることができました。

なお、スタンバイで restore_command を必ず失敗するコマンド（例えば Linux における false コマンド）に置き換えて設定して昇格させても概ね同様の結果になります。この場合には WAL アーカイブは無視されますがオンライン WAL ディレクトリにある分は適用されます。厳密にいえばこちらの方が pg_standby の fast モードに近いと言えます。ただし、平常時は restore_command に WAL ファイルをコピーするコマンドが必要ですので、あらかじめ false 等に設定しておくわけにはいきません。

6. 免責事項

本ドキュメントは SRA OSS, Inc. 日本支社により作成されました。しかし、SRA OSS, Inc. 日本支社は本ドキュメントにおいて正確性、有用性、その他いかなる保証をするものではありません。本ドキュメントを利用する場合、利用者の責任において行なって頂くものとなります。