PostgreSQL15 検証レポート

1.2 版
2022 年 10 月 20 日

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

目次

1. はじめに 2
2. 概要 2
3. 検証のためのセットアップ 3
  3.1. ソフトウェア入手 3
  3.2. 検証環境 3
  3.3. インストール 3
4. 主な機能追加 5
  4.1. 性能向上 5
    4.1.1. NOT IN 句の改善 5
    4.1.2. ソートの性能改善 7
    4.1.3. ウィンドウ関数の性能改善 10
    4.1.4. psql \copy の性能改善 12
    4.1.5. Zstandard 圧縮サポート 13
    4.1.6. 先読みによるリカバリ性能改善 15
  4.2. SQL 機能追加 16
    4.2.1. MERGE 文 16
    4.2.2. SQL/JSON 対応の拡張 20
    4.2.3. 正規表現関数の追加 24
    4.2.4. 多重範囲型への集約 25
  4.3. DDL 機能追加 26
    4.3.1. ロジカルレプリケーションの拡張 26
    4.3.2. 呼び出し元権限で実行されるビュー 32
  4.4. 運用管理 33
    4.4.1. モジュールによる WAL アーカイブ 33
    4.4.2. JSONLOG 形式 34
    4.4.3. モニタリングビューの追加 35
5. 非互換変更 36
  5.1. 実行時統計情報の共有メモリ格納 36
  5.2. データベース作成時のデフォルト権限変更 37
  5.3. 排他的バックアップモードの廃止 37
  5.4. Python 2 のサポート終了 39
  5.5. array_to_tsvector 関数の変更 39
6. 免責事項 40

Copyright © 2022 SRA OSS LLC All rights reserved.

1. はじめに

本文書は PostgreSQL 15 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 15 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。
2022 年 5 月 19 日にリリースされた PostgreSQL 15 beta1 を使用して検証を行い、その後、2022 年 8 月 11 日にリリースされた beta3 での変更内容を反映して、本文書を作成しています。その後、2022 年 10 月 13 日リリース時点までに、「SQL/JSON 対応の拡張」の取り下げやいくつかの機能変更があったため、加筆修正をしました。

2. 概要

PostgreSQL 15 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
  - NOT IN の性能改善
  - ソートの性能改善
  - ウィンドウ関数の性能改善
  - psql ¥copy の性能改善
  - Zstandard 圧縮サポート
  - 先読みによるリカバリ性能改善
- SQL 機能追加
  - MERGE 文
  - SQL/JSON 対応の拡張　（こちらはリリース前に取り下げられました）
  - 正規表現関数の追加
  - マルチ範囲型への集約
- DDL 機能追加
  - ロジカルレプリケーションの拡張
  - 呼び出し元権限で実行されるビュー
- 運用管理
  - モジュールによる WAL アーカイブ
  - JSONLOG 形式
  - モニタリング機能の拡充

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 15 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/15/release-15.html

Copyright © 2022 SRA OSS LLC All rights reserved.

3. 検証のためのセットアップ

3.1. ソフトウェア入手

PostgreSQL 15（ベータ版を含む）は以下 URL のページからダウンロード可能です。ソースコード 、Windows むけバイナリのインストーラ、RPM yum リポジトリが用意されています。

```
https://www.postgresql.org/download
```

3.2. 検証環境

検証環境として、仮想化基盤上の RHEL 8.x (x86_64) 互換環境の仮想マシンを使用しました。

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

3.3. インストール

gcc、zlib-devel、readline-devel 、libicu-devel、openssl-devel、libzstd-devel、lz4-devel の各パッケージがあらかじめインストールされている状態で、以下のオプションにてソースコードのビルドを行いました。事前に postgres ユーザで読み書き可能な /usr/local/pgsql/15 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/v15beta1/postgresql-
15beta1.tar.bz2
                                                    《実際は 1 行》
$ tar jxf postgresql-15beta1.tar.bz2
$ cd postgresql-15beta1
$ ./configure --prefix=/usr/local/pgsql/15 --enable-debug \
  --with-openssl --with-icu --with-zstd --with-lz4
$ make world
$ make install-world
```

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

Copyright © 2022 SRA OSS LLC All rights reserved.

```
$ cat > ~/pg15.env <<'EOF'
VER=15
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
$ . ~/pg15.env
```

データベースクラスタを作成します。ロケール無し（C ロケール）、UTF8 をデフォルトとします。

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

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

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

PostgreSQL を起動します。

```
$ pg_ctl start
```

検証用のデータベースを作成します。

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

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

```
$ psql -U postgres -d db1
psql (15beta1)
Type "help" for help.

db1=#
```


Copyright © 2022 SRA OSS LLC All rights reserved.

4. 主な機能追加

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

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

```
/usr/local/pgsql/15/share/doc/html/
```

また、以下 URL にて PostgreSQL 15 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/15/
```

4.1. 性能向上

4.1.1. NOT IN 句の改善

本バージョンでは「NOT IN (《多数の要素》)」を伴う問い合わせの性能改善が行われています。これまでは順に値を比較して実行されていましたが、内部的にハッシュテーブルを使うように変更されました。

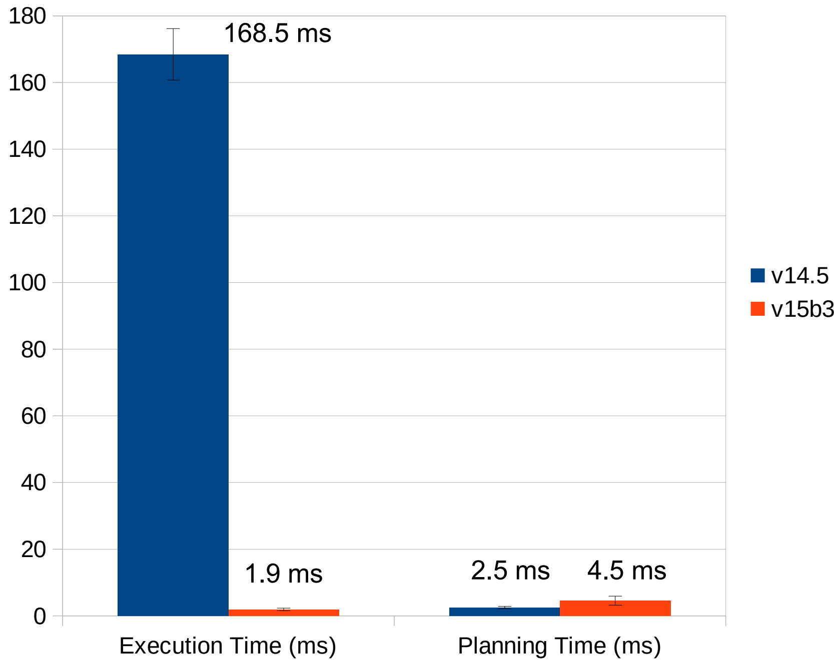
以下の手順で効果を確認しました。

```
（サンプルテーブルを作成し、1 万件を投入）
db1=# CREATE TABLE t_notin (id int primary key, dat text);
db1=# INSERT INTO t_notin SELECT g, 'x' FROM generate_series(1, 10000) g;

（NOT IN 句を使った問い合わせの文字列を作る SQL を用意）
db1=# SELECT 'SELECT * FROM t_notin WHERE id NOT IN (' ||
              string_agg(id::text, ',') || ');' FROM (
         SELECT id FROM t_notin ORDER BY random() LIMIT 5) v;
                        ?column?
-------------------------------------------------------------------
 SELECT * FROM t_notin WHERE id NOT IN (8261,3916,6501,5306,3699);
(1 row)

（NOT IN 句に 5000 件の値が並び、explain (analyze) を伴う問い合わせ文字列を生成）
db1=# SELECT 'explain (analyze) SELECT * FROM t_notin WHERE id NOT IN (' ||
              string_agg(id::text, ',') || ');' FROM (
```


Copyright © 2022 SRA OSS LLC All rights reserved.

```
        SELECT id FROM t_notin ORDER BY random() LIMIT 5000) v;
《出力省略》

（前回 SQL の出力を SQL 文として実行）
db1=# \gexec
                                QUERY PLAN
---------------------------------------------------------------------------
Seq Scan on t_notin  (cost=12.50..207.50 rows=5000 width=6) (actual
time=0.301..1.829 rows=5000 loops=1)
   Filter: (id <> ALL ('{9373,7209,2954, 《中略（合計 5000 要素が並ぶ）》
            5888,5810,2992,4198,911,1610,8445,7730,323,929}'::integer[]))
   Rows Removed by Filter: 5000
 Planning Time: 3.623 ms
 Execution Time: 2.060 ms
(5 rows)
```

このように 5000 個の定数要素を持つ NOT IN による問い合わせを実行して、プラン作成時間と実行時間を調べました。以下の結果が得られました。



棒グラフに付加された誤差範囲線は、上下に標準偏差だけ伸ばしたもの。同条件ごと 5 回実施。

Copyright © 2022 SRA OSS LLC All rights reserved.

プラン作成時間が若干増えている一方で、実行時間が大幅に短縮されていることが確認できました。
なお、NOT IN ではなく IN については、以前から内部的にハッシュ処理を使う動作となっています。

4.1.2. ソートの性能改善

本バージョンでは、外部ソートのアルゴリズムが変更されました。外部ソートは、ソート処理を work_mem 設定をサイズ上限とするメモリ上では処理しきれない場合の処理方式で、一時ファイルの書き出しと読み込みを伴ってソート処理を実行します。

以下の手順で PostgreSQL 14 と 15 で外部ソート処理の性能を比較します。

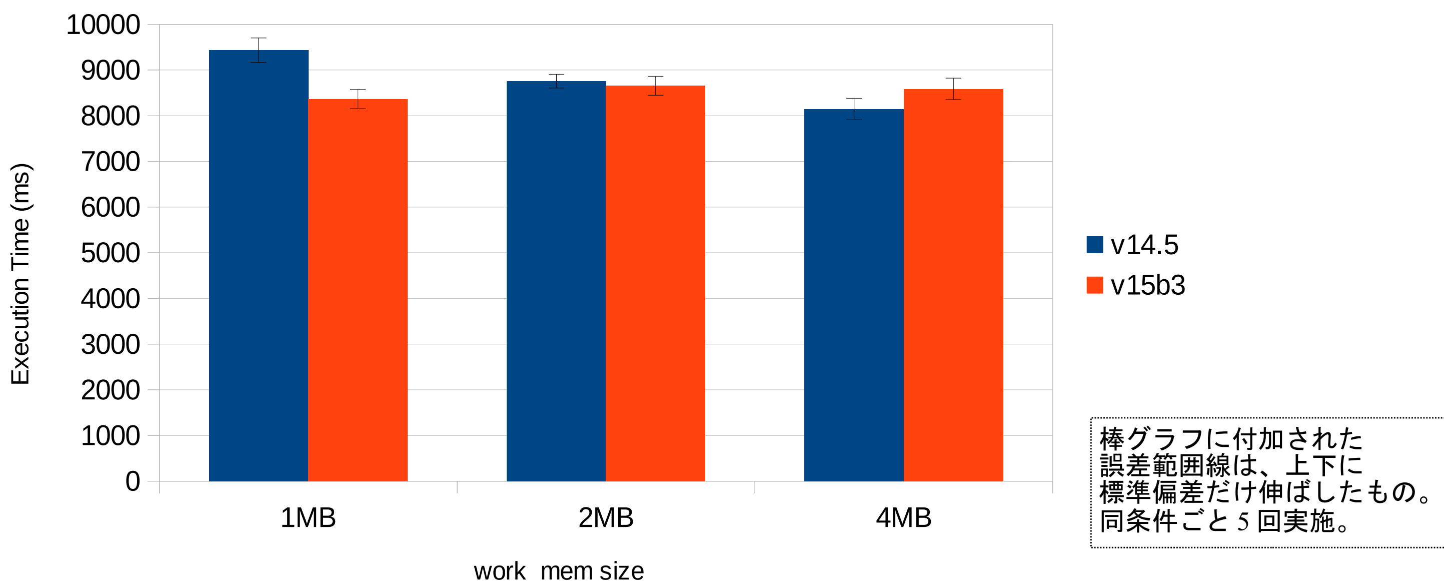
```
（サンプルテーブルを作成し、1000 万件を投入 ／ dat 列は md5 ハッシュ値で 32 文字の英数字）
db1=# CREATE TABLE t_sort1 (id int, dat text);
db1=# INSERT INTO t_sort1 SELECT g, md5(g::text)
        FROM generate_series(1, 10000000) g;

（パラレル処理の影響を除外するため 0 に設定）
db1=# SET max_parallel_workers_per_gather TO 0;

（work_mem を 1MB、2MB、4MB と変えて検証する）
db1=# SET work_mem TO '1MB';

（dat 列でソートして 500 万件目に続く 1 行だけを取り出す）
db1=# explain (analyze, buffers)
      SELECT * FROM t_sort1 ORDER BY dat LIMIT 1 OFFSET 5000000;

----------------------------------------------------------------------------
 Limit  (cost=2850780.44..2850780.44 rows=1 width=37)
        (actual time=8141.964..8141.966 rows=1 loops=1)
   Buffers: shared hit=83334, temp read=201242 written=231010
   ->  Sort  (cost=2838280.44..2863280.08 rows=9999858 width=37)
             (actual time=7443.007..8002.996 rows=5000001 loops=1)
        Sort Key: dat
        Sort Method: external merge  Disk: 459920kB
        Buffers: shared hit=83334, temp read=201242 written=231010
        ->  Seq Scan on t_sort1
             (cost=0.00..183332.58 rows=9999858 width=37)
```


Copyright © 2022 SRA OSS LLC All rights reserved.

```
                (actual time=0.015..505.314 rows=10000000 loops=1)
                Buffers: shared hit=83334
 Planning Time: 0.086 ms
 Execution Time: 8173.950 ms
(10 rows)
    → 上記出力は 15b3 バージョンのもの
      実際には各バージョン、各 work_mem 値で実行する
```

EXPLAIN (ANALYZE) コマンドで実行プランを出力させると PostgreSQL 14、15 とも同じ形になり、ソート部分で「Sort Method: external merge」と表示されます。データ件数は 1000 万件ですが、ソート件数は約 500 万件であることに注意してください。

本検証はメモリ 4GB の仮想マシンで shared_buffers=1024MB として実行しました。バッファにデータが載る過程での実行結果を除外するため、繰り返しの実行で所要時間が減っていく過程の結果は除外しています。各実行の所要時間（Execution Time）は以下の結果になりました。



work_mem が 1MB の場合に特にバージョン 15 の所要時間が短くなりました。一方で work_mem が 2MB、4MB と増えても速くなっていかない結果になっています。本修正の作者も開発過程において、特に work_mem が小さい場合において性能向上することを示すベンチマーク結果を報告しています[1]。これは本テスト結果とも一致します。

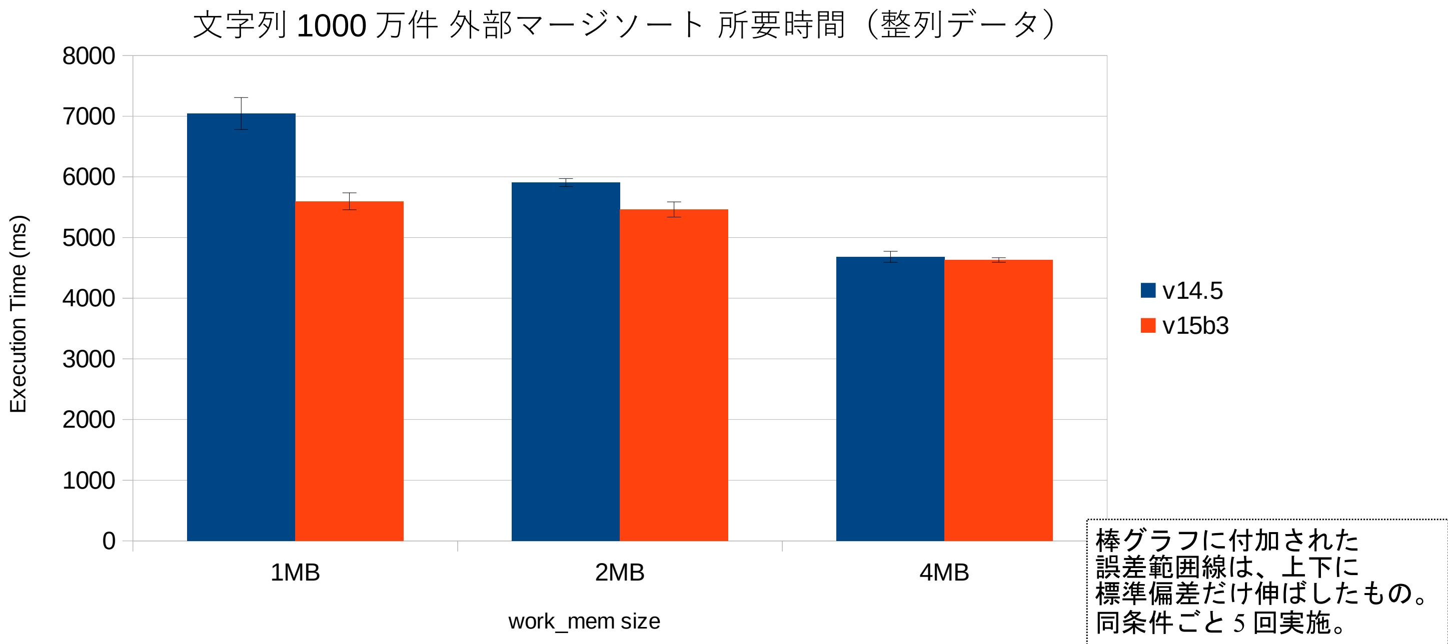
1 https://www.postgresql.org/message-id/flat/420a0ec7-602c-d406-1e75-1ef7ddc58d83%40iki.fi

Copyright © 2022 SRA OSS LLC All rights reserved.

予めデータがソート済みである場合についても以下の手順で検証しました。

```
（CLUSTER コマンドを使い t_sort1 テーブルを dat 列の値の順でソートした物理配置とする）
db1=# CREATE INDEX ON t_sort1 (dat);
db1=# CLUSTER t_sort1 USING t_sort1_dat_idx;
db1=# DROP INDEX t_sort1_dat_idx;

（同様に dat 列でソートして 500 万件目に続く 1 行だけを取り出す処理で、各条件で所要時間計測）
db1=# SET max_parallel_workers_per_gather TO 0;
db1=# SET work_mem TO '1MB';
db1=# explain (analyze, buffers)
      SELECT * FROM t_sort1 ORDER BY dat LIMIT 1 OFFSET 5000000;
```

ソート済みデータに対するソート処理の比較では以下の結果が得られました。



ソート済みデータに対するソート処理では全体的にバージョン 15 がバージョン 14 よりも短い所要時間で済んでいます。work_mem が少ない場合について特に優れた性能を示す点は同様となりました。なお、このテストでは完全にソートされたデータを使っていますが、意図としては「大部分がソート済みのデータに対するソート処理」を模しています。

Copyright © 2022 SRA OSS LLC All rights reserved.

4.1.3. ウィンドウ関数の性能改善

本バージョンではウィンドウ関数 row_number()、rank()、count() の性能を向上させる改良が行われました。これらの関数を使った問い合わせに対する新たな処理方法が追加され、それが使える場合には使うように実行プランが生成されるようになりました。

以下の手順で動作を確認しました。

(サンプルテーブル作成し、10 万件のデータを投入)

```
db1=# CREATE TABLE t_win1 (id int, c1 int, v1 text);
db1=# INSERT INTO t_win1 SELECT g, ceil(random() * 1000), md5(g::text)
       FROM generate_series(1, 100000) g;
```

(row_number()を使った問い合わせの実行プランを比較)

```
db1=# explain SELECT * FROM (
        SELECT *, row_number() over (order by c1) rn FROM t_win1) t
        WHERE rn <= 10;
```

《バージョン 14.5 の出力》

```
                                  QUERY PLAN
----------------------------------------------------------------------------
 Subquery Scan on t  (cost=180533.84..210533.84 rows=333333 width=49)
   Filter: (t.rn <= 10)
   ->  WindowAgg  (cost=180533.84..198033.84 rows=1000000 width=49)
         ->  Sort  (cost=180533.84..183033.84 rows=1000000 width=41)
               Sort Key: t_win1.c1
               -> Seq Scan on t_win1 (cost=0.00..19346.00 rows=1000000
width=41)
```

《バージョン 15b3 の出力》

```
                                QUERY PLAN
----------------------------------------------------------------------------
 WindowAgg  (cost=152431.13..171730.62 rows=1102828 width=49)
   Run Condition: (row_number() OVER (?) <= 10)
   ->  Sort  (cost=152431.13..155188.20 rows=1102828 width=41)
         Sort Key: t_win1.c1
         -> Seq Scan on t_win1 (cost=0.00..20374.28 rows=1102828 width=41)
```

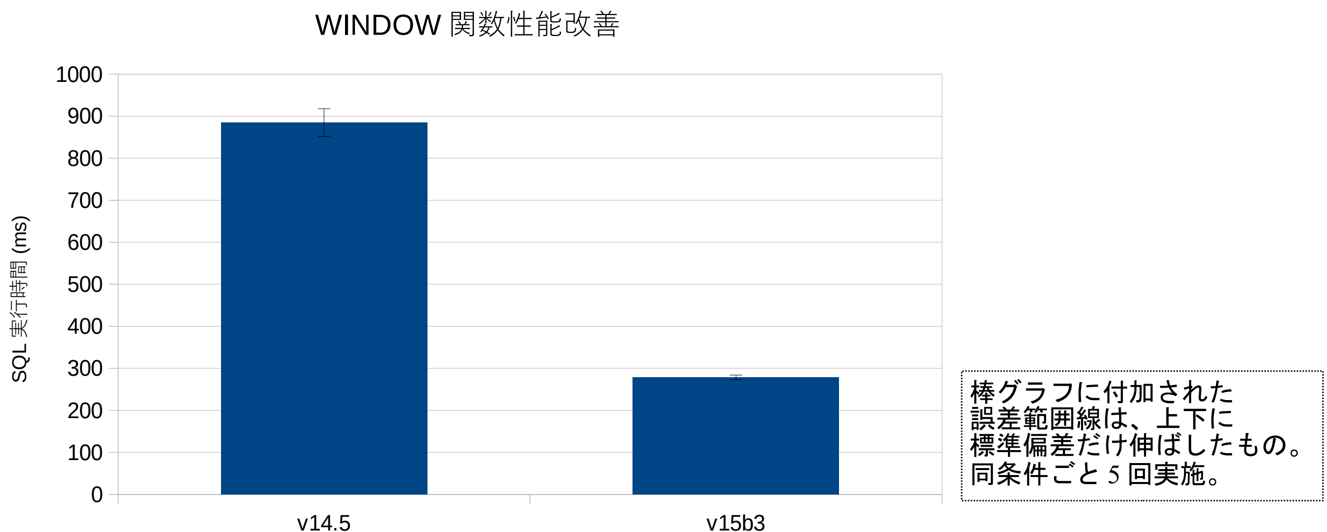

Copyright © 2022 SRA OSS LLC All rights reserved.

バージョン 14 ではウィンドウ関数による集約処理（WindowAgg）を行った後に、その結果に対して 、row_number が 10 以下であるものをフィルタ抽出している一方、バージョン 15 では WindowAgg 内で row_number が 10 以下という条件を合わせて処理していることが分かります。なお、本問い合わせの戻り行数は明らかに 10 行ですので、実行プランにおける行数予測は、両バージョンとも正しくありません。

この SQL について以下のように所要時間を計測しました。work_mem を大きくしてインメモリソートを行わせることで、PostgreSQL15 の外部マージソート改善の影響を除いています。

```
（両バージョンで row_number() を使った問い合わせを実行して所要時間を計測）
db1=# SET max_parallel_workers_per_gather TO 0;      -- パラレル処理の影響を排除
db1=# SET work_mem TO '1GB';                    -- インメモリ quicksort を行わせる
db1=# \timing on
db1=# SELECT * FROM (
        SELECT *, row_number() over (order by c1) rn FROM t_win1) t
        WHERE rn <= 10;
   id    | c1 |                v1                | rn
--------+----+----------------------------------+----
 611175 |  1 | 3490df2f6590e001629fa50fe9176dbb |  1
  《出力中略》
 584229 |  1 | 9d8820ebfecfb1f9d65795340261d323 | 10
(10 rows)
Time: 853.522 ms
```

その結果、以下のように SQL 所要時間としても短縮していることが確認できました。バージョン 15 では所要時間が 3 分の 1 程度に短縮されています。



Copyright © 2022 SRA OSS LLC All rights reserved.

しかしながら、本性能改善は非効率な実行プランを改善できるようにするものです。したがって、大きな性能改善が生じるのは旧バージョンにおける動作に非効率な実行プランが生じていた場合に限られます。また 、WindowAgg 実行プラン要素の中で Run Condition: を使う機能が無くとも、実行プラン改善ができる場合もあります。例えば、本例であれば「WHERE rn <= 10」の代わりに「LIMIT 10」と記述する方法もありました。

4.1.4. psql \copy の性能改善

本バージョンで psql の \copy コマンドの性能改善が適用されました。特に列が少なく行が多いテーブルの読み込み（\copy ... FROM ...）で性能が向上します。これまで 1 行ごとに PostgreSQL のワイヤプロトコルにおける CopyData メッセージを出していたものを、8KB 単位にまとめるように変更されました。

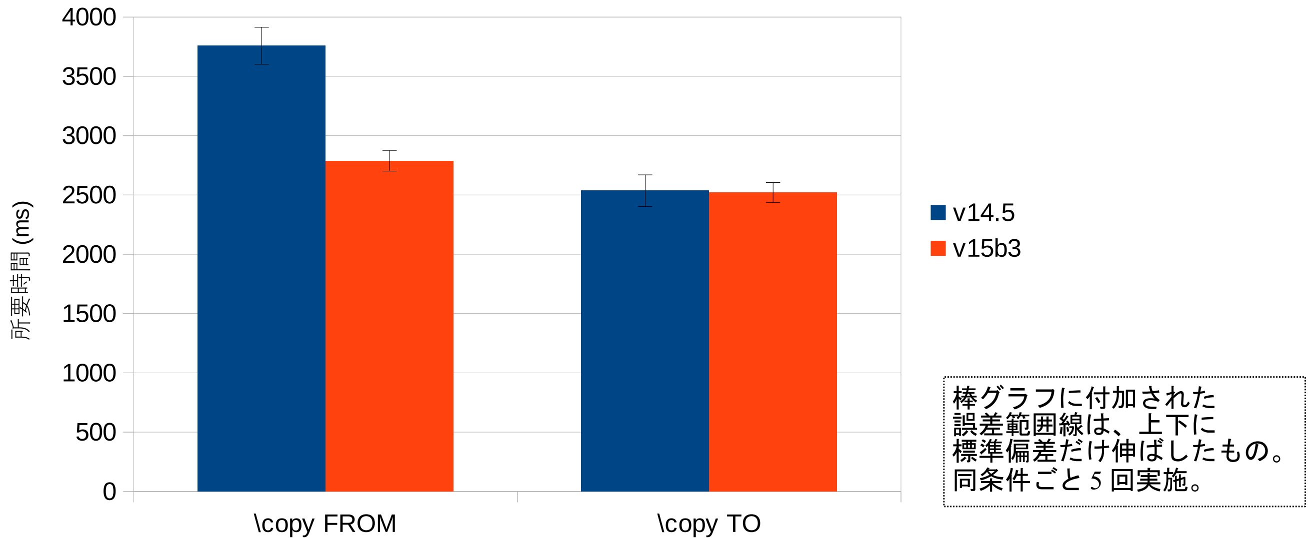
以下の手順で効果を確認しました。

```
（サンプルテーブル作成し、100 万件のデータを投入）
db1=# CREATE TABLE t_narrow (id int);
db1=# INSERT INTO t_narrow SELECT generate_series(1, 10000000);

（データの\copy による書き出し、読み込みを繰り返す）
db1=# \timing on
db1=# \copy t_narrow TO 't_narrow.dat'
COPY 10000000
Time: 2644.596 ms (00:02.645)
db1=# TRUNCATE t_narrow;
TRUNCATE TABLE
Time: 52.545 ms
db1=# \copy t_narrow FROM 't_narrow.dat'
COPY 10000000
Time: 3837.728 ms (00:03.838)
db1=# \copy t_narrow TO 't_narrow.dat'
《以下繰り返し》
```

計測結果を以下グラフに示します。

バージョン 15 では \copy FROM の所要時間が 3 割ほど短縮されています。\copy TO では違いがありません。このテーブルは 32bit 整数の列を 1 つだけ持ち、データが 100 万件ありますので、最も「細長い」テーブルと言えます。したがって、この結果が最大限効果がある場合であって、実務上で生じるテーブルにおける効果はこれよりは小さくなると言えます。

Copyright © 2022 SRA OSS LLC All rights reserved.



4.1.5. *Zstandard* 圧縮サポート

本バージョンから、WAL 圧縮と pg_basebackup に Zstandard による圧縮がサポートされました。ここでは特に WAL 圧縮について、性能影響と WAL 量軽減効果について以下の手順で検証しました。

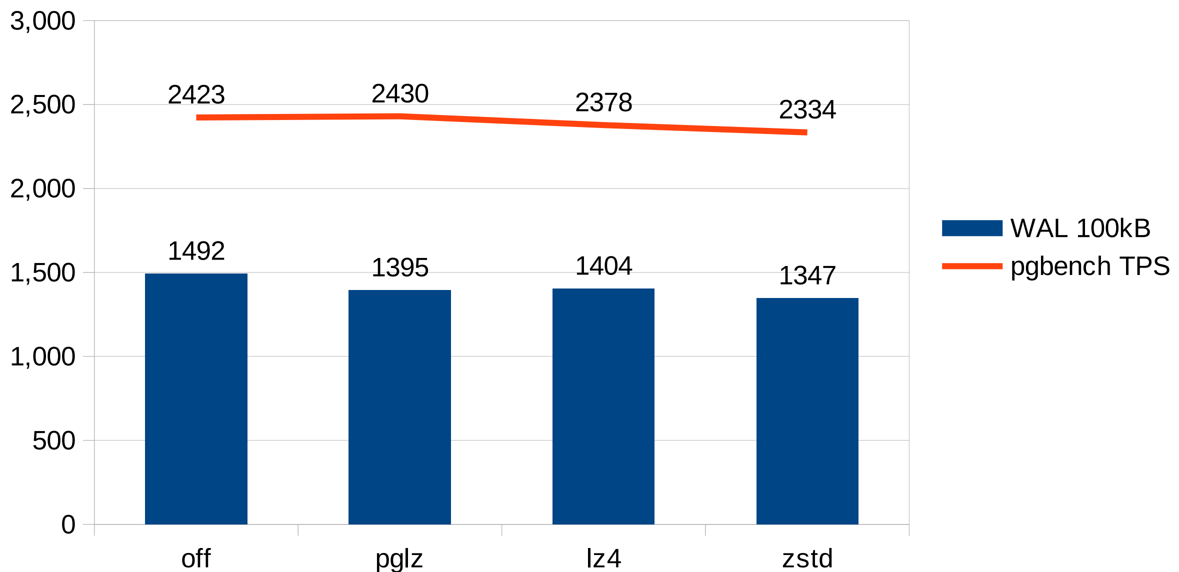
```
《wal_compression 設定を off、pglz、lz4、zstd に設定》
$ vi $PGDATA/postgresql.conf

$ pg_ctl restart                                  # PostgreSQL 再起動
$ psql -c 'SELECT pg_current_wal_lsn()'           # 現在の WAL LSN を取得
 pg_current_wal_lsn
--------------------
 1/C8AB4520
(1 row)

$ pgbench -i -s 10 -d postgres                    # pgbench 初期化
  《出力省略》
$ pgbench -c 5 -t 10000 postgres                  # pgbench 実行
  《出力中略》
latency average = 2.142 ms
tps = 2334.241867 (without initial connection time)
```


Copyright © 2022 SRA OSS LLC All rights reserved.

```
$ psql -c 'SELECT pg_current_wal_lsn()'              # 再度 WAL LSN を取得
 pg_current_wal_lsn
--------------------
 1/D0E42920
(1 row)

《差分から WAL の進行バイト数を計算》
$ psql -c "SELECT '1/D0E42920'::pg_lsn - '1/C8AB4520'::pg_lsn"
 ?column?
-----------
 137946112
(1 row)
```

以下グラフはこの手順を設定毎に実行した結果です。

pgbench TPS は WAL 書き込みを進めるために実行した pgbench コマンドの性能スコアです。各 1 回しか実施していませんが、書き込みトランザクションへの性能影響の参考として掲載しました。



WAL 出力サイズは、wal_compression = off が最も大きく、それに次いで pglz と lz4 （off より 5%～10%

Copyright © 2022 SRA OSS LLC All rights reserved.

縮小）、zstd が最も小さい（ off よりも 10%程度縮小）という結果になりました。pgbench 結果は 4 者の間での差異は小さいですが、傾向としては WAL 出力サイズが小さいものほどスコアが下がる（遅い）という結果になりました。

WAL 圧縮が効くのはフルページ書き出しが生じた場合に限られますが、WAL 出力には通常の WAL レコード出力も大きな割合を占めるため、圧縮方式の違いは大きな差異としては現れなかったと考えられます。

4.1.6. 先読みによるリカバリ性能改善

本バージョンから、クラッシュリカバリやストリーミングレプリケーション、バックアップからのリカバリにおける WAL 適用で、WAL ファイルの先読みができるようになりました。これは posix_fadvise システムコールを使うことで実現しています。そのため、サポートされるのは posix_fadvise をサポートする OS（Linux でサポートされています）に限られます。本機能による WAL 適用の性能改善はストレージ I/O の効率化によります。

以下の設定パラメータが追加されています。

パラメータ	説明
recovery_prefetch	先読みをするか。try（デフォルト、サポートされているプラットフォームであれば先読みする）、on、off で指定する。
wal_decode_buffer_size	先読みするサイズ。

以下のように性能改善を確認しました。

```
《ベースバックアップと WAL アーカイブを用意》
$ mkdir /var/lib/pgsql/data15a
$ vi $PGDATA/postgresql.conf                    # 以下の設定をする
   archive_mode = on
   archive_command = 'cp %p /var/lib/pgsql/data15a/%f'
$ pg_ctl restart
$ pg_basebackup -D /var/lib/pgsql/backup15
$ pgbench -i -s 10
$ pgbench -c 10 -t 100000                       # -t の数はもっと多くてもよい
$ pg_ctl stop

《リカバリを実行／アーカイブ WAL ファイルは予め全て pg_wal に入れておく》
$ cp -R /var/lib/pgsql/backup15 /var/lib/pgsql/restore15
$ rm -f /var/lib/pgsql/restore15/log/*
```


Copyright © 2022 SRA OSS LLC All rights reserved.

```
$ cp /var/lib/pgsql/data15a/* /var/lib/pgsql/restore15/pg_wal/
$ vi /var/lib/pgsql/restore15/postgresql.conf        # 以下の設定をする
   archive_mode = off
   recovery_prefetch = on または off

$ pg_ctl start -D /var/lib/pgsql/restore15
$ pg_ctl stop -D /var/lib/pgsql/restore15
$ cat /var/lib/pgsql/restore15/log/* | tail -100     # ログから所要時間を調べる
《ログ内容は抜粋のみ》
2022-08-24 16:29:11.303 JST [2503] LOG:  starting PostgreSQL 15beta3 on
x86_64-pc-linux-gnu, compiled by gcc (GCC) 8.5.0 20210514 (Red Hat 8.5.0-
10), 64-bit
2022-08-24 16:29:16.468 JST [2507] LOG:  redo done at 2/675B060 system
usage: CPU: user: 4.38 s, system: 0.37 s, elapsed: 4.86 s
2022-08-24 16:29:16.743 JST [2503] LOG:  database system is ready to accept
connections
$ rm -rf /var/lib/pgsql/restore15                    # 次試験のために削除
```

結果は以下の通りで大きな効果は得られませんでした。これは実施した環境が仮想マシンであって、ストレージ I/O は下位の各レイヤで様々な最適化が加わっているため、 posix_fadvise システムコールの効果があらわれなかったと考えられます。

パラメータ	Redo Elapsed	起動開始から接続受け付けまで
recovery_prefetch = on	4.79s	5.44s
recovery_prefetch = off	4.86s	5.32s

4.2. SQL 機能追加

4.2.1. MERGE 文

本バージョンから SQL で MERGE 文がサポートされました。MERGE 文は、行のマージ（併合）を行います。テーブルの結合条件を元に合致する場合と合致しない場合それぞれで INSERT、UPDATE、DELETE あるいは、何もしない（DO NOTHING）ことを単一の SQL 内で実行できます。

Copyright © 2022 SRA OSS LLC All rights reserved.

以下に MERGE 文の実行例を示します。

```
（t_master と t_work は、t_work にだけ del 列がある以外は同定義のテーブル、
 最初に t_master の行内容を全て t_work にコピー）
db1=# CREATE TABLE t_master
      (id int PRIMARY KEY, c1 text, c2 text, ts timestamp);
db1=# CREATE TABLE t_work
      (id int PRIMARY KEY, c1 text, c2 text, ts timestamp, del boolean);
db1=# INSERT INTO t_master VALUES (1, 'aa', 'AA', '2022-08-01 00:00'),
                                  (2, 'bb', 'BB', '2022-08-01 00:00'),
                                  (3, 'cc', 'CC', '2022-08-01 00:00');
db1=# INSERT INTO t_work SELECT *, NULL FROM t_master;
db1=# SELECT * FROM t_work;
 id | c1 | c2 |         ts          | del
----+----+----+---------------------+-----
  1 | aa | AA | 2022-08-01 00:00:00 |
  2 | bb | BB | 2022-08-01 00:00:00 |
  3 | cc | CC | 2022-08-01 00:00:00 |
(3 rows)

（t_work で行の変更や追加を実施）
db1=# INSERT INTO t_work VALUES (4, 'dd', 'DD', '2022-08-02 10:00');
db1=# UPDATE t_work SET c1 = 'bbb', c2 = 'BBB', ts = '2022-08-03 11:00'
      WHERE id = 2;
db1=# UPDATE t_work SET c1 = NULL, c2 = NULL, del = true,
       ts = '2022-08-03 12:00' WHERE id = 3;
db1=# SELECT * FROM t_work;
 id | c1  | c2  |         ts          | del
----+-----+-----+---------------------+-----
  1 | aa  | AA  | 2022-08-01 00:00:00 |
  4 | dd  | DD  | 2022-08-02 10:00:00 |
  2 | bbb | BBB | 2022-08-03 11:00:00 |
  3 |     |     | 2022-08-03 12:00:00 | t
(4 rows)
```


Copyright © 2022 SRA OSS LLC All rights reserved.

```
（MERGE 文で t_work のより新しい変更を t_master に反映）
db1=# MERGE INTO t_master m USING t_work w ON m.id = w.id
      WHEN MATCHED AND m.ts < w.ts AND w.del IS true THEN DELETE
      WHEN MATCHED AND m.ts < w.ts THEN
        UPDATE SET c1 = w.c1, c2 = w.c2, ts = w.ts
      WHEN NOT MATCHED
        THEN INSERT VALUES (w.id, w.c1, w.c2, w.ts);

db1=# SELECT * FROM t_master;
 id | c1  | c2  |         ts
----+-----+-----+---------------------
  1 | aa  | AA  | 2022-08-01 00:00:00
  2 | bbb | BBB | 2022-08-03 11:00:00
  4 | dd  | DD  | 2022-08-02 10:00:00
(3 rows)
```

t_work 上で変更した行の追加、変更、削除を MERGE 文 1 つで t_master に反映させることができました 。MERGE 文ではターゲットテーブル（ここでは t_master）とソーステーブル（ここでは t_work）をソーステーブル側を主とした外部結合をして、各結合行ごとに WHEN 句の条件に合った 1 つの処理を行います。本例であれば以下の外部結合結果を眺めると動作が理解しやすいでしょう。

```
db1=# SELECT * FROM t_master m RIGHT JOIN t_work w ON (m.id = w.id);
 id | c1 | c2 |         ts          | id | c1  | c2  |         ts          | del
----+----+----+---------------------+----+-----+-----+---------------------+----
  1 | aa | AA | 2022-08-01 00:00:00 |  1 | aa  | AA  | 2022-08-01 00:00:00 |
     → この結合行に対しては、t_work の ts がより新しくないので特に処理なし
  2 | bb | BB | 2022-08-01 00:00:00 |  2 | bbb | BBB | 2022-08-03 11:00:00 |
     → この結合行に対しては、t_master の c1、c2、ts を t_work の値で UPDATE
  3 | cc | CC | 2022-08-01 00:00:00 |  3 |     |     | 2022-08-03 12:00:00 | t
     → この結合行に対しては、t_work の del が真なので t_master の行を DELETE
    |    |    |                     |  4 | dd  | DD  | 2022-08-02 10:00:00 |
     → 結合できる行が無い場合には、t_work のデータで t_master に INSERT
```

ソーステーブルには、ビューやサブクエリも指定できます。例えば、大きな行数のテーブルに本処理を行うには、タイムスタンプの ts 列にインデックスを作成したうえで、以下のようにソーステーブルをサブクエリにして、ある時点以降の行だけを対象として処理させる方法が考えられます。

Copyright © 2022 SRA OSS LLC All rights reserved.

（ある時点以降の変更だけ反映させる）

```
db1=# MERGE INTO t_master m USING
      (SELECT * FROM t_work WHERE ts > '2022-08-02 00:00') w ON m.id = w.id
      WHEN MATCHED AND w.del IS true THEN DELETE
      WHEN MATCHED AND m.ts < w.ts THEN
        UPDATE SET c1 = w.c1, c2 = w.c2, ts = w.ts
      WHEN NOT MATCHED
        THEN INSERT VALUES (w.id, w.c1, w.c2, w.ts);
```

◆ *他の方式との比較*

PostgreSQL には、対象テーブルに行があれば UPDATE して、無ければ INSERT するという処理を記述する方法がいくつかあります。PostgreSQL 9.5 からサポートされている INSERT INTO ... ON CONFLICT 構文も利用可能ですし、さらに以前から利用可能であった WITH 句と RETURNING 句を使った書き方もできます。

これらは構文の違いだけではなく、動作に差異と特徴があります。以下表に示します。

方式	行データを INSERT または UPDATE する書き方	処理内容
MERGE	MERGE INTO t_master t USING (VALUES (1, 'aaa', 'AAA', now())) AS s (id, c1, c2, ts) ON t.id = s.id WHEN MATCHED THEN UPDATE SET c1 = s.c1, c2 = s.c2, ts = s.ts WHEN NOT MATCHED THEN INSERT VALUES (s.id, s.c1, s.c2, s.ts);	対象テーブルと対象データを外部結合してから、結合結果の行について UPDATE または INSERT のいずれかを実行します。
INSERT／ON CONFLICT	INSERT INTO t_master VALUES (1, 'aaa', 'AAA', now()) ON CONFLICT (id) DO UPDATE SET c1 = EXCLUDED.c1 , c2 = EXCLUDED.c2, ts = EXCLUDED.ts;	まず、INSERT を試みて、行が既に存在するために失敗したものについて、UPDATE を行います。
WITH／RETURNING	WITH val AS (SELECT * FROM (VALUES (1, 'aaa', 'AAA', now())) v (id, c1, c2, ts)), upd AS (UPDATE t_master SET c1 = val.c1, c2 = val.c2, ts = val.ts FROM val WHERE t_master.id = val.id RETURNING t_master.id) INSERT INTO t_master SELECT * FROM val WHERE id NOT IN (SELECT id FROM upd);	まず、UPDATE を試みて、該当行が無いものについて、INSERT を行います。

Copyright © 2022 SRA OSS LLC All rights reserved.

3つの方法は、INSERT、UPDATE どの順番で試みるかと言う点が異なります。このため、テーブルのデータや INSERT または UPDATE されるデータ内容によって、処理速度で有利な方法が異なります。

3つの方法ともターゲットとなるテーブルに複数行を INSERT または UPDATE することが可能です 。VALUES 句の部分に複数の値を書いたり、サブクエリで置き換えることで実現できます。

4.2.2. SQL/JSON 対応の拡張

注意

SQL/JSON 対応の拡張は 2022 年 9 月 1 日に取り消しされましたので、下記の検証結果は 15 正式版とは異なります。

本バージョンでは、SQL/JSON 標準に基づく JSON データを生成するコンストラクタ関数といくつかの JSON データに対する問い合わせ関数、変換関数のサポートが追加されました。

追加されたコンストラクタ関数を以下表に示します。戻り値型は RETURNING 句によって json 型か jsonb 型かを指定できます。

関数名	説明
JSON	テキストデータから JSON を生成します。
JSON_OBJECT	テキスト配列から JSON オブジェクトを構築します。
JSON_OBJECTAGG	提供されたデータを JSON オブジェクトに集約します。
JSON_ARRAY	提供された SQL または JSON データから JSON 配列を構築します。
JSON_ARRAYAGG	提供された SQL または JSON データを JSON 配列に集約します。
JSON_SCALAR	SQL データから JSON スカラー値を生成します。

以下にこれら関数の使用例を示します。

（デフォルトの戻り値型は json 型であるため重複したキーも保持）

```
db1=# SELECT JSON('{"a":123, "b":[10,20,30], "a":"bar"}');
                 json
----------------------------------------
 {"a":123, "b":[true,"foo"], "a":"bar"}
```

（戻り値を jsonb 型に指定すると「"a":123」は「"a":"bar"」で上書きされる）

```
db1=# SELECT JSON('{"a":123, "b":[10,20,30], "a":"bar"}' RETURNING jsonb);
```


Copyright © 2022 SRA OSS LLC All rights reserved.

```
            json
----------------------------------
 {"a": "bar", "b": [10, 20, 30]}
```

（JSON_OBJECT でスカラ値からオブジェクトを構成）

```
db1=# SELECT JSON_OBJECT('num' VALUE 123, 'reg_date' : CURRENT_DATE);
            json_object
------------------------------------------
 {"num" : 123, "reg_date" : "2022-08-10"}
```

（JSON_OBJECTAGG は複数行データから 1 つのオブジェクトを構成）

```
db1=# SELECT g AS n, g^3 AS cube FROM generate_series(1, 3) g;
 n | cube
---+------
 1 |    1
 2 |    8
 3 |   27

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

（JSON_ARRAY はスカラ値や json/jsonb データの並びから JSON 配列を構成）

```
db1=# SELECT JSON_ARRAY(1, true, JSON '{"a":null}');
     json_array
-----------------------
 [1, true, {"a":null}]
```

（JSON_ARRAYAGG は複数行から JSON 配列を構成、並び順の指定も可能）

```
db1=# SELECT JSON_ARRAYAGG(v ORDER BY v DESC) FROM generate_series(1, 3) v;
 json_arrayagg
---------------
 [3, 2, 1]
```


Copyright © 2022 SRA OSS LLC All rights reserved.

```
（JSON_SCALAR 関数はスカラ値 1 つから json/jsonb 型データを生成）
db1=# SELECT JSON_SCALAR(123.45), pg_typeof(JSON_SCALAR(123.45));
 json_scalar | pg_typeof
-------------+-----------
 123.45      | json
```

json 型のデータを文字列型や bytea 型に変換する JSON_SERIALIZE も追加されました。この関数に jsonb 型の値を与えるとエラーにならない一方、意味のある出力が得られないことに注意が必要です。

```
（json 型データを文字列型や bytea 型に変換）
db1=# SELECT JSON_SERIALIZE('{ "a" : 1 } ' RETURNING bytea);
      json_serialize
----------------------------
 \x7b20226122203a2031207d20
```

また、下記の JSON 問い合わせ関数も追加されました。JSON_TABLE 以外は機能的には既存の関数で実現できるものですが、SQL/JSON 標準に基づくため移植性に優れる意味があります。これらは JSON PATH を使うため、対応しているのは jsonb 型のみです。

関数名	説明
JSON_EXISTS	JSON PATH で該当する要素があるか、を返します。
JSON_QUERY	JSON PATH で要素を抽出します。
JSON_VALUE	JSON PATH で要素を取り出し、SQL スカラ値として返します。得られたものがスカラ値でない場合には NULL が返ります。
JSON_TABLE	JSON データに問い合わせを行い、その結果をリレーショナルビューとして表示し通常の SQL テーブルと同様にアクセスすることができます。

以下にこれら関数の使用例をいくつか示します。

```
（JSON_QUERY は JSON PATH 問い合わせを行う）
db1=# SELECT JSON_QUERY (
       JSON('{"data":[{"name":"Alice", "att":{"age":18}},
                      {"name":"Bob", "att":{"age":21}}]}' RETURNING jsonb),
       '$.data[1].att');
```


Copyright © 2022 SRA OSS LLC All rights reserved.

```
 json_query
-------------
 {"age": 21}
```

（JSON_VALUE では問い合わせ結果がスカラ値でないと NULL になる）

```
db1=# \pset null *null*
db1=# SELECT JSON_VALUE (
       JSON('{"data":[{"name":"Alice", "att":{"age":18}},
                      {"name":"Bob", "att":{"age":21}}]}' RETURNING jsonb),
       '$.data[1].att');
 json_value
------------
 *null*
```

（JSON_TABLE は jsonb 型データから表データを構成、ORDINALITY で序数を追加できる）

```
db1=# SELECT * FROM JSON_TABLE (
      '[{"name":"Alice"}, {"name":"Bob"}]'::jsonb,
      '$[*]' COLUMNS (id FOR ORDINALITY, name text PATH '$.name'));
 id | name
----+-------
  1 | Alice
  2 | Bob
(2 rows)
```

（各列の値について JSON データのどの位置を取り出すかを JSON PATH で指定できる）

```
db1=# SELECT * FROM JSON_TABLE (
      JSON('{"data":[{"name":"Alice", "att":{"age":18}},
                     {"name":"Bob", "att":{"age":21}}]}' RETURNING jsonb),
      '$.data[*]' COLUMNS (id FOR ORDINALITY, name text PATH '$.name',
                           age int PATH '$.att.age'));
 id | name  | age
----+-------+-----
  1 | Alice |  18
  2 | Bob   |  21
(2 rows)
```


Copyright © 2022 SRA OSS LLC All rights reserved.

4.2.3. 正規表現関数の追加

本バージョンで以下の正規表現関数が追加されました。他のリレーショナルデータベース製品で実装されている関数に合わせた仕様となっており、移植性を向上させます。

関数名	説明
regexp_count	POSIX 正規表現パターンが文字列にマッチする個数を返します。
regexp_instr	POSIX 正規表現パターンが文字列に N 番目にマッチする開始位置または終了位置を返します。
regexp_like	POSIX 正規表現パターンが文字列にマッチするかを真偽で返します。
regexp_substr	POSIX 正規表現パターンにマッチする部分文字列を返します。
regexp_replace	POSIX 正規表現パターンにマッチする部分文字列を指定文字列に置き換えたものを返します。本関数は従来からありましたが、対象文字範囲の引数指定に対応しました。

以下にこれら関数の実行例をいくつか示します。

```
（「前の文字と同じ文字が 1 つ続く」というパターンの出現回数は 5 回）
db1=# SELECT regexp_count('ああ、すもももももももものうち', '(.)\1');
 regexp_count
--------------
            5

（「前の文字と同じ文字が 1 つ続く」パターンを 1 文字目から探して 2 番目の出現位置は 5 文字目、
  第 5 引数に 1 を与えると終了位置の次文字位置である 7 文字目が返る）
db1=# SELECT regexp_instr('ああ、すもももももももものうち', '(.)\1', 1, 2);
 regexp_instr
--------------
            5

db1=# SELECT regexp_instr('ああ、すもももももももものうち', '(.)\1', 1, 2, 1);
 regexp_instr
--------------
            7
```


Copyright © 2022 SRA OSS LLC All rights reserved.

```
（「前の文字と同じ文字が１つ続く」パターンの１番目の部分文字列は「ああ」）
db1=# SELECT regexp_substr('ああ、すもももももももものうち', '(.)\1', 1);
 regexp_substr
---------------
 ああ

（「前の文字と同じ文字が１つ続く」パターンを 4 文字目から探して２番目の出現を置換）
db1=# SELECT regexp_replace('ああ、すもももももももものうち', '(.)\1', '■■',
                                     4, 2);
        regexp_replace
------------------------------
 ああ、すもも■■ももももうち
(1 row)
```

4.2.4. 多重範囲型への集約

本バージョンで、多重範囲型（マルチ範囲型）への集約関数 range_agg() が追加されました。複数行の範囲型データを多重範囲型に集約することができます。

以下の動作例で機能を示します。

```
（休暇予定を格納する t_vacation テーブルを作成して、データを投入）
db1=# CREATE TABLE t_vacation (empid int, vacation_days daterange);
db1=# INSERT INTO t_vacation VALUES
      (101, daterange('2022-08-08', '2022-08-10', '[]')),
      (102, daterange('2022-08-15', '2022-08-18', '[]')),
      (103, daterange('2022-07-25', '2022-07-29', '[]')),
      (101, daterange('2022-08-15', '2022-08-16', '[]')),
      (102, daterange('2022-09-01', '2022-09-01', '[]'));

（range_agg() を使用して多重範囲型データに集約する）
db1=# SELECT empid, range_agg(vacation_days) FROM t_vacation
      GROUP BY empid ORDER BY empid;
 empid |                       range_agg
-------+-----------------------------------------------------
   101 | {[2022-08-08,2022-08-11),[2022-08-15,2022-08-17)}
```


Copyright © 2022 SRA OSS LLC All rights reserved.

```
  102 | {[2022-08-15,2022-08-19),[2022-09-01,2022-09-02)}
  103 | {[2022-07-25,2022-07-30)}
(3 rows)
```

4.3. DDL 機能追加

4.3.1. ロジカルレプリケーションの拡張

本バージョンではロジカルレプリケーション（論理レプリケーション）について様々な機能追加が行われました。それらのうち主要なものについて機能を確認しました。本節の動作確認のために以下の手順で、データベースクラスタをもう１つ作成しています。

```
（pg_basebackup にて 5433 ポートで動作するデータベースサービスを作成して起動）
$ vi $PGDATA/postgresql.conf    # ロジカルレプリケーション対応のため以下設定に修正
   wal_level = logical
$ pg_ctl restart
$ pg_basebackup -D /var/lib/pgsql/data15r -R
$ pg_ctl start -D /var/lib/pgsql/data15r -o '-c port=5433'
$ pg_ctl promote -D /var/lib/pgsql/data15r
    《いずれのコマンドも出力省略》
```

◆ ***スキーマ内の全テーブルをパブリケーション指定***

これまでテーブルのパブリケーションを指定するのに、個別のテーブルを列挙する方法と「FOR ALL TABLES」としてデータベース内の全テーブルを指定する方法がありましたが、これに加えて「FOR ALL TABLES IN SCHEMA ...」という指定スキーマ内の全テーブルを指定できるようになりました。

```
（サンプルのスキーマとテーブルを作成）
$ psql -p 5432 db1
db1=# CREATE SCHEMA scm1;
db1=# CREATE TABLE scm1.t1 (id int primary key, val text);
db1=# CREATE TABLE scm1.t2 (id int primary key, timestamp text);
db1=# \q
$ psql -p 5433 db1
    《記載省略 ／ 5433 ポートのサーバにも同様にスキーマとテーブルを作成》
```


Copyright © 2022 SRA OSS LLC All rights reserved.

```
（以下のようにしてスキーマ一括でパブリケーションを指定できる）
$ psql -p 5432 db1
db1=# CREATE PUBLICATION pub1 FOR TABLES IN SCHEMA scm1;

（サブスクリプション作成の手順は、従来同様で違いはなし）
$ psql -p 5433 db1
db1=# CREATE SUBSCRIPTION sub1 CONNECTION
      'host=localhost port=5432 user=postgres dbname=db1' PUBLICATION pub1;
```

以上の手順でスキーマ scm1 内の両テーブルについてロジカルレプリケーションが構成できました。

◆ *パブリケーションする行・列を限定*

テーブルをパブリケーションする際に行を限定することができるようになりました。

```
（WHERE 句条件を付けてパブリケーションを作成）
$ psql -p 5432 db1        # パブリケーション側（ポート 5432）
db1=# CREATE TABLE public.t_rep1 (id int PRIMARY KEY, c1 text);
db1=# CREATE PUBLICATION pub2 FOR TABLE public.t_rep1 WHERE (id < 100);

$ psql -p 5433 db1        # サブスクリプション側（ポート 5433）
db1=# CREATE TABLE public.t_rep1 (id int PRIMARY KEY, c1 text);
db1=# CREATE SUBSCRIPTION sub2 CONNECTION
      'host=localhost port=5432 user=postgres dbname=db1' PUBLICATION pub2;

（id が 100 以上の行はレプリケーションされない動作になる）
$ psql -p 5432 db1        # パブリケーション側（ポート 5432）
db1=# INSERT INTO t_rep1 VALUES (1, 'A'), (2, 'a'), (101, 'B'), (102, 'b');

$ psql -p 5433 db1        # サブスクリプション側（ポート 5433）
db1=# SELECT * FROM t_rep1;
 id | c1
----+----
  1 | A
  2 | a
(2 rows)
```


Copyright © 2022 SRA OSS LLC All rights reserved.

上記のようにテーブル単位のパブリケーション定義で WHERE 句を付加すると、その式を満たす行だけが伝搬するようになります。

パブリケーション側で UPDATE を行って、INSERT 時に伝搬しなかった行がパブリケーションの WHERE 句条件を満たすようになると、その行がサブスクリプション側に追加されます。逆に、パブリケーション側で UPDATE を行った結果、パブリケーションの WHERE 句条件を満たさなくなった場合、サブスクリプション側から行が消えます。以下に実行例を示します。

```
（パブリケーション側で）
db1=# UPDATE t_rep1 SET id = 52 WHERE id = 102;

（サブスクリプション側で）
db1=# SELECT * FROM t_rep1;
 id | c1
----+----
  1 | A
  2 | a
 52 | b
(3 rows)

（パブリケーション側で）
db1=# UPDATE t_rep1 SET id = 102 WHERE id = 52;

（サブスクリプション側で）
db1=# SELECT * FROM t_rep1;
 id | c1
----+----
  1 | A
  2 | a
(2 rows)
```

パブリケーションするテーブルの列についても限定できます。

```
（パブリケーション側で）
db1=# CREATE TABLE public.t_rep2 (id int PRIMARY KEY, c1 int, c2 int);
db1=# ALTER PUBLICATION pub2 ADD TABLE public.t_rep2 (id, c1);
```


Copyright © 2022 SRA OSS LLC All rights reserved.

```
（サブスクリプション側で）
db1=# CREATE TABLE public.t_rep2 (id int PRIMARY KEY, c1 int);
db1=# ALTER SUBSCRIPTION sub2 REFRESH PUBLICATION WITH (COPY_DATA);

（パブリケーション側で）
db1=# INSERT INTO t_rep2 VALUES (1, 10, 100), (2, 20, 200);

（サブスクリプション側で）
db1=# SELECT * FROM t_rep2;
 id | c1
----+----
  1 | 10
  2 | 20
(2 rows)
```

上記例では t_rep2 テーブルについて、対象列を id 列、c1 列のみと指定してパブリケーション pub2 に加えています。サブスクリプション側の t_rep2 テーブルの定義には c2 列がありません。

これまでサブスクリプション側のテーブルに余計な列があっても（それらの列には値が供給されない形で）レプリケーションが可能でしたが、この機能によりサブスクリプション側の列が少ない場合でもレプリケーションを構成できるようになります。

◆ ***サブスクライバでの適用時エラー対応の強化***

ロジカルレプリケーションでは、サブスクリプション側のテーブルにトリガや制約を加えたり、独自にデータを追加することができるため、制約違反などによりサブスクリプション側のテーブルにパブリケーション側の更新データを反映できないケースが発生します。

以下に例を示します。

```
（サブスクリプション側でテーブルに制約を追加）
db1=# ALTER TABLE scm1.t1 ADD CHECK (id != 0);

（パブリケーション側で制約違反になる行を追加、その後、違反にならない行を追加）
db1=# INSERT INTO scm1.t1 (id, val) VALUES (0, 'ZERO');
db1=# INSERT INTO scm1.t1 (id, val) VALUES (2, '正しい値');
```

このようにするとサブスクリプション側では id=0 の行を反映できないのみならず、それ以降の変更が適

Copyright © 2022 SRA OSS LLC All rights reserved.

用できなくなります。サブスクリプション側の PostgreSQL ログには以下のようなメッセージが繰り返し発生します。

```
（サブスクリプション側のサーバログメッセージ）
ERROR:  new row for relation "t1" violates check constraint "t1_id_check"
DETAIL:  Failing row contains (0, ZERO).
CONTEXT:  processing remote data for replication origin "pg_16771" during
"INSERT" for replication target relation "scm1.t1" in transaction 1226820
finished at 2/C00F788
```

本バージョンから、これを解消するシンプルなコマンドが用意されました。エラーメッセージに現れる LSN 番号を引数に与えて ALTER SUBSCRIPTION .. SKIP .. コマンドを実行することで、エラーを引き起こしているトランザクションの適用を飛ばすことができます。

```
（サブスクリプション側で）
db1=# ALTER SUBSCRIPTION sub1 SKIP (LSN = '2/C00F788');
db1=# SELECT * FROM scm1.t1;
 id |    val
----+----------
  1 | A
  2 | 正しい値
(2 rows)
```

従来は同様のことを行うために、より煩雑な手順を行う必要がありました。なお、従来同様に制約を外したり、一意性制約等で競合するデータをサブスクリプション側で削除する、という解消手段も引き続き利用可能です。

さらに本バージョンでは、ロジカルレプリケーションの適用で繰り返しエラーになる状況に陥った場合に自動的にレプリケーションを無効化する機能も追加されています。サブスクリプションにオプション disable_on_error を設定すると、変更データの適用で一度エラーが出ると、その時点でサブスクリプションが無効化されます。このため、解消されるまで繰り返しエラーが出るということを回避できます。

以下に動作例を示します。

```
（サブスクリプション側で）
db1=# ALTER SUBSCRIPTION sub1 SET (disable_on_error = true);
db1=# INSERT INTO scm1.t2 VALUES (100, CURRENT_TIMESTAMP);  -- 先に行を挿入
```


Copyright © 2022 SRA OSS LLC All rights reserved.

```
（パブリケーション側で）
db1=# INSERT INTO scm1.t2 VALUES (100, '2022-08-01 00:00'); -- 主キーが同値
db1=# INSERT INTO scm1.t2 VALUES (101, '2022-08-01 00:00');
   -- これはエラーにならない行だが手前でエラーになるので反映されない

（サブスクリプション側のログメッセージ – 以下が１回だけ出力される）
ERROR:  duplicate key value violates unique constraint "t2_pkey"
DETAIL:  Key (id)=(100) already exists.
CONTEXT:  processing remote data for replication origin "pg_16771" during
"INSERT" for replication target relation "scm1.t2" in transaction 1226822
finished at 2/C00FCD0
LOG:  logical replication subscription "sub1" has been disabled due to an
error

（サブスクリプション側で）
db1=# \dRs                                  # sub1 の Enabled が f になっている
        List of subscriptions
 Name |  Owner   | Enabled | Publication
------+----------+---------+-------------
 sub1 | postgres | f       | {pub1}
 sub2 | postgres | t       | {pub2}
(2 rows)

（サブスクリプション側で、エラートランザクションをスキップして、sub1 を再び有効化）
db1=# ALTER SUBSCRIPTION sub1 SKIP (LSN = '2/C00FCD0');
db1=# ALTER SUBSCRIPTION sub1 ENABLE;

（サブスクリプション側で、次トランザクションの変更が反映される）
db1=# SELECT * FROM scm1.t2;
 id  |           timestamp
-----+-------------------------------
   1 | 2022-08-24 17:49:59.008369+09
 100 | 2022-08-24 22:31:23.671253+09
 101 | 2022-08-01 00:00
(3 rows)
```


Copyright © 2022 SRA OSS LLC All rights reserved.

4.3.2. 呼び出し元権限で実行されるビュー

ビューに security_invoker 属性が追加されました。

security_invoker が設定されている場合、ビューに対する問い合わせでは、問い合わせを実行するユーザの権限で、ビューの元となるテーブルに対するアクセスするようになります。これまではビューへのアクセスは必ずビューの所有者の権限で実行されていました。

以下に動作確認結果を示します。

```
（接続できるデータベースロール user1 と user2 を用意して、
 public スキーマにオブジェクト作成権限を与えます）
db1=# CREATE ROLE user1 LOGIN;
db1=# CREATE ROLE user2 LOGIN;
db1=# GRANT CREATE ON SCHEMA public TO user1, user2;

（user1 で接続して、テーブル t1 と t1 を参照するビュー v1 を作成）
$ psql -U user1 -d db1
db1=> CREATE TABLE t1 (c1 int, c2 text);
db1=> INSERT INTO t1 (c1, c2) values (1, 'aaa');
db1=> CREATE VIEW v1 AS SELECT * FROM t1;

（user2 に v1 の SELECT 権限 を付与）
db1=> GRANT SELECT ON v1 TO user2;
db1=> \q

（security_invoker オプションなしの場合、user2 で v1 を参照できる）
$ psql -d db1 -U user2
db1=> SELECT * FROM v1;
 c1 | c2
----+-----
  1 | aaa
(1 row)

（v1 に security_invoker オプション付与）
$ psql -d db1 -U user1
db1=> ALTER VIEW v1 SET (security_invoker);
db1=> \q
```


Copyright © 2022 SRA OSS LLC All rights reserved.

```
（security_invoker オプション付きビューは、ビューについて GRANT されていてもエラーになる）
$ psql -d db1 -U user2
db1=> SELECT * FROM v1;
ERROR:  permission denied for table t1
```

上記のように、security_invoker オプションなしのビューには、テーブルへの SELECT 権限がなくともビューに対して SELECT 権限を持つロール（user2）でビューを参照できることから、ビューの所有者（user1）の権限でクエリが実行されていることが分かります。対して、security_invoker オプション有のビューには、テーブルへの SELECT 権限がないロール（user2）ではビューが参照できません。

security_invoker 属性を伴うビューを使うことで、テーブルの各ユーザに対する権限設定を尊重するビューを作ることができるようになりました。これまではビューの「問い合わせに名前を付けてテーブルのようにアクセスする機能」と「決まったロールの権限で動作すること」が切り離せませんでした。

4.4. 運用管理

4.4.1. モジュールによる WAL アーカイブ

本バージョンから、WAL アーカイブにライブラリモジュールが利用可能になりました。従来のコマンド呼び出しと比べ、処理オーバーヘッドが軽減されます。

サンプル実装として contrib モジュールに basic_archive モジュールが用意されており、以下のように設定して使用できます。「archive_command = 'test ! -f 《アーカイブディレクトリ》/%f && cp %p 《アーカイブディレクトリ》 /%f'」を指定したのと概ね同じ動作をします。

```
（WAL アーカイブ用ディレクトリを用意）
$ mkdir /var/lib/pgsql/data15a/

（basic_archive を使用する設定をして、再起動して反映）
$ vi $PGDATA/postgresql.conf              # 以下を設定
   archive_mode = on
   archive_library = 'basic_archive'
   archive_command = ''
   basic_archive.archive_directory = '/var/lib/pgsql/data15a'
$ pg_ctl restart
```


Copyright © 2022 SRA OSS LLC All rights reserved.

4.4.2. JSONLOG 形式

設定パラメータ log_destination に jsonlog を指定することで、JSON フォーマットでログを出力することが可能になりました。ログファイルの拡張子は、.json となります。拡張子 .log のログファイルにはログが.json ファイルに出力されている旨を示すメッセージが出力されます。

以下に出力例を示します。

```
$ vi $PGDATA/postgresql.conf    # 以下を設定
   log_destination = jsonlog

$ pg_ctl restart
$ tail -n 3 $PGDATA/log/postgresql-2022-06-27_172350.json
{"timestamp":"2022-06-27 17:23:50.183
JST","pid":1997,"session_id":"62b96916.7cd","line_num":6,"session_start":"20
22-06-27 17:23:50 JST","txid":0,"error_severity":"LOG","message":"database
system is ready to accept
connections","backend_type":"postmaster","query_id":0}
{"timestamp":"2022-06-27 17:28:50.270
JST","pid":1999,"session_id":"62b96916.7cf","line_num":1,"session_start":"20
22-06-27 17:23:50 JST","txid":0,"error_severity":"LOG","message":"checkpoint
starting: time","backend_type":"checkpointer","query_id":0}
{"timestamp":"2022-06-27 17:28:50.290
JST","pid":1999,"session_id":"62b96916.7cf","line_num":2,"session_start":"20
22-06-27 17:23:50 JST","txid":0,"error_severity":"LOG","message":"checkpoint
complete: wrote 3 buffers (0.0%); 0 WAL file(s) added, 0 removed, 0
recycled; write=0.001 s, sync=0.004 s, total=0.021 s; sync files=2,
longest=0.003 s, average=0.002 s; distance=0 kB, estimate=0
kB","backend_type":"checkpointer","query_id":0}

$ cat $PGDATA/log/postgresql-2022-06-27_172350.log
2022-06-27 17:23:50.169 JST [1997] LOG:  ending log output to stderr
2022-06-27 17:23:50.169 JST [1997] HINT:  Future log output will go to log
destination "jsonlog".
```


Copyright © 2022 SRA OSS LLC All rights reserved.

4.4.3. モニタリングビューの追加

本バージョンでは、データベースサーバの状態を取得するためのビューや関数がいくつか追加されています。

関数／ビュー	説明
ビュー pg_ident_file_mappings	pg_ident.conf ファイル内容を出力する。
ビュー pg_stat_subscription_stats	サブスクリプションごとにロジカルレプリケーションの同期エラー発生数を出力。
関数 pg_stat_reset_subscription_stats(oid)	引数にはサブスクリプションの oid を指定。 pg_stat_subscription_stats ビューで出力される値をリセット。
関数 pg_ls_logicalsnapdir()	$PGDATA/pg_logical/snapshots/ ディレクトリ内容（ファイル名、サイズ、変更日付時刻）を出力。
関数 pg_ls_logicalmapdir()	$PGDATA/pg_logical/mappings/ ディレクトリ内容（ファイル名、サイズ、変更日付時刻）を出力。
関数 pg_ls_replslotdir(text)	引数にはスロット名を指定。 $PGDATA/pg_replslot/《スロット名》/ ディレクトリ内容（ファイル名、サイズ、変更日付時刻）を出力。

いくつかのビュー、関数について、実行例を示します。

（pg_ident.conf にエントリを記載、反映して、pg_ident_file_mappings を確認）

```
$ vi $PGDATA/pg_ident.conf
    # MAPNAME        SYSTEM-USERNAME          PG-USERNAME
    root_postgres    root                     postgres
$ pg_reload
$ psql -c 'SELECT * FROM pg_ident_file_mappings;'
 line_number |    map_name    | sys_name | pg_username  | error
-------------+---------------+----------+--------------+--------
          43 | root_postgres | root     | postgres     | *null*
(1 row)
```


Copyright © 2022 SRA OSS LLC All rights reserved.

```
（pg_stat_subscription_stats ビューを確認 - サブスクリプション側のサーバで実行）
db1=# SELECT pg_stat_reset_subscription_stats(16771);
db1=# SELECT * FROM pg_stat_subscription_stats;
 subid | subname | apply_error_count | sync_error_count |          stats_reset
-------+---------+-------------------+------------------+-------------------------------
 16779 | sub2    |                 2 |                0 | *null*
 16771 | sub1    |                 0 |                0 | 2022-08-25 10:45:21.167318+09
(2 rows)
```

5. 非互換変更

本章では PostgreSQL 15 の PostgreSQL 14.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. 実行時統計情報の共有メモリ格納

本バージョンから実行時統計情報が共有メモリ上に格納されるようになりました。サービスを停止する際にはファイルとして書き出し、次に起動するときに再び読み込みます。これまでは、プロセス間通信で統計情報をやり取りして、蓄積した値はファイルに格納していました。そのうえで性能改善のために、その格納ディレクトリを RAM ディスクに割り当てるといったことが行われていました。

このため設定パラメータが以下のように変更されました。

追加／廃止	設定パラメータ	説明
廃止	stats_temp_directory	稼働中の実行時統計上書き出し先ディレクトリ。
追加	stats_fetch_consistency	同トランザクション内で累積値について複数回アクセスを受けた場合の挙動を cache（キャッシュ値を返す／項目ごとにキャッシュを持つ、デフォルト）、none（新しい値を返す）、snapshot（キャッシュ値を返す、全項目についてキャッシュを持つ） から指定。

Copyright © 2022 SRA OSS LLC All rights reserved.

5.2. データベース作成時のデフォルト権限変更

本バージョンより PostgreSQL クラスタ内でデータベースを作成する際の権限設定について、新しいデフォルトが導入されました。新しいデータベースが作成された時、データベース所有者とスーパーユーザ以外は、デフォルトの public スキーマでオブジェクトを作成することができなくなりました。旧バージョンからデータベースをリストアした場合も、この新しい所有者権限が適用されます。

以下に動作例を示します。

```
（PostgreSQL 14 で public スキーマにオブジェクトを作成する例）
$ psql -U postgres -d db1 -c 'CREATE ROLE user1 WITH LOGIN';
CREATE ROLE
$ psql -U user1 -d db1 -c 'CREATE TABLE public.table1(id integer)';
CREATE TABLE

（PostgreSQL 15 で public スキーマにオブジェクトの作成が失敗する例）
$ psql -U postgres -d db1 -c 'CREATE ROLE user1 WITH LOGIN';
CREATE ROLE
$ psql -U user1 -d db1 -c 'CREATE TABLE public.table1(id integer)';
ERROR:  permission denied for schema public
LINE 1: CREATE TABLE public.table1(id integer)
```

template1 データベースにて public スキーマに対する CREATE 権限を PUBLIC に対して付与することで、データベース作成時デフォルトのアクセス許可を従来と同じにできます。

5.3. 排他的バックアップモードの廃止

長らく非推奨となっていた排他的バックアップモードが廃止されました。

排他的バックアップモードでは、オンライン物理バックアップ取得中にデータベースサーバが突然停止すると、ファイルを除去する手順を行わない限り、次にサーバの起動に失敗する可能性があるという問題がありました。

この変更に伴い、pg_start_backup() は pg_backup_start() に、pg_stop_backup() は pg_backup_stop() へと関数名が変更されました。また、pg_backup_start_time() と pg_is_in_backup() のそれぞれの関数は廃止されました。

Copyright © 2022 SRA OSS LLC All rights reserved.

◆ ***basebackup_to_shell でストレージ機能によるバックアップ***

仮想・物理を問わずストレージ層のスナップショット機能を使ってベースバックアップを取得する場合、これまでしばしば排他バックアップモードが使われてきました。排他モードでは pg_start_backup() と pg_stop_backup() を同一セッションで実行する必要が無いため、独立したジョブとして「PostgreSQL バックアップ開始」「ストレージ機能でバックアップ取得」「PostgreSQL バックアップ終了」を順に実行する設計にできるためです。

本バージョン以降の代替手段として、contrib モジュールの basebackup_to_shell を使う方法があります。

basebackup_to_shell はベースバックアップの書き出し先にコマンドを指定するためのものです。指定したコマンドの標準入力にバックアップデータが流し込まれますが、それを使用せずに、ストレージ層のスナップショット機能でバックアップを取ることも可能です。以下に例を示します。

```
（バックアップスクリプトを用意）
$ vi /usr/local/pgsql/15/bin/basebackup.sh
$ chmod +x /usr/local/pgsql/15/bin/basebackup.sh

（モジュールをロードするようにし、コマンドとして用意したスクリプトを指定）
$ vim $PGDATA/postgresql.conf          # 以下の設定を記述
  basebackup_to_shell.command = '/usr/local/pgsql/15/bin/basebackup.sh %f %d'
  shared_preload_libraries = 'basebackup_to_shell'
$ pg_ctl restart
```

バックアップスクリプト内容としては以下のようになります。%f にベースバックアップ対象が渡され、対象要素ごとに何回かスクリプトが呼ばれます。本例では 1 回だけ処理したいため、%f がデータベースクラスタ本体の「base.tar」であるときだけ処理を行うようにしています。

```
#!/bin/bash
cat - > /dev/null                # 標準入力に流し込まれるデータは捨てる
if [ "${1}" = "base.tar" ]; then
  《ここにストレージ層以下の機能を使ったスナップショット取得のコマンドを記述》
fi
exit 0
```

用意したスクリプトによるベースバックアップ取得には以下のコマンドを実行します。

```
$ pg_basebackup -t shell: -X none
```

「-t shell:」を指定すると「basebackup_to_shell.command」で指定したコマンドが使われます。「:」の右に文

Copyright © 2022 SRA OSS LLC All rights reserved.

字列を付加すると、コマンド文字列中「%d」に展開されますが、本例では使用していません。ベースバックアップ取得中に生じた WAL をストリームで取得するデフォルト動作（-X stream）はサポートされないため、ここでは -X none を指定してます。

5.4. *Python 2* のサポート終了

PL/Python 拡張モジュールの plpythonu、plpython2u が廃止されました。Python 2.x 系統のサポート終了に伴い、上記拡張モジュールも廃止されました。

回避策は無いため、PostgreSQL 15 で使うためには PL/Python スクリプト実装を Python 3.x に移行する必要があります。

5.5. *array_to_tsvector* 関数の変更

array_to_tsvector() 関数に空文字列の配列要素が渡された場合にエラーを返すようになりました。旧バージョンから移行した場合に、新バージョンでエラーが生じる可能性があります。また、式インデックスで array_to_tsvector() 関数を使っていた場合には、ダンプをリストアする段階でエラーが発生する可能性があります。

（PostgreSQL 14 での実行例）

```
db1=# SELECT array_to_tsvector(ARRAY['', 'foo']);
 array_to_tsvector
-------------------
 '' 'foo'
(1 row)
```

（PostgreSQL 15 での実行例）

```
db1=# SELECT array_to_tsvector(ARRAY['', 'foo']);
ERROR:  lexeme array may not contain empty strings
```

対処としては移行前に空文字列の要素を取り除くことです。tsvector 上の空要素はテキスト検索では役に立たないため、無くても問題ありません。

Copyright © 2022 SRA OSS LLC All rights reserved.

6. 免責事項

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

Copyright © 2022 SRA OSS LLC All rights reserved.