PostgreSQL 9.6 beta3 検証レポート

1.0 版
2016 年 8 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....2
2. 概要....2
3. 検証のためのセットアップ....2
3.1. ソフトウェア入手....2
3.2. 検証環境....2
3.3. インストール....3
4. 主要な追加機能....4
4.1. パラレルクエリ....4
4.1.1. パラレル処理の動作概要....4
4.1.2. 設定パラメータ....6
4.1.3. 動作検証....6
4.1.4. パラレルクエリ - その他....11
4.2. ストリーミングレプリケーションの拡張....12
4.2.1. 複数の同期スタンバイに対応....12
4.2.2. 変更データ適用を保証するモード....13
4.2.3. pg_basebackup の --slot オプション....14
4.3. VACUUM における XID 凍結処理の改善....15
4.3.1. XID 凍結 - 従来動作の問題点....15
4.3.2. XID 凍結 - 新実装の仕組み....16
4.3.3. XID 凍結でのスキップ動作の検証....17
4.4. 全文検索におけるフレーズ検索....18
4.5. 多 CPU 同時実行における性能改善....20
4.6. postgres_fdw の改善....21
4.6.1. 結合/ソートをリモートサーバ側で処理....21
4.6.2. DELETE/UPDATE をリモート側だけで実行....25
4.6.3. 関数・演算子のリモート処理....26
4.6.4. postgres_fdw - その他の改善....27
4.7. 各種モニタリングビューの拡張....27
4.7.1. pg_stat_activity....27
4.7.2. pg_stat_progress_vacuum....28
4.8. 古いスナップショットの強制廃棄....28
5. 免責事項....29

1. はじめに

本文書は PostgreSQL 9.6 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 9.6 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。
2016 年 7 月 21 日リリースされた PostgreSQL 9.6 beta3 を使用して検証を行って、本文書を作成しています。

2. 概要

PostgreSQL 9.6 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。

- パラレル問い合わせ処理
- VACUUM における XID 凍結処理の改善
- ストリーミングレプリケーションの拡張
- 全文検索におけるフレーズ検索
- 多 CPU サーバにおける性能向上
- postgres_fdw の改善

この他にも細かな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 9.6 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。また、これらは beta3 バージョン時点での機能実装です。バージョン 9.6 の本リリースまでにいくつか削除や変更される可能性があります。

```
https://www.postgresql.org/docs/devel/static/release-9-6.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 9.6 のベータ版は以下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。

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

3.2. 検証環境

検証環境として、HP ProLient MicroServer 上の CentOS 6.8 / x86_64 等を使用しました。いずれも小型サーバマシンであり、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。

3.3. インストール

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。

```
$ cd postgresql-9.6beta3
$ ./configure --prefix=/usr/local/pgsql/9.6.b3 --enable-debug
$ 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 > 9.6.b3.env <<'EOF'
PGHOME=/usr/local/pgsql/9.6.b3
export PATH=$PGHOME/bin:$PATH
export LD_LIBRARY_PATH=$PGHOME/lib:$LD_LIBRARY_PATH
export PGDATA=/usr/local/pgsql/data9.6.b3
EOF
$ . 9.6.b3.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。

```
$ cat >> $PGDATA/postgresql.conf << EOF
logging_collector = on
log_line_prefix = '%t %p '
EOF
```

PostgreSQL を起動します。

```
$ pg_ctl -w start
```

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、合わ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ベータ版に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/9.6beta3/share/doc/html/
```

また、以下 URL にて開発中バージョンのマニュアル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/9.6/static/
```

4.1. パラレルクエリ

9.6 からパラレルクエリのサポートが始まりました。現在のところ、参照のみのクエリでテーブルをシーケンシャルスキャンする時、ハッシュ結合や入れ子ループ結合(Hash Join, Nested Loop)や集約(Aggregate)で並列化することが可能です。

4.1.1. パラレル処理の動作概要

パラレル処理は複数のワーカプロセスを使って実現されています。内部処理として PostgreSQL9.4 から導入されたバックグラウンドワーカプロセスの仕組みを使っています。

複数のワーカプロセスを使って、操作対象となったリレーションを同時並列にアクセスすることで処理効率を向上させます。元のバックエンドプロセスはグループリーダとなり、各ワーカプロセスを起動し、処理を振り分け、結果の集計処理(Gather)を行ないます。(図 1)

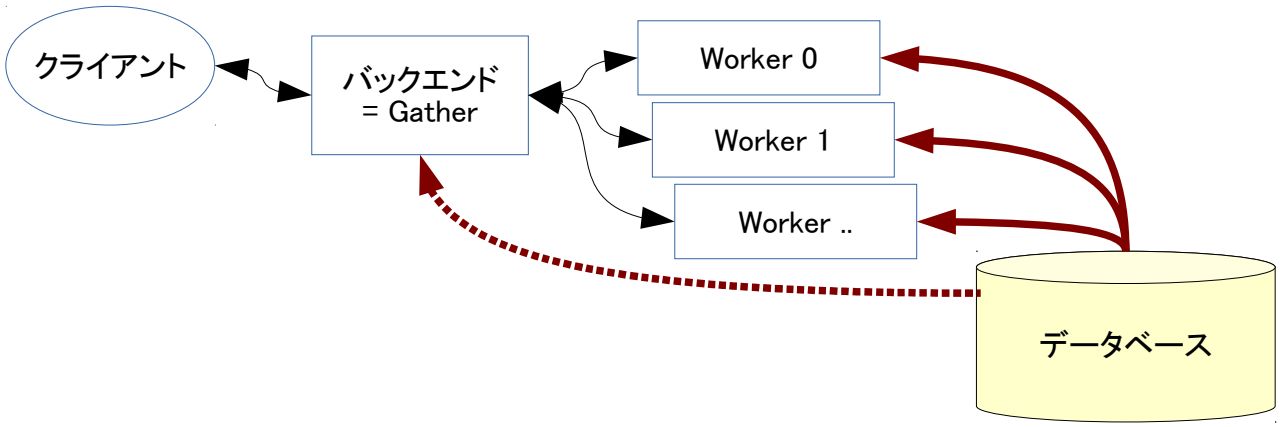

図 *1:* パラレル処理概要

パラレル処理を行なうかはプランナによって決定されます。プランナが参照する GUC パラメータに付いては次節を参考にして下さい。

Seq Scan 操作で、対象となるリレーションのサイズと min_parallel_relation_size を元にワーカプロセスの個数を決めています。対数オーダーの計算がされており、デフォルトの min_parallel_relation_size = 8M なら、3 倍の 3000 ページ(24M バイト)以上のリレーションサイズなら並列数 2 でのプラン検索が行なわれます。現在は対数計算で使う係数として 3 が使われていまう。8M=1000 ページ以上なら 1 並列、3000 ページ以上なら 2、9000 で 3、27000 で 4、81000 は 5、...と並列数を増やしていく方法になっています。(図 2)

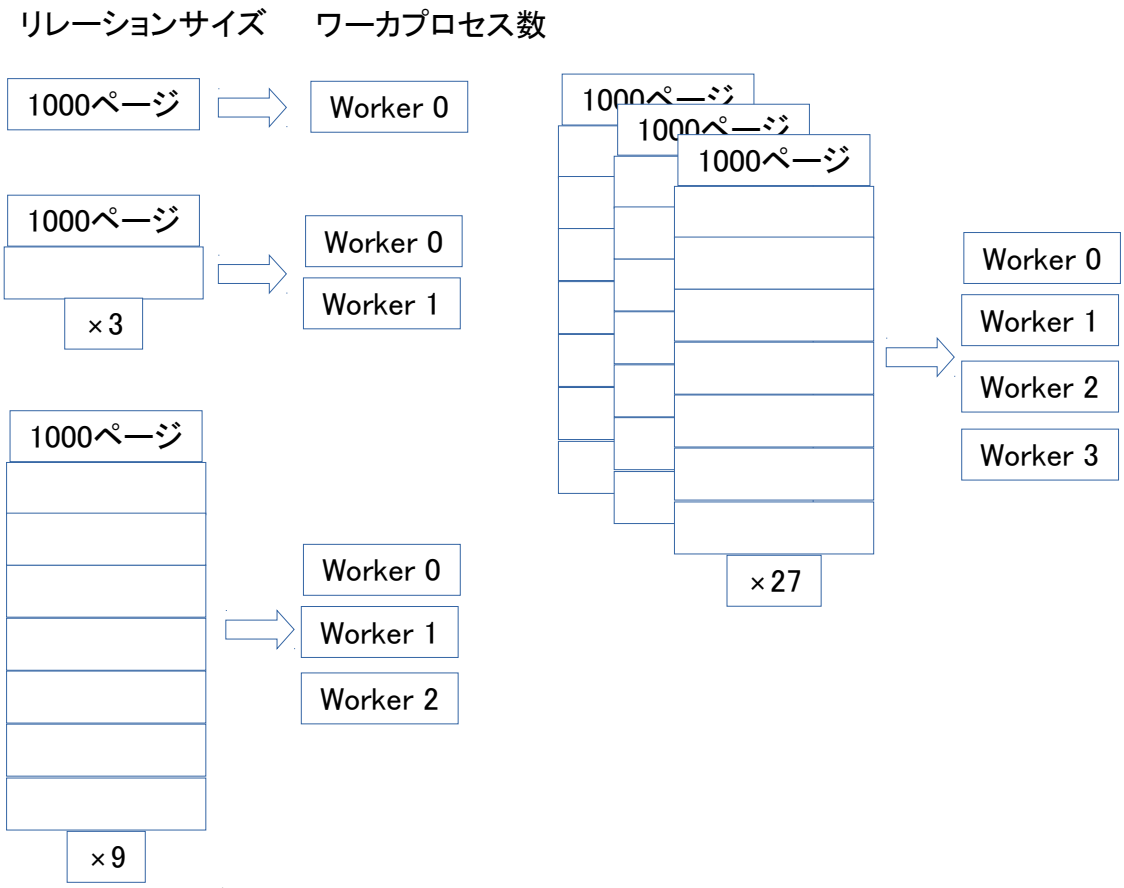

図 2: リレーションサイズとワーカプロセス数の関係

リレーションサイズを元にした計算をさせずに、ワーカ数を設定することもできます。ALTER TABLE を使って指定すると、リレーションサイズに関係なくワーカプロセス数を指定できます。

```
ALTER TABLE tbl  SET (parallel_worker = 8);
```

ただし、ひとつのクエリで使えるワーカ数の上限 max_parallel_workers_per_gather を越えることはありません。また、プランナがパラレルクエリを採用するかどうかは、これら指定だけでは決まらない点に留意し

てください。

4.1.2. 設定パラメータ

新たな GUC パラメータが増えています。パラレルクエリに関連する GUC を以下にあげます。

名称	デフォルト	概要
max_parallel_workers_per_gather	2	ひとつの Gather に対するワーカプロセスの最大数
force_parallel_mode	off	強制的にパラレルクエリを行なわせる（テスト用）
parallel_setup_cost	1000	ワーカプロセスの起動にかかるコスト
parallel_tuple_cost	0.1	1 行の結果をワーカプロセスから転送するコスト
min_parallel_relation_size	8M	パラレル処理する最小のリレーションサイズ
max_worker_processes	8	(9.4 から)バックグラウンドワーカプロセスの最大数

これら設定は、パラレルクエリのプラン選択と、ワーカプロセス数に影響をあたえます。

□ *パラレルクエリのプラン選択*

ワーカプロセスを起動するためのコスト（parallel_setup_cost）や、各ワーカが計算した結果を集計するコスト（parallel_tuple_cost）を加えて、プラン全体のコストが決定されます。そのためワーカを起動する方が遅かったり、予測される行数が多く各ワーカからの集計コストが高くなる（parallel_tuple_cost * 予測行数分のコストが発生する）と判断されると、パラレル処理が選択されない事になります。

□ *ワーカプロセス*

システム全体のバックグラウンドワーカの総数は max_woker_processes になります。同時実行されている別のセッションでもワーカプロセスが使われるなど、パラレルクエリで必要とするワーカが不足する場合は、ワーカプロセスの数を減らして実行されます。同時に多数のセッションを実行すると、パラレル処理された時と、されなかった時で、応答性能にばらつきが出る可能性があります。

各ワーカプロセスは、必要に応じて作業メモリ（work_mem）を使います。プロセス数やメモリ使用量の見積りの際には max_worker_processes も考慮する必要があります。

4.1.3. 動作検証

pgbench で生成したデータを使って、パラレルクエリの動作を確認します。

□ *準備*

pgbench -i -s 20 でデータを生成すると pgbench_accounts のサイズが約 256M バイトになり、最大 4 個のワーカで処理する程度のデータ量になります（min_parallel_relation_size=8M なら 27000 ページ=216MB 以

上でワーカ数が4）。

```
$ pgbench -i -s 20
$ psql
db=# ¥d+
                    List of relations
 Schema |       Name       | Type  |  Owner   |  Size   | Description
--------+------------------+-------+----------+---------+-------------
 public | pgbench_accounts | table | postgres | 256 MB  |
 public | pgbench_branches | table | postgres | 40 kB   |
 public | pgbench_history  | table | postgres | 0 bytes |
 public | pgbench_tellers  | table | postgres | 48 kB   |
(4 rows)
```

GUC 設定はデフォルトのままとします。

```
max_worker_processes = 8
max_parallel_workers_per_gather = 2
min_parallel_relation_size = 8M
parallel_setup_cost = 1000
parallel_tuple_cost = 0.1
force_parallel_mode = off
```

□ *EXPLAIN による表示*

EXPLAIN (ANALYZE, VERBOSE)を使うことで、各ワーカプロセスの実行時間が表示されます。
Gather と名付けられたプランの下でワーカプロセスが動作する様子を確認することができます。

○ Parallel Seq Scan

パラレルクエリの基本となる処理です。通常の Seq Scan 処理が行なわれる部分を置き換えます。
Parallel Seq Scan はページ内のデータを各ワーカプロセスに割り当てて処理させます。1ページ分の処理が終ったら次のページに進み、データをワーカに割り当てる処理を繰り返します。
Gather は下位のパラレル処理からの結果を集計するノードになります。

```
db=# EXPLAIN (ANALYZE, VERBOSE) SELECT * FROM pgbench_accounts WHERE abalance >= 10000;
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
```

```
-----------------------------------------
 Gather  (cost=1000.00..44203.77 rows=1 width=97) (actual time=102.834..102.834 rows=0
loops=1)
   Output: aid, bid, abalance, filler
   Workers Planned: 2
   Workers Launched: 2
   ->  Parallel Seq Scan on public.pgbench_accounts  (cost=0.00..43203.67 rows=1 width=97)
(actual time=97.912..97.912 rows=0 loops=3)
         Output: aid, bid, abalance, filler
         Filter: (pgbench_accounts.abalance >= 10000)
         Rows Removed by Filter: 666667
         Worker 0: actual time=98.090..98.090 rows=0 loops=1
         Worker 1: actual time=93.319..93.319 rows=0 loops=1
 Planning time: 0.086 ms
 Execution time: 103.757 ms
(12 rows)
```

○ 集約

Parallel Seq Scan から得られたデータを並列に集約することができます。以下に例を示します。

集約は、各部分毎に行なわれる処理（Partial Aggregate）と、全体の結果をまとめる処理（Finalize Aggregate）で行なわれます。集約関数は PARALELL SAFE としてパラレル処理に対応している必要があります。これについては後述いたします。

```
db=# EXPLAIN (ANALYZE, VERBOSE) SELECT avg(abalance) FROM pgbench_accounts;
                                                        QUERY PLAN
-------------------------------------------------------------------------------------------
-------------------------------------------------------------------
 Finalize Aggregate  (cost=44203.88..44203.89 rows=1 width=32) (actual
time=274.627..274.627 rows=1 loops=1)
   Output: avg(abalance)
   ->  Gather  (cost=44203.67..44203.88 rows=2 width=32) (actual time=274.377..274.564
rows=3 loops=1)
         Output: (PARTIAL avg(abalance))
         Workers Planned: 2
         Workers Launched: 2
```

```
      ->  Partial Aggregate  (cost=43203.67..43203.68 rows=1 width=32) (actual
time=267.440..267.440 rows=1 loops=3)
            Output: PARTIAL avg(abalance)
            Worker 0: actual time=259.645..259.645 rows=1 loops=1
            Worker 1: actual time=268.961..268.961 rows=1 loops=1
            ->  Parallel Seq Scan on public.pgbench_accounts  (cost=0.00..41120.33
rows=833333 width=4) (actual time=0.090..193.384 rows=666667 loops=3)
                  Output: abalance
                  Worker 0: actual time=0.053..196.047 rows=516670 loops=1
                  Worker 1: actual time=0.023..193.935 rows=973560 loops=1
 Planning time: 0.530 ms
 Execution time: 275.931 ms
(16 rows)
```

○ JOIN

Parallel Seq Scan のノードは、他のノードと nested loop や hash join であれば並列に結合処理を実施することができます。以下に例を示します。

```
db=# EXPLAIN (ANALYZE, VERBOSE) SELECT * FROM pgbench_accounts AS a
      JOIN pgbench_branches AS b ON a.bid = b.bid WHERE b.bbalance >= 1000;
                                                QUERY PLAN
-----------------------------------------------------------------------------------------------
-----------------------------------------------------
 Gather  (cost=1001.26..55663.26 rows=100000 width=461) (actual time=9.723..9.723 rows=0
loops=1)
  Output: a.aid, a.bid, a.abalance, a.filler, b.bid, b.bbalance, b.filler
  Workers Planned: 2
  Workers Launched: 2
  ->  Hash Join  (cost=1.26..44663.26 rows=100000 width=461) (actual time=0.510..0.510
rows=0 loops=3)
      Output: a.aid, a.bid, a.abalance, a.filler, b.bid, b.bbalance, b.filler
      Hash Cond: (a.bid = b.bid)
      Worker 0: actual time=0.732..0.732 rows=0 loops=1
      Worker 1: actual time=0.674..0.674 rows=0 loops=1
      ->  Parallel Seq Scan on public.pgbench_accounts a  (cost=0.00..41120.33 rows=833333
```

```
width=97) (actual time=0.072..0.072 rows=1 loops=3)
          Output: a.aid, a.bid, a.abalance, a.filler
          Worker 0: actual time=0.059..0.059 rows=1 loops=1
          Worker 1: actual time=0.059..0.059 rows=1 loops=1
      ->  Hash  (cost=1.25..1.25 rows=1 width=364) (actual time=0.073..0.073 rows=0
loops=3)
          Output: b.bid, b.bbalance, b.filler
          Buckets: 1024  Batches: 1  Memory Usage: 8kB
          Worker 0: actual time=0.156..0.156 rows=0 loops=1
          Worker 1: actual time=0.047..0.047 rows=0 loops=1
          ->  Seq Scan on public.pgbench_branches b  (cost=0.00..1.25 rows=1 width=364)
(actual time=0.072..0.072 rows=0 loops=3)
                Output: b.bid, b.bbalance, b.filler
                Filter: (b.bbalance >= 1000)
                Rows Removed by Filter: 20
                Worker 0: actual time=0.155..0.155 rows=0 loops=1
                Worker 1: actual time=0.047..0.047 rows=0 loops=1
 Planning time: 0.351 ms
 Execution time: 11.206 ms
(26 rows)
```

□ *各プロセスの動作状況の確認*

現在の仕組みでは詳細な情報が記録されません。

ps コマンドで、ワーカプロセスの動作状況を見ても、クエリ元となるバックエンドプロセスの PID 情報しか見えません。

pg_stat_activity にもワーカプロセスに対応する情報が入りますが、ワーカプロセス自身の基本的な情報しかありません。

```
$ ps axf
(抜粋)
 5293 ?        Rs     0:00  ¥_ postgres: postgres db 192.168.210.1(33643) SELECT
 5299 ?        Ss     0:00  ¥_ postgres: bgworker: parallel worker for PID 5293
 5300 ?        Ss     0:00  ¥_ postgres: bgworker: parallel worker for PID 5293

=# SELECT * FROM pg_stat_activity;
-[ RECORD 1 ]----+------------------------------
```

```
datid            | 16578
datname          | db
pid              | 5778
usesysid         | 16557
usename          | postgres
application_name | pgbench
client_addr      |
client_hostname  |
client_port      |
backend_start    | 2016-07-28 20:51:17.450156+09
xact_start       | 2016-07-28 20:51:17.405282+09
query_start      |
state_change     |
wait_event_type  |
wait_event       |
state            |
backend_xid      |
backend_xmin     | 1849
query            |
```

4.1.4. パラレルクエリ - その他

パラレルクエリ処理に関連して、関数の属性が増えました。

```
CREATE FUNCTION name (…) … PARALLEL { UNSAFE | RESTRICTED | SAFE }
```

psql の \df+ で確認することができます。

UNSAFE はパラレル実行できない関数に指定します。RESTRICTED はグループリーダ（上位の Gather ノード）なら実行できる関数になります。副作用のある関数などは UNSAFE 指定をする必要があります 。PARALLEL UNSAFE な関数が使われていたら、パラレルクエリは使われません。

また、トランザクション隔離レベル SERIALIZABLE ではパラレルクエリは使えません。

4.2. ストリーミングレプリケーションの拡張

PostgreSQL9.0 からストリーミングレプリケーション機能が追加され、PostgreSQL 本体だけでトランザクションログデータ（WAL データ）の転送に基づくマスタ・スレーブ型のレプリケーションが実現可能となりました。9.6 バージョンでは、ストリーミングレプリケーションについていくつか重要な拡張が適用されています。

4.2.1. 複数の同期スタンバイに対応

9.5 まではストリーミングレプリケーションの同期スタンバイは 1 つだけでした。設定パラメータ synchronous_standby_names に複数指定したとしても、そのうちの一つは同期スタンバイ（sync）になりますが、残りは同期スタンバイの交代候補である「潜在的な同期スタンバイ （potential）」となります。

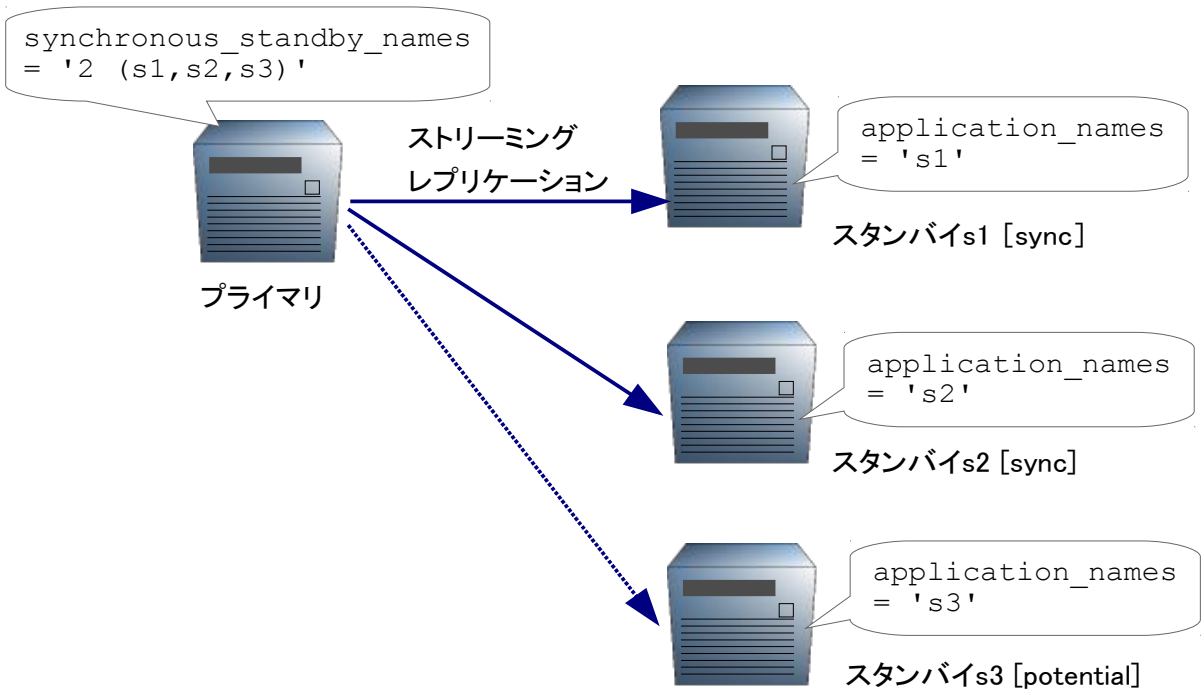

図 3: 複数の同期スタンバイを指定

9.6 では図 3 のように複数のサーバを同期スタンバイとすることができるようになりました。

複数の同期スタンバイを指定するため、synchronous_standby_names に新たな書式が加わりました。下記のように「《同期スタンバイの数》(スタンバイ名, ...)」として、同期スタンバイの数を指定します。

（従来からの指定方法 – 9.6 で従来通りの書き方も可能です）

```
synchronous_standby_names = 's1, s2, s3'          # 3 つのうち 1 つが同期スタンバイとなる
```

（新しい指定方法）

```
synchronous_standby_names = '1 (s1,s2,s3)'      # 3つのうち1つを同期スタンバイとする
synchronous_standby_names = '2 (s1,s2,s3)'      # 3つのうち2つを同期スタンバイとする
```

3つのうち1つの同期スタンバイがダウンした場合には、潜在的な同期スタンバイ（potential）であったサーバが同期スタンバイに昇格します。

複数の同期スタンバイを設定することにより、データ保全性をより高いレベルで保証する構成を組むことができます。また、変更データ適用を保証させる設定（「4.2.2 変更データ適用を保証するモード」参照）を与えた複数の同期スタンバイサーバを持つことで、参照負荷分散を狙った構成を組むのにも役立ちます。

4.2.2. 変更データ適用を保証するモード

従来、ストリーミングレプリケーションで同期モードを設定しても、それは、変更差分情報を持つWALデータがスタンバイサーバへの転送、もしくは、転送後のストレージ書き込みを保証するのみでした。データ保全という意味では有用ですが、プライマリでのコミット完了後でも、スタンバイの問い合わせ結果に変更が反映されているとは限りません。

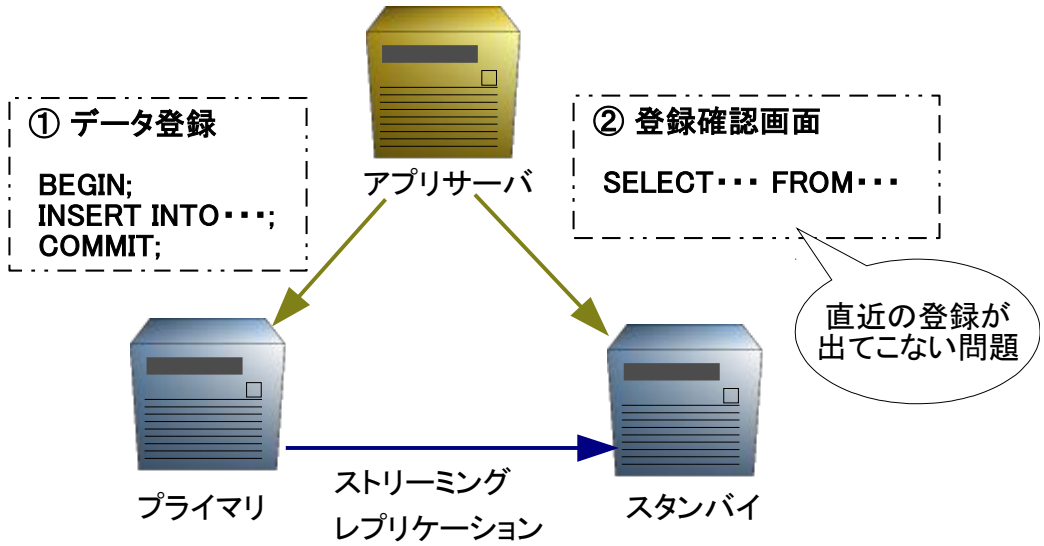

図 4: 参照更新振り分けでの問題点

図6のような、更新問い合わせをプライマリに、参照問い合わせをスタンバイに振り分けるシステム設計とした場合、変更直後にスタンバイサーバにて参照問い合わせを行うと、変更前の結果が返ってしまう問題がありました。入力フォーム画面で登録ボタンを押した後、「登録内容の確認」が表示される、といった動作はアプリケーションで頻繁にあらわれます。

9.6では、設定パラメータ synchronous_commit に設定可能な選択肢に remote_apply が加わりました。同期モードでストリーミングレプリケーションを構成している場合に remote_apply を指定すると、コミット処

理をするにあたり、WAL データを転送して、スタンバイからデータベースに内容を反映させたという報告を受けてから、応答を返すようになります（図 2）。

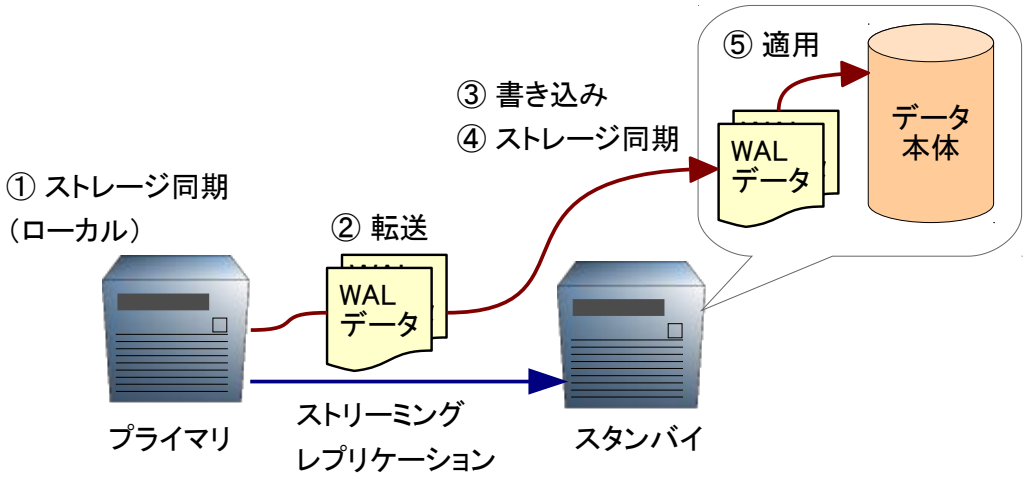

図 *5:* 同期レプリケーションにおける *WAL* 転送と適用

synchronous_commit の設定と動作は以下表の通りです。なお、同期レプリケーションモードでない場合（synchronous_standby_names に設定が無い場合）には、off 以外のどれを設定してもローカルストレージ書き込みを待つ動作となります。表内の数字は図 5 に対応しています。

synchronous_commit 設定	同期レプリケーションのコミット時の動作
off	WAL データのローカルストレージ書き込みも、同期スタンバイへの転送も、待たない。
local	WAL データのローカルストレージ書き込みを待つ。［①］
remote_write	WAL データのローカルストレージ書き込みと、同期スタンバイへ転送を待つ。［① ② ③ ］
remote_apply	WAL データのローカルストレージ書き込みと、同期スタンバイへの転送および同期スタンバイでの適用を待つ。［① ② ③ ⑤］
on	WAL データのローカルストレージ書き込みと、同期スタンバイへの転送および同期スタンバイでのストレージ書き込みを待つ。［① ② ③ ④］

4.2.3. pg_basebackup の *--slot* オプション

pg_basebackup に --slot オプションが追加されました。

ストリーミングを通して WAL ファイルも合わせて取得する「-X stream」オプションと共に使用します。ベースバックアップ用のレプリケーションスロットを作成済みのサーバに対して、そのスロット名を「--slot《スロット名》」でオプション指定して pg_basebackup 実行することで、以下のような必要な WAL ファイルが既に削除されてしまったためにベースバックアップ取得に失敗するケースを防ぎます。

```
$ pg_basebackup -D /data/pg/basebackupdir -X stream
pg_basebackup: unexpected termination of replication stream: ERROR:  requested WAL segment
000000010000000000000016 has already been removed
```

従来は「-X stream」を指定することが、「ERROR: requested WAL segment ... has already been removed」を防ぐ最善の方法でしたが、より確実な方法が提供されました。この機能を実現するために内部的にはストリーミングレプリケーションプロトコルにおけるコマンドの拡張が行われました。

4.3. *VACUUM* における *XID* 凍結処理の改善

PostgreSQL の運用には VACUUM 処理が必要です。VACUUM には、データ格納サイズの肥大化を防ぐ「不要領域の整理」、XID（トランザクション ID）値の周回を防ぐ「XID 凍結」、プランナが使う統計情報を更新する「テーブル統計情報の更新」という 3 つの役割があります。このうち「XID 凍結」の処理について、本バージョンで改善されました。

4.3.1. *XID* 凍結 - 従来動作の問題点

多くのサーバで自動 VACUUM 機能を通して VACUUM 処理を実行しています。自動 VACUUM は、不要行バージョン（一つの行でも複数バージョンが同時に存在するので「行バージョン」と呼びます）の数・割合が増えたのを契機に駆動する場合と、データ上の最も古い XID と現在の XID との差が広がっていることを契機に駆動する場合があります。後者の場合には XID 凍結の VACUUM 処理が行われます。

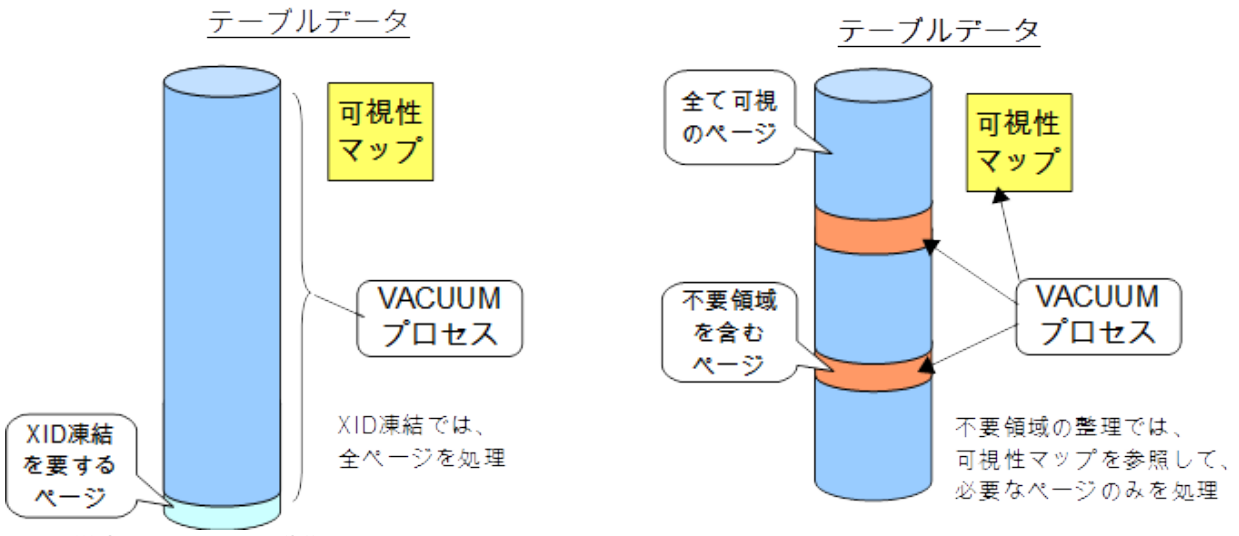

図 *6:* 従来の *VACUUM* 動作

通常の VACUUM 処理では、可視性マップ（ビジビリティマップ、Visibility Map）を使って、処理をしなくてよいと分かっているページを予め覚えておいて、これらをスキップします。ところが、XID 凍結の VACUUM 処理では、対象テーブルの全てのページを走査します。このため大きなテーブルでは処理に時間を要しました（図 6）。

自動 VACUUM における XID 凍結の VACUUM 処理では、閾値となる設定 autovacuum_freeze_max_age はデフォルト が 2 億トランザクションと大きい一方、サイズは大きけれども平素アクセス量が少ないというテーブルも対象となりうるため、「忘れたころに突然対象に負荷の高い処理が走る」という挙動が生じて、システム運用者を悩ませていました。

なお、vacuum_freeze_table_age （デフォルト 1.5 億トランザクション）は、自動 VACUUM でどのテーブルを処理するかの選択には使われませんが、VACUUM をすると決まったあと、合わせて XID 凍結処理をするかどうかの判定に使われます。更新と自動 VACUUM がよく行われているテーブルで、あるときだけ XID 凍結を伴う重い処理が実行されたという挙動をひき起こす閾値となります。

4.3.2. XID 凍結 - 新実装の仕組み

PostgreSQL 9.6 では、VACUUM の XID 凍結処理においても、可視性マップを使って、処理をしなくてよいと分かっているページをスキップするようになりました。これは自動 VACUUM、コマンドの VACUUM とも共通の動作です。これを実現するために可視性マップは、従来からの「ページ内容が全て可視か？（=前回 VACUUM 以降の変更ない）」ということに加えて、「ページ内容がすべて XID 凍結済みか？」という情報も持つようになりました。

これにより、既に大部分のページは XID 凍結済みとなっているテーブルに対する、VACUUM での XID 凍結処理が、短時間で終了するようになります。

DISABLE_PAGE_SKIPPING

コマンドの VACUUM では DISABLE_PAGE_SKIPPING オプションが加わりました。この指定をすると 、VACUUM 処理において以下のページスキップをしなくなります。

- 不要領域の整理が不要なのでスキップ
- XID 凍結が不要なのでスキップ
- 他のプロセスが当該ページを使用中なのでスキップ

本オプションは通常は使用しません。壊れた、あるいは、壊れているかもしれない可視性マップを再作成したいときに使用します。

4.3.3. XID 凍結でのスキップ動作の検証

以下に、大部分は古い変動のないデータが格納されたテーブルで、XID 凍結の自動 VACUUM が動作する例を、PostgreSQL9.5 と 9.6 で実行した結果を示します。

```
-- 「古くて量が多いデータ」を作成。VACUUM FREEZE で XID 凍結済み状態にする。
db1=# CREATE TABLE t_great_history (id int primary key, memo text, ts timestamp(0));
CREATE TABLE
db1=# INSERT INTO t_great_history SELECT g, 'The old historical data,  ' || g,
       '1970-04-01'::timestamp(0) + (g || 'min')::interval
        FROM generate_series(1, 5000000) as g;
db1=# VACUUM FREEZE t_great_history;
VACUUM

-- １件だけデータ追加。
db1=# INSERT INTO t_great_history VALUES (5000001, 'recent one', now());
INSERT 0 1

db1=# SELECT * FROM t_great_history WHERE id >=  5000000;
  id    |            memo             |         ts
--------+-----------------------------+---------------------
 5000000 | old historical data 5000000 | 1979-10-03 05:20:00
 5000001 | recent one                  | 2016-07-08 10:59:43
(2 rows)
db1=# ¥q
```

```
-- pgbench コマンドによる他テーブルの更新で XID を進める。結果を採取するために、
   autovacuum_freeze_max_age = 2000000 、log_autovacuum_min_duration = 0 と設定しておく。
$ pgbench -i db1
$ pgbench -c 4 -t 500000 -n db1 -N </dev/null &>pgbench.out &
```

PostgreSQL 9.5 では以下のように t_great_history に全スキャンの自動 VACUUM が走ります。pages: や tuples: にあらわれる不要領域の整理は 0 件ですので、これは XID 凍結のための実行であることがわかります。また、スキップは行われず、全件がスキャンされています。

```
LOG:  automatic vacuum of table "db1.public.t_great_history": index scans: 0
      pages: 0 removed, 41667 remain, 0 skipped due to pins
      tuples: 0 removed, 5000001 remain, 0 are dead but not yet removable
      buffer usage: 41667 hits, 56116 misses, 5 dirtied
      avg read rate: 4.222 MB/s, avg write rate: 0.000 MB/s
      system usage: CPU 1.52s/1.57u sec elapsed 111.88 sec
```

PostgreSQL 9.6 では以下のように 自動 VACUUM での XID 凍結にスキップが効いています。

```
LOG:  automatic vacuum of table "db1.public.t_great_history": index scans: 0
      pages: 0 removed, 41667 remain, 0 skipped due to pins, 41666 skipped frozen
      tuples: 0 removed, 4999912 remain, 0 are dead but not yet removable
      buffer usage: 55 hits, 13712 misses, 0 dirtied
      avg read rate: 7.620 MB/s, avg write rate: 0.000 MB/s
      system usage: CPU 0.16s/0.09u sec elapsed 14.05 sec
```

4.4. 全文検索におけるフレーズ検索

全文検索でフレーズ検索が拡張されました。

フレーズ検索とは、単語の並びに対して検索するものです。従来から 問い合わせデータ（tsquery 型データ）の中で以下のようにクオートでくくった複数単語の並びを記述することは可能でした。しかし、これは同義語辞書による置き換えを働かせるには有用ですが、辞書に当てはまらない場合には、結局複数単語の AND 条件検索となっていました。

```
-- 以下のような問い合わせをしたとしても、クオートで囲った部分が辞書になければ、
db1=# SELECT id, doc FROM t_doc
    WHERE tsv @@ to_tsquery('english', E'¥'design of the database¥'');
```

```
-- to_tsquery 部分は以下のように評価され、2 単語の単なる AND 条件検索になります。
db1=# SELECT to_tsquery('english', E'¥'design of the database¥'');
   to_tsquery
---------------------
 'design' & 'databas'
(1 row)
```

9.6 から tsquery データ型の表現で <-> が追加されました。これにより前の単語から続くフレーズであるという指定ができます。以下の実行例で、<3> となっているのは、<-> が 3 つ並んでいるという意味です。全文検索において英語では of や the は無視されるストップワードになっていますので、間に 2 語挟むという条件だけ保持されています。<N> という表現は to_tsquery に与える文字列に記述することもできます。

また、フレーズの tsquery データを生成する phraseto_tsquery 関数も追加されました。さらに、tsquery データ型同士をつなげてフレーズを構成する PostgreSQL の演算子としての <-> も追加されています。

```
-- <-> を使って design のあと 2 語挟んで database が続くという条件が指定できます。
db1=# SELECT to_tsquery('english', 'design <-> of <-> the <-> database');
    to_tsquery
-----------------------
 'design' <3> 'databas'
(1 row)

-- フレーズの tsquery データを生成する phraseto_tsquery 関数も用意されています。
db1=# SELECT phraseto_tsquery('full text searching');
     phraseto_tsquery
-------------------------------
 'full' <-> 'text' <-> 'search'
(1 row)

-- 演算子 <-> でフレーズを構成することもできます。
db1=# SELECT to_tsquery('full') <-> to_tsquery('text');
   ?column?
-------------------
 'full' <-> 'text'
(1 row)
```

4.5. 多 CPU 同時実行における性能改善

9.6 では多 CPU マシンでの同時実行動作における性能改善を狙った改修がいくつか加わっています。

本レポートでは、高スペックマシンでの実機検証は実施していません。本節では、どのような仕組みが入ったかを説明するのみとします。

以下の図 7 は、PostgreSQL の内部構造と、9.6 で加えられた多 CPU マシンでの同時実行動作における改善点を示したものです。PostgreSQL の構成要素として、プロセス群（図の右側）、いくつかの種類の共有メモリ（図の左側上）、いくつかのストレージ上のファイル群（図の左下）を模式的に記載しています。

図中の「TX 管理」とはトランザクション管理のことです。pg_clog や pg_subtrans などのファイル群とそれらに対するバッファ用メモリ領域をあらわしています。

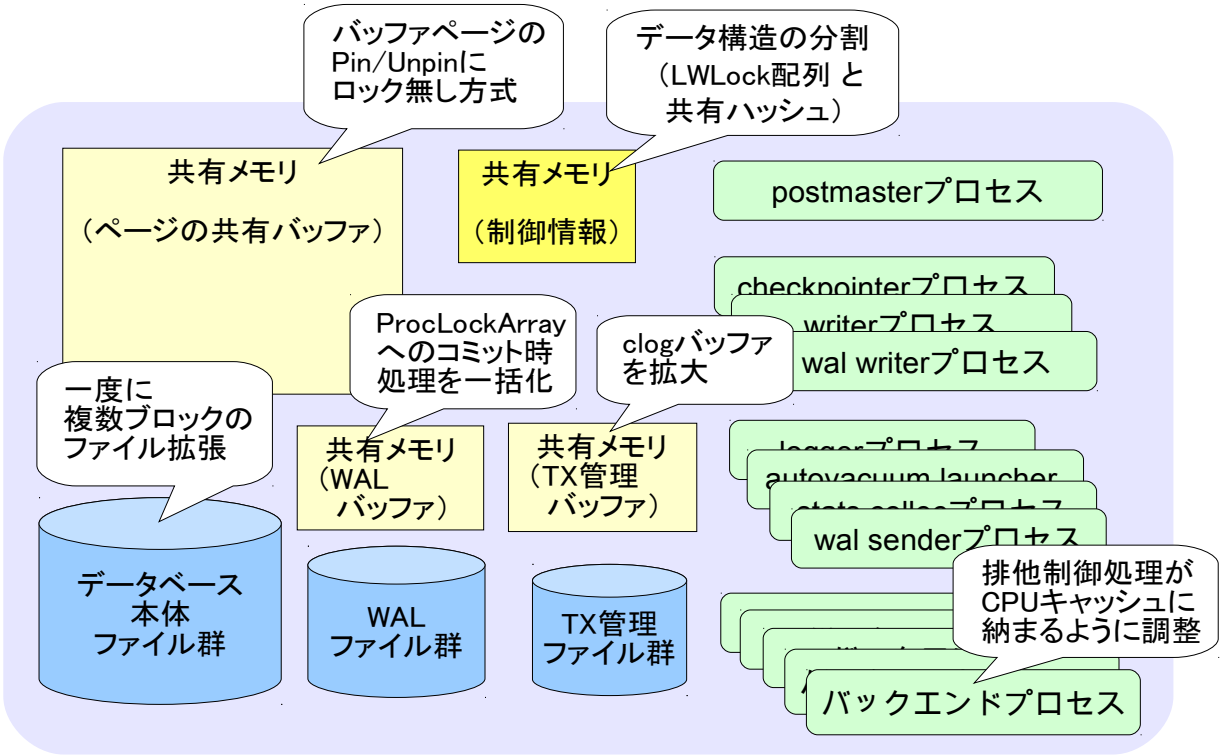

図 7: *PostgreSQL* 内部構造における多 *CPU* 同時実行の改善点

多数の CPU あるいは CPU コア数を持つマシンで、多数の同時接続を受け付けてトランザクション実行する場合、様々な箇所・段階でのロック競合が性能を引き下げる要因となります。9.6 では、これらの競合を回避する修正がいくつか適用されました。

□ *共有データ構造の分割*

各同時実行プロセスが共有して参照し、変更するときには排他制御が必須となる、共有メモリ上の制御情

報で、データ構造の分割が行われています。これにより、排他制御の単位が小さく分割されて、競合が減ることが見込めます。LWLock（ライトウェイトロック、内部ロック）用の配列と、各種データ使われる共有ハッシュにおける一部データ構造について、分割されました。

- ***共有バッファページの Pin/Unpin 実装の変更***

共有バッファのページのアクセスにあたっては、ピンする、ピンを外す（Unpin）というある種の排他制御が使われています。これは共有バッファのページ入れ替えのための制御であるため、参照のみのアクセスでも発生します。この部分の実装が、スピンロック（ビジーウィエイトによる排他制御）を使わない方式に改められました。

- ***ProcLockArray へのコミット時処理の一括化***

書き込みトランザクションでコミットをすると WAL に書き込みが発生します。このとき、共有メモリ上の制御情報 ProcLockArray に対する書き込みアクセスが発生します。本修正では同時実行の複数コミットの処理を一括実行するようにして、排他競合待ちを軽減します。なお、WAL 書き込みに関係があるので図では WAL バッファ、WAL ファイル群の近くに記載しています。

- ***一度に複数ブロックのファイル拡張***

PostgreSQL では 1 つのテーブルのデータ本体が 1 つのファイルに格納され、データ量が増えるとブロック（ページ、通常 8KB）単位で拡張されます。ファイル拡張の際には強い排他ロックが発生するので、大量の書き込みアクセスがあって、次々にファイル拡張が起きているとき、ロック競合が性能低下の要因となりました。本修正は一度に複数ブロックを拡張することで、これを軽減します。

そのほか、clog バッファサイズが増やされたり、排他制御処理について CPU 内のキャッシュに収まるように調整する等の変更が加わっています。

4.6. postgres_fdw の改善

postgres_fdw は、外部 PostgreSQL サーバのデータを仮想的にローカルテーブルのように扱える、外部データラッパ（Foreign Data Wrapper）です。PostgrSQL 9.3 から付属するようになりました。

PostgreSQL 9.6 では、より多くの処理をリモート側（外部サーバ側）で実行するように改善されました。また、そのほか、いくつか改善がほどこされています。

4.6.1. 結合/ソートをリモートサーバ側で処理

外部テーブルへの参照クエリを発行する際に、結合処理とソート処理をあらかじめ外部サーバ側で処理させることができるようになりました。

9.5 と 9.6 で比較検証した結果を以下に示します。

□ *検証準備*

リモート側のインスタンスを作成し、出来合いの postgres データベースにテスト用の t0 テーブルを作成しておきます。

```
$ initdb --no-locale --encoding=UTF8 -D "${PGDATA}remote"   -- 別インスタンスを作成して、
$ pg_ctl start -D "${PGDATA}remote" -o '-p 5433'            -- 5433 ポートで動作させます
$ psql -p 5433 -d postgres
postgres=# CREATE TABLE t0 (i int);
postgres=# INSERT INTO t0 VALUES (generate_series(1, 1000));
```

postgres_fdw モジュールを利用可能にし、外部サーバ、ユーザマッピングを定義します。また、外部テーブル用のスキーマを作成し、外部サーバのテーブル定義を読み込みます。

```
  --以下の操作は全て、ローカル側の操作です
$ psql db1
db1=# CREATE EXTENSION postgres_fdw;                                -- 拡張モジュール導入
db1=# CREATE SERVER server1 FOREIGN DATA WRAPPER postgres_fdw
       OPTIONS (dbname 'postgres', port '5433');                    -- 外部サーバ定義

db1=# CREATE USER MAPPING FOR public SERVER server1 OPTIONS (USER 'postgres');
                                                                    -- ユーザマッピング定義

db1=# CREATE SCHEMA remote1;          -- 外部テーブル用スキーマを作成
db1=# IMPORT FOREIGN SCHEMA public FROM SERVER server1 INTO remote1;
                                      -- スキーマ内に外部テーブル定義を作成
```

□ *結合処理の検証*

9.5 と 9.6 の実行計画の違いを以下のクエリで検証します。

```
SELECT * FROM remote.t0 a JOIN remote.t0 b ON a.i = b.i AND a.i = 500;
```

以下の EXPLAIN (ANALYZE, VERBOSE) の結果から、9.5 ではリモートサーバに対して SELECT を 2 回行っていて結合はローカルサーバで行っていることが分かります。これに対して 9.6 では結合する SQL 全体をリモートサーバで実行しています。

```
-- 9.5 の場合
db1=# EXPLAIN (ANALYZE, VERBOSE)
```

```
    SELECT * FROM remote.t0 a JOIN remote.t0 b ON a.i = b.i AND a.i = 500;
                                    QUERY PLAN
-------------------------------------------------------------------------------------------------
---------------------------------
 Nested Loop  (cost=200.00..296.58 rows=225 width=8) (actual time=2.785..2.790 rows=1
loops=1)
   Output: a.i, b.i
   ->  Foreign Scan on remote.t0 a  (cost=100.00..146.86 rows=15 width=4) (actual
time=1.537..1.538 rows=1 loops=1)
         Output: a.i
         Remote SQL: SELECT i FROM public.t0 WHERE ((i = 500))
   ->  Materialize  (cost=100.00..146.94 rows=15 width=4) (actual time=1.236..1.238 rows=1
loops=1)
         Output: b.i
         ->  Foreign Scan on remote.t0 b  (cost=100.00..146.86 rows=15 width=4) (actual
time=1.223..1.225 rows=1 loops=1)
               Output: b.i
               Remote SQL: SELECT i FROM public.t0 WHERE ((i = 500))
 Planning time: 0.449 ms
 Execution time: 3.597 ms
(12 rows)

-- 9.6 の場合
db1=# EXPLAIN (ANALYZE, VERBOSE)
    SELECT * FROM remote.t0 a JOIN remote.t0 b ON a.i = b.i AND a.i = 500;
                                  QUERY PLAN
-------------------------------------------------------------------------------------------------
--------------------------
 Foreign Scan  (cost=100.00..198.75 rows=225 width=8) (actual time=1.677..1.678 rows=1
loops=1)
   Output: a.i, b.i
   Relations: (remote.t0 a) INNER JOIN (remote.t0 b)
   Remote SQL: SELECT r1.i, r2.i FROM (public.t0 r1 INNER JOIN public.t0 r2 ON (((r2.i =
500)) AND ((r1.i = 500))))
 Planning time: 0.361 ms
```

```
Execution time: 2.217 ms
(6 rows)
```

□ *ソート処理の検証*

9.5 と 9.6 の実行計画の違いを以下のクエリで検証します。

```
SELECT * FROM remote.t0 ORDER BY i DESC;
```

以下の EXPLAIN (ANALYZE, VERBOSE) の結果から、9.5 ではローカルサーバでソートしていて、9.6 ではリモートサーバでソートしていることが分かります。

```
-- 9.5 の場合
db1=# EXPLAIN (ANALYZE, VERBOSE) SELECT * FROM remote.t0 ORDER BY i DESC;
                                                          QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=366.15..373.46 rows=2925 width=4) (actual time=6.383..6.694 rows=1000 loops=1)
   Output: i
   Sort Key: t0.i DESC
   Sort Method: quicksort  Memory: 71kB
   ->  Foreign Scan on remote.t0  (cost=100.00..197.75 rows=2925 width=4) (actual time=0.958..5.341 rows=1000 loops=1)
         Output: i
         Remote SQL: SELECT i FROM public.t0
 Planning time: 0.171 ms
 Execution time: 7.646 ms
(9 rows)

-- 9.6 の場合
db1=# EXPLAIN (ANALYZE, VERBOSE) SELECT * FROM t0 ORDER BY i DESC;
                                      QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
 Foreign Scan on remote.t0  (cost=100.00..205.60 rows=2925 width=4) (actual time=2.404..6.852 rows=1000 loops=1)
```

```
  Output: i
  Remote SQL: SELECT i FROM public.t0 ORDER BY i DESC NULLS FIRST
Planning time: 0.219 ms
Execution time: 7.697 ms
(5 rows)
```

4.6.2. DELETE/UPDATE をリモート側だけで実行

9.5 以前の外部テーブルに対する DELETE、UPDATE 処理は、最初にリモート側で SELECT ... FOR UPDATE を実行して、その結果を取得のうえ、改めて 1 行ずつ処理をしていました。これはローカル側にあるテーブルのデータと突き合わせて行選択する場合には必要な手順といえます。

9.6 では、処理内容リモート側だけで完結する場合には、リモート側だけで処理を行うようになりました。これにより余計な処理が減って性能向上が実現できています。

改善された外部テーブルに関する DELETE 処理について、9.5 と 9.6 で比較検証した結果を以下に示します。本検証は前項の検証準備で作成した同じテーブルを用います。

□ DELETE 処理での検証

9.5 と 9.6 の実行計画の違いを以下のクエリで検証します。

```
DELETE FROM remote.t0;
```

以下の EXPLAIN (ANALYZE, VERBOSE) の結果を見ると、9.6 では SELECT ... FOR UPDATE が実行されていません。また、9.5 と比較して実行時間(Execution Time)も短くなっています。

```
-- 9.5 の場合
db1=# EXPLAIN (ANALYZE, VERBOSE) DELETE FROM remote.t0;
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
-------------------------------
 Delete on remote.t0  (cost=100.00..191.90 rows=2730 width=6) (actual
time=169.518..169.518 rows=0 loops=1)
   Remote SQL: DELETE FROM public.t0 WHERE ctid = $1
   ->  Foreign Scan on remote.t0  (cost=100.00..191.90 rows=2730 width=6) (actual
time=2.292..12.185 rows=1000 loops=1)
         Output: ctid
         Remote SQL: SELECT ctid FROM public.t0 FOR UPDATE
 Planning time: 11.733 ms
```

```
 Execution time: 203.697 ms
(7 rows)

-- 9.6 の場合
db1=# EXPLAIN (ANALYZE, VERBOSE) DELETE FROM remote.t0;
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
---------------------------------
 Delete on remote.t0  (cost=100.00..191.90 rows=2730 width=6) (actual time=3.371..3.371
rows=0 loops=1)
   ->  Foreign Delete on remote.t0  (cost=100.00..191.90 rows=2730 width=6) (actual
time=3.367..3.367 rows=1000 loops=1)
       Remote SQL: DELETE FROM public.t0
 Planning time: 0.141 ms
 Execution time: 3.675 ms
(5 rows)
```

4.6.3. 関数・演算子のリモート処理

postgres_fdw では、WHERE 句に書かれた組み込みの関数や演算子についてはリモートで処理しようとします。しかし、あとから追加した関数や演算子はリモート処理の対象になりません。9.6 で、こういった関数や演算子についてもリモート実行させる設定が用意されました。

CREATE SERVER コマンドの OPTIONS で extensions パラメータに拡張モジュール名をカンマ区切りで指定することで、その拡張モジュールに属する関数や演算子をリモート実行させることができます。

```
db1=# CREATE EXTENSION pgcrypto;
db1=# CREATE EXTENSION hstore;                          -- 拡張モジュールを追加
db1=# ALTER SERVER server1
        OPTIONS (ADD extensions 'pgcrypto,hstore');     -- 追加指定なので ADD で記述
```

もちろん、ローカル側にもリモート側にもこれら拡張モジュールが予めインストールされてなければいけません。さもないと SQL 実行時にエラーが生じます。

```
-- リモート側に拡張モジュールが無い場合のエラー例
ERROR:  function public.digest(text, text) does not exist
HINT:  No function matches the given name and argument types. You might need to add
explicit type casts.
```

```
CONTEXT:  Remote SQL command: SELECT id, v FROM public.t1 WHERE ((public.digest(v,
'sha1'::text) = E'¥¥x666f6f'::bytea))
```

4.6.4. postgres_fdw - その他の改善

リモートサーバへの接続を、同じリモートユーザにマップされている複数のローカルユーザ ID にて共有するようになりました。ただし、これはローカルサーバにおける単一のセッション内での共有です。複数セッションで共有するわけではありません。セッション内でユーザを切り替えて SQL 実行するときに、リモートサーバへの接続数を節約することができます。

また、問い合わせのキャンセルをしたときに、それがリモートサーバにも伝搬するようになりました。これまでは、リモートに投げた問い合わせはそのまま終わるまで実行されていました。

4.7. 各種モニタリングビューの拡張

4.7.1. pg_stat_activity

実行中のセッションの情報を返す pg_stat_activity ビューが、ロック待ちについて拡充されました。ロック待ちをしているかを示す waiting カラムに代えて、以下二つのカラムが追加されました。ロック待ちしていないときには、どちらも NULL になります。

カラム	意味
wait_event_type	以下の 4 ついずれか、または NULL になります。 'Lock' SQL レベルのロック（重量ロック）を待っている。 wait_event にロックタイプ（pg_locks の locktype カラムと同じ）を表示。 'LWLockNamed' 名前付きの軽量ロックを待っている。 wait_event に軽量ロック名を表示。 'LWLockTranche' 軽量ロックのグループの一つを待っている。 wait_event に軽量ロック名を表示。 'BufferPin' バッファ Pin（共有バッファページの入れ替えのための排他制御）を待っている
wait_event	wait_event_type に応じた待機イベント種別の詳細をあらわす文字列

軽量ロック名には「WALWriteLock」「CheckpointLock」など、処理箇所を示す名前がつけられています。

4.7.2. pg_stat_progress_vacuum

PostgreSQL 9.6 から、実行中 SQL の進捗を示すための基盤的な仕組みが導入されました。

通常の SQL コマンド（SELECT 等）の進捗表示は未だ実装されていませんが、VACUUM コマンドについては進捗が pg_stat_progress_vacuum ビューに表示されるようになりました。大きなテーブルの VACUUM 処理がいつ終わるのかを予測するのに役立ちます。

実行中の VACUUM 処理中セッションの数だけ以下のような情報が返ります。「処理フェーズ」については、マニュアルに詳細が記載されています。

```
db1=# SELECT * FROM pg_stat_progress_vacuum ;
-[ RECORD 1 ]------+--------------
pid                | 6106
datid              | 16384
datname            | db1              -- データベース名
relid              | 16603            -- 処理中のテーブルの OID
phase              | scanning heap    -- 処理フェーズ（７種のフェーズ内の１つ）
heap_blks_total    | 8334             -- 処理中テーブル（ヒープ本体）のブロック数
heap_blks_scanned  | 2775             -- 処理中テーブルの現時点でスキャン済みブロック数
heap_blks_vacuumed | 0                -- 処理中テーブルの現時点で整理済みブロック数
index_vacuum_count | 0                -- 現時点で処理済みのインデックス個数
max_dead_tuples    | 291              -- インデックスでの蓄積可能なデッドタプル数
num_dead_tuples    | 1                -- インデックスでの現時点で収集済みデッドタプル数
```

4.8. 古いスナップショットの強制廃棄

VACUUM をかけているはずなのにデータが肥大化するケースがあります。VACUUM VERBOSE コマンドの出力や設定 log_autovacuum_min_duration によるログ出力で以下のように「cannot be removed yet」と出て、不要行が回収されません。これは、当該行バージョンを参照するスナップショットが未だ存在してるために処理が保留されている結果です。

```
INFO: "t1": found 0 removable, 12345 nonremovable row versions in 100 out of
1000 pages
DETAIL: 12345 dead row versions cannot be removed yet.
```

PostgreSQL 9.6 から、古いスナップショットを強制廃棄できるようになりました。以下の設定で時間（'1day' であるとか '10min'）を指定します。指定時間が経過すると、強制廃棄の対象となります。

```
old_snapshot_threshold  = -1          # 時間を指定 ／ デフォルトは無効の意味で -1
```

old_snapshot_threshold = 1min と、極端に短く設定したうえで PostgreSQL 再起動をして、以下のように 2 つの接続からトランザクションを実行してみます。

```
-- 接続その 1
db1=# START TRANSACTION ISOLATION LEVEL
      REPEATABLE READ;
db1=# SELECT * FROM t1;
 id | v
----+---
  1 | A
(1 row)
                                                  -- 接続その 2
                                                  db1=# DELETE FROM t1;
db1=# SELECT pg_sleep(60); -- 60 秒経過
db1=# SELECT * FROM t1;
ERROR:  snapshot too old   -- エラーが発生した
```

REPEATABLE READ トランザクションであっても、指定した時間を経過したあと削除済みデータを参照しようとするとエラーになることが確認できました。

5. 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。