PostgreSQL インサイド

リレーションバッファー削除の最適化によるリカバリーの高速化 PostgreSQL 14 でコミットされた機能の先行紹介

技術者 Blog

Kirk Jamison

FUJITSU Limited
Software Products Business Unit Data Management Division

はじめに

富士通のグローバル PostgreSQL 開発チームは、PostgreSQL の次期バージョンで実装される最新機能についてのブログをシリーズで公開しています。私は富士通 PostgreSQL チームのシニアディレクターである Amit Kapila が率いるコミュニティー活動に専念する、このチームのメンバーです。このブログでは、PostgreSQL 14 で提供される予定の機能改善について紹介したいと思います。

関連情報

- 富士通 PostgreSQL チームのシニアディレクター Amit Kapila を紹介：技術者 Blog

背景 – リレーションバッファーの削除とバッファー・プールのスキャン

近年のデータ大量化に伴い、大きな共有バッファー（shared_buffers）が設定されたシステムは、より一般的になってきています。大きな共有バッファーが設定されている場合、VACUUM や TRUNCATE のようなリレーションバッファーを削除するコマンドのフェイルオーバーや WAL の再実行には、バッファー・プール全体をスキャンしなければならないため、非常に長い時間がかかることがあります。
特に、プライマリーサーバー上で VACUUM / TRUNCATE のトランザクションが複数（例えば数千件）実行されている場合に、性能が低下する可能性があります。ここからは専門的な話になりますが、この場合、各トランザクションはテーブルを切り捨て（VACUUM は、テーブルごとに新しいトランザクションを内部的に開始します）、バッファー・プール全体がトランザクションごとにスキャンされます。フェイルオーバーのテストとして PostgreSQL 13 を使用すると、更新がスタンバイサーバーに反映されるまでに数十秒かかります。特定のテストケースの例は、後のセクションで示します。性能はサーバーの仕様によって異なりますが、調査の結果、PostgreSQL 13 における性能の低下は共有バッファーのサイズとトランザクション数の増加に直接関係していることが分かりました。
私は、FUJITSU Software Enterprise Postgres でお客様の要件を満たすため、この問題に対する解決策を提示しなければなりませんでした。並行して、私はオープン・ソース・ソフトウェア（以下、OSS）コミュニティーに設計を提案し、特にリレーションのバッファーを削除する際における上記の性能問題について、PostgreSQL（ターゲットは PostgreSQL 14）の信頼性を向上させることに関心を持ってもらうようにしました。

機能概要 – リレーションサイズの判定とバッファーの検索

ここでも専門的な話をします。私のパッチでは、リレーション内で削除されるバッファーの数が設定されたしきい値以下の場合にバッファー・プール全体のスキャンが回避されるよう、リレーションバッファーを削除するためのリカバリーパスが最適化されています。これは、削除されるリレーション以外に余分なものが残らないように、リレーションの正確なサイズを決定することによって行われます。キャッシュされたリレーションサイズがあると判断したら、バッファーマッピングハッシュテーブルで無効化されるバッファーを検索します。
VACUUM と TRUNCATE のパスは異なっており、少し異なる変更が必要です。VACUUM / autovacuum のパスは、単一のリレーション用に DropRelFileNodeBuffers()で最適化されます。TRUNCATE のパスは、複数のリレーションを削除できる DropRelFileNodesAllBuffers()で最適化されます。パフォーマンス改善の目的は同じですが、パッチを分離する必要があるため、2 つのパッチをコミットしました。

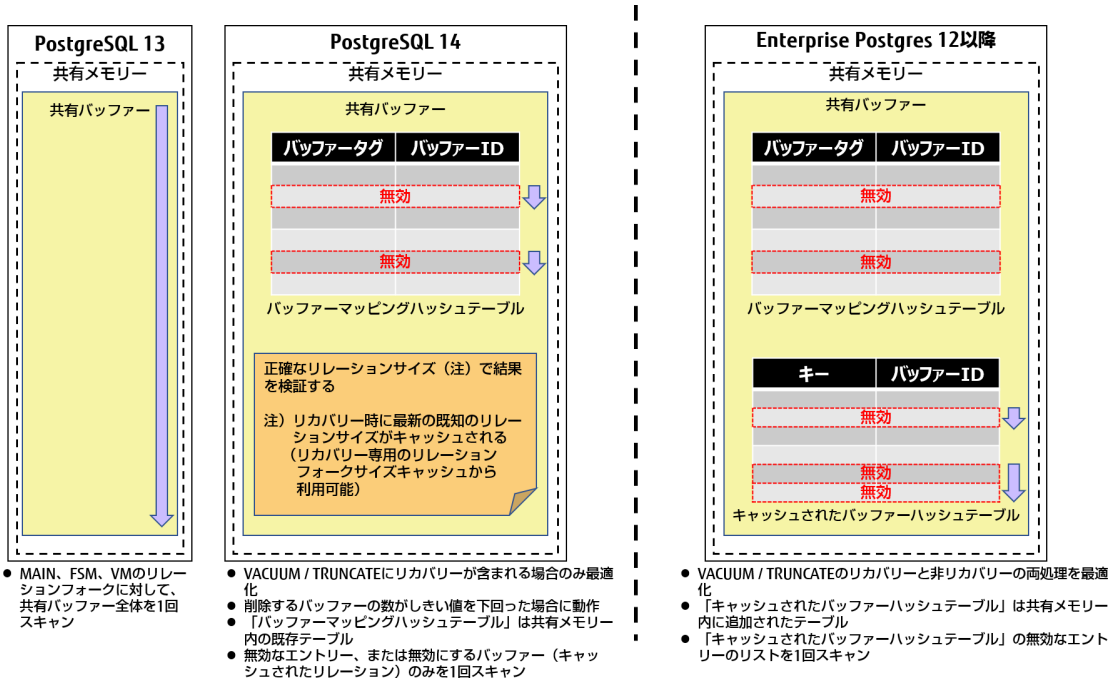


図 1：VACUUM および TRUNCATE でのリレーションバッファーの無効化

前のセクションで述べたように、この改善は元々、FUJITSU Software Enterprise Postgres 12 用に開発されました。これと PostgreSQL コアにコミットされたパッチの実装には違いがあることに注意してください。FUJITSU Software Enterprise Postgres 12 の機能は、バッファーハッシュテーブルのキャッシュを使用し、共有メモリー内に無効化されるバッファーのリンクリストを 2 重に保持することで、VACUUM と TRUNCATE のリカバリーパスと非リカバリーパスの両方をカバーします。しかし、私がこの設計についてコミュニティーと話し合ったところ、バッファー割当てには余分なオーバーヘッド（コスト）があり、PostgreSQL の一般的なワークロードに影響を与える可能性があることがわかりました。そのため、コミュニティー内のコミッター（Andres Freund 氏と Amit Kapila）の提案されたアイデアを使用して、別のアプローチをしなければなりませんでした。それは、最新の既知のリレーションサイズをキャッシュし、無効にするバッファーのハッシュテーブル検索を行うことで、バッファー割り当てのオーバーヘッドを回避し、リカバリーパスのみを最適化するというものです。

ユーザーのメリット

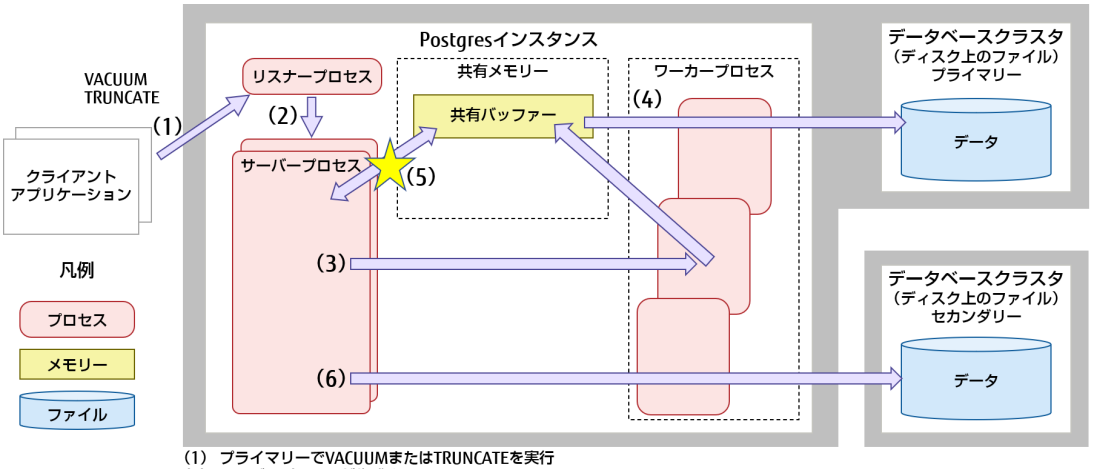
コミットされた機能は、大きなバッファー・プールを使用する大規模なサーバー・システムにとって有益です。一般に、コミットされたリカバリーの最適化は次のユースケースに役立ちます。
以下のどちらかのケース：

a) VACUUM コマンドまたは autovacuum がリレーションの最後にある空のページを切り詰めた場合

b) リレーションが作成されたのと同じトランザクションで切り詰められている場合

かつ、

c) リレーションファイルを削除する必要がある操作、例えば TRUNCATE、DROP TABLE、ABORT、CREATE TABLE などを実行する場合



（1） プライマリーでVACUUMまたはTRUNCATEを実行
（2） サーバープロセスが生成される
（3） ワーカープロセスによって共有バッファーに変更が加えられる
（4） 共有バッファー上の変更がワーカープロセス（チェックポインターまたはバックグラウンドライター）によってプライマリーに反映される
（5） 無効な共有バッファーが削除される（ここで速度向上）
（6） プライマリーへの変更は、WALの送信でセカンダリーに反映（レプリケート）される（ここで速度向上）

図 2：最適化されたリカバリーパスのトランザクションプロセスの概要（レプリケーションの場合）

これにより、リカバリーのパフォーマンスが 100 倍以上向上するケースがあります。特に、小さなテーブル（1,000 個のリレーションでテストしました）がいくつかトランケートされている場合や、サーバーに大きな値の共有バッファーが設定されている場合に顕著です。主なアピールポイントは、トランケートされたリレーションのキャッシュサイズがある場合、リカバリー中にバッファー・プール全体のスキャンをスキップできることです。これは、shared_buffers のサイズに関わらず、ユースケース（a）、（b）、（c）でのリカバリー時の WAL 適用時間を短縮します。（128 メガバイトから 100 ギガバイトまでテストしました）

テストの結果は以下の表のとおりです。なお、実行時間はテストを 5 回実行したときの平均値です。また、改善率（フェイルオーバー時間の削減率）は、下記の計算式で算出しました。

（（PostgreSQL 13 の時間 - PostgreSQL 14 の時間）÷ PostgreSQL 13 の時間） × 100

1,000 個のリレーションに対する VACUUM フェイルオーバー / リカバリー性能の結果

shared_buffers のサイズ	PostgreSQL 13	PostgreSQL 14	改善率
128 メガバイト	0.306 秒	0.306 秒	0%
1 ギガバイト	0.506 秒	0.306 秒	39.53%
20 ギガバイト	14.522 秒	0.306 秒	97.89%
100 ギガバイト	66.564 秒	0.306 秒	99.54%

1,000 個のリレーションに対する TRUNCATE フェイルオーバー / リカバリー性能の結果

shared_buffers のサイズ	PostgreSQL 13	PostgreSQL 14	改善率
128 メガバイト	0.206 秒	0.206 秒	0%
1 ギガバイト	0.506 秒	0.206 秒	59.29%

shared_buffers のサイズ	PostgreSQL 13	PostgreSQL 14	改善率
20 ギガバイト	16.476 秒	0.206 秒	98.75%
100 ギガバイト	88.261 秒	0.206 秒	99.77%

上記のベンチマークから、PostgreSQL 14 での結果は TRUNCATE と VACUUM の両方共、すべての shared_buffers の設定で一定であることがわかります。バッファーのスキャンと無効化は共有バッファーのサイズに依存しません。
このテストケースと仕様の詳細については、以下のコミュニティースレッドを参照してください。

- RE: [Patch] Optimize dropping of relation buffers using dlist
  https://www.postgresql.org/message-id/OSBPR01MB2341B75F355B285199AD1BD3EFD00%40OSBPR01MB2341.jpnprd01.prod.outlook.com

開発の舞台裏：コミュニティーメンバーとの協働

OSS コミュニティーでは、コミッターである Tom Lane 氏、Robert Haas 氏、Andres Freund 氏、Thomas Munro 氏、Amit Kapila が、このソリューションに関する議論に重要なポイントを提供してくれました。また、レビュアーである堀口 恭太郎氏や当社シニアプロエンジニアの綱川 貴之が、よりシンプルで高品質なコードになるようにパッチの完成を手伝ってくれました。
私はレビュー担当者のコメントに返信したり、テストケースとテスト結果を使用してパフォーマンス改善の証拠を投稿したりするために多くの時間を費やしました。この機能は大きく変更されたため、実装するのに約 1 年かかりました。私が提出したパッチが高品質であることを証明するために、テストを使用してコミュニティーを説得しなければなりませんでした。
フィードバックのいくつかが私の最初のコードに大きな変更をもたらしたので、この経験にはやりがいと充実がありました。コミュニティー活動の醍醐味は、最初は相反するアイデアやレビューがあることが多いのですが、最終的には合意に達することです。批判的なフィードバックと、レビュアーやコミッターの継続的なサポートがなければ、私はプロジェクトを成功させることはできなかったでしょう。
富士通のグローバル PostgreSQL 開発チームのロードマップの一部として、特にシステム停止やフェイルオーバーの発生に対する PostgreSQL の信頼性向上を目指しています。今回のプロジェクトでは、Amit Kapila と綱川 貴之の指導のもと、パッチのコミットに成功しました。私のテストが有効であることを確認するために、私のチームの他のメンバーも私のテストを再実行し、その結果をコミュニティーのメーリング・リストのスレッドに投稿しました。
今回コミットされたパッチは、私にとっての 2 回目と 3 回目の主要機能になります。私のコードが PostgreSQL のコアコードに反映されているのを見ると、非常に満足しています。このコミュニティーに貢献し続ける自信を深めました。

将来を見据えて – リレーションサイズのキャッシングにおけるさらなる改善

リレーションサイズの共有キャッシュを提供することについて、別のコミッターである Thomas Munro 氏による議論と開発が進行中です。

- Cache relation sizes?
  https://www.postgresql.org/message-id/flat/CAEepm%3D3SSw-Ty1DFcK%3D1rU-K6GSzYzfdD4d%2BZwapdN7dTa6%3DnQ%40mail.gmail.com

開発がうまくいけば、VACUUM と TRUNCATE の非リカバリーパスや通常の操作をカバーするように機能を拡張することができるでしょう。

富士通のグローバル PostgreSQL 開発チームは、PostgreSQL コアへの統合を目的とした開発項目を挙げ、コミットの目標時期を決めて取り組んでいます。これらの開発項目は、富士通の顧客ニーズに対応し、PostgreSQL コミュニティーに貢献します。近い将来、私たちの開発作業についてさらに情報を共有できることを楽しみにしています。

2021 年 4 月 23 日