PostgreSQL ベースの OLXP システム実現のための同時実行制御手法

橋田 拓志[1,a)] 田原 司睦[1] 中村 実[1] 宇治橋 善史[1] 河場 基行[1] 原田 リリアン[1]

概要：近年、トランザクション処理と高速分析処理を兼ね備えた OLXP の必要性が高まっている。我々は、OSS の RDBMS である PostgreSQL 9.4 をベースとして、OLXP システムを実現するための設計・開発を行っている。OLXP システムの実現には、トランザクション処理が必要とする一貫性と分析処理の高速性とを、透過的なインターフェースで実現することが大きな課題となる。我々は、分析処理の高速化のためにカラム形式のストアを導入した。本論文では、このカラムストアへの更新と、PostgreSQL と同様のデータ一貫性を保証するための多版型同時実行制御の低オーバーヘッドな実現方式および性能評価について述べる。

Concurrency control technique for OLXP system based on PostgreSQL

Hashida Takushi, Tabaru Tsuguchika, Nakamura Minoru, Ujibashi Yoshifumi, Kawaba Motoyuki, Harada Lilian

Abstract: The importance of database systems to support mixed online transaction processing (OLTP) and online analytical processing (OLAP) workloads, the so-called OLXP workloads, has attracted much attention lately. We have developed an OLXP database system based on the PostgreSQL which is an OSS RDBM system. In order to implement OLXP system, realizing consistency of transactional processing and high performance of OLAP simultaneously and transparently is an important issue. We have introduced a column store to realize high OLAP performance in our previous works. In this paper, we present our implementation and evaluation results of its low overhead method to update the column store while incurring minimal OLTP performance degradation, and a multi version concurrency control method that ensures data consistency with the original PostgreSQL.

1. はじめに

リレーショナルデータベースマネージメントシステム(RDBMS)において、一件あたりのデータ量は比較的小さいが挿入・更新・削除の操作が短期間に大量に発生するオンライントランザクション処理(Online Transaction Processing：OLTP)のワークロードと、すでに蓄積されたデータを広範囲に読み込み複雑な統計処理を行うオンライン分析処理(Online Analytical Processing：OLAP)のワークロードの二つが良く知られている。OLTP ワークロードは更新操作が行単位に行われるため、RDBMS のデータストアの構造としてデータを行ごとに格納する行形式を用いると、効率よく処理できることが知られている。一方、OLAP においては、蓄積されたデータの中で特定の列に注目して分析することが多く、列ごとにデータを格納する列形式が向いていることが知られている。

行形式データストアと列形式データストアは相反する性質を持っているため、一つの RDBMS で処理することは困難である。例えば、列形式データストアを用いる場合、1 行の更新を行う際に、物理的に離れた位置を複数更新する必要があるため、IO の増大を招くなどの問題がある。そのためそれぞれのワークロードに特化した別々のシステムを用意して OLTP システムから OLAP システムへ ETL(Extract Transform Load)ツールなどを用いて一定間隔でデータをロードするという使い方が一般的である。

近年、ビジネスの世界では OLTP システムの持つデータをリアルタイムに分析できる RDBMS の存在が必要とされるようになってきた。このような OLTP と OLAP が混在したワークロードは OLXP と呼ばれ[4]、OLXP ワークロードを効率的に処理可能なデータベースは OLXP システムと呼ぶことができる。このような OLXP システムは研究レベル・商業レベルの両方で発表されている(SAP HANA[1]、Oracle Database 12c[2]、HyPer[3])。OLXP システムは、いずれもリアルタイム分析を可能にしているが、形式が異なる 2 種のデータストアを透過的に扱おうとすると、OLTP と OLAP それぞれが分離した専用システムに比べどちらかの最大処理性能が落ちてしまうという問題がある。

我々は、高い OLTP 性能と高い OLAP 性能を両立した OLXP システム実現のため、OSS の RDBMS である PostgreSQL 9.4[5] をベースとした OLXP システム（カラムストアインデックス：**CSI**）を開発し、OLAP 性能を向上させることに成功している[6]。本論文では、OLXP 実現に向けた処理オーバーヘッドにフォーカスし CSI の OLTP 性能を落とさないための工夫と、OLTP 処理と OLAP 処理をユーザーが意識することなく扱うことができる透過的なインターフェースの実現について説明する。

2 章では、PostgreSQL を OLXP システムにするために必

[1] （株）富士通研究所
Fujitsu Laboratories Ltd.
a) hashida.takushi@jp.fujitsu.com

要な要件について述べる。3 章、4 章では、CSI インデックスの構成要素の実装、およびそれらに対するオペレーションについてそれぞれ説明する。5 章では、OLAP 処理の際の CSI からのデータの読み取りと、それを行う際に問題となる同時実行制御について説明する。最後に、6 章で CSI の OLTP、OLAP 各ワークロードに対する性能インパクトについて評価する。

2. OLXP の実現要件

本論文では、PostgreSQL をユーザーから OLTP と OLAP が透過的に扱える OLXP システムにするために必要な要件として、以下を設けた。

① OLTP システムである PostgreSQL に、高速な OLAP 処理機能を導入する
② OLAP 処理機構の導入による OLTP 機能の低下は最低限に抑える
③ OLTP 処理であるか OLAP 処理であるかは、システムが自動的に判断し、ユーザーは特別なキーワードなどでこの区別を指定する必要がない
④ OLAP 処理が OLTP 処理によって更新された最新のデータを遅延や矛盾なく扱うことができる

①の項目については、カラムナを用いた vector-at-a-time な処理や、並列化機構の導入により、実現されている[6]。

RDBMS では、検索や更新の高速化を目的として、テーブルの特定のカラムにインデックスを作成する場合があるが、そのインデックス管理のために、OLTP 性能の低下が生じる。②の項目については、CSI では、PostgreSQL の標準の B-tree インデックスをテーブルに対して作成した際の OLTP 性能の性能劣化と同程度の性能劣化に抑えることを目指す。

③、④は、ユーザーが意識することなく OLTP も OLAP も高速に扱うことができるという OLXP システムの透過的なインターフェースの機能面の実現に関するものである。本論文では、①や②のような性能面の改善を実現したうえで、④の機能を損なわないような実装について述べる。

3. カラムストアインデックス CSI の構成

CSI の全体の構成を図 1 に示す。カラムストレージは主に PostgreSQL のリレーション（テーブルやインデックスを格納する論理的なファイル構造）から成り立っており、永続化やリレーションを構成するページのバッファの機構は PostgreSQL の仕組みを利用する。CSI を用いる場合でも、PostgreSQL のオリジナルテーブルはなくなるわけではなく、行形式テーブルのほうが効率良く処理できるクエリに関しては、そちらを用いて実行される。

実際にカラム形式で情報を記録する領域は Read Optimized Storage（ROS）と呼ばれ、一カラムにつき一つのリレーションとなっている、その他に、ライトバッファである Write Optimized Storage（WOS）や、PostgreSQL のオリジナルテーブル上の行 ID（Tuple-ID:TID）から ROS 上での位置(Columnar Record Identifier:CRID)を探索するのに必要となる TID-CRID ツリーなどの構造を持つ。カラムストレージへのアクセス手段は、主に PostgreSQL のオリジナルテーブルと WOS とのやり取りを行うインデックスアクセスメソッド、ROS や TIDCRID ツリーを管理する ROS 制御コマンド、CSI のクエリエンジンに対してカラムからデータを取り出して提供するカラムストアフェッチからなる。また、WOS に存在している情報をカラムストアフェッチへと提供するためにクエリ毎に作成されるメモリ上の構造として、Local ROS が存在する。Local ROS は、トランザクションを実現するために、クエリ毎に異なる行の見え方を吸収する役割も果たす。

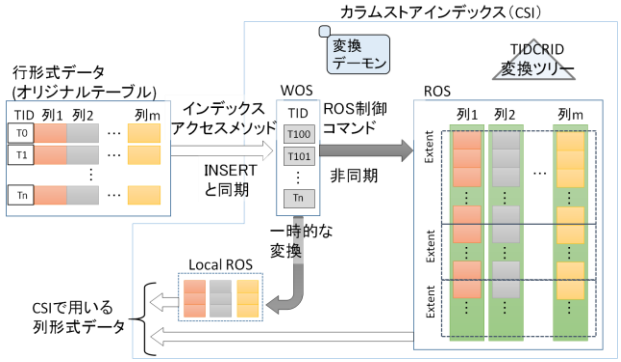

図 1　CSI の全体構成

3.1 カラムストアインデックスの利用方法

我々は CSI を PostgreSQL のインデックスの一つとして実現する。従って、CSI はその名前の通り、インターフェースとしてはインデックスであるかのように振る舞う。ほかのインデックス同様、ユーザーが CSI を使うためには、以下のように CREATE INDEX 文を実行する。

```
CREATE INDEX indexname ON tablename USING csi (a, b, c);
```

CSI は PostgreSQL が提供しているカスタムインデックス機構を利用して実装されている。この機能は、INSERT や UPDATE の実行に対してインデックスごとに設定されているコールバック関数(インデックスアクセスメソッドと呼ばれる)を呼び出すことで、インデックスの実装ごとの処理を行うものである。

PostgreSQL の CREATE INDEX 文が持つ USING 句は B-tree や hash などインデックスの実装アルゴリズムを指定するために使われ、”csi”キーワードを指定することにより、独自追加した CSI のインデックス実装を呼び出すことができるようになる。

ON 句でカラムストアを設けるテーブル名を指定し、そ

の後に ON 句で指定したテーブルの中のカラムストアを作成したいカラム名を列挙する。ここで指定するカラム名のリストは PostgreSQL の B-tree インデックスの場合はインデックスのソートキーとなるが、CSI では列形式データへ変換するカラムを指定する効果のみでカラムの並び順にも意味はない。CREATE INDEX 文が実行されると、図 1 に示したような各種テーブルが作成される。

3.2 Write-Optimized-Store (WOS)

カラムストアは OLTP のように頻繁にデータ行の DELETE/UPDATE が発生するワークロードでは効率が悪い。そのため列形式 RDBMS では、列形式ストアの前段として Write-Optimized-Storage(WOS)[7]あるいは Delta Store などと呼ばれる行形式格納のキャッシュを設け、短期的な更新は行形式で貯めるのが一般的である。

CSI でも OLTP 性能へ与えるインパクトの低減を図るため、WOS を設け、WOS へ一定量のデータが貯まった後に非同期なタイミングで列形式に変換する方式をとる。WOS に行形式でデータをためるだけでも更新の効率をあげることができるが、CSI ではさらに更新オーバーヘッドを小さくするため、更新に必要な記録サイズを減らす工夫をした。CSI では、実際のデータの代わりに、図 2 のように PostgreSQL がテーブル内の行(タプル)の位置を記録するために使用している Tuple-ID(TID)と呼ぶ情報のみを WOS に記録することで INSERT/DELETE に対するオーバーヘッドを減らした。TID は 32 ビットのページ番号と 16 ビットのページ内アイテム番号が対になった 48 ビットの識別子で、タプル内の実際のデータよりサイズが小さいため記録するコストを下げることができる。実際にはこれに加え、後述する多版同時実行制御（MVCC）の実現のため、WOS 上にオリジナルテーブルと同等の生成・削除に関するトランザクション情報を保持している。このように、WOS は PostgreSQL の行データが持つのと同じだけの管理情報をもつことになる。

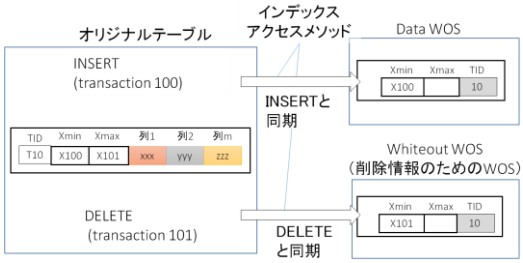

図 2　WOS 更新のイメージ

CSI はタプルの挿入を記録する Data WOS とタプルの削除を記録する Whiteout WOS の二種類 WOS を持つ。インデックスへ挿入が行われた場合は、Data WOS 内に挿入した行の TID を追記する。行の削除が行われた場合は、Whiteout WOS 内に削除した行の TID を追記する。図 2 のように、属性としてオリジナルテーブル上での TID だけをもつ Data WOS のタプルは、該当するオリジナルテーブルのタプルの Xmin と同じ Xmin の値を持つ。一方で Whiteout WOS では、オリジナルテーブルの Xmax と同じ Xmin を持つ。

3.3 Read-Optimized-Store（ROS）

ROS は CSI のカラムストアである。管理オーバーヘッドの削減と同時読み出しの効率化のため、ROS は一定量の行（最大 262,144 行）のデータをまとめたエクステント(extent)単位で管理される。WOS から ROS への変換は、Data WOS に 1 エクステントを構成するために必要な数だけタプルが貯まった時に変換を開始し、実際には 4.1 節で説明する可視条件を満たすタプルのみが ROS に格納される。WOS には TID のみが記録されているので、ROS へ格納するカラムデータはインデックス対象のテーブルから TID を元にフェッチされる。

従来タプルという形で 1 行のデータが共通で持っていたトランザクション情報をコンパクトに効率的に扱うために、ROS ではエクステント毎に Xgen と Xdel という管理情報を設け、エクステントの生成・削除を実施したトランザクション ID を記録している。カラムストアの 1 要素ずつにトランザクション情報を持たせると、容量オーバーヘッドが増大する。また、MVCC のためカラムの一要素ごとに生成・削除情報を参照し、トランザクションから可視か不可視かの判定を行うのは多くの分岐予測ミスを発生させ高速実行の妨げる問題を発生する。エクステントという大きな単位での可視性の判定を行うことでこれら 2 つの問題を解決した。

また ROS からのデータ削除処理の効率を上げるために Delete Vector を導入した。Whiteout WOS に存在する TID に対応する行を ROS から削除する際に、エクステントの一部が削除される度に、既存のエクステントを再作成するのはコストが高い。そこで、削除された行を記録する Delete Vector を導入し、非同期的に一括して領域を回収(4.3「空行の回収」)する。

ROS 内のデータに対しては、独自の論理的なレコード番号として、CRID（Columnar Record Identifier）が割り当てられる。ROS はカラム毎に別々のカラムデータとして格納されるが、オリジナルテーブルで同じレコードに属するデータには同一の CRID が割り振られる。CRID は、エクステントの番号と、エクステント内の位置だけで一意に決まる番号である。

3.4 TID-CRID 変換ツリー

PostgreSQL のオリジナルテーブルからデータが削除される際に、Whiteout WOS 上にはオリジナルテーブル上での TID が記録されているが、この TID から Delete Vector へと反映するためには、この TID が ROS 上のどの CRID を削除するものかを判定する必要がある。そのため、TID から CRID へと高速に変換するための機構が必要となる。そこで、図 3 のような TID をキーとしたツリーを作成する。

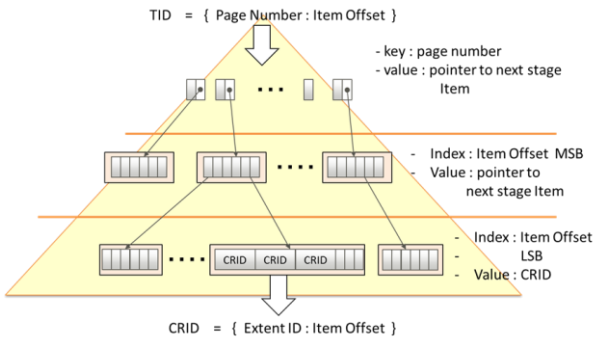

図 3　TIDCRID 変換ツリー

TIDCRID 変換ツリーのエントリー情報は、CRID が決定する時、つまり WOS から ROS へデータを移動する際に行われる。

4. ROS の制御（ROS Control Command）

本節では、主に ROS に対する制御について述べる。ROS へのデータ更新や不要領域回収などの ROS に対する制御は、PostgreSQL9.3 から導入された Dynamic Background Worker（DBW）によって非同期的に実行される。これらの処理を ROS 制御コマンドと呼ぶことにする。DBW は、一つの CSI インデックスに対して同時に一つしか存在しないように制御するため、複数の ROS 制御コマンドが同時に実行されることはない。処理が終了すると同時に DBW も終了する。エクステントを生成するコマンドは、実行時にエクステントごと生成タイミング情報として自身のトランザクション ID を Xgen、Xdel として、エクステントに記録する。この情報は、5 章で説明する、MVCC の制御で使われる。

4.1 Data WOS から ROS への変換

ROS 上に新規にエクステントを生成し、WOS にあるデータを列形式へ変換しながらエクステントに格納する。ROS への変換は、WOS 上に存在する行数をカウントし、1 エクステント分のデータが存在するときに実行される。Data WOS 上の全てのタプルが変換されるわけではなく、タプルを INSERT したトランザクションの状態によって挙動が異なる。具体的には、INSERT したトランザクションが、現存するすべてのクエリからコミット済みに見えているときに限り ROS への変換が実行される。例えば未コミットである場合や、トランザクションの分離条件のために特定のクエリからは不可視状態であるようなタプルは、ROS に移すことはできない。これにより、エクステントの可視条件判定を簡単にしている（図 4）。

対象となった WOS 上のタプルから、TID を取り出し、その TID を検索キーとしてオリジナルテーブルから実際のタプルの情報を読み取り、カラム形式に構成し直してエクステントを作成し、ROS に書き込む。ROS に書き込んだ WOS 上のデータは WOS から削除する。この削除は PostgreSQL の通常のテーブルの行削除と同様 Xmax を設定することで行われ、ロールバックや WAL の生成なども同様に行われる。ROS へ移すかどうかの判定条件は INSERT したトランザクションの状態なので、タプルがオリジナルテーブルから既に DELETE を経て削除されている場合もあるが、その場合タプルは読むことができず、また実際に ROS に移す必要もない。その場合、WOS からのタプルの削除のみを実施する。

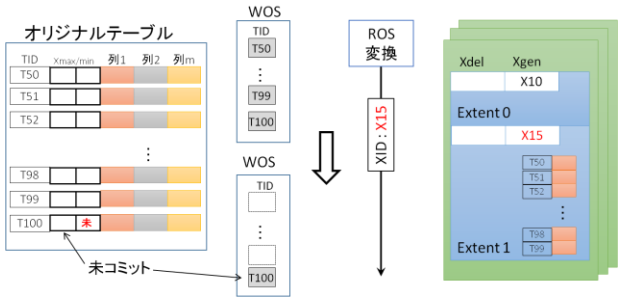

図 4　Data WOS から ROS への変換

4.2 Whiteout WOS から Delete Vector への変換

Delete Vector は、削除された行を表すフラグビットの集まりで、Whiteout WOS から変換される特殊なカラムであると考えられる（図 5）。

Whiteout WOS にあるタプルから Delete Vector への変換条件は Data WOS と同様の条件で、DELETE したトランザクションが現存するすべてのクエリからコミット済みに見えているものとなる。集めた TID のリストを、TIDCRID 変換ツリーを用いて CRID のリストへと作成し、リストから Delete Vector を作成する。

Delete Vector の変更は、通常のカラムと異なり、唯一 in-place にストレージに対して行われるため、変換が途中で失敗した場合、中途半端に反映された Delete Vector が残ってしまう可能性がある。一方で Delete Vector への変換のトランザクションがコミットされていない限り、Whiteout WOS は、変換実行前の状態に戻る。Whiteout WOS の情報が失われない限り、Delete Vector への変換は再度実行が可能である。Whiteout WOS から Delete Vector への変換条件を考えれば、一度 Delete Vector を打つことになった行を、再度見る必要があるクエリは存在しないので、一度立ててしまった削除フラグを消去する必要はない。

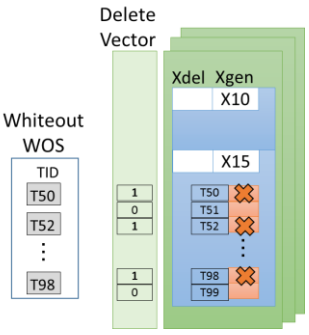

図 5　デリートベクター

4.3 空行の回収

ROS に格納されたエクステントのうち、Delete Vector によって削除されたレコードの割合が一定率を超えた場合などエクステントに含まれる有効な行数が一定量以下になった場合に、空行回収を実施する。

空行回収は、5 章にあるような検索クエリに用いるカラムストアからのデータフェッチを用いて行われる。カラムストアからのデータフェッチを用いると、Delete Vector が反映された実際に可視であるタプルが得られる。これをソースとして、4.1 節のオリジナルテーブルからタプルを抽出した以降と同じ処理を行うことで、有効な行のみを新しいエクステントへと詰めなおすことが可能である。新たに空いたスペースには、Data WOS から ROS へと変換可能な WOS 中のデータを詰めなおす。

その際、ソースとなるエクステントの Xdel と、新しいエクステントの Xgen に自身のトランザクション ID を設定する。

4.4 不要エクステントの回収

前節の空行の回収を行うと、一部のデータは複数のエクステントにコピーが存在する状態になるため、コピー元のエクステントはいずれ必要がなくなる。すべてのクエリから不要になったエクステントは、回収することができる。

また、何らかの異常で作成に失敗したエクステントも、回収される。

5. カラムストアからのデータフェッチ

OLAP 処理では、カラムストアから分析に必要なカラムのデータを読み取る。この際、行ベースのテーブルを使ってデータを読み取った場合と、結果が異なってしまわないようにする必要がある。つまり、ユーザーから見ると、常に結果は同じで、カラムストアが使用された場合には単に latency が向上したように見えなければならない

表 1　CSI の同時実行制御

	①オリジナルテーブル更新	②CSI を利用するクエリ	③CSI を更新するクエリ
①	PostgreSQL が自動的に処理する、不要。	Local ROS で吸収する。Local ROS 作成のための WOS 読み取りルールには、PostgreSQL の関数を流用する	ROS への変換対象となるのは、古い①の結果のみで、コミット前などの新しいデータは変換しないとする
②		CSI クエリは更新を行わないので、不要	エクステント単位の可視チェックを実施
③			変換は排他的に行われるため、不要

このような結果を得るためには、OLAP クエリと並行して走行している、データベースの更新クエリや、ROS 制御コマンドの実行により生成・削除されたデータの扱い、つまり、同時実行制御が必要である。

クエリの種類は、「①オリジナルテーブルを更新するクエリ」、「②CSI を利用するクエリ」、「③CSI を更新するクエリ（ROS 制御コマンド）」3 種類に分類可能で、それぞれの関係について、表 1 にまとめた。

それぞれの関係についてみていくと、まず、①と①の関係は、これは CSI とは無関係である。②と②同士は、同時に行われるが、CSI を利用するクエリは更新を行わないため、問題はない。③と③同士は、排他的に実行される。

①と③の関係については、4.1 節で述べたとおり、変換コマンドの対象となるのは、十分以前更新されたようなデータだけとなるよう、判定を行っている。これにより、WOS には最近更新された（同時実行制御に関係するような）データが残っていることになるが、本章で説明する Local ROS を利用することで、①と②の関係を解決する。

最後に、「②CSI を利用するクエリ」の走行途中に、③が実行される、つまり新しいエクステント生成やコピーが行われる場合については、エクステントに備わる Xgen、Xdel を利用した独自のエクステント単位の可視チェックを行い、データの重複や欠落を起こさないようにする。

5.1 PostgreSQL の同時実行制御（Snapshot Isolation）

表 1 の①同士や、①と②の場合には、PostgreSQL の同時実行制御を利用するため、PostgreSQL の本来持つ同時実行制御の方法について解説する。

PostgreSQL は、複数のクライアントから同時にデータの読み書きを許しており、内部的に多版同時実行制御（Multi Version Concurrency Control：MVCC）を用いて、データの一貫性を管理している。同時実行されるクエリは、現在の実際のデータではなく、ある時点におけるデータベースのスナップショットを参照する。

実際にデータその物のスナップショットをそれぞれのクエリで持つのは現実的ではないため、クエリの開始時にその時点で使用中のトランザクション ID（XID）の「スナップショット」を作成する。このスナップショットに存在しない XID は、すでにコミットかロールバックされている。また、すべてのタプルは、Xmin、Xmax という、自身を生成および削除したトランザクションを記録するフィールドを持っている。クエリは例えば Xmin に記録された XID がスナップショットに含まれているかどうかでそのタプルを INSERT したトランザクションが完了済みかどうかを判定し、その後そのトランザクションがコミット成功したかどうかを判定する。

PostgreSQL では、スナップショットを用いたトランザクションの分離には二つのレベルがある。各クエリの開始時までに終了したトランザクションの結果が見える Read Committed 分離レベルと、トランザクションの開始時点に終了した結果が見える Repeatable Read 分離レベルである。CSI もこの二つのレベルに対応している。スナップショットに加え、述語ロックを使うことで実現される Serializable

分離レベルには対応しておらず、トランザクション分離レベルとして Serializable を指定した場合は、自動的に CSI を使用しない実行となる。

5.2 Local ROS と Local Delete List

WOS から ROS への変換は、クエリ実行とは非同期に実施されるため、クエリ実行時に、WOS の中に未変換のデータが残っていることが考えられる。我々は、これらを一時的にメモリ上で列形式へと変換する。これは、クエリ単位のスコープと寿命を持った ROS と見なすことが可能で、これを Local ROS と呼ぶ。Local ROS の作成と ROS への変換は排他的に実施されるが、一度 Local ROS を作成してしまえば、それらを用いたクエリの実行は、同時に行われる。

WOS から ROS への変換条件を満たしていないような、「あるクエリからは見えて、あるクエリからは見えない」タプルは、すべて未変換データ中に含まれていると考えられる。こういった、クエリごとの可視性の差を吸収するため、Local ROS の作成時に可視性をチェックする。

Data WOS に格納されているタプルは、PostgreSQL の通常のデータと同じ形式なので、それぞれのタプルの可視性をチェックすることが可能である。Data WOS に格納されているタプルは、オリジナルのテーブルと同じ Xmin を持っているので、Local ROS 作成時にこれらの可視性をオリジナルテーブルの可視性チェックに用いるのと同じ Snapshot を用いてチェックすることで PostgreSQL 本来のクエリとデータの一貫性を維持することが可能である。

また、Local ROS 同様、Whiteout WOS にあるデータから、削除されたアイテムの一時的なリストを作成する。これは Local Delete List と呼ばれる。Whiteout WOS から Delete Vector への変換の特徴として、唯一、作成済みのエクステントに対する変更を加えるという点がある。そのため、CSI を用いたクエリ実行で用いるエクステントに対して、Delete Vector の変更が同時に行われる場合があるが、クエリ実行前に作成された Local Delete List により、すでに削除されたように見えている行に対して Delete Vector がセットされるだけであるので問題はない

5.3 エクステント単位の可視チェック

CSI を用いたクエリの実施中にも、各種 ROS 操作は同時に実行されることがあるため、クエリの実施途中にエクステントが増減することがある。従って、エクステント単位での可視性チェックが必要となる。

4 章で述べたとおり、ROS への変換は、エクステント単位で行われるが、このエクステントに ROS 変換を行った際の XID を Xgen、Xdel として記録している。カラムストアインデックスを用いた検索クエリは、クエリの開始時に最後に成功した ROS 変換の XID を Local XID（LXID）として取得する。この LXID と、エクステントに記録された Xgen、Xdel を比較し「 Xgen ≦ LXID ＜ Xdel」が成立するエクステントのみを読むことができる。

5.2 節で説明した Local ROS と、本節で説明したエクステントごとの可視性のチェックを組み合わせることで、オリジナルの結果と同様の結果が得られるという例を以下に示す。図 6 に示す一つ目の例は、CSI を用いた実行の途中に ROS への変換が実施される例である。CSI を用いるクエリは開始時に最後に成功した ROS 制御コマンドのトランザクション ID=X10 を LXID として取得し、Local ROS を作成する。クエリの実施途中に、ROS 変換が行われ、Xgen=X15 をもつエクステント 1 ができるとする。クエリは、「LXID ＜Xgen」となるため、エクステント 1 を読むことはできない。もしこのエクステント 1 を読んでしまった場合、Local ROS とエクステント 1 との間でデータの重複が起こる可能性がある。

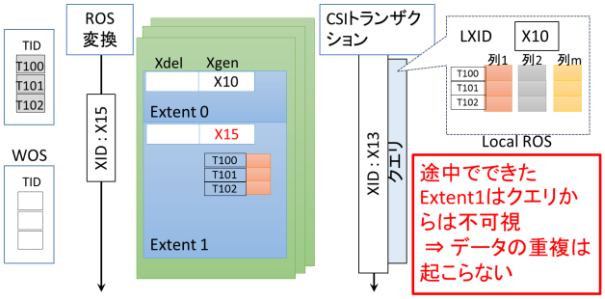

図 6　同時実行制御の例①

図 7 に示す次の例は、クエリの実行前に、別のトランザクションによって、データの INSERT が行われた場合である。CSI を用いるクエリを含む XID=X25 のトランザクションが開始した後に、XID=X26 と X27 のトランザクションが開始し、それぞれ TID=T103 と T104 のデータを挿入した。X26 は X25 で CSI を用いるクエリが始まる前にコミットされたとすると、X25 の CSI を用いるクエリは、未変換のデータのうち T103 のデータだけを Local ROS に取り込むことになる。Local ROS を用いることで、ROS への変換を経ないデータが WOS にあっても PostgreSQL の Snapshot Isolation と同様のリードセットを、得られることがわかる。

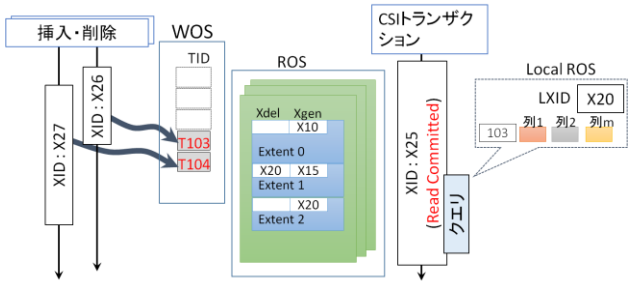

図 7　同時実行制御の例②

また、このエクステント単位の可視性の判断は、ROS 変換のロールバックにも有効である。ROS 変換が、何らかの理由で失敗した場合、次に CSI を用いた処理を実行するクエリは、LXID として、ひとつ前に成功したトランザクション ID を取得する。そうすることで、失敗したトランザクションにより、エクステントが生成されていたとしても、

そのエクステンションは以降のトランザクションから取得されることはない。

5.4 Repeatable Read の実現

CSI の LXID と Xgen、Xdel を用いた独自のエクステント単位可視チェックは、Read Committed 分離レベルであるため、WOS を Repeatable Read 分離レベルで読むと、Local ROS と ROS エクステントでデータの重複が起きてしまう。WOS に対してデータを追加するのはインデックスアクセスメソッドのみで、WOS からデータを削除するのは ROS 制御コマンドのみであるという特徴を利用することで、重複を起こさない Local ROS を作成することができる。

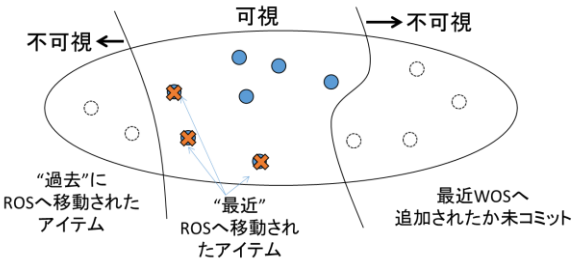

図 8　Repeatable Read の実現

まず、WOS から、Repeatable Read 分離レベルのスナップショットを用いてデータを読み取る。このとき不可視となるデータは、トランザクションの開始以前に ROS へ移されたデータと、トランザクション開始以後に INSERT されたデータとなる。可視となったデータの中には、トランザクション開始以後に WOSROS 変換で削除されたデータも含まれている。したがって、Read Committed 的にエクステントを読むと、重複する。そこで、図 8 のように可視となったデータセットの中から、コミット済みとなっている Xmax が設定されているデータを単純に取り除けばよい。

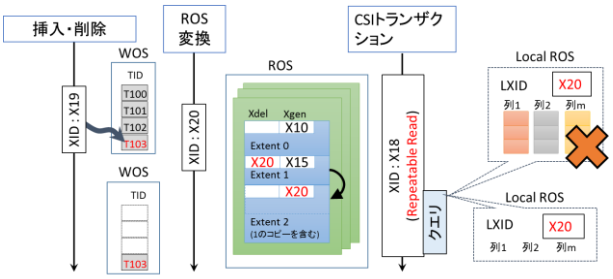

図 9　Repeatable Read の場合の同時実行制御の例

Repeatable Read の場合の同時実行制御の具体例を図 9 に示す。CSI を用いるクエリを含むトランザクション X18 は Repeatable Read 分離レベルである。X18 と並行して、X19 で T103 のデータを挿入し、それがコミットされる以前に、X20 の ROS 変換クエリが、十分以前に挿入された T100 から T102 のデータを ROS へ移動させたとする。これらのクエリのコミット後に、X18 内で、CSI を利用するクエリを実行する。LXID は X20 となり、X20 で作成されたエクステントは可視となる。

Data WOS を PostgreSQL の Repeatable Read 分離レベルにしたがって読むと、T103 はトランザクションの開始以後にコミットされたものであるので不可視で T100 から T102 は同様の理由で WOS から削除されているが、X18 からは可視となるが、T100 から T102 にはコミット済みの Xmax=X20 が設定されているので不可視とし、Local ROS には含めないようにする。これにより、ROS から読んだデータとの重複を防ぐことができる。

6. 性能評価

OLXP を実現するための工夫として、OLTP 性能を損なわないための WOS の軽量な更新と、透過的な OLAP 実行を実現するための Local ROS を用いた同時実行制御を導入した。それぞれの OLTP、OLAP 性能へのインパクトを評価した。

6.1 CSI の更新性能

CSI の OLTP 性能の評価として、CSI を生成した場合の更新性能の評価を行った。ベンチマークソフトとして、PostgreSQL に付属する pgbench を用いた。pgbench は TPC-B におおよそ基づいたシナリオを実施するベンチマークである。3 つのテーブルに対して、それぞれの UPDATE と 1 つの SELECT 文を実施するという単純なベンチマークソフトで、ピーク性能時の更新性能を測定することができる。測定では、テーブルに PostgreSQL の B-tree インデックスを付けた場合、CSI インデックスを付けた場合（通常時および ROS 変換中）それぞれについて、最大スループットを測定し、インデックスを付けない場合との比較を行った。

ベンチマーク評価は、表 2 の評価装置を用いて行われた。測定結果を図 10 に示す。

表 2　評価装置諸元

Machine	Primergy RX300S7
Processor	Intel Xeon CPU E5-2680 @ 2.70GHz 8-cores x2 (HTT off)
Memory	128 GB
OS	RHEL 6.4

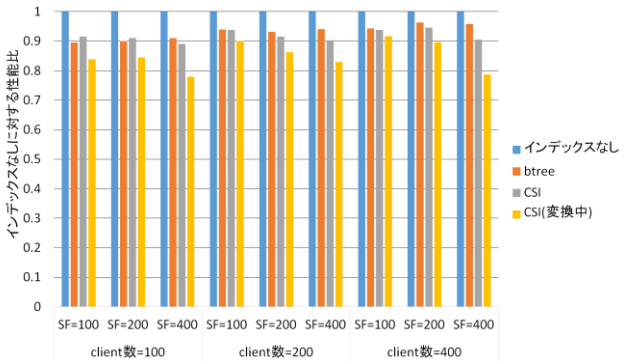

図 10　CSI インデックスを付加した場合の OLTP 性能インパクト

図 10 に示す通り、CSI インデックスをテーブルに付与することによる更新操作の最大スループットの低下は 7.2%、程度であった。 B-tree インデックスの場合のピーク性能の低下は、平均 6.9%であるため、それと比べるとやや悪いものの同程度のオーバーヘッドであることがわかった。また低下率は接続クライアント数やテーブルのサイズ（SF）による変動はあまり見られなかった。

UPDATE は、実際には INSERT と DELETE の 2 つの処理が実行されるため Data WOS と Whiteout WOS 両方に書き込みが発生しさらに、CSI インデックスでは書き込みのデータ形式が通常のタプルであるためタプルのヘッダ情報など書き込み量に対するオーバーヘッドがあるのに対し、B-tree は、INSERT の際はデータの更新を行うが、DELETE に対しては何も行わないといった点が、性能低下率の差の原因になっていると考えられる。

CSI 変換中は、カラム作成時の WAL 書き込みが集中するため、INSERT 性能が低下してしまうが、平均して 15.1%の性能低下となった。今後の課題として、OLTP が最大スループットに近い状態のときは、ROS 変換を遅延させるなどの対策が必要である。

6.2 Local ROS 作成のオーバーヘッド

次に、OLAP クエリに対する Local ROS 作成の作成コストについて評価した。OLXP を実現するために、ROS 更新の残存データで Local ROS を作成するという手法を選んだが、OLAP クエリの純粋なクエリ実行時間に対してどの程度のオーバーヘッドとなるかを調査した。図 11 は、TPC-H の lineitem テーブルのうち、集計に用いるいくつかのカラムに対して CSI インデックスを作成し、WOS に残っているデータ行数に対する Local ROS と Local Delete List の作成時間をプロットしたものである。 Dynamic Background Worker による、ROS への変換が滞りなく進んでいれば、WOS に残っているデータは平均 1 エクステント、つまり 262,144 個のデータ程度に収まる。したがって、CSI のクエリ実行には 0.75 秒程度のオーバーヘッドが生じる。この値は、WOS に含まれるデータ量のみで決まるため、データ全体の規模には影響されないコンスタントな値である。そのため、分析対象のデータ量の増加にしたがって無視できる値になると考えられる。

0.75 秒程度のオーバーヘッドが無視できないような場合、現状クエリ実行の並列化前に実施されている Local ROS 作成も並列化することで、オーバーヘッドを減らす方法が考えられる。また、ROS 変換を起動する閾値は、PostgreSQL の設定変数として、コンフィグファイルなどで設定できるようにしてあるため、想定されるデータサイズによっては ROS 変換をより早いタイミングで実施するようにすることで、WOS にデータがたまりすぎないようにし、後から空行回収時に隙間を埋めていく方法をとることで、Local ROS 作成の時間を減らすことが可能であると考えられる。

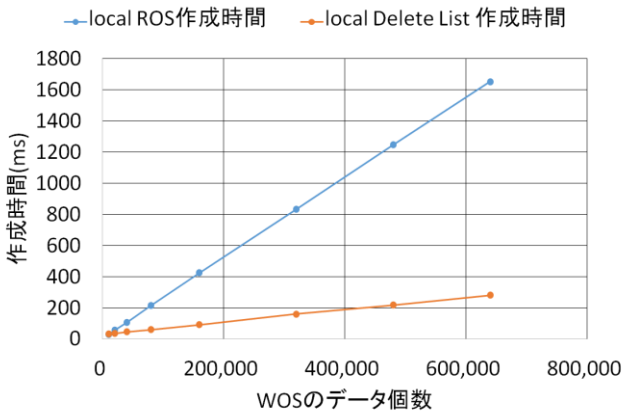

図 11　Local ROS 作成時間の評価

7. まとめ

OLTP 向けの OSS である PostgreSQL を用い、OLXP を実現するためのカラムストアインデックスのストレージ構造及びトランザクション動作に対する制御の実装について述べた。実装したシステムでは、PostgreSQL の OLTP 性能をなるべく損なうことなく、またインデックス作成以外の特別な操作を必要とせずに、透過的なインターフェースで OLAP を実施することが可能な OLXP システムを実現している。そのための工夫として、WOS を用いた軽量な更新と、Local ROS を用いた同時実行制御を導入した。

Pgbench を用いた評価の結果、カラムストアインデックスを付加した場合でも、挿入性能の低下は 7.1%程度で、既存のインデックスと同程度であった。Local ROS の作成は、OLAP クエリの性能のオーバーヘッドとなるため、その作成時間を評価した。TPC-H ベンチマークのテーブル定義を用いた場合、作成時間はデータ規模によらず平均 0.75 秒程度で、分析対象のデータの増大にしたがって無視できる程度となった。今後は CH-benCHmark および実際のアプリケーションを用いて詳細な性能評価を行う予定である。

参考文献

[1] V. Sikka, F. Färber, W. Lehner, S. K. Cha, T. Peh, and C. Bornhövd, "Efficient Transaction Processing in SAP HANA Database: The End of a Column Store Myth", Proc. SIGMOD, 2012.

[2] Oracle Database In-Memory, Oracle, http://www.oracle.com/technetwork/database/in-memory/overview/twp-oracle-database-in-memory-2245633.html

[3] Florian Funke, Alfons Kemper, and Thomas Neumann, "Compacting Transactional Data in Hybrid OLTP&OLAP Database", Proc. VLDB, 2012.

[4] Hasso Plattner,"The Impact of Columnar In-Memory Databases on Enterprise Systems", Proc. VLDB, 2014.

[5] PostgreSQL, http://www.postgresql.org/

[6] 中村 実他,PostgreSQL をベースとしたカラムストア機構の実現検討,Proc. DEIM, 2015

[7] Mike Stonebraker, et al., "C-store: a column-oriented DBMS", Proc. VLDB, 2005.