SCMの利用を想定したOLAP高速化のための列指向バッファ更新用インデックスの提案

塩井　隆円†　　横田　治夫††　　波多野　賢治†††

† 同志社大学大学院 文化情報学研究科　〒610–0394 京都府京田辺市多々羅都谷 1-3
†† 東京工業大学 情報理工学院　〒152–8550 東京都目黒区大岡山 2 丁目 12–1
††† 同志社大学 文化情報学部　〒610–0394 京都府京田辺市多々羅都谷 1-3
E-mail: †shioi@ilab.doshisha.ac.jp, ††yokota@cs.titech.ac.jp, †††khatano@mail.doshisha.ac.jp

あらまし　DWH は基幹系業務システムで管理される業務データを DBMS のスキーマを変えて格納し，効率良くデータ分析を行うことで意思決定支援に役立てられてきた．列指向型 DBMS はメモリ上で行指向にデータを格納し，論理スキーマを分割して物理的に格納することで一つのシステムで効率良く DWH に利用することができる．しかしながら，大量に蓄積したデータを高速にデータ分析するために様々なパタンでスキーマを物理的に格納し，メモリ上で実行する OLTP によって更新されるデータをディスク上のデータとのソート状態を保ちつつ複数のスキーマに対して格納するため，揮発性のメモリ上で実行する OLTP の増加に伴い大量データを扱う OLAP に使用できるメモリキャッシュ領域が少なく，一つのシステムで OLTP と OLAP に対応したことが OLAP 性能の低下を招く．そこで，本稿では不揮発性のメモリである SCM の利用を想定して OLTP の実行した後に参照効率の高い列指向データの更新を行うインデックスを提案し，インデックスを利用して効率的に Materialization を行うことで OLAP の高速化を行った．
キーワード　DB システムアーキテクチャ，DWH，インデックス・データ構造

1. はじめに

これまで，業務で蓄積した大量データを企業の意思決定に役立てるために Data Warehouse（以下，DWH）を利用したデータ分析が行われてきた．基幹系業務システムで主に採用される RDBMS は表形式で管理するデータを一行単位でディスク上に格納する行指向型データ格納方式を採用し，業務で生じるトランザクションを複数の行で処理する Online Transactional Processing（以下, OLTP）を一行ずつ扱うことが可能である [5]．OLTP を効率良く実行する RDBMS では，行指向に蓄積されたデータに対してデータ分析系の処理である Online Analytical Processing（以下，OLAP）を実行する際に，一行単位で分析対象の列データを取得するため処理に不要な列データも取得してしまう場合が多く，OLTP と OLAP の性能は物理的に格納されるテーブルのスキーマに依存するトレードオフの関係にある [1,3]．そのため，大量の蓄積した業務データに対する OLAP を高速に実行するためには OLTP を効率良く実行した RDBMS 等のスキーマを分割して DWH に格納する必要があった [17]．

近年 DWH に利用される列指向型 DBMS [6,16] は，図 1 のようにユーザが定義したスキーマの OLTP をメモリ上で実行し，効率良く OLAP の実行を行うためにスキーマを分割してディスク上に格納するため一つの DBMS で OLTP と OLAP に対してスキーマを変えて対応する DWH である．
また，列指向型 DBMS では行指向型データ格納方式とは対象的に表形式のデータをディスク上に一列単位で同じ型のデータを連続して格納する列指向型データ格納方式に合わせた問合せ処理方法を採用するため，OLAP 問合せ処理に対するソートや圧縮を利用した最適化が可能となる [2,18]．しかしながら，列単位でソートや圧縮を行うことで OLTP によって反映されるデータに対してもソートや圧縮を行う必要があるため，OLTP の際にはディスク上のデータとメモリ上のデータを読み出し，OLAP の際にもメモリ上のディスク上に反映されていないデータとディスク上のデータを読み出さなければならない．さらに，列指向型 DBMS はディスク上に格納したスキーマの列データに対するソートパタンを変えて同じ列データであっても複製する materialization によって OLAP の様々な問合せに対応するため，ディスク上に複製された列データに対してもソートパタンに合わせて OLTP を反映させる必要がある [7,10,16]．ユーザが発行する OLTP の問合せは元の論理スキーマに対して実行され，分割して複製された複数の物理スキーマに対してディ

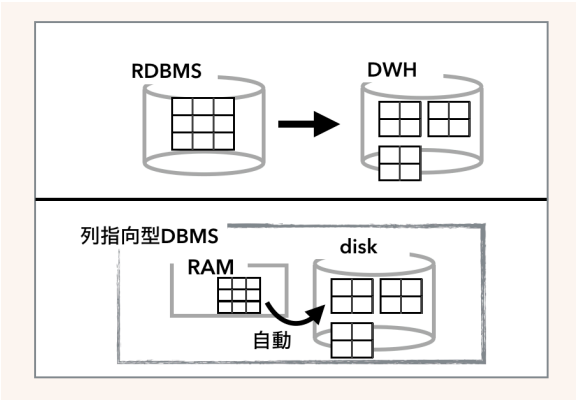

図 1　DWH

スクからデータを読み出しつつメモリ上でソートや圧縮処理を行い，一定の行数をまとめてディスク上に INSERT することで反映させるため，揮発性のメモリ上で処理するデータやログ処理が増加してしまうことで大量データを分析する OLAP に利用できるキャッシュ領域が減ることになる．データ更新頻度の少ない DWH の特徴を活かし，OLAP を高速化するためにディスク上で複数の materialization を行う列指向型 DBMS は，メモリ上で実行される OLTP のメモリ使用領域が大きくなるため OLTP の頻度が増えた際のリアルタイムなデータ分析における OLAP 実行速度が遅くなってしまう [9].

そこで本稿では，不揮発性のメモリである SCM に着目し，SCM 上で行指向に OLTP を実行した後に DRAM 上に保持した列指向データを更新するインデックスを提案することで，提案するインデックスを利用した DRAM 上でのキャッシュ領域を効率良く利用する materialization を行う DB アーキテクチャを考案する．提案するインデックスを利用する DB アーキテクチャは，列指向型 DBMS がディスク上にスキーマを分割して格納する materialization によって増加するメモリ上の OLTP 負荷を減らすために，インデックスを用いて DRAM 上のキャッシュ領域に materialization を行うことで，OLTP / OLAP 性能の向上を図るため DWH のリアルタイム性能向上に役立てられると考えられる．

2. 列指向型 DBMS

DWH では格納されるデータの特徴に応じてユーザが発行する分析系の問合せ処理は似通いやすく，問合せに応じてデータを運用することで OLAP 性能を向上させることができる．そして，列指向型 DBMS はメモリ上で行指向に OLTP を実行した後にディスク上にスキーマを分割して格納するため，一つのシステムで OLTP を実行しつつ DWH に利用される．しかし，ディスク上で分割されたスキーマに複製される列データに対しても OLTP によるデータ更新を反映する必要があり，データを反映する際にはディスク上のデータを読み出し，反映されるデータとディスク上のデータのソート状態を保ちつつ蓄積したデータを圧縮する処理が必要となる．本節では，本稿で提案するインデックスを利用したメモリ上の materialization 処理のために参考とした DWH に利用される列指向型 DBMS の特徴について 2.1 節で説明を行い，従来の列指向型 DBMS が行うディスク上に materialization したテーブルに対してメモリ上で実行する OLTP による更新中データのソートとディスク上データへの反映手法について 2.2 節，メモリキャッシュ上の列指向データに対する検索のために参考としたソートされた列指向データの sparse な読み出しについて 2.3 節で説明を行う．

2.1 materialization

列指向型データ格納方式では列単位で格納されるデータは各列のデータがディスクページ単位で連続して格納されるため，メモリに読み込まれるデータに対して CPU が SIMD で処理を行う単位を各列のデータの行に合わせて連続して処理することができる．各列のデータが連続して SIMD のレジスタに読み込まれるため，データの問合せ処理も各列単位で実行することで列指向型データ格納方式を利用した場合に CPU が並列して列データを処理することができる [13]．また，行指向型データ格納方式においても現代の CPU の進化に合わせて効率的にデータを処理するように最適化することが可能である [11] が，表形式の各列単位で格納されたデータはページ単位で列指向に連続してデータを格納することで同じデータ型のデータが連続するためデータ圧縮効率が高いというメリットがある [2, 18]．しかし，DBMS の問合せ処理では問合せや格納されるデータのスキーマ，各列内のカーディナリティに依存して行指向／列指向に格納された双方のデータ格納方式においても効率良く問合せ処理が可能となるため，ディスクページ内で CPU キャッシュ効率を向上させるために行指向と列指向に連続してデータを格納する方法に PAX [4] 等が提案されている．列指向型 DBMS においても PAX と同じように行指向／列指向にディスクページ内のデータを格納することで，関連の強い複数の列データを物理的に近い場所に格納することで行指向データのメリットを取り入れることが可能となる [12]．ページ内に複数の列データを格納する方法は列指向型 DBMS の多くに採用されており，Vertica [10] の Projection では同じ列データでも複数の Projection に物理的に格納され，それぞれの列データが関連の強い列同士で様々なパタンでそれぞれの Projection の主な列データをソートキーとみなしてソートすることで様々な問合せパタンに対応することが可能となるため柔軟に高速な OLAP を可能としている．列指向型 DBMS で利用されている materialization は，OLAP 高速化のためにディスク上で様々なスキーマが物理的に格納されるため，メモリ上で実行する OLTP によって更新されるデータをディスク上のデータと合わせてソートした状態を保つために複数のスキーマに対する OLTP の負荷が増加してしまうというデメリットがある．

2.2 update

列指向型 DBMS は図 2 のように OLTP による行指向のデータ更新をメモリ上で実行し，セグメントと呼ばれる一定の数十万行単位のデータを列指向に変換することで，ディスク上に列指向にソートして圧縮したデータを格納するデルタバッファを用いることが多い．

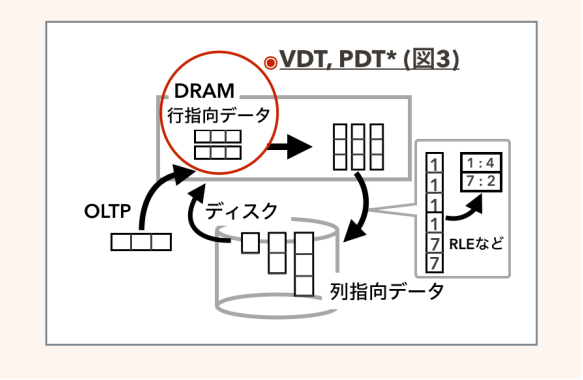

図 2 列指向型 DBMS

デルタバッファに格納されるデータは，メモリ上で実行する OLTP を Vertica [10] が採用する VDT (Value-based Delta

Tree) のように INSERT と DELETE をそれぞれのテーブルに格納し，デルタバッファ上の INSERT テーブルとディスク上のテーブルを Merge Union，デルタバッファ上の DELETE テーブルとディスク上のテーブルを Merge Diff することでデルタバッファ上に蓄積した複数の OLTP のデータをディスク上に更新する．Vertica の VDT はメモリ上のデルタバッファのサイズを小さく保つために，デルタバッファのデータをディスクに反映する際にディスク上のデータを 15 万行単位のセグメント等を利用して読み出して Merge しているが，Projection 内のソートキーとデルタバッファ内のソートキーをすべて読み込むキー同士の MergeUnion が必要となる．そのため，ソートキーの列データを基にしてデルタバッファ上のデータが属する範囲を特定することですべてのデータを読み出さないように工夫する PDT (Positional Deleta Tree) [7] が提案されている．

列指向型 DBMS では関連した列を Vertica の Projection のようにテーブル単位で物理的に格納して扱うため，テーブル内のソートキー単位で行を特定する．PDT では，メモリ上にディスク上のソートキーである SID と OLTP によって更新されるデルタバッファ上のソートされた RID を基に B-tree のように管理することで Merge を行うことが可能である．メモリ上で実行される OLTP は図 3 のように insert table と delete table，そして列単位の update table に分けて格納される．また，delete table はソートキーの列データのみを格納し，update table はそれぞれの列の修正する値のみを格納している．最初にディスク上から読み出したスキーマのソートキーを SID として木構造のピボットとして格納し，delta は $INSERT - DELETE$ の数をそれぞれ格納しておき，Δ も前回のトランザクションから delta と同じように $INSERT - DELETE$ から計算することで右側のノードの最小 RID を $RID = SID + \Delta$ で計算することができるため，ディスクから読み出したソートキーである SID を基にした木構造を用いて，メモリ上で実行した OLTP によって更新されたデータに対してもテーブル上の順番を保つための RID を格納することなく計算できる．

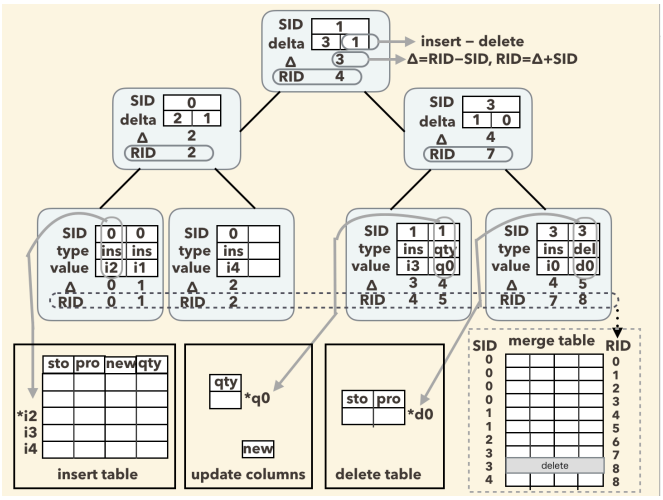

図 3　PDT: Positional Delta Tree [7]

2.3　sparse index

PDT のピボットとしてディスクから読み出す SID を効率良く特定するために，列指向データのソート状態を利用して sparse に読み出すことで高速に検索する sparse index が利用されている [15]．MemStore [14] では，列指向データを sparse に読み出すために OLTP でメモリ上に格納するデータとディスク上に格納するデータの単位を一定のセグメントとして，圧縮されたセグメント単位の列指向データに対して最大値，最小値をメタデータとして保持しておくことで，デルタバッファから書き込まれるメモリ上に格納されたセグメント単位の列指向データの最大値より大きい最小値かつ最小値より大きい最大値を持つディスク上に格納されたセグメント単位の列指向データを各セグメント単位で取得し，空のセグメントを対象となったセグメント数+1 分用意することでまとめてソートして格納することで最大値，最小値等のメタデータを利用してデータ範囲を絞ることで OLTP のデータ更新処理を行っている．また，DWH に格納されるデータに対するユーザの問合せは似通うため OLTP によるデルタバッファを利用したセグメント単位のインデックスだけでなく，ユーザが利用した OLAP の問合せに合わせて効率良く格納する列指向データの順番を連続させて木構造で管理することで DBMS チューニングの必要もなく自動的に OLAP を高速化する列指向型インデックスが存在する [8]．これらの sparse に読み出す工夫は，OLTP の際には PDT が管理する SID を検索する際にディスク上の全ての SID を読み出さないようにするために利用され，OLAP の際には処理に利用するデータの範囲を特定することで不要なデータを読み出さずに済むため，OLTP/OLAP 双方においてソートして格納された列指向データを効率良く利用することが可能である．

3.　提 案 手 法

列指向型 DBMS は列指向型データ格納方式の特徴に合わせて CPU や安価に大規模化したメモリ等の現代のデバイス技術を効率良く利用するために最適化することで高速に OLAP を可能としたため，列指向型データ格納方式が着目されるようになった背景として DBMS が依存するデバイス技術の進歩が挙げられる．そして近年 DBMS アーキテクチャの変革が起きる可能性があるとして着目されているデバイス技術に，不揮発性のメモリである SCM がある．データを記憶するために新たに利用できる SCM は，データを記憶した後に電源が切れた場合でもそのデータを永続して記憶することができるため，従来の DRAM やディスクの二種類を記憶領域として利用するよりも DBMS の可用性が向上すると考えられる．そのため，SCM の利用が現実化した場合や SCM として Flash Memory を代替して利用する際の DBMS アーキテクチャを構成するアイデアが効果的であると考えられる程，SCM と呼ばれる多くのデバイス技術の進歩を促すことが可能である．そこで，本稿では SCM の効率的な利用を想定した DBMS のアーキテクチャを考案する．

従来の列指向型 DBMS では，ディスク上に列データの複製を行い複数のスキーマを格納することで OLAP の最適化を行っ

てきた．しかしながら，OLAP 高速化のために materialization した複数のスキーマに対する OLTP のディスク上への反映処理が増えてしまい，ソートや圧縮のためにディスク上データの読み出し処理に加え，OLAP で更新中のデータに対する検索処理が必要となり更新の多いリアルタイムなデータ分析が求められるアプリケーションでは利用できない．

本稿では，SCM を効率良く利用することで DWH に利用される列指向型 DBMS の OLTP 性能と OLAP 性能を向上させ，リアルタイムデータ分析基盤として利用可能とするために，図 4 のように従来の列指向型 DBMS がメモリ上で行ってきた OLTP を SCM 上で行指向型データ格納方式を用いて実行し，OLTP を実行しないメモリ上で利用可能となるメモリキャッシュ領域をユーザの問合せに合わせた materialization のために利用するデータ反映用の update インデックスを SCM 上に作成する．

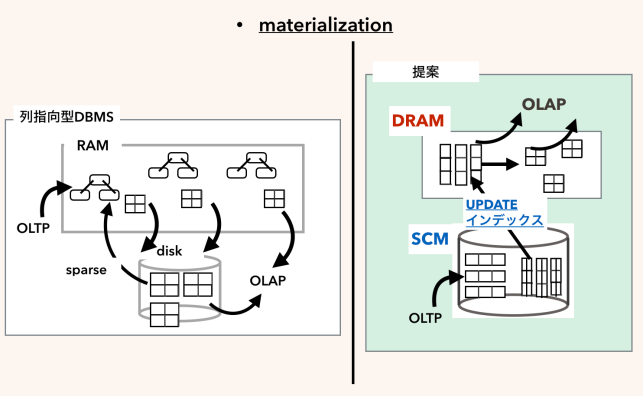

図 4　materialization の違い

また，従来の列指向型 DBMS のデルタバッファは DRAM 上で実行していたため，データが揮発することを防ぐためにログ領域をディスク上に保持する必要があることや DRAM 上のキャッシュ領域が少なくなってしまうというデメリットもあった．そこで，図 5 のように SCM 上で行指向型データ格納方式を利用して OLTP を実行し，OLAP 高速化のためにメモリ上にキャッシュした列指向データに対して OLTP の更新を update インデックスを用いて反映する．

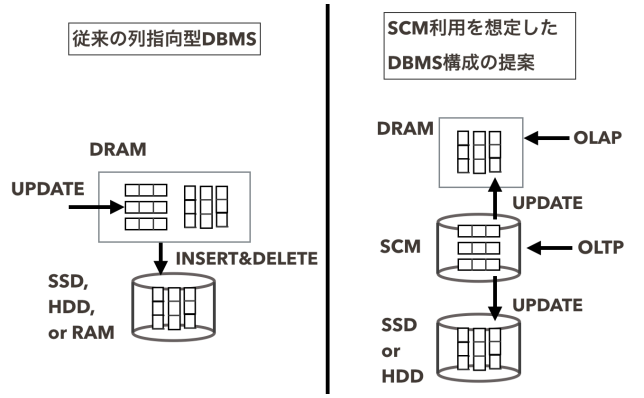

図 5　UPDATE 処理における従来の列指向型 DBMS と提案する構成の比較

本稿では，大規模な SCM の実用は未だ成されていないため，図 6 のように SCM を効率的に利用するための DBMS アーキテクチャの評価のために，行指向型データ格納方式を採用する RDBMS を用いて行指向にデータを格納し，DRAM 上には UPDATE 用のインデックスとしてハッシュテーブルを作成し，列指向のテーブルデータを作成した．

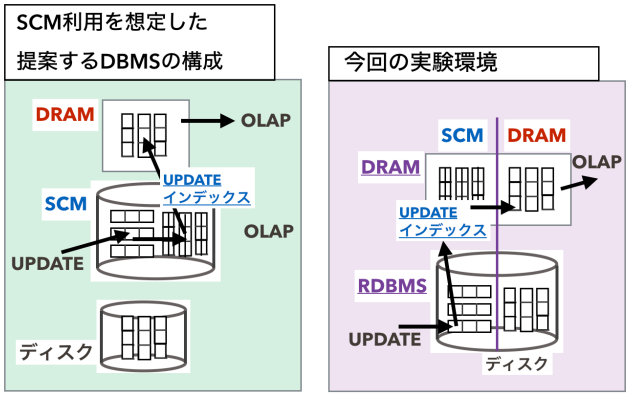

図 6　提案する UPDATE 用インデックスの実験環境

UPDATE 用のインデックスとして作成するハッシュテーブルは，図 7 のように RDBMS に格納される物理的な格納位置である TiD を基に作成することで，RDBMS で UPDATE された行の，DRAM 上の UPDATE 対象の列の行を一意に特定することで高速にバッファ上の列データを UPDATE することが可能となる．また，OLAP 性能を向上させるためにバッファデータを列指向データとして 15 万行のセグメント単位で各列ごとにソートして最大値の max と最小値の min を持たせている．

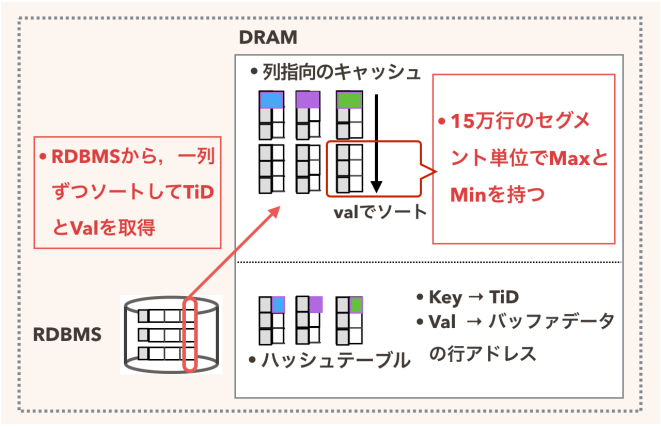

図 7　ハッシュテーブルの作成

UPDATE の処理は図 8 で示すように，図 7 で述べた RDBMS の TiD が，本稿で利用する PostgreSQL で採用される MVCC (Multi Version Concurrency Control) では UPDATE した後に変更されてしまう．そのため，本稿で提案する UPDATE 用インデックスでは以下の順番で UPDATE 用インデックスの処理を実行する．

(1) 問合せ解析　UPDATE が行われる問合せ内の条件句を取得（Condition1）

＊本実験環境で利用する PostgreSQL では UPDATE 後に TiD が変わってしまうため，UPDATE 後の TiD を取得し，ハッシュテーブルのキーと DRAM 上のバッファデータの TiD も更新する．

(2)UPDATE の準備　Condition1 を基に RDBMS，もしくは列指向のバッファデータから物理位置を取得（ID1）

(3)UPDATE の実行　ID1 を基に UPDATE を実行し，同時に UPDATE が行われた行の物理位置を取得（ID2）

＊ ID1 を基に UPDATE を実行しその TiD を一行単位で取得することで別の行を誤って UPDATE しないようにする必要がある．

(4) ハッシュテーブルの UPDATE　更新前のハッシュテーブルの key を基に val のアドレスを取得し，ハッシュテーブルの key と DRAM 上の TiD を ID2 に入れ替える

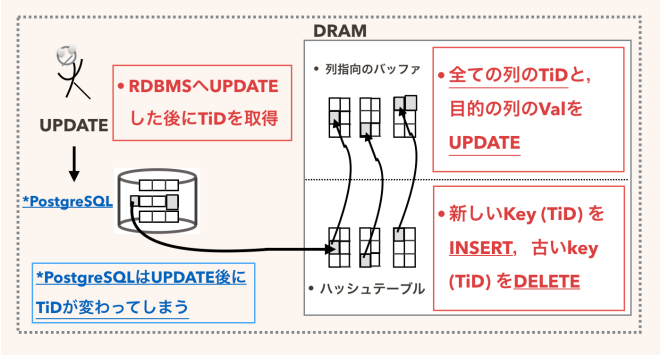

図 8　UPDATE 用インデックス

データの更新処理の中でも UPDATE はデータをデルタバッファ上に INSERT し DELETE は論理的な削除フラグを作成することで INSERT と DELETE を分割して処理することが一般的であったが，UPDATE 用のインデックスではキャッシュ上のデータとディスク上のデータを一意に特定して上書きを行うために TiD を用いてハッシュテーブルを作成しバッファ処理を行う．UPDATE を効率良くバッファデータに反映することで，リアルタイムに OLTP を更新しつつ OLAP を高速に実行するリアルタイム OLAP の高速化を行うことが可能となる．

また，列指向型データ格納方式は大量データから少量のデータを取得する問合せの実行が行指向型 DBMS のインデックスに比べて不利であるとされている．本研究で考案した SCM の利用を想定した DBMS アーキテクチャでは，OLAP 高速化のためにハッシュテーブルを利用してキャッシュされたデータを UPDATE することで，これまで列指向型 DBMS でデルタバッファのために利用されていたメモリ領域を多く利用することが可能であり，UPDATE されたキャッシュデータはそのまま OLAP のために再利用することが可能となる．本研究で提案する UPDATE 用のインデックスは，物理的な格納位置を取得するのみで列指向データの行を一意に特定することが可能であるため，キャッシュしたデータに対してより高速に検索する

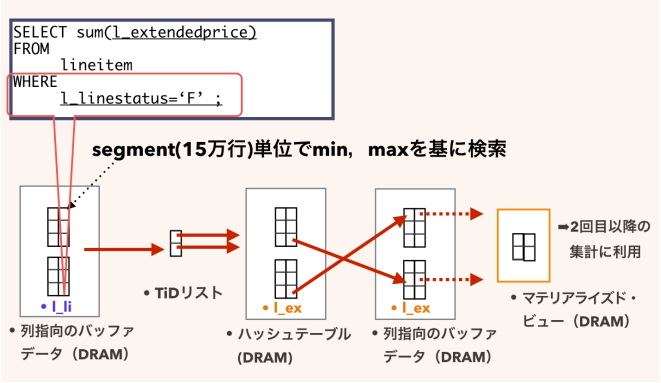

図 9　OLAP 高速化のためのマテリアライズドビュー（Projection）の作成

ことができる．

そこで，図 9 のように OLAP の問合せで利用される条件句から列指向のバッファデータを利用してセグメント単位で min，max を基に検索を行い，条件に当てはまる TiD のリストを作成した後に UPDATE 用のハッシュテーブルを利用して集計対象の列データを取得することで，Projection のようにキャッシュ上の列データから問合せ内で必要となる列データをソートしてキャッシュするマテリアライズドビューを作成することで OLAP の高速化を行う．

4. 評　　価

本実験では Amazon Web Service の Amazon Elastic Compute Cloud を利用して r4.8xlarge の計算機（Intel(R) Xeon(R) CPU E5-2686 v4 @ 2.30GHz，Red Hat Enterprise Linux Server release 7.1 (Maipo)，EBS 汎用 SSD (gp2) 500GB，DDR4 244GiB のメモリ）と gcc 4.8.5，GLib 2.49.7 を用いて UPDATE 用のインデックスとマテリアライズドビューを作成し，RDBMS として PostgreSQL9.4.4 を利用した．また，実際にハッシュテーブルを用いて UPDATE と Aggregation の問合せ処理を行う際に本研究で提案した DB アーキテクチャの有用性を図るために，TPC-H ベンチマーク（スケールファクター：SF=1，10）のデータを用いて OLTP ／ OLAP 性能の評価を行った．本稿で使用する PosgreSQL は UPDATE 後に物理的な格納位置である id が変わってしまうため，提案した update インデックスでは UPDATE であってもメモリキャッシュ上の列指向データの行キーとして持たせている TiD の更新が INSERT と DELETE の組み合わせで実行する動作と同じである．そこで，本実験では UPDATE の問合せを用いて，selectivity 約 50%の以下の問合せを用意し，UPDATE に掛かる時間を評価した．

```
UPDATE lineitem
  SET l_extendedprice=l_extendedprice+5
  WHERE l_returnflag='N'
```

列単位でキャッシュされたデータを UPDATE した後に TPC-H ベンチマークの問合せ#1 を実行した．また，ハッシュテーブルを用いてマテリアライズドビューを作成して Aggregation し

た場合とハッシュテーブルのみを用いて Aggregation した場合を想定して実験を行った．ハッシュテーブルを用いた Aggregation では，OLAP 問合せの条件句に当てはまる行の TiD を取得し，問合せ内で SELECT 句や GROUPBY 句で利用する列の各列ごとのハッシュテーブルに対して TiD の key を基に DRAM 上のデータを一行ずつ集計した．マテリアライズドビューの作成は，TPC-H ベンチマークの問合せ#1 に記述される条件句に当てはまる行に対して Projection のように GROUPBY 句を基にソートキーとしてソートし，Aggregation は問合せ内で記述される SELECT 句に基づいて行った．

図 10 に示す表は，上から RDBMS で実行した UPDATE の問合せを実行した場合の実行時間，中段の表はハッシュテーブルを用いて列指向のバッファデータを 10 回 UPDATE した際の平均実行時間，下段の表は PosgreSQL に UPDATE の問合せを実行した後の列指向バッファデータを UPDATE した際の平均実行時間である．

RDBMS Selectivity 50%	SF1	SF10
1 回目 (vacuum直後)	122.3秒	1946.3秒
2 回目	315.4秒	1439.8秒
3 回目	538.3秒	2530.4秒
4 回目	485.0秒	1641.9秒
平均 (10回)	460.0秒	1876.6秒

バッファデータのUPDATE	SF1	SF10
平均 (10回)	34.3秒	303.5秒

UPDATEインデックス：	SF1	SF10
RDBMS + バッファ更新	494.3秒	2180.1秒

図 10　UPDATE インデックスの実行時間

図 10 の結果から，追記型の PosgreSQL の Vacuum（不要領域回収）直後では，SF1 の lineitem テーブルの場合ではデータ量が小さく最も高速に UPDATE の実行が可能だが，SF10 の場合ではデータ量が多く空の利用可能な領域にデータを挿入することが可能であるため，UPDATE によってできた隙間が却って平均実行時間を短縮したと考えられる．また，今回提案したバッファデータの UPDATE では主に以下の 2 つの処理

（1）WHERE 句を基に列指向のバッファデータから，UPDATE 対象の TiD を取得（更新前 TiD）

（2）RDBMS で UPDATE された TiD（更新後 TiD）は，更新前 TiD を基にハッシュテーブルから全ての列データの更新後 TiD を INSERT，更新前 TiD の DELETE

を行う必要があるため，SF10 の場合では約 300 秒ほどバッファデータの更新に時間が掛かってしまった．今後，UPDATE によって変わってしまう TiD を効率良く更新するように UPDATE 用インデックスの処理方法を考える必要がある．

図 11，図 12 では TPC-H ベンチマークの問合せ 1 の条件を変えて Selectivity を変えながら実験することで，大量データから少量データを検索する列指向型データ格納方式の不利な点にハッシュテーブルの検索効率の高さがどれほど貢献できるかを確認した．

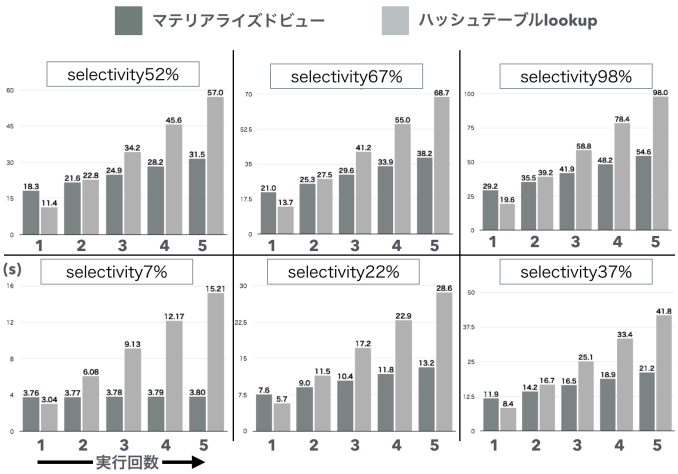

図 11　SF1 のときの selectivity7〜98%のマテリアライズドビューの評価

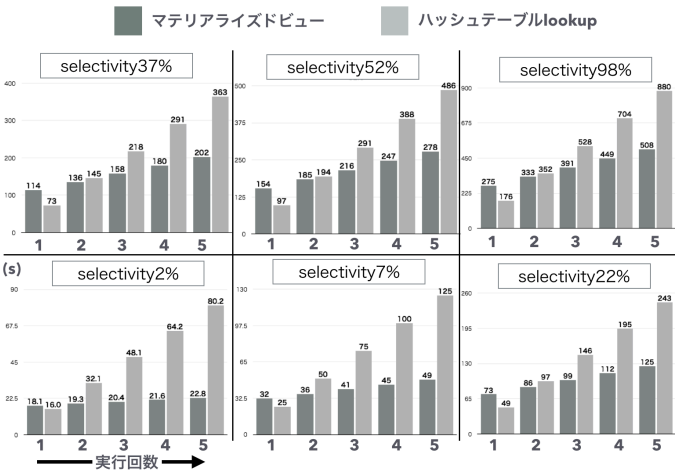

図 12　SF10 のときの selectivity2〜98%のマテリアライズドビューの評価

Selectivity が小さい程，ハッシュテーブルから高速にマテリアライズドビューを作成でき，マテリアライズドビュー自体の検索時間も早いことが分かった．また，Selectivity が大きくなると，ハッシュテーブルからマテリアライズドビューを作成する時間が遅くなってしまうことが分かった．さらに，Selectivity が 98%のように大きくなる場合には，マテリアライズドビューを作成する時間は最も遅くなってしまうが，マテリアライズドビュー自体の集計時間がハッシュテーブルを介して Aggregation するよりも早いため実行回数を重ねるほどマテリアライズドビューを作成して Aggregation した方がハッシュテーブルを利用するよりも効率的な集計が可能である．

本節の実験から，Selectivity が小さいほどハッシュテーブル

を介した Aggregation に時間が取られないためマテリアライズドビューの作成時間が早く Selectivity が大きい場合においてはマテリアライズドビューの作成に時間が掛かっていた．この結果は，従来の列指向型データ格納方式のデメリットである大量データから少量データを検索する問合せに対して効果的であると考えられ，また大量のデータに対しては従来の列指向型 DBMS が行ってきた最適化手法によって高速な OLAP の実行に役立てられると考えられるため，本研究で提案した DBMS のアーキテクチャにおいては，Selectivity の小さいマテリアライズドビューを Projection のようにメモリ上に管理することで様々な問合せに対応することができるため列指向型 DBMS の OLAP 高速化が可能である．そのため，Selectivity が大きい場合にキャッシュデータをそのまま Aggregation した場合とマテリアライズドビューの実行時間を比較する実験を様々な OLAP の問合せを用いて今後行う必要がある．

5. おわりに

1970 年代に提案された列指向型データ格納方式は，計算機を構成する CPU やメモリ，ディスク等のストレージ技術の進化に伴い列指向の問合せ処理実行に最適化することで 2000 年代に提案された列指向型 DBMS の登場以来再び注目を集めるようになったデータ管理方法である．現在では 2000 年代に提案された列指向型 DBMS を基にした多くの商用 DBMS が利用されており，DBMS に関連する研究において近年の企業が先行する多くの技術研究だけではなくその知的資産を生み出す学術も重要だと考えられる．特に DBMS は計算機を構成するデバイス技術などに性能が依存することも多く，デバイスの特徴に合わせた DBMS アーキテクチャを構成する必要がある．本研究では，近年着目されている新たなデバイス技術の SCM の特徴を効率良く利用するための DBMS を想定した DBMS アーキテクチャについて提案した．

SCM はディスクより高速かつ不揮発な RAM とされているため，これまで列指向型 DBMS が DRAM 上で行っていたデルタバッファを SCM 上で実行することで実行時間の掛かる OLAP の高速化のために DRAM 上でより多くのキャッシュ領域を持つことが可能となる．そこで，リアルタイムにデータを更新しながらデータを分析するリアルタイム OLAP の高速な実行のために，本研究ではメモリ上のキャッシュされたデータに対してリアルタイムに UPDATE の実行を行う列指向のバッファデータの更新用インデックスをハッシュテーブルを用いて作成することで，TPC-H ベンチマークの SF1，SF10 のそれぞれを利用した場合に RDBMS の Vacuum 直後の Selectivity50%の UPDATE 時間に比べてそれぞれ 28%，16%の実行時間でバッファデータの UPDATE が可能となり，Vacuum 後から 10 回の平均実行時間に比べてそれぞれ 7%，16%の実行時間で UPDATE が可能となった．

また，OLTP の UPDATE のために利用した DRAM 上にキャッシュされた列指向データに対する UPDATE 用インデックスを用いることで，OLAP の Aggregation 対象の行を一意に取得することが可能となるため，UPDATE 用インデックスを用いて Aggregation を行う場合と，UPDATE 用インデックスを用いて高速にマテリアライズドビューを作成し，Aggregation を行う場合の実行時間の比較を実行回数と行の Selectivity（テーブルの行数に対する行の選択率）に着目して評価を行った．

何れの Selectivity であってもマテリアライズドビューを作成するために一回目の Aggregation の実行時間に掛かった時間的コストを，二回目の実行で取り戻すことが可能となった．Selectivity が 7%以下の場合では，ハッシュテーブルを利用してマテリアライズドビューを構築する時間と，マテリアライズドビュー自体の検索時間が早く，二回目の実行回数で大きく高速化できることが分かった．そのため，Selectivity が 7%以下の場合には予めマテリアライズドビューを作成することでもう一度同じ問合せが実行された場合に問合せ処理を高速に実行することができる．近年では安価に大規模化なメモリ領域を利用することができるため，Selectivity が小さいほど作成されるマテリアライズドビューのデータ量も少なく，提案した SCM を利用するための DBMS のアーキテクチャにおいて積極的にマテリアライズドビューを作成することによって OLAP の高速化に役立てることが可能である．

本研究で提案した DBMS アーキテクチャでは，ディスクよりも高速にアクセスできる SCM 上に格納する行指向データと，DRAM 上にキャッシュする列指向データをハイブリッドに利用することで，従来のユーザの問合せに応じて列指向データを並び替える OLAP 高速化のための列指向のインデックスよりも様々なユーザが初めて実行するような OLAP 問合せに対して行指向／列指向データを併用することで，初めて実行するような問合せに対しても柔軟な問合せ処理が可能となり OLAP を高速化できると考えられるため，行指向／列指向データを柔軟に利用するための行指向型／列指向型問合せ処理方法を今後考える必要がある．また，SCM の実用的な利用ができない場合であっても NAND Flash Memory 等を用いて SCM の利用を仮定した本稿の DBMS アーキテクチャを，Flash Memory 上の OLTP ／ OLAP 実行速度に着目して実用的な評価を行う必要がある．

謝　辞

本研究の一部は JSPS 科研費 26280115，16H02908，文部科学省私立大学戦略的研究基盤形成支援事業 S1411030 の助成を受けたものである．

文　献

[1] D. Abadi. Hybrid Row-Column Stores: A General and Flexible Approach. http://blogs.teradata.com/data-points/hybrid-row-column-stores-general-flexible-approach/, March 2015. Retrieved September 2, 2015.

[2] D. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 671–682. ACM, June 2006.

[3] D. J. Adadi, S. R. Madden, and N. Hachem. Column-stores vs. row-stores: how different are they really? In *Proceed-*

ings of the 2008 ACM SIGMOD international conference on Management of data, pages 967–980. ACM, June 2008.

[4] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving Relations for Cache Performance. In *Proceedings of the 27th International Conference on Very Large Data Bases*, pages 169–180. VLDB Endowment, June 2001.

[5] H. Garcia-Moluna, J. D. Ullman, and J. Widom. *Database Systems: Pearson New International Edition: The Complete Book*. Pearson, August 2013.

[6] S. Héman, N. Nes, M. Zukowski, and P. Boncz. Vectorized data processing on the cell broadband engine. In *Proceedings of the 3rd international workshop on Data management on new hardware*, page 4. ACM, 2007.

[7] S. Héman, M. Zukowski, N. J. Nes, L. Sidirourgos, and P. Boncz. Positional update handling in column stores. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 543–554. ACM, 2010.

[8] S. Idreos. Database cracking: Towards auto-tuning database kernels. *CWI and University of Amsterdam*, 2010.

[9] A. Jacobs. The Pathologies of Big Data. *Communications of the ACM*, 52(8):36–44, August 2009.

[10] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear. The vertica analytic database: C-store 7 years later. *Proceedings of the VLDB Endowment*, 5(12):1790–1801, 2012.

[11] S. Padmanabhan, T. Malkemus, A. Jhingran, and R. Agarwal. Block oriented processing of relational database operations in modern computer architectures. In *Data Engineering, 2001. Proceedings. 17th International Conference on*, pages 567–574. IEEE, April 2001.

[12] B. Peter, A., M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *Proceedings of Conference of Innovative Database Research*, pages 225–237. CIDR, 2005.

[13] B. Răducanu, P. Boncz, and M. Zukowski. Micro adaptivity in vectorwise. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1231–1242. ACM, June 2013.

[14] A. Skidanov, A. J. Papito, and A. Prout. A column store engine for real-time streaming analytics. In *Data Engineering (ICDE), 2016 IEEE 32nd International Conference on*, pages 1287–1297. IEEE, May 2016.

[15] D. Ślezak and V. Eastwood. Data warehouse technology by infobright. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 841–846. ACM, 2009.

[16] M. Stonebraker, D. J. Asadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-Store: A Column-Oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases*, pages 553–564. VLDB Endowment, August 2005.

[17] J. Thomas and S. Dessloch. Near real-time data warehousing using state-of-the-art ETL tools. In *Enabling Real-Time Business Intelligence*, pages 100–117. Springer Berlin Heidelberg, September 2010.

[18] K. Wu, E. J. Otoo, and A. Shoshani. Compressing bitmap indexes for faster search operations. In *Scientific and Statistical Database Management, 2002. Proceedings. 14th International Conference on*, pages 99–108. IEEE, July 2002.