分散データベース環境における マテリアライズドビューの同期手法の実装とその評価

松澤 裕史† 大川 昌弘†† 福田 剛志††

† 日本アイ・ビー・エム (株) 東京基礎研究所
†† 日本アイ・ビー・エム (株) 大和ソフトウェア開発研究所
†,†† 〒 242-8502 神奈川県大和市下鶴間 1623-14
E-mail: †{matuzawa,mohkawa,fukudat}@jp.ibm.com

あらまし OLAP 分析では，クエリ処理を効率良く行うため，事前に集約演算した結果をマテリアライズドビューとして保持しておき，これを利用して高速な演算が行われている．マテリアライズドビューは，データベースの更新に合わせて，適宜，更新が必要である．非分散環境でのマテリアライズドビューの更新は，ベーステーブルの変更データを利用して効率的に差分更新する方法が知られている．しかしながら，分散データベース環境に同様の方法を適用すると，長時間ベーステーブルをロックする必要があり，本来のデータベースの更新を伴う業務への影響が大きいため，頻繁に更新ができず，業務の空き時間等に再作成を行わなければならないのが現状である．本論文では，分散データベース環境におけるマテリアライズドビューの効率的な更新を行う方法を提案し，その実験結果を示す．

キーワード 分散 DB，情報統合，異種 DB，OLAP，データウェアハウス，マテリアライズドビュー

A Method of Materialized View Refresh on Distributed Database System

Hirofumi MATSUZAWA†, Masahiro OHKAWA††, and Takeshi FUKUDA††

† Tokyo Research Laboratory, IBM Japan, Ltd.
†† Yamato Software Laboratory, IBM Japan, Ltd.
†,†† 1623-14, Shimotsuruma, Yamato-shi, Kanagawa 242-8502, Japan
E-mail: †{matuzawa,mohkawa,fukudat}@jp.ibm.com

Abstract Materialized views are used to reduce the computational cost of OLAP operations. The problem of view maintenance on a single database server is well studied and implemented in commercial DBMS's. However, existing view maintenance methods do not work well in distributed environment. In this paper, we propose a method of distributed view maintenance that reduces the lock duration.

Key words Distributed Database, Information Integration, OLAP, Data Warehouse, Materialized View

1. はじめに

多くの企業で，データウェアハウスや OLAP(On-Line Analytical Processing) [1] が用いられている．通常，データの集計や集約演算には時間がかかるが，OLAP で高速なクエリ処理を行うために，事前に計算した中間の演算結果を別のテーブルに保存しておき，投入されたクエリに対してその事前計算結果を参照することで，高速に処理ができるような手段が取られている．このような事前計算結果を格納するテーブルは，一般に，マテリアライズドビュー (Materialized View) あるいはマテリアライズド参照表 (Materialized Query Table 略して MQT) [2] [3] と呼ばれている．本稿では，これを MQT と呼ぶ．

MQT は事前計算の結果であるため，参照しているテーブル (ベーステーブルと呼ぶ) の内容が更新されると MQT も更新する必要が生じる．データベース (DB) 製品では，DB が 1 つのサーバー上に構築されている時には，MQT を適宜，更新する機構が提供されている．MQT の更新手法には 2 種類あり，MQT を再作成する方法，及び，ベーステーブルの変更データから MQT の差分のみを計算して MQT に反映させる方法である．基本的には，後者の方が，前者に比べ高速に MQT の更新を行うことが可能である．

一方で，多くの企業などでは，業務毎，部門毎にデータベースを持ち，データが分散されて管理されている．また，近年の企業間の合併など，元々異なる組織で保持されていたシステムの統合が行われることがある．その際，[2] [3] [4] [5] などを用いて，データベースを分散させたままで，仮想的な統合データ

ベースを構築することが出来る．

このような分散データベース環境下においても，OLAP を用いる際，MQT 及び，その更新も非常に重要である．しかしながら，従来手法を用いて変更データから MQT の差分のみを計算して MQT へ反映させる方法は，次章で説明するように，本来の DB の更新を伴う業務へ与える影響が大きい．そのため，業務に影響のない夜間を利用したバッチ処理等で，MQT を再作成する方法が採用されているのが現状である．

我々は，分散データベース環境下において，差分のみを計算して MQT を更新する際，業務への影響を減らす効率的な手法の設計を行い，そのプロトタイプを実装した．本稿では，2. 章にて従来技術について説明し，3. 章において基本方針について述べる．4. 章において本手法を実現するプロトタイプについて説明し，最後に，5. 章にてパフォーマンスの測定とその評価について述べる．

2. 従来技術

2.1 MQT の更新手法

事前に MQT が作成されている場合，クエリが発行されるとデータベースは SQL 文を解析して，より効率的な計算に利用可能な MQT があるかどうかを調べ，もし，あれば，その MQT を利用して計算を行う(注1)．MQT の事前計算結果を利用できれば，計算する際，参照するテーブルを全て読む必要がないので，クエリのレスポンスタイムが大幅に改善される．

MQT は，ある時刻において SQL 文で計算した演算結果を保持しているため，ベーステーブルの情報が MQT の計算後に変更された場合，MQT の内容と実際の SQL 文の結果に不整合が生じる．そのため，ベーステーブルに変更があれば，整合性をとるために MQT の保持内容の更新 (リフレッシュ) を行う必要がある．図 1，図 2 に，DB2 [2] [3] で MQT を作成するための SQL 文の例を示す．

```
CREATE TABLE TPCH.MQT1 AS (
  SELECT
     T2.O_SHIPPRIORITY AS O_SHIPPRIORITY,
     T2.O_ORDERSTATUS AS O_ORDERSTATUS,
     T4.R_NAME AS R_NAME,
     T5.N_NAME AS N_NAME,
     T3.P_MFGR AS P_MFGR,
     YEAR( T2.O_ORDERDATE) AS year,
     MONTH( T2.O_ORDERDATE) AS month,
     SUM(T1.L_DISCOUNT) AS L_DISCOUNT,
     SUM(T1.L_EXTENDEDPRICE) AS L_EXTENDEDPRICE,
     SUM(T1.L_QUANTITY) AS L_QUANTITY,
     SUM(T1.L_TAX) AS L_TAX,
     COUNT_BIG(*) AS "COUNT_*"
   FROM
     TPCH.LINEITEM AS T1, TPCH.ORDERS AS T2,
     TPCH.PART AS T3, TPCH.REGION AS T4,
     TPCH.NATION AS T5, TPCH.CUSTOMER AS T6
   WHERE
     T1.L_ORDERKEY=T2.O_ORDERKEY AND T1.L_PARTKEY=T3.P_PARTKEY AND
     T6.C_CUSTKEY=T2.O_CUSTKEY AND T5.N_NATIONKEY=T6.C_NATIONKEY AND
     T4.R_REGIONKEY=T5.N_REGIONKEY
   GROUP BY
     T2.O_SHIPPRIORITY, T2.O_ORDERSTATUS, T4.R_NAME, T5.N_NAME,
     T3.P_MFGR, YEAR( T2.O_ORDERDATE), MONTH( T2.O_ORDERDATE)
)DATA INITIALLY DEFERRED REFRESH DEFERRED
 MAINTAINED BY SYSTEM
```

図 1 集約演算を用いた MQT の例

Fig. 1 Example of MQT with aggregate functions

現在，DB2 や Oracle [4] などに代表される DB 製品では，全てのベーステーブルが MQT と同じ DB サーバーにある場合，MQT の定義がある条件(注2)を満たせば，ベーステーブルの変更データから MQT の差分を演算して効率的に MQT を更新することが可能である．本稿では，MQT の差分のみを計算し，これを適用して MQT を更新する方法を**差分リフレッシュ**と呼ぶ．一方，ベーステーブルの全データを用いて MQT を再作成して更新する方法を**フルリフレッシュ**と呼ぶ(注3)．ベーステーブルとの整合性が重要な場合には，差分リフレッシュが利用されるが，MQT の頻繁な更新が本来の更新業務に影響を及ぼすようなケース，差分が多量でむしろ処理が遅くなるようなケースなどでは，フルリフレッシュが利用されている．

```
CREATE TABLE TPCH.MQT2 AS (
  SELECT
    T1.O_ORDERSTATUS AS O_ORDERSTATUS,
    T1.O_ORDERDATE AS O_ORDERDATE,
    T2.C_NAME AS C_NAME,
    T3.S_NAME AS S_NAME,
    T4.L_PARTKEY AS L_PARTKEY,
    T4.L_QUANTITY AS L_QUANTITY
  FROM
    TPCH.ORDERS AS T1,   TPCH.CUSTOMER AS T2,
    TPCH.SUPPLIER AS T3, TPCH.LINEITEM AS T4
  WHERE
    T1.O_CUSTKEY=T2.C_CUSTKEY AND T1.O_ORDERKEY=T4.L_ORDERKEY AND
    T3.S_SUPPKEY=T4.L_SUPPKEY AND
    T1.O_ORDERPRIORITY="1-URGENT" AND T1.O_ORDERSTATUS="F"
)DATA INITIALLY DEFERRED REFRESH DEFERRED
 MAINTAINED BY SYSTEM
```

図 2 ジョイン演算からなる MQT の例

Fig. 2 Example of MQT with join operation

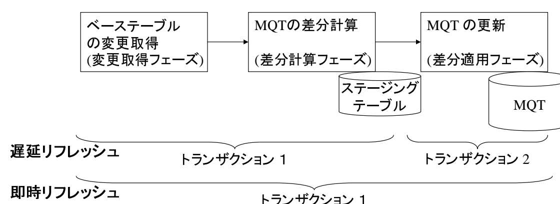

図 3 差分リフレッシュ(従来手法)

Fig. 3 Conventional Refresh Method

図 3 に差分リフレッシュの従来方法を示す．差分リフレッシュには，二つの方法がある．最初の方法は，**遅延リフレッシュ**と呼ばれる方法であり，ベーステーブルが更新されると，そのトランザクション内において，変更データを取得し (変更取得フェーズ)，他のベーステーブルから MQT の差分を計算し，ステージングテーブルと呼ばれる中間テーブルに挿入する (差分計算フェーズ)．ユーザがリフレッシュ用コマンドを実施することで，別のトランザクション (差分適用フェーズ) にて，(複数の変更分をまとめて) ステージングテーブルに格納されている MQT の差分を適用し，MQT を更新する．

もう一つの方法は，**即時リフレッシュ**と呼ばれる方法であり，ベーステーブルが変更されると，そのトランザクション内で変更取得，差分計算，差分適用の全フェーズを実施する．即時リフレッシュを用いると，自動的に MQT が更新され，常にベー

(注1)：利用可能であっても演算が効率的でない場合には MQT を利用しない

(注2)：Group-By がある場合 COUNT(*) または，COUNT_BIG(*) を含めないといけない．Having 文節は許されない．など

(注3)：差分リフレッシュ，フルリフレッシュをそれぞれ，高速リフレッシュ，完全リフレッシュと呼んでいる DB 製品もある

$$
\begin{aligned}
dMQT1_{(t_n,t_o)} &= MQT1_{t_n} - MQT1_{t_o} \\
&= MQT1_{t_n} - T1_{t_o} \bowtie T2_{t_o} \bowtie T3_{t_o} \\
&= T1_{t_n} \bowtie T2_{t_n} \bowtie T3_{t_n} - (T1_{t_n} - dT1_{(t_n,t_o)}) \bowtie (T2_{t_n} - dT2_{(t_n,t_o)}) \bowtie (T3_{t_n} - dT3_{(t_n,t_o)})
\end{aligned}
$$

図 4　MQT1 の時刻 $t = t_o$ から $t = t_n$ までの差分

Fig. 4　Delta of MQT1 at $t = t_n$ from $t = t_o$

ステーブルと整合が取れた状態の保持が可能である．

上記の差分計算フェーズについて説明する．MQT には，通常，複数のテーブルを用いた SQL 文の演算結果が保持されている．例えば，MQT1 が T1，T2，T3 の 3 つのテーブルを参照する SQL 文の演算結果を保持すると仮定して，ここでは，$MQT1 = T1 \bowtie T2 \bowtie T3$ ($\bowtie$ はジョインを表す) と表す．実際には，図 1，図 2 にあるような WHERE 節，GROUP BY 節を含むが，以下の説明では，これらを省略する．

時刻 t におけるテーブル T，MQT の状態をそれぞれ T_t，MQT_t と記述する．時刻 $t = t_o$ を初期状態として，$MQT1_{t_o} = T1_{t_o} \bowtie T2_{t_o} \bowtie T3_{t_o}$ が成り立っているとする．その後 $t = t_n$ までに，テーブル T1，T2，T3 のデータが更新され，それぞれが $T1_{t_n}$, $T2_{t_n}$, $T3_{t_n}$ となったとする．その時の差分データをそれぞれ $dT1_{(t_n,t_o)}$, $dT2_{(t_n,t_o)}$, $dT3_{(t_n,t_o)}$ とすると，$T1_{t_n} = T1_{t_o} + dT1_{(t_n,t_o)}$ である (T2, T3 についても同様)．$MQT1$ の差分を $dMQT1$，$MQT1_{t_o}$ に差分を加えた最新の状態を $MQT1_{t_n}$ とすると，図 4 の式が成り立つ．差分データ dT には，データの追加，削除，更新が含まれている．

各テーブルの差分 $dT1, dT2, dT3$ が取得できれば，上記の式を実行して，$MQT1$ の差分 ($dMQT1$) を計算することができるので，これをステージングテーブルに挿入することができる．差分適用フェーズにおいて，$MQT1_{t_o}$ に差分 $dMQT1_{(t_n,t_o)}$ を適用することで MQT1 を更新し，$MQT1_{t_n}$ が得られる．

従来の差分リフレッシュ方法では，図 3 に示すように，変更取得と差分計算のフェーズが一つのトランザクションで実行されており，演算中はデータベースの整合性を保つために，演算前に T1, T2, T3 のテーブルをロックし，演算終了後にアンロックされるまで，ベーステーブルを更新することができない．

OLAP では，通常，ファクトテーブルと呼ばれるテーブルに格納されるデータを用いて分析を行うが，テーブルが更新される頻度も一つのファクトテーブルに集中するという特徴があることが知られている．一般に，長い時間，テーブルをロックしておくことは，DB の更新業務に多大な影響があるので，特に，ファクトテーブルを長時間ロックさせないことは重要である．

2. 2　分散データベース環境への適用

次に，従来の差分リフレッシュ手法を分散データベース環境に適用した場合を考える．図 5 に，複数のリモートデータベースから構成される分散データベース環境の例を示す．リモート DB にあるテーブルに対して，テーブル本体をリモート DB に置いたまま，統合サーバー側で仮想的なテーブルとしてニックネームを付与し，テーブルへの参照を可能とする環境が構築されている．図 5 において，T1，T2，T3 がリモートデータベース上にあるベーステーブル本体であり，それらに対するニックネームが N1，N2，N3 である．即ち，統合サーバーは，テーブル N1 を参照することで，実際には，リモート DB に本体があるテーブル T1 を参照することができる．また，MQT は統合サーバーに配置される．

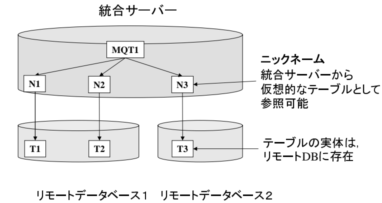

図 5　分散データベース環境

Fig. 5　Distributed Database System

変更取得フェーズにおいて，ベーステーブル本体は他のリモートサーバー上にあるので，統合サーバーは，直接，差分を取得することができない．そのため，トリガーなどの仕組みを用いて，取得する必要がある．ベーステーブルが更新されるとそのトランザクション内で，ベーステーブルをロックしたまま，差分データを取得し，差分計算フェーズで図 4 のように MQT の差分を計算する．

この時，同一サーバー内にないテーブル間の演算 (例えば，図 5 で N1(T1) と N3(T3)) については，ベーステーブルの内容を (必要な分だけ) 統合サーバー上にネットワーク経由でコピーしてから行われるので差分計算フェーズに時間がかかる．そのため，ベーステーブルが長時間ロックされ，本来の更新業務に多大な影響を与えることになる．従って，できるだけ業務への影響が少ない方法で差分リフレッシュを行うことが必要である．

3.　分散 DB 環境における差分リフレッシュ手法

3. 1　提案手法の基本的な方針

我々は，次のような方針により従来の MQT の差分リフレッシュ手法を拡張し，分散データベース環境において，一回あたりのロックする時間を削減し，業務に影響の少ない処理方法を実現した．図 6 に，その概略を示す．

ベーステーブルの変更取得と MQT の差分計算の分割

従来手法では，変更取得フェーズと差分計算フェーズを一つのトランザクションで実施していた．我々は，図 6 に示すように，変更取得フェーズと差分計算フェーズを別々のトランザクションで実施することにした．そのために，各ベーステーブルの変更データを入れるための**デルタテーブル**を用意し，変更取得フェーズでは，トリガーなどを用いて，変更データをデルタテーブルに追加し，差分計算プロセスではデルタテーブルを用いて変更データから MQT の差分計算を行う．

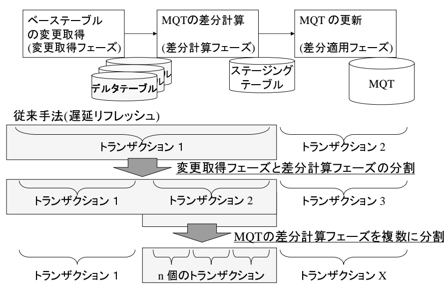

図 6　トランザクションの分割
Fig. 6　dividing one transaction into several transactions

$$
\begin{aligned}
& dMQT1_{t_n} \\
& = dT1_{(t_n,t_o)} \bowtie (T2_{t_n} - dT2_{(t_n,t_o)}) \bowtie (T3_{t_n} - dT3_{(t_n,t_o)}) \quad \ldots(1) \\
& + T1_{t_n} \bowtie dT2_{(t_n,t_o)} \bowtie (T3_{t_n} - dT3_{(t_n,t_o)}) \quad \ldots(2) \\
& + T1_{t_n} \bowtie T2_{t_n} \bowtie dT3_{(t_n,t_o)} \quad \ldots(3)
\end{aligned}
$$

図 7　MQT1 の差分計算を 3 つに分割する
Fig. 7　Calculation of delta MQT1 by three transactions

MQT の差分計算を複数のトランザクションへ分割

従来手法では，図 4 の式を 1 回のトランザクションで実施していた．これは長時間のロックが必要となり，業務への多大な影響が発生する．そこで，MQT の差分を演算する SQL 文を変更し，それぞれのベーステーブルの変更データ毎に分割して MQT の差分データを演算するようにした．この分割により，トランザクションも複数に分割できるので，1 回のロックにかかる時間を短縮することができる．ただし，1 回のロック時間は減少するが，MQT の更新全体の時間は長くなる．また，MQT の定義とある時刻における MQT の計算結果の間には差が発生することになるが，これについて 3.2.2 節にて説明する．

我々は，MQT の差分計算のトランザクション分割方法として，ベーステーブルの差分であるデルタテーブル毎にトランザクションを分割する (Each Delta At Once 方式) だけでなく，その演算をさらに細かく分割し，より多数のトランザクションに分割する方式 (One By One 方式) についても実装した．この二つの方式について，以下に説明する．

3.2　Each Delta At Once 方式

差分計算を以下のように複数のトランザクションに分割する．図 4 の式は，図 7 の式に展開することができる．図 7 の (1) の式で T1 のデルタテーブル dT1 に関する MQT の差分データを演算する．(2) では dT2 に関して，(3) では dT3 に関して，それぞれ MQT の差分データを演算する．この演算結果が MQT の差分データとなるが，差分計算フェーズにおいて，式 (1)，(2)，(3) をそれぞれ別のトランザクションで実施し，これをステージングテーブルに格納する．3 つの計算が終わった後に，差分適用フェーズで MQT に反映させる．このようにベーステーブル毎にその差分を MQT の差分として計算していく方法を Each Delta At Once 方式と呼ぶ．

このようにトランザクションを分割すると，各ベーステーブル Tn の変更データに関する MQT の差分計算中は，そのデルタテーブル dTn は一貫性を保つ必要があるが，そのベーステーブル Tn は演算に用いないため，dTn に関する MQT の差分の計算中に Tn を更新されても一貫性が保持される．Tn が変更されると変更データが dTn にも追加されるが，デルタテーブルに計算中のデータか，計算中に追加されたデータかを区別するためのフラグを持たせることにより，計算中の dTn の一貫性を保持できるようにした．

OLAP では頻繁に更新されるベーステーブルは 1 つのファクトテーブルに集中するという性質があるが，この手法は，あるベーステーブルの変更データに関して MQT の差分データの計算中であっても，そのベーステーブルを更新可能な状態にできるというメリットがあり，業務への影響を抑えることができる．ここで，例えば，一番頻繁に更新が行われるテーブルを T1 として，T1 の差分を計算するのが (1) の式である．(1) の式では，テーブル T2，T3 についてはロックする必要があるが，T1 の差分データはデルタテーブルに格納されており，(1) の式の計算中も T1 のロックが不要である．式 (2)，(3) の計算には テーブル T1 をロックする必要があるが，T1 に比べて，T2，T3 の更新はそれほど多くないので，比較的，短時間で処理が可能であり，ロックによる業務への影響が軽減される．

3.2.1　アルゴリズム

これらを実現するアルゴリズムを以下に述べる．ただし，$t = t_0$ において，MQT が再作成され MQT_{t_o} が生成されているものとする．

(1) T1 の差分データ処理 ($t = t_1$ における処理)

(1-1)　ベーステーブル T2，T3 を共有モード(注4)でロックする．この計算中は，dT1，(T2 − dT2)，(T3 − dT3) についても一貫性を保つ必要があるが，前述のように dT1 に関しては，デルタテーブルのフラグにより一貫性が保持される．また，ベーステーブルのロックにより，そのデルタテーブル (dT2，dT3) も更新されないでの計算中の一貫性が保持される．

(1-2)　$dT1_{(t_1,t_0)} \bowtie (T2_{t_1} - dT2_{(t_1,t_0)}) \bowtie (T3_{t_1} - dT3_{(t_1,t_0)})$ の演算結果をステージングテーブルに挿入する．

(1-3)　$dT1_{(t_1,t_0)}$ のデータを削除する．計算中に dT1 に追加されたデータはフラグにより区別されているので，計算中の追加データは削除されない．

(2) T2 の差分データ処理 ($t = t_2$ における処理)

この時点で テーブル T1 が $t = t_1$ 以降に，更新されている可能性がある．この場合，デルタテーブル dT1 にも差分が挿入されている．テーブル T2 の差分データに関する MQT の差分の計算に使用するのは，この演算を開始する時点で MQT に反映されているデータであり(注5)，T1 については $t = t_1$ 時点でのデータ ($T1_{t_1} = T1_{t_0} + dT1_{(t_1,t_0)}$)，T3 については $t = t_0$ 時点でのデータである ($T3_{t_0}$)．これらは $T1_{t_1} = (T1_{t_2} - dT1_{(t_2,t_1)})$，$T3_{t_0} = (T3_{t_2} - dT3_{(t_2,t_0)})$ で得ることができる．差分計算にはこれらを用いる．

(2-1)　dT2, (T1 − dT1), (T3 − dT3) の一貫性を保つため，同様に，T1 と T3 を共有モードでロックする．

(注4)：テーブルの更新はできないが参照は可能なロック方法
(注5)：ステージングテーブルに残っている MQT の差分も含む

dTi の差分データに関する MQT の差分を時刻 $t=t_i$ において計算する時，これを $dMQT(dTi)_{t=t_i}$ と表すと，

$$dMQT(dT1)_{t=t_1} = dT1_{(t_1,t_0)} \bowtie (T2_{t_1} - dT2_{(t_1,t_0)}) \bowtie (T3_{t_1} - dT3_{(t_1,t_0)}) ... (Tn_{t_1} - dTn_{(t_1,t_0)}) \quad ... (1)$$
$$dMQT(dT2)_{t=t_2} = (T1_{t_2} - dT1_{(t_2,t_1)}) \bowtie dT2_{(t_2,t_0)} \bowtie (T3_{t_2} - dT3_{(t_2,t_0)}) ... (Tn_{t_2} - dTn_{(t_2,t_0)}) \quad ... (2)$$
$$dMQT(dT3)_{t=t_3} = (T1_{t_3} - dT1_{(t_3,t_1)}) \bowtie (T2_{t_3} - dT2_{(t_3,t_2)}) \bowtie dT3_{(t_3,t_0)} \bowtie (T4_{t_3} - dT4_{(t_3,t_0)}) ... (Tn_{t_3} - dTn_{(t_3,t_0)}) \quad ... (3)$$
$$\vdots$$
$$dMQT(dTn)_{t=t_n} = (T1_{t_n} - dT1_{(t_n,t_1)}) \bowtie (T2_{t_n} - dT2_{(t_n,t_2)}) ... (T(n-1)_{t_n} - dT(n-1)_{(t_n,t_{n-1})}) \bowtie dTn_{(t_n,t_0)} \quad ... (n)$$

図 8　Each Delta At Once 方式での計算式

Fig. 8　Calculation of delta MQT by Each Delta At Once

(2-2)　$dT2_{t_2} \bowtie (T1_{t_2} - dT1_{(t_2,t_1)}) \bowtie (T3_{t_2} - dT3_{(t_2,t_0)})$ の演算結果をステージングテーブルに挿入する．

(2-3)　$dT2_{(t_2,t_0)}$ のデータを削除する．

(3) T3 の差分データ処理 ($t=t_3$ における処理)

(1) や (2) と同様な処理を行う．ステージングテーブルに挿入するのは，$dT3_{t_3} \bowtie (T1_{t_3} - dT1_{(t_3,t_1)}) \bowtie (T2_{t_3} - dT2_{(t_3,t_2)})$ の演算結果である．

3.2.2　MQT の定義と実際の MQT の不整合について

このように，ベーステーブル毎にその差分を，複数のトランザクションに分割して計算を行うとトランザクションの実施中に発生したベーステーブルへの変更 ($dT1_{(t_3,t_1)}, dT2_{(t_3,t_2)}$) が，MQT の更新直後の MQT_{t_3} に反映されずに残っている．

しかしながら，これらの変更はデルタテーブルに格納されているので，同様の手順で $t=t_4$ で dT1 の差分の処理，$t=t_5$ において dT2 の差分の処理，$t=t_6$ で dT3 の差分の処理を行う際に解消される．ただし，$t=t_6$ で同じように $dT1_{(t_6,t_4)}, dT_{(t_6,t_5)}$ が反映されないで残る．

ある時刻において，MQT に反映されていない変更データがない状態にしたい場合には，(1)(2)(3) の処理を終了した直後に，全デルタテーブルをロックして MQT の差分を計算すればよい．このとき，反映されずに残っているデルタテーブルの差分は，式 (1)(2)(3) の計算時間に更新されたデータであるため，非常に小さく，計算時間も最小限に抑えることが可能である．

この Each Delta At Once 方式の計算結果は，厳密には，図 7 とは異なっている．これは，トランザクションを分割して計算を行っている間にベーステーブルが変更されてしまうからである．また，3 個のベーステーブルからなる例を示したが，MQT が 4 個以上のベーステーブルから定義される場合でも同様の方法で実現可能である．MQT が n 個のテーブルで定義され，$t=t_0$ で MQT が更新されているとして，その計算式は，図 8 のように一般化できる．また，図 8 の式 (n) までの計算が終わった状態で各テーブル Ti のデルタテーブルには，それぞれ $dTi_{(t_n,t_i)}$ が MQT に反映されずに残っている．

3.3　One By One 方式

もう一つの One By One 方式は，Each Delta At Once 方式の各トランザクションをさらに細かく分割する手法である．

One By One 方式では，デルタテーブル $dT1$ に関する MQT の差分を計算する式 $[dT1_{(t_1,t_0)} \bowtie (T2_{t_1} - dT2_{(t_1,t_0)}) \bowtie (T3_{t_1} - dT3_{(t_1,t_0)})]$ を更に分割し，図 9 に示すように分割する．最初のトランザクションで $dT1 \bowtie (T2 - dT2)$ の結果を中間テーブル $dT1M12$ に挿入し，次のトランザクションにおいて $dT1M12 \bowtie (T3 - dT3)$ を計算する．これを Each Delta At Once 方式同様に，全てのデルタテーブルに関して MQT の差分の計算を行う．

$$\begin{aligned} dT1M12 &= dT1 \bowtie (T2 - dT2) \\ &= dT1 \bowtie T2 - dT1 \bowtie dT2 \end{aligned}$$
$$\begin{aligned} dMQT(T1) &= dT1M12 \bowtie (T3 - dT3) \\ &= dT1M12 \bowtie T3 - dT1M12 \bowtie dT3 \end{aligned}$$

図 9　SQL の分割

Fig. 9　Division of SQL

ここで，One By One 方式では，次の点に考慮する必要がある．図 9 では，$dT1$ と $(T2 - dT2)$ を先に計算する場合の式であったが，計算の順序を変更し，$dT1 \bowtie (T3 - dT3)$ を先に計算して中間テーブル $dT1M13$ を作成し，次に，$dT1M13 \bowtie (T2 - dT2)$ を計算する方法も可能である．n 個のベーステーブルから定義される MQT の場合，一つのベーステーブルに関して MQT の差分を計算するための他のベーステーブルの順番の決め方は $(n-1)!$ 通りあるので，この中から最適なものを選択する必要がある．これは，計算に必要な SQL 文を全ての組み合わせについて生成し，データベースの SQL の処理コストを見積もる機能を用いることで，最適な順序を予測することが可能である．

One By One 方式では，Each Delta At Once 方式よりトランザクション数が増え，中間テーブルへの挿入などのオーバーヘッドが発生するため，MQT の差分リフレッシュ1 回分の合計処理時間が長くなる．しかし，トランザクション毎の処理時間は非常に小さくなるため，その間にロックされている時間，及び，ロックされているテーブル数が少なくなるので，業務への影響は最低限に抑えることができる．

また，Each Delta At Once 方式同様，トランザクションの実行中にベーステーブルが更新されデルタテーブルに挿入されるという状況が発生する．One By One 方式においても同様に，処理されずに残った差分については，次回に MQT へ反映させることも可能であり，One By One 方式の実施直後に，全テーブルをロックして従来の差分リフレッシュを行うことで反映されなかった差分を MQT へ反映させることが可能である．

4.　プロトタイプシステムの構成

4.1　システム構成

我々は，IBM DB2 [2] 及び WebSphere Information Integrator [3] を情報統合データベースとして用いて，分散データベース環境下で差分リフレッシュを行うプロトタイプシステムを構

築した．我々が構築したプロトタイプは，主に前処理部と実行部からなる．また，実行部が処理を行うのに必要な SQL 文を格納するデータベース (Info DB) を保持する．

前処理部では MQT の定義 (SQL 文) を解析することで，本手法で差分リフレッシュを実現するのに必要な一連の SQL 文を生成し，データベース (Info DB) に登録する．Java プログラムを利用する場合は，そのクラス名を Info DB に登録する．また，リフレッシュを行う間隔などのパラメータ等の情報も Info DB に登録して利用する．

実行部は，基本的に Info DB に登録されている SQL 文を実行することで差分リフレッシュを実施する．即ち，ユーザからのリクエストに応じて，MQT，ステージングテーブル，差分リフレッシュを行うのに必要なデルタテーブルなどのデータベースのオブジェクトを作成したり，登録されている SQL 文を利用 (実行) して差分リフレッシュを行う．

一つの MQT に関して，我々のプロトタイプが差分リフレッシュを行うのに必要なデータベースの構成例を図 10 に示す

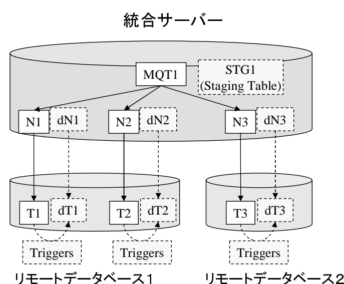

図 10　MQT の差分リフレッシュ DB 構成例

Fig. 10　An example of the MQT Refresh component

与えられた MQT に対して，前処理部で作成した SQL 文に従い，実行部が必要なテーブルを生成する．また，リモートデータベース上に各ベーステーブルに対応するデルタテーブルを生成し，トリガーを用いてベーステーブルの差分がデルタテーブルに反映できるようにした．生成されたデルタテーブルはニックネームを付与し，統合サーバーから参照可能にする．また，MQT の差分データを保持するステージングテーブルを統合サーバー上に生成する．

4.2　変 更 取 得

ベーステーブルにデータが挿入・更新・削除されるとトリガーなどにより，デルタテーブルにその情報が追加される．デルタテーブルは MQT に演算結果を保存するために必要なベーステーブルの列のほかに，それぞれ INTEGER 型の OPERATIONTYPE 列と PROCESSFLAG 列を持つ．OPERATIONTYPE は 1 か −1 の値を保持し，1 は挿入，−1 は削除を意味する．更新の場合には，古いデータの削除と新しいデータの挿入が行われたものとしてデルタテーブルに追加される．

ベーステーブルが更新されると，そのデルタテーブルに PROCESSFLAG を 0 にセットした差分情報が挿入される．デルタテーブルを処理する際に，PROCESSFLAG を 1 にセットし，その差分に関する MQT の差分を演算し，処理が終了したら PROCESSFLAG が 1 のレコードだけを削除する．PROCESSFLAG の導入により，デルタテーブルの処理中にベーステーブルが更新されても，PROCESSFLAG=0 としてその情報をデルタテーブルに追加することができるため，デルタテーブルを使った計算開始後に追加されたレコードと処理対象のレコードを区別することができる．

4.3　差 分 計 算

4.3.1　ステージングテーブルの構造

ステージングテーブルは，MQT の差分データを格納しておくテーブルであるが，これには既存技術を用いる．ここで，MQT のタイプには 2 種類あり，ステージングテーブルも MQT のタイプに合わせて構築される．MQT は，その定義として，SQL 文が集約関数を使うタイプ (集約 MQT，図 1 参照) と集約関数を使わないタイプ (ジョイン MQT，図 2 参照) がある．集約 MQT では，ステージングテーブルは MQT が持つ列と同じ列を持つ．集約 MQT の場合は，挿入，削除の件数が MQT の中で $COUNT$，SUM 関数によって保持されているが，ジョイン MQT の場合には，MQT のテーブル中にないために，ジョイン MQT 用のステージングテーブルには MQT が持つ列に加え，挿入・削除を表すフラグを保持する必要があり，そのために，INTEGER 型の OPERATIONTYPE 列を加える．

4.3.2　差分計算のための SQL 文の作成

差分計算のための SQL 文の作成方法は次の通りである．

Each Delta At Once 方式の場合

3. 章で述べた T1 の差分処理の式 (図 7(1)) は，展開すると以下のようになる．

$$
\begin{aligned}
\mathrm{dT1} \bowtie (\mathrm{T2} - \mathrm{dT2}) \bowtie (\mathrm{T3} - \mathrm{dT3}) = {} & \mathrm{dT1} \bowtie \mathrm{T2} \bowtie \mathrm{T3} && \ldots \text{(A)} \\
& - \mathrm{dT1} \bowtie \mathrm{dT2} \bowtie \mathrm{T3} && \ldots \text{(B)} \\
& - \mathrm{dT1} \bowtie \mathrm{T2} \bowtie \mathrm{dT3} && \ldots \text{(C)} \\
& + \mathrm{dT1} \bowtie \mathrm{dT2} \bowtie \mathrm{dT3} && \ldots \text{(D)}
\end{aligned}
$$

(A) 〜(D) について，SQL 文を生成する．生成された SQL 文を実行することで，T1 の差分データに関する MQT の差分データが得られる．T2 や T3 の差分データに関する MQT の差分データを得る場合も同様に作成する．

One By One 方式の場合

One By One 方式の場合も計算を分割しているので，そのための SQL 文を作成する必要がある．DB のコンパイラは，DB へのクエリを効率良く計算するために，テーブル毎の演算順序を決定したり，MQT を計算に利用する可否を判断するなど，SQL 文の分割や変換を行っている．

One By One 方式でも基本的には，これと同様の方法で必要な SQL を作成している．Each Delta At Once 方式と同様に，OPERATIONTYPE，PROCESSFLAG を用いて SQL 文を変換する．また，中間計算結果を一時的に保持するため，一時テーブルを用意する必要があり，SQL 文で一時テーブルの作成や一時テーブルへの挿入を実行する．

4.4　差 分 適 用

ステージングテーブルの内容を MQT へ適用するため方法は，従来手法と同様である．本手法では，そのための SQL 文を用意し，これを実行する．

5. 実験と評価

プロトタイプを用いてパフォーマンスを測定した．統合サーバーとリモート DB の構成に関して，下記に示す 4 種類の分散環境を構築した．データは TPCH [11] を用い，8 つのベーステーブルを 3 つのリモート DB に分散させて配置した．リモート DB 上のテーブルをニックネーム経由で参照する集約 MQT(図 1) とジョイン MQT(図 2) を統合サーバーに作成した．

分散環境	統合サーバー	リモート DB1	リモート DB2	リモート DB3
Config1	PC1	PC2	– (リモート DB が 1 つ)	
Config2	PC1	PC2	PC2	PC2
Config3	PC1	PC4	PC3	PC2
Config4	PC2	PC2	PC2	PC2

データサイズはデフォルトの 1GB とし，差分データは TPCH が提供するツールで作成し，ORDERS テーブルに対して 300 行，LINEITEM テーブルに対して 1159 行の差分データを挿入したり，削除したりして，MQT のリフレッシュを行った．

なお，実験に用いたマシンは，それぞれ，下記のスペックである．PC1: ThinkPad Z60t (CPU: Pentium M 1.6GHz, Memory: 2GB, OS: Windows XP Professional 2002 SP2), PC2: IBM IntelliStation Z Pro (Intel Xeon 3GHz, 4GB, Windows 2003 Server R2 SP1)，PC3: IBM IntelliStation Z Pro (Intel Xeon 3GHz, 2.5GB, Windows XP Professional 2002 SP2), PC4: IBM IntelliStation M Pro (Intel Pentium IV 2.0GHz, 768GB, Windows XP Professional 2002 SP2)

5.1 パフォーマンスの比較

本手法を用いた MQT の更新について，各分散環境の上でパフォーマンスを測定した．その結果を集約 MQT とジョイン MQT に関して，それぞれ表 1，表 2 に示した．

Config4 は同一の PC を用いて分散環境を構築しており，ネットワークのコストが無視できるので，他の分散環境よりも良いパフォーマンスが得られている．Config2 は Config1 と比較して，ベーステーブルが (同一 PC ではあるが) 複数の DB に分散されており，その分のオーバーヘッドで処理時間が長くなっていると思われる．さらに，Config3 は分散した DB と統合サーバーを全て異なる PC 上に配置しており，最もネットワークのコストが高い構成となっている．

フルリフレッシュは，他の差分リフレッシュ手法と比較して，その処理に長時間を要している．この間，ベーステーブルがロックされるため，業務に影響があることは明白である．

従来手法と Each Delta At Once 方法を比較した場合，どの分散環境においても差分更新（差分計算＋差分適用）の処理時間は，それほど差が出ていないが，ベーステーブルのロック時間を見ると，大きく減少していることが確認できる．本手法の目的がロック時間の削減による業務への影響を減らすことであり，良い結果を得ることが出来たと言える．

次に，One By One 方式について考察する．更新時間の合計については，One By One 方式は SQL 分割，中間ファイルへの読み書きなどのオーバーヘッドが大きく，最も処理時間が遅い方式となっている．しかしながら，本実験では LINEITEM を中心にベーステーブルの更新を実施しており，OLAP で更新頻度の多いベーステーブルに偏りがあることを考えると LINEITEM のロック時間を削減することは業務への影響を削減するという点で最も効果的な方法と考えることができる．実験の結果から，One By One 方式において LINEITEM のロック時間が大きく削減できていることが確認できた．しかしながら，Config3 に関しては，我々の期待を裏切る結果となっており，原因の調査が必要であると考えている．

ロックする対象のテーブル数は，Each Delta At Once が他のベーステーブル全てであるのに対し，One By One は 1 つのベーステーブルだけであり，更新されるベーステーブルが一つのテーブルに偏る場合には，One By One 方式で業務への影響を減らすことができるが，Config2, Config3 のケースなど，ORDERS テーブルのロック時間が非常に長いケースがあるため，このような環境においては，Each Delta At Once を用いるなど，二つの方式を使い分ける必要があると思われる．

仮に，図 9 のテーブル T1, T2, T3 が全て同一のリモート DB 上にあり，中間テーブルである $dT1M12T1$ をリモート DB に作成できれば，ネットワークコストを削減することが可能である．しかしながら，現在の実装では，中間テーブルを全て，統合サーバーに配置しているため，ネットワークのコストがかかり，One By One 方式の所要時間は，我々の期待よりも長時間であった．実装の変更により，One By One 方式は，さらに高速化されることが期待される．

5.2 One By One の計算順序

One By One 方式の最適な計算順序については，DB の見積もり機能を利用して，計算順序を選択している．表 3 は，Config 4 において，表 1，表 2 と同じ計算を One By One の計算順序を変更して行った測定結果である．

表 3 One By One の計算順序を変更した場合

集約 MQT	データ更新方法	追加 (sec)	削除 (sec)
One By One	MQT の差分計算	測定失敗	測定失敗

ジョイン MQT	データ更新方法	追加 (sec)	削除 (sec)
One By One	MQT の差分計算	1.64	1.63
	MQT へ差分適用	0.27	23.5
	ORDERS のロック	0.047	0.046
	LINEITEM のロック	0.047	0.047

ベーステーブルのサイズを n として，計算順序の組み合わせは $(n-1)!$ 個あるので，その中から計算コストの見積もりの結果が中ぐらいな計算順序を選択した．集約 MQT については，PC のリソースが足りなくなり，途中で Error となり，MQT が作成できなかった．また，ジョイン MQT についても 表 2 よりも，処理計算に時間を要する結果となった．

このように，どのテーブルから計算していくかという順番を決めることも One By One 方式を採用する際には重要な要素であることが確認できた．

6. 関 連 研 究

マテリアライズドビューの更新に関しては, [6] で概要を見る

表 1　集約 MQT のパフォーマンス

		Config 1		Config 2		Config 3		Config 4	
	データ更新方法	挿入 (sec)	削除 (sec)	挿入 (sec)	削除 (sec)	挿入 (sec)	削除 (sec)	挿入 (sec)	削除 (sec)
フルリフレッシュ		830.5	903.0	912.0	933.4	1101.7	1067.9	616.7	553.0
差分リフレッシュ 従来手法	MQT の差分計算	3.91	1.95	15.3	12.8	16.4	12.6	8.52	4.99
	MQT へ差分適用	7.11	6.78	7.02	7.00	7.05	6.97	7.28	8.28
	ORDERS のロック	3.91	1.95	15.3	12.8	16.4	12.6	8.52	4.99
	LINEITEM のロック	3.91	1.95	15.3	12.8	16.4	12.6	8.52	4.99
差分リフレッシュ Each Delta At Once	MQT の差分計算	1.95	1.92	13.7	11.7	12.5	12.2	5.33	5.73
	MQT へ差分適用	7.31	6.86	7.20	7.17	7.42	7.45	7.45	7.31
	ORDERS のロック	0.88	1.16	9.13	7.59	7.89	7.73	3.45	4.00
	LINEITEM のロック	0.17	0.016	4.45	4.06	4.14	3.97	1.69	1.53
差分リフレッシュ One By One	MQT の差分計算	22.6	12.5	23.8	23.9	106.0	98.9	14.5	6.84
	MQT へ差分適用	7.16	6.80	6.78	6.81	6.75	6.84	7.20	6.84
	ORDERS のロック	4.59	2.24	15.0	15.1	15.0	15.0	0.91	0.89
	LINEITEM のロック	2.42	2.28	2.44	2.52	76.1	77.2	0.89	0.84

表 2　ジョイン MQT のパフォーマンス

		Config 1		Config 2		Config 3		Config 4	
	データ更新方法	挿入 (sec)	削除 (sec)	挿入 (sec)	削除 (sec)	挿入 (sec)	削除 (sec)	挿入 (sec)	削除 (sec)
フルリフレッシュ		45.0	55.0	64.2	87.6	61.4	59.7	161.7	158.6
差分リフレッシュ 従来手法	MQT の差分計算	1.06	1.03	14.1	6.89	6.95	6.80	1.74	1.73
	MQT へ差分適用	0.30	30.8	0.172	15.8	0.31	16.0	0.30	12.8
	ORDERS のロック	1.06	1.03	14.1	6.89	6.95	6.80	1.74	1.73
	LINEITEM のロック	1.06	1.03	14.1	6.89	6.95	6.80	1.74	1.73
差分リフレッシュ Each Delta At Once	MQT の差分計算	1.17	1.13	6.89	6.30	7.39	7.70	0.44	0.25
	MQT へ差分適用	0.53	30.6	0.28	17.9	0.16	18.9	0.13	13.0
	ORDERS のロック	0.55	0.66	4.00	4.28	4.38	4.47	0.22	0.16
	LINEITEM のロック	0.25	0.094	2.81	1.86	2.28	2.00	0.13	0.031
差分リフレッシュ One By One	MQT の差分計算	292.9	305.2	61.9	59.8	63.8	60.7	1.69	1.61
	MQT へ差分適用	0.047	31.1	0.11	19.9	0.83	19.5	0.13	13.3
	ORDERS のロック	2.63	5.09	57.7	55.5	59.8	57.5	0.70	0.67
	LINEITEM のロック	0.13	0.20	0.062	0.047	0.41	0.27	0.031	0.015

ことができる．また，差分リフレッシュに関して，[7] [8] [9] [10] など，多くの研究がなされている．Salem [9] らは，タイムスタンプを利用して，さらにロックの時間を減らそうとしているが，機能が複雑で演算量も多い．また，分散環境では，サーバー間で正確な時間を合わせる必要がある．Agrawal [10] らは，本手法の One By One 方式に類似した差分リフレッシュ手法を提案しているが，我々のようにリモート DB にデルタテーブルを持たず，キューを用いて逐次，統合サーバに到着した差分データに関する MQT の差分を計算しており，時間のずれは発生しないが，処理が細かく処理時間を要すると思われる．

7. おわりに

分散データベース環境における MQT の差分リフレッシュ手法について，従来手法を適用することは，業務に与える影響が大きいため，改善が必要であった．我々は分散データベース環境における MQT の差分リフレッシュ手法として，業務に与える影響を抑える方法を提案し，プロトタイプを実装した．この方法は，ロック時間を削減することができ，かつ，シンプルなメカニズムで実現可能である．そして，実験により業務に与える影響を抑える差分リフレッシュが実現できたことを確認した．

One By One の方式の実装の改善や，本手法に適用可能な MQT の定義 (SQL) 文の拡張などを行っている予定である．

文　献

[1] E. F. Codd, S. B. Codd, and C. T. Sally. “Beyond decision support”. Computer World, 27(30), July 1993.
[2] IBM DB2, http://www-06.ibm.com/jp/software/data/db2/
[3] IBM WebSphere Information Integrator V8.2, “IBM DB2 Information Integrator, Federated Systems Guide, Version 8.2”, http://www-06.ibm.com/jp/software/websphere/ii/v82/
[4] Oracle Database, http://www.oracle.co.jp/database/index.html
[5] LinkMax es 3.2 http://www.i-ze.com/main/product_i_outline.html
[6] A.Gupta, and I. S. Mumick. “Maintenance of materialized views: Problems, techniques, and applications”. Bulletin of the IEEE Technical Committee on Data Engineering, 18(2):3-19, 1995.
[7] Y. Zhuge, H. Garcia-Molina, J. Hammer, J. Widom, “View maintenance in a warehouse environment”, SIGMOD 95.
[8] S. Adali, K. Candan, Y. Papakonstantinou, and V. S. Subrahmanian. “Query caching and optimization in distributed mediator systems”. In Proc. of the ACM SIGMOD International Conf. on Management of Data, pages 137–148, Montreal, Canada, June 1996
[9] K. Salem, K. Beyer, B. Lindsay, “How To Roll a Join: Asynchronous Incremental View Maintenance”, In Proc. of ACM SIGMOD International Conf. on Management of Data, pages 129-140, Texas, United States, 2000.
[10] D. Agrawal, A. El Abbadi, A. Singh, T. Yurek, “Efficient View Maintenance at Data Warehouses”, In Proc. of ACM SIGMOD International Conf. On Management of Data, pages 417–427, Arizona, United States, 1997.
[11] http://www.tpc.org/tpch/