2005－DBS－137（Ⅱ）（70）
2005／7／14

クラスタ DBMS に適したインデックス構成法

伊藤 大輔　清水 晃　馬場 恒彦†

あらまし　近年TCO削減の手段としてクラスタサーバを用いた複数業務の統合運用に注目が集まり、その結果ミドルウエアには柔軟な構成変更を行うことが要求されている。我々は無共有型 DBMS においてもこの要求を満たせるよう、データ領域リマッピング機能を提案している。提案手法を用いることで物理的なデータ移動を伴わない高速な構成変更が実現される。本稿では、B-Tree インデックスにもリマッピング機能を適用することで、再作成不要なインデックス構成方法を提案した。実験の結果、提案方式を用いることで、オンライン業務システムの性能に影響を与えることなくインデックスを再構成できる見通しを得た。

Proposal of Index Architecture for Clustered DBMS

Daisuke ITO　Akira SHIMIZU　and　Tsunehiko BABA‡

Abstract　The demand of consolidating two or more business operations with cluster server rises as a means of the TCO reduction in recent years and as a result it is demanded for the middleware to do a flexible composition change. To fill this demand with shared-nothing DBMS, we have been proposing the data area remapping function. To use the proposal function, we can change the composition of the DBMS without any data migration. In this paper, applying the function to a B-Tree index, we proposed new index architecture needs no re-creation and demonstrated efficiency for the online business system.

1. はじめに

企業間の競争激化に伴い、ビジネスプロセスの継続的な変化に追随可能で総保有コスト (TCO) の低い IT プラットフォームが要求されている。この要求を満たすソリューションの一例として、クラスタサーバ上での複数業務の統合運用が挙げられる。これは、従来は業務ごとに別々のシステムとして設計され、また運用されてきた複数のオンライン業務システムを、単一のクラスタサーバ上のシステムとして統合運用するソリューションである。一般にオンライン業務システムには「受注業務は月曜日に高負荷になる」「給与計算業務は月末に高負荷になる」といった、予測可能な負荷変動がある。そこで、予測可能な負荷変動に合わせて業務ごとのサーバ割り当て台数を変更することで、サーバの総保有数を削減する事が可能となる。また、ブレードサーバに代表される管理性に優れたクラスタサーバを用いることで、運用コストを削減することが可能となる。このようなクラスタサーバ上での統合運用を実現するためには、IT プラットフォームを構成するミドルウエアの構成を多数のサーバ上で台数の増減に応じて高速に変更できることが要求される。

現在のオンライン業務システムの主流は Web 層 / アプリケーション (AP) 層 / データベース (DB) 層からなる Web3 階層システムであり、統合運用の有力ターゲットである。ここで、Web 層および AP 層は多数のサーバ上で台数の増減に合わせて高速にミドルウエアの構成を変更できるが、データ管理を担う DB 層では高速なミドルウエアの構成変更は困難であった。DB 層を構成するミドルウエアの一種であり、数百台規模の高いスケーラビリティを持つことで知られている無共有型データベース管理システム (DBMS) に着目すると、DBMS を構成するサーバごとに個別に割り当てられるストレージ上にデータが重複なく配置されるためスケーラビリティが高い反面、構成変更時にはストレージ間でデータの移動が生じ、高速な構成変更を行えない。

† 株式会社日立製作所中央研究所, {d.ito, ashimizu, tsn} _at_ crl.hitachi.co.jp
‡ Central Research Lab., Hitachi, Ltd., {d.ito, ashimizu, tsn} _at_ crl.hitachi.co.jp

そこで、我々は無共有型 DBMS においてスケーラビリティを維持したまま高速な構成変更を可能にするための拡張機能としてデータ領域リマッピング機能を提案している[1]。データ領域リマッピング機能では、共有ストレージを予めサーバ台数の数倍から数十倍程度の小領域に分割しておき、データをこれらに分割して保存する。これら小領域を、マッピング管理機構を用いて複数のサーバに重複なく割り当て、構成変更の際には、割り当て変更を行うことでデータ移動を生じない高速な構成変更を可能にする。

一方、DBMS では検索高速化のためにインデックスと呼ばれる索引を用いる。インデックスは表の各行を特定の列に対してソートした索引であり、特に B-Tree と呼ばれる木構造インデックスが広く用いられている。しかし、従来手法による B-Tree インデックスはソートされた索引であるため、DBMS の構成変更を行った場合に再作成が必要になる。そのため、B-Tree インデックスにもデータ領域リマッピング機能を適用し、再作成を伴わない高速な構成変更を実現することが望まれる。

そこで、本稿ではデータ領域リマッピング機能に適した、構成変更後も再作成不要なインデックス構成方法の提案と、オンライン業務システムを模した評価実験を実施し、結果を示す。

2. データ領域リマッピング機能

2.1. 従来手法によるクラスタ DBMS の課題

近年、大規模なデータを扱う場合に、従来のメインフレームに代表される高価な大型サーバの代わりに複数の安価な小型サーバからなるクラスタ DBMS を用いる運用が、TCO 削減の観点から注目されている。クラスタ DBMS を構成可能な DBMS アーキテクチャの 1 つに無共有型 DBMS がある。無共有型 DBMS にはサーバごとに個別に割り当てられるストレージをデータ領域として用いるためデータが重複なく配置され、データアクセスの際に複数サーバにまたがる排他制御が必要ないという特長がある。また、無共有型 DBMS では、少なくとも 1 台のユーザからのクエリを受け付けるフロントエンドサーバと、複数台の実際にデータ処理を行うバックエンドサーバを用いる。

無共有型のクラスタ DBMS では、単一の表を複数のバックエンドサーバに対し行ごとに分割して格納する。このとき、表を構成するある列を分割キー列と指定し、その分割キー列の値を用いて分割先を決定する。ここで、バックエンドサーバ間の格納行数に偏りが生じると処理速度が劣化するため、行数を均等に分割する必要がある。分割先を決定する方式として一般的なものに、ハッシュ分割方式とキーレンジ分割方式が挙げられる。このうち、分割キー列の値の分布によらずデータを均等に分割可能な方式はハッシュ分割方式である。ハッシュ分割方式では、分割キー列にハッシュ関数を適用し、その剰余を用いて各行を各サーバに均等に分配する。図 1にハッシュ分割方式を用いて 3 分割した、フロントエンドサーバ 1 台 / バックエンドサーバ 3 台からなる無共有型クラスタ DBMS の構成例を示す。

2.2. データ領域リマッピング機能

データ領域リマッピング機能は、エンタープライズ環境において急速に普及しつつある SAN ストレージを前提としている。従来の無共有型 DBMS において SAN ストレージを用いる場合、バックエンドサーバごとに独立した論理ボリュームを割り当てることで無共有型の構成を取っていた。

一方、我々は SAN ストレージを用いた場合、バック

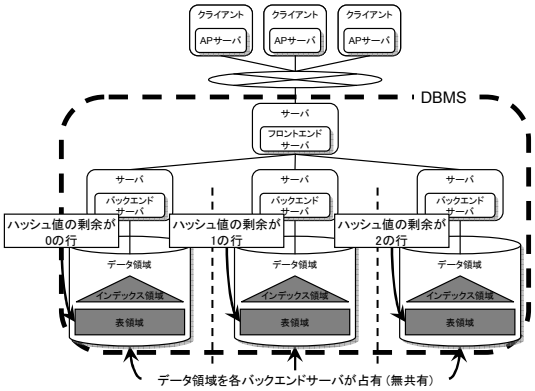


図 1 :無共有型クラスタ DBMS の構成

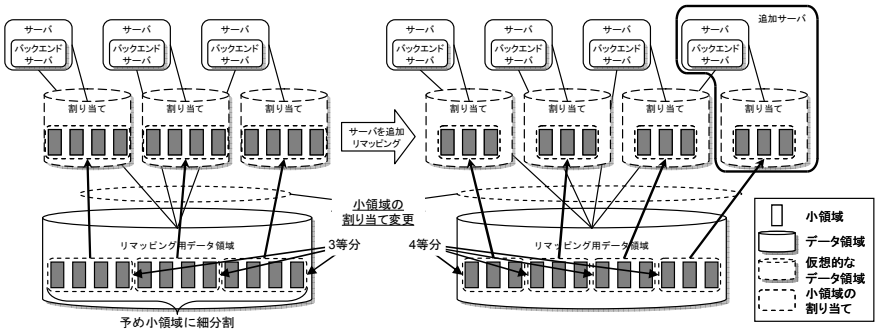


図 2：データ領域リマッピング操作の例

エンドサーバは物理的には任意のバックエンドサーバに割り当てられた論理ボリュームにアクセスできることに着目した。そこで SAN ストレージ上に共有ボリュームを準備し、共有ストレージを予め適当な定数の小領域に分割し、ハッシュ分割法を用いてデータをこれら小領域に分割して保存する。これら小領域はマッピング管理機構を用いてバックエンドサーバに重複なく割り当てる。構成変更に伴うバックエンドサーバ数の変更の際には、小領域とバックエンドサーバの再割り当て (リマッピング) を行う。リマッピング操作のとして、図 2に新規 DB サーバを追加する際のデータ領域リマッピング操作の例を示す。このように物理的なデータ移動の代わりに小領域のリマッピングを行うことで、構成変更所要時間を大幅に短縮することが可能となる。データ領域リマッピング機能の詳細に関しては、文献[1]を参照されたい。

3. データ領域リマッピングとインデックス

3.1. DBMS 向けインデックスの要件

DBMS では表検索高速化のためにインデックスと呼ばれる索引構造を用いる。一般に、DBMS ではデータを長期保存するストレージとして RAID 装置に代表されるディスク装置を用いる。ディスク装置はブロック I/O デバイスの一種であり、データの入出力はある程度のサイズのブロックを用いて行われる。そのため、1Byte のデータの入出力を行う場合にもブロックサイズ分のデータを入出力する必要がある。そこで、インデックスにはこのようなディスク装置の I/O 特性に適した構造が要求される。

また、DBMS では大量のデータを扱う必要がある。データの種類によっては日々データ量が増加し続ける。このようなデータの例として、顧客の購買履歴や口座の入出金情報が挙げられる。ここでデータ量の増加速度は正確に見積もることが難しく、結果として 1 年後のデータ量が正確に見積もれないということが容易に起こり得る。そこで、インデックスの構造には、予測の難しいデータ量の増大に耐えうるスケーラビリティを持つこともあわせて要求される。

以上のことから、DBMS 向けの汎用インデックス構造として、以下の 2 つの条件を満たすことが望まれる。

1. ディスク装置の I/O 特性に適していること
2. 高いスケーラビリティを有すること

インデックス構造として広く知られたものに、2 分木インデックス、B-Tree インデックス、ハッシュインデックスがある。これら 3 つのインデックスはそれぞれ長所短所を有するが、上記の DBMS 向け汎用インデックス構造には表 1 に示すように B-Tree インデックスが適している。

なお、B-Tree インデックスから派生したインデックス構造として著名なものに B+-Tree および B*-Tree インデックスがあるが、表 1に示した基本性質については B-Tree インデックスと等しい。これら B-Tree および派生したインデックスは多くの商用 DBMS で標準的なインデックス構造として実装されている。

3.2. データ領域リマッピング環境において B-Tree インデックスを用いた際の課題

データ領域リマッピング機能を適用した環境では、

表 1：主要インデックス構造の特性

種別	1. I/O 特性	2. スケーラビリティ
2 分木	×	○
B-Tree	○	○
ハッシュ	○	×

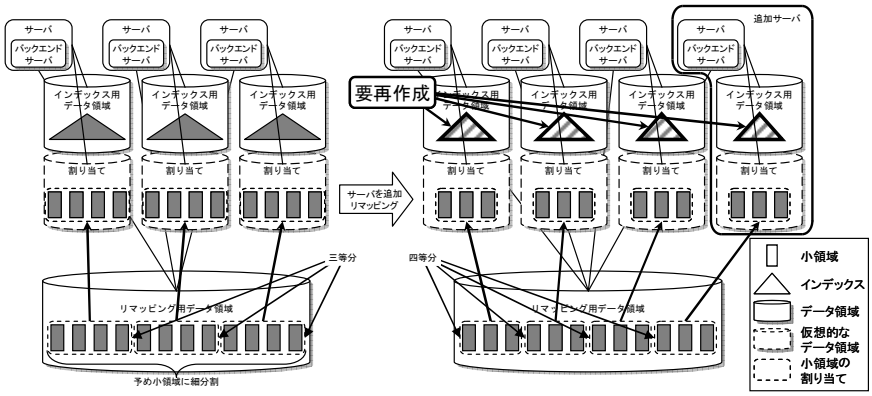


図 3：従来方式インデックスの課題

バックエンドサーバの台数を変更する際に小領域の割り当てを変更することで、物理的なデータ移動を伴わない構成変更を実現する。ここで、B-Tree インデックスが表を構成する行をある列に従いソートした構造であるため、バックエンドサーバ間で行の移動が生じた場合には B-Tree インデックスの更新が必要になる。なお、ここではソートに用いる列をインデックスキー列と呼び、特に表の分割キー列をインデックスキー列とするインデックスを分割キーインデックスと呼ぶ。

データ領域リマッピング機能ではリマッピングによる小領域の割り当て変更に伴い一度に大量の行が移動するが、物理的な移動を伴わないため移動する行数と割り当て変更所要時間の間に相関関係はない。しかし B-Tree インデックスの更新は行数に依存した時間を要するため、大量の行が移動する場合にはデータ領域リマッピング機能による割り当て変更と比較して長時間を要する。図 3にバックエンドサーバ 3 台構成からなる DBMS に 4 台目のバックエンドサーバを追加した場合の例を示す。データ領域にデータ領域リマッピング機能を適用することで全てのバックエンドサーバの割り当てデータに変化が生じるため、全てのバックエンドサーバにおいてインデックスの更新が必要になる。例えば、5章の実験で用いた我々のプロトタイプ上では 1,500 万行の表および 1 つのインデックスを用いた場合に、リマッピング操作が数十秒で完了するのに対し、インデックスの再作成には数分を要する。そのため、データ領域リマッピング機能に適したインデックスの構成方法が必要になる。

4. 提案方式

4.1. 小インデックス方式

3.2節に記した課題は、B-Tree インデックスがリマッピング可能な構成ではないため、リマッピングによる論理的なデータ移動に対して物理的な B-Tree インデックスの再作成が必要になったことが原因である。そのため、B-Tree インデックスもリマッピング可能な構成とすることで、課題を解決できる。

そこで、表領域の細分化された小領域ごとに B-Tree インデックスを作成し、表領域とインデックス領域を合わせて割り当て変更することで、リマッピング可能な B-Tree インデックスを構成する。3 バックエンドサーバ構成からなる DBMS に 4 台目のバックエンドサー

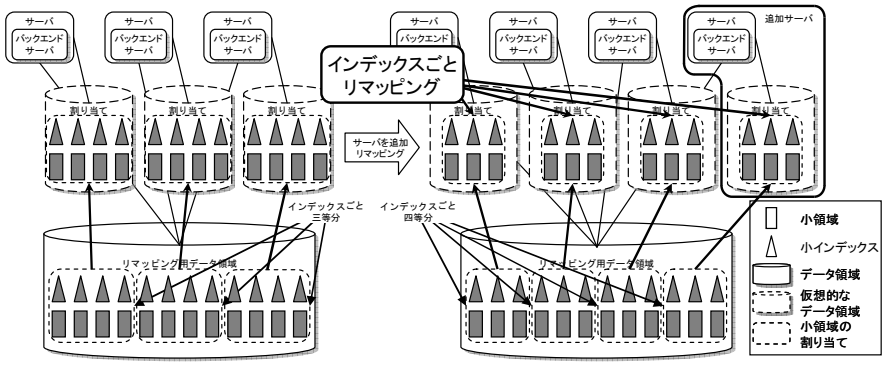
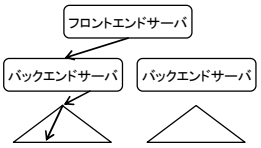
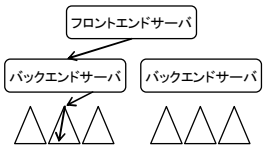


図 4：リマッピング可能なインデックスの構成

表 2：インデックス探索の I/O パターンと特徴

評価項目	大インデックス	小インデックス
I/O パターン		
特徴	フロントエンドサーバによる検索対象絞込み後、特定のバックエンドサーバのインデックスのみ探索を行う。	フロントエンドサーバによる検索対象の絞込み後、特定のバックエンドサーバ内でさらに検索対象となる小インデックスの絞込みを行う。

△ インデックス　⟶ 探索順序

バを追加した場合の、リマッピング可能な B-Tree インデックスの構成例を図 4に示す。図に示したとおり、一対一に対応する表とインデックスの小領域が対になって割り当て変更されるため、表とインデックスの整合性が保たれる。ここで、図 3に示した複数の小領域にまたがる B-Tree インデックスを大インデックスと呼び、また図 4に示したリマッピング可能な小領域ごとの B-Tree インデックスを小インデックスと呼ぶ。

4.2. オンライン業務システムへの応用

小インデックスはリマッピング可能である反面、大インデックスとは構造が異なるため、インデックス探索の性能に影響を与える。小インデックスを用いた場合に探索所要時間が変化する要因として探索対象となる小インデックスの切り替えに伴う CPU コストの変化とインデックス探索に伴う I/O 回数の変化が挙げられる。しかし、一般に I/O 時間の単位は CPU 時間の単位より 3 桁程度大きいことから、I/O 回数の変化が支配的であると予想した。

インデックス探索の I/O パターンにはいくつかの種類があるが、ここでは表 2を用いて Web3 階層システムのようなオンライン業務システムで主に用いられる分割キーインデックスに対する等号条件検索を用いた場合の I/O パターンに関して考察する。分割キーインデックスに対する等号条件検索の場合、まずフロントエンドサーバよって探索対象データを持つ可能性のあるバックエンドサーバが絞り込まれる。大インデックスの場合にはこの時点で探索対象となるインデックスがただ 1 つに絞り込まれ、インデックス探索が開始される。一方の小インデックスの場合には、1 つのバックエンドサーバに複数の探索対象インデックスが属するため、ハッシュ値を用いた絞込みをもう一度行って探索対象を 1 つに絞り込み、探索を行う。インデックスの探索は、大 / 小インデックス共に B-Tree インデックスの深さ方向の探索のみである。小インデックスは、インデックス片 1 つあたりの葉ページ数が大インデックスと比較して少ないため、深さは同じか低くなる。このため、インデックス探索の I/O 回数は等しいか少なくなる。

このように、小インデックスの場合には大インデックスの場合と比較して探索対象インデックスを 1 つに絞り込むまでに 1 ステップ余計に要するが、I/O 回数は等しいかもしくは減少するため、分割キーインデックスに対する等号条件検索の性能に関して従来と同等もしくはより高速になると予想される。

5. 実験と考察

5.1. 実験の目的

ここでは4.2節に記した机上評価から得た課題の確認と、小インデックス方式をプロトタイプ DBMS 上で動作させ、小インデックス方式のみを用いた運用の実現可能性を評価することを目的とする。具体的には、バックエンドサーバ追加に伴い 1 バックエンドサーバあたりの小領域割り当て数が変化することを想定し、小インデックスの分割数を変化させ、小インデックスと大インデックスの探索性能を比較する。より大きい分割数においても高い性能を維持することは、より多くのバックエンドサーバ数に対してデータを均等に分割可能であることを意味し、ビジネスの継続性の観点から望ましい特性である。

5.2. 実験方法と実験モデル

5.2.1. 実験方法

本実験では小インデックスの分割数を変更する際に、データ量一定かつバックエンドサーバを 1 台として、同一バックエンドサーバ内での分割表の分割数と小イ

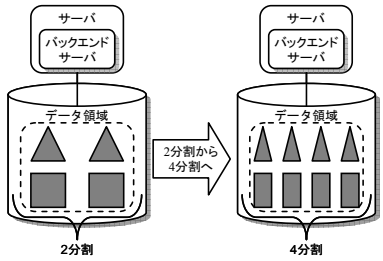


図 5：小インデックスの分割数の変更

ンデックスの分割数を変更する。図 5に 2 分割から 4 分割の例を示す。データ量を一定にする事で、データ量に関するスケーラビリティの影響を受けることなく小インデックスを導入することによる性能変化を観測できる。また、バックエンドサーバ数を 1 にすることで、バックエンドサーバ数に関するスケーラビリティおよびバックエンドサーバ間の TCP/IP 通信の遅延といった不確定要素の影響を受けることなく、小インデックスの導入による性能変化を観測できる。

5.2.2. 実験環境

実験では 64 台程度のバックエンドサーバに対してほぼ均等にデータを分配することを目指し、最大 128 分割された分割表用データ領域と小インデックス用データ領域、非分割の大インデックス用データ領域を用いる。なお、これら 3 種類のデータ領域はすべて I/O の最小単位であるページのサイズを 4KB、複数ページをまとめて管理する単位であるセグメントのサイズを 128 ページとした。実験はフロントエンドサーバおよびバックエンドサーバをそれぞれ専用サーバに分離した 2 台構成で行う。実験環境の模式図を図 6に示す。また、実験で用いたサーバ類の仕様を表 3に示す。

5.2.3. スキーマ

実験では業界標準ベンチマークである TPC-H ベンチマークの orders 表および表データを用いる。サーバの主記憶のサイズと比較して十分に大きなサイズとなるよう、表データのサイズを表すパラメータであるスケールファクタを 10 とした。この場合、orders 表は 15,000,000 行 (1.7GB) になる。TPC-H ベンチマークの詳細は文献[2]を参考されたい。

5.2.4. バッファサイズ

小インデックスを用いた際の性能変化要因は、4.2節に記したとおり、CPU 時間と比べて I/O 時間が長いことと予想される。一般に DBMS では CPU 時間と I/O 時間の差を縮めるために、主記憶の一部を I/O バッファとして割り当てる。I/O バッファの量が大きいほど

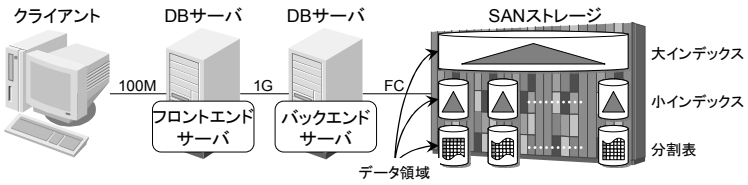


図 6：実験環境の模式図

表 3：実験環境

項目	内容
データベースサーバ	IA-32 上の Redhat8.0 (Linux2.4.20-20.8 標準構成) × 2 CPU： Xeon@2.4GHz Mem： 1GB LAN： GEth HBA： qla2300
SAN ストレージ	SANRISE9570V キャッシュ： 512MB

実 I/O の回数が減り、見かけ上 I/O 時間が短縮される。そのため、I/O バッファを大きく取れる環境では、小さく取った環境よりも小インデックス方式の導入による性能変化は小さくなる。

しかし主記憶はストレージに対して容量が小さく高価なため、一般に必要最小限の I/O バッファ量を割り当てて運用される。B-Tree インデックスに対する I/O バッファ量の割り当て方法として、リーフ以外のデータが載る量を目安とする方法が知られている。本実験では I/O バッファ量を大インデックスのリーフ以外のデータが載る量とし、約 12MB の主記憶を割り当てる。

5.2.5. クエリ

実験ではオンライン業務システムを前提とした分割キーインデックスに対する等号条件検索を行う以下のクエリを用いる。ここでは大 / 小インデックスの探索時間差が顕著に現れるよう、インデックス検索結果の数え上げ関数を用いてインデックスだけを探索するようにする。また、複数回の実行に備えて条件節中に ${param}で示したパラメータを含める。

```
select count(*) from orders where o_orderkey=${param};
```

ここでは大 / 小インデックスの組み合わせごとに変数を変えながら 2,000 回のウォームアップを行いインデックス用 I/O バッファを実運用に近い状態とし、その後 100 回の測定を行い、その平均値を用いて評価する。なお、これら 2,100 個の変数はクエリ毎に同じ順で与え、大 / 小インデックス間で条件に差がでないようにする。また、測定はクライアントとフロントエンドサーバ間の通信の影響を受けないよう、DBMS 内部の統計情報を参照して行う。

5.3. 実験結果と考察

5.2.5節に記した通り、100 回の測定の平均値を取得した実験結果を表 4 に示す。なお分割数を変えても大インデックスの構成は影響を受けないため、大インデックスに関しては 1 分割 (非分割) の場合の値のみを測定した。また、1 分割の場合には大 / 小インデックスの構成に差が生じないため、小インデックスの場合の測定を行っていない。また、図 7に大インデックスの 1 分割 (非分割) の場合を 1 とした相対値のグラフを示す。

実験の結果、大 / 小インデックスの間に所要時間の有意な差は見られなかった。これは4.2節の予想に合致した結果である。なお、小インデックスを用いた場合には、インデックスの小領域 1 つあたりのリーフノード数が減少するため、インデックスの深さが 4 段から 3 段に減少した。その結果、I/O が 1 回少なくなったが、ここで用いた実験環境では 1I/O の所要時間は数十マイクロ秒～数ミリ秒程度であり、トランザクション所要時間の有意な差にはならなかった。

6. 従来研究との比較

無共有型クラスタ DBMS においてデータの再配置が発生した際の B-Tree インデックス再構成について扱った従来手法は主に表データがキーレンジ分割方式によって分割されていることを前提としている。[3]では B-Tree インデックス再構成を行う 2 つの両極端な方式である OAT と BULK を提案している。OAT はバックエンドサーバ間で 1 ページずつ表データを移動し、そのページに含まれる数行のデータに関してインデッ

表 4：クエリ実行時間の平均値

インデックス	分割数ごとのクエリ実行時間 (ms)							
	1	2	4	8	16	32	64	128
大	49.06	-	-	-	-	-	-	-
小	-	48.12	48.64	48.79	48.76	50.77	48.74	48.59

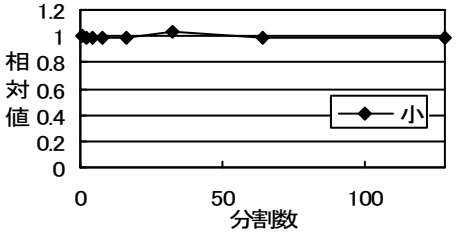


図 7：大インデックスを基準とした相対値

クス再構成を行う方式であり、低速だが他のオンライン処理の性能劣化が少ない。一方の BULK は複数の表データページを一括移動し、インデックスも一括して再構成する方式であり、このときインデックスの再構成が連続したキー値の挿入であることが保証されるため、シーケンシャル I/O による高速化が望める反面、リソース使用量が多い。提案手法のように大量の表データが一括移動する場合に[3]を応用すると BULK を用いる事になるが、表データがハッシュ分割方式で分割されている場合にはインデックスの再構成を 1 行ずつ行う必要があり、シーケンシャル I/O による高速化は望めない。

一方、[4]および[5]は効率的な負荷分散を目的とした改良 B-Tree を提案している。[4]は無共有型クラスタ環境における更新処理の負荷と参照処理の負荷分散のバランスを考慮した B-Tree 構造として Fat-Btree を提案している。Fat-Btree は全バックエンドサーバに分割配置された全表データに対する仮想的な B-Tree インデックスのうち、各バックエンドサーバにその保持するリーフノードおよびその祖先ノードだけをコピーした構造であり、アクセスは多いが更新の少ないルート近傍は多くのバックエンドサーバにコピーされ、逆に更新は多いがアクセスの少ないリーフ近傍はコピーされないという特徴を持つ。この構造のメリットとして、負荷分散のため隣接バックエンドサーバ間で表データが移動した際のインデックス更新コストが低いことが挙げられる。また、[5]は Fat-Btree と似た構造であるが、無共有型クラスタ環境におけるノード間の負荷偏り制御に主眼を置いた aB+-Tree を提案している。aB+-Tree では各バックエンドサーバに属する B+-Tree インデックスに深さに関する制約を設け、偏り制御の際に隣接バックエンドサーバ間でインデックスの部分木およびそこに属するデータページの一括移動を行う。深さに関する制約によって部分木は容易に「接木」可能であり、BULK と同じくシーケンシャル I/O によって高速なインデックス再構成が実現される。このように両手法とも表データがキーレンジ分割方式によって分割されていることを前提としており、提案手法のようにハッシュ分割方式で分割されたデータが一括移動する場合に応用すると[3]の場合と同じくインデックスの再構成を 1 行ずつ行う必要があり、シーケンシャル I/O による高速化は望めない。

7. まとめと今後の課題

本稿では無共有型 DBMS に対してデータ領域リマッピング技術のような高速にデータ移動および構成変更を行う技術を適用した場合に顕在化するインデックスの再編成問題に対して、オンライン業務システムへの応用を想定し小インデックスを用いて問題を解決する方式を提案した。また、オンライン業務システムを想定したクエリを用いて大 / 小インデックスの性能比較を行い、性能差が観測されないことを実測した。この結果、提案方式を用いることで、オンライン業務システムの性能に影響を与えることなくインデックスを再構成できる見通しを得た。

今後の課題として、クエリパタンの異なるオンライン業務システム以外のシステムへの応用があげられる。

文　献

[1] 伊藤大輔, 牛嶋一智, “無共有型 DBMS 向けデータ領域リマッピング機能の開発,” FIT2004, 分冊 D, D-016, Sep, 2004.

[2] Transaction Processing Performance Council, “TPC BENCHMARK™ H (Decision Support) Standard Specification Revision 2.1.0,” on http://www.tpc.org/tpch/spec/tpch2.1.0.pdf, Aug, 2003.

[3] Kiran J. Achyutuni, Edward Omiecinski and Shamkant B. Navathe, “Two techniques for on-line index modification in shared nothing parallel databases,” Proc. 1996 ACM SIGMOD International Conference on Management of Data, pp. 125 – 136, Quebec, Canada, June, 1996.

[4] Haruo Yokota, Yasuhiko Kanemasa and Jun Miyazaki, "Fat-Btree: An Update-Conscious Parallel Directory Structure", Proc. of 15th International Conference on Data Engineering, pp. 448 - 457, Mar, 1999.

[5] Mong Li Lee, Masaru Kitsuregawa, Beng Chin Ooi, Kian-Lee Tan and Anirban Mondal, “Towards self-tuning data placement in parallel database systems”, Proc. 2000 ACM SIGMOD International Conference on Management of Data, pp. 225 – 236, Texas, United States, May, 2000.