情報処理学会論文誌　データベース　Vol. 4　No. 4　11–33 (Dec. 2011)

タプル再分散不要の並列データベース構成法

油井　誠[†1]　小島　功[†1]

本論文では無共有計算機設計においてデータウェアハウス処理を行ううえでタプルの再分散の問題に着目し，タプルの再分散を必要としない並列データベース構成法を述べる．特に Φ ハッシュ分割と呼ぶ，タプルの再分散を必要としないテーブル分割手法を提案する．Φ ハッシュ分割ではノード数に対するスケーラビリティを維持しながら，TPC-H などの複雑なデータ分析問合せを並列処理することができる．TPC-H の SF=100 による評価実験で，提案手法が MapReduce に基づく競合システム Hive に対して顕著な性能面での優越（3.1 倍〜19.9 倍）があることを示すとともに，我々の問合せ処理手法の現実装における有効範囲と制限に考察を与える．

A Parallel Database Architecture Avoiding Tuple Redistribution

Makoto Yui[†1] and Isao Kojima[†1]

This paper describes a parallel database architecture avoiding tuple redistribution. We focus on the tuple redistribution issue; it becomes problematic on processing data warehouse queries on a shared-nothing architecture. And then, we propose a novel table partitioning technique, named Φ hash partitioning, that can avoid redistribution of tuples. The Φ hash partitioning can handle complex analytical queries, as ones in TPC-H, in parallel. Moreover, the partitioning scheme does not have a scalability limit on the number of nodes. The results of experimental evaluation showed that our system is much (3.1 to 19.9 times) faster than a MapReduce-based system (Hive) on TPC-H SF=100. We also give a consideration on the capabilities and limitations of the current implementation of our query processing scheme.

†1 産業技術総合研究所情報技術研究部門
Information Technology Research Institute, National Institute of Advanced Industrial Science and Technology

1. まえがき

無共有設計の計算機クラスタで巨大な構造化データのデータ分析を行ううえでの選択肢として，並列データベースシステムを用いるか，あるいは MapReduce[1] に基づくシステムを用いるかが開発者の主要な選択肢となっている．理想的には，細粒度の負荷分散や高い耐障害性といった MapReduce の利点と，効率や性能面の並列データベースの利点は同時に提供されるべきである．HadoopDB[2] や Osprey[3] は，無共有型並列データベース上でリレーションを水平分割したうえで，MapReduce 型のタスク実行方法を採用することで，並列データベース上に耐障害性と細粒度の負荷分散を実現している．

こうしたデータベース管理システムと MapReduce 型のタスク実行のハイブリッド手法が注目を集めている中で，本論文では，データ分割に基づく並列演算を行うときに共通する課題であるタプルの再分散（redistribution）に着目する．並列ハッシュ結合[4),5)] では，ハッシュバケットを処理するデータベースノードを割り当てて，割り当てられたノードがそのバケットに対する関係演算を処理することで並列処理を実現している．しかし，複数の異なる属性を利用した結合演算を並列処理するうえではタプルの再分散が不可欠である．タプルの再分散では，リレーションをディスクから読み込み，タプルごとにハッシュ値を計算し，それを基に担当ノードを決定して割り当て，必要に応じて索引の構築や統計情報の収集を行うといったプロセスが発生する．そのため，タプルの再分散は無共有型計算機クラスタにおける並列データベース処理の潜在的な問題であり，再分散のオーバヘッドは並列データベースの並列処理による性能向上を阻害する[6]．特に，無共有型計算機クラスタにおいてネットワークは唯一の共有リソースであること，多くの計算機クラスタで帯域の狭い 1 ギガビットイーサネットが依然として利用されていること，Amazon EC2 のような仮想化されたデータセンタで並列システムを運用する場合にネットワークのスループットが不安定でボトルネックになりがちであること[7] を考慮すると，データ交換量を低減することは鍵となる．

本論文では，再分散のオーバヘッドを低減するために，*Field-interleaving*（Φ）ハッシュ分割と呼ぶ新しいテーブル分割手法を提案する．Φ ハッシュ分割は，データベーススキーマ（より正確には，データベースカタログの参照整合性制約）からテーブル分割方法を導出する．そして，タプルのノードへの割当てを行うときに，テーブルがどのような属性を用いて分割されたかという情報を分割されるタプルごとに追加属性として差し込む．こうしてタプルに差し込まれたメタデータは，問合せ処理時に該当タプルが与えられた問合せにおいて必要か否かを判断するための材料となる．問合せ処理器は，与えられた問合せを MapReduce

の処理モデルで並列処理する場合に必要となる shuffle キーの集合（以降，これを問合せが期待する分割属性と呼ぶ）を問合せの結合条件や group by 句から求め，差し込まれたメタデータと shuffle キーの集合を利用した選択演算を問合せに加えたうえで問合せを評価する．本論文の貢献は次のとおりである．

- Φ ハッシュ分割と呼ぶタプル再分散を必要としないテーブル分割手法を提案する．このテーブル分割手法は，ノード数増加に対するスケーラビリティを保証したうえで，複雑なデータ分析問合せを独立並列（Independent Parallel）に処理することができる．また，データベーススキーマからテーブル分割方法を自動的に導出するため，複雑な要因が絡むテーブル分割属性の決定にデータベース管理者（DBA）の介入を必要としない．
- Φ ハッシュ分割と主記憶データベースの組合せによる主記憶を有効活用した MapReduce 処理手法を紹介する．Φ ハッシュ分割の導入の狙いは，データの移動をなくしてディスク I/O やネットワークの負荷といった shuffle 操作のオーバヘッドを下げることだけでなく，データの移動にともなうメモリからのデータ入出力を避け，無共有型並列データベースの各データベースノードで問合せを極力インメモリ（memory mapped データに対する map 処理と単一の reducer による処理）で処理することにある．本論文では，MonetDB/MR を主記憶データベースの一種である MonetDB[8] をベースとして，Φ ハッシュ分割を利用した並列データベースシステム MonetDB/MR を設計・構築し，キャッシュの有無の両方の場合の評価を与えることで，動的なデータ分散を抑えることの効果とその因子を分析する．
- 32 ノード構成で TPC-H SF=100（約 100 GB のデータベース）で評価を与え，提案手法の実装である MonetDB/MR が MapReduce に基づいたデータ分析システム Hive に対して 3.1 倍〜19.9 倍，豊富なメモリを積んだ単一ノード構成の MonetDB に対して平均 5.8 倍最大 51.89 倍の優れた性能を示すことを述べる．さらに，提案システムの比較から MapReduce ベースのシステムの改善可能な点に考察を与える．

本論文の構成は次のとおりである．2 章で並列データベースにおいて SQL 問合せがどのように並列処理されるかを説明する．3 章で既存のテーブル分割手法の問題点をあげ，4 章で提案する Φ ハッシュ分割手法を述べる．5 章で設計・構築した MonetDB/MR の構成を述べる．6 章で提案手法を評価する．7 章で関連研究について述べ，8 章でまとめる．

2. 並列 SQL 問合せ処理

本章では，並列データベースにおいて SQL 問合せがどのように並列処理されるかを説明する．まず 2.1 節で並列 SQL 問合せ処理の基本となる並列ハッシュ結合について説明し，2.2 節で問合せ処理の中でタプルの再分散がどのように行われるか，例を述べる．

2.1 並列ハッシュ結合

並列（結合）演算にハッシュ分割を適用したアルゴリズムは文献 4) で提案された．並列ハッシュ結合では，バケットを処理するデータベースノードを割り当てて，割り当てられたノードがそのバケットに対する関係演算を処理することで並列処理を実現する．ハッシュ分割によるクラスタリングは，結合演算だけでなく，射影演算，集合演算といったデータベースの操作をデータ並列に評価するうえで有用である．

図 1 に示すように，2 つのリレーションそれぞれのタプルが結合属性のハッシュ値によってハッシュバケットごとにクラスタリングされていれば，異なるハッシュバケットに格納されたタプル間で結合処理は発生しない．したがって，それぞれサイズ N，M のリレーションの結合演算の逐次実行の場合の総処理時間 T は次のようになる（ここで，s はバケット数，n_i，m_i は i 番目のバケットのサイズ）．

$$N = \sum_{i=1}^{s} n_i, M = \sum_{i=1}^{s} m_i$$

$$T \propto \sum_{i=1}^{s} n_i \times m_i$$

一方，クラスタリングを行わない場合には，$T \propto N \times M$ である．並列ハッシュ結合で

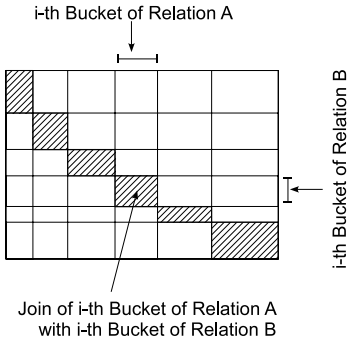


図 1　ハッシュ結合の図解
Fig. 1　Graphic illustration of hash join.

は，図 1 の斜線部分が実際に処理される．その他の部分の計算を除去できるため，ハッシュ分割を用いたクラスタリング手法は関係演算の負荷を劇的に減らすことができる．こうした利点からハッシュ分割に基づく並列処理は，Teradata などの商用並列データベースや MapReduce [1] で広く利用されている．

2.2　タプルの再分散

並列ハッシュ結合演算を効率的に処理するには，結合演算の結合属性（join attributes）を分割属性（partitioning attributes）として用いて，あらかじめハッシュ分割しておくことが肝である．結合属性を分割属性として選ぶことで，(1) 結合処理の突き合わせコストを下げ，(2) タプルの再分散処理を不要とし，また (3) 個々のパーティションごとに独立並列な問合せ実行を容易に（1 度のタスク配布と結果の集約で）実現することができる．たとえば，リレーション R1 とリレーション R2 がそれぞれ属性 A の値に基づいてハッシュ分割されているとき，R1 と R2 を属性 A に基づいて等結合するクエリは独立並列に実行可能である．

一方で，問合せごとに期待する分割要求は異なるため，事前に最適なデータ分割方法を選んだとしてもタプルの再分散が必要となり並列処理の適用範囲が狭まる．リレーション R1 とリレーション R2 がそれぞれ属性 A に基づいてハッシュ分割，リレーション R3 が属性 C に基づいてハッシュ分割されていると仮定すると，問合せ select * from R1, R2, R3 where R1.A = R2.A and R2.B = R3.B の並列実行プランはたとえば**図 2** のようになり，実行コストの高いタプルの再分散（Shuffle 操作）が不可避である．

なお，あらかじめハッシュ分割をしておくことの有用性は MapReduce 処理においても同様に重要である．HadoopDB [2] は結合属性をデータ分割属性に用いて，あらかじめデータをハッシュ分割しておくことを前提としている．同様に，Hadoop++ [9] は，Reduce フェーズで結合処理を行うために Shuffle 操作で大量のデータ交換が必要なことが MapReduce の潜在的なボトルネックであり，データのロード時にデータセットのスキーマ構成と問合せのワークロードを考慮することが Hadoop の性能を高めるうえで最も重要であると結論している．ただし，データ分割要求が問合せによって異なることを認めながらも，その対処はユーザ自身がスキーマ構成とワークロードを考慮するものとしている点で，システム側で本質的な解決策は示されていない．Jiang らは，あらかじめ行ったデータの分割情報を利用した結合処理を Partition Join として Hadoop に導入している[10]．なお，文献 10) でもデータの分割方法は利用者が指示するものであり，そのアドホックなデータ分割は異なるデータ分割を要求する複数の問合せ，あるいは複数のジョブに対して強固でない．

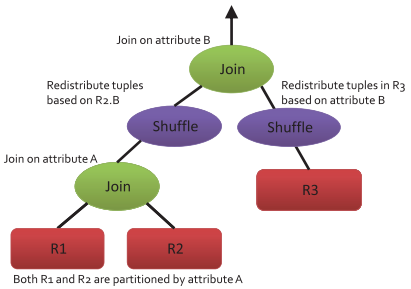


図 2　タプルの再分散が行われる例
Fig. 2　An example of tuple redistribution.

3.　既存のテーブル分割手法の問題点

テーブル分割の問題点は，次のとおりである．target を射影される属性とし，qualification を等結合に利用される属性とするようなクエリを $Q = \{target|qualification\}$ とし，3 つのリレーションの結合が行われる問合せを $Q = \{R_1.A, R_3.B|R_1.A = R_2.A \wedge R_2.B = R_3.B\}$ とする．ここで，すべての 3 つのリレーションを再分散なしに結合できるようにテーブル分割することは，結合結果の各タプルを一意に特定するような属性が存在する場合を除いて，従来，不可能とされてきた[11]．なぜならば，第 1 の結合述語はリレーション R_2 が属性 A によって分割されていることを期待するが，第 2 の結合述語はリレーション R_2 が属性 B で分割されていることを要求するからである．この 2 つのテーブル分割要求に矛盾が存在する．

この問題に対処するために，分散 Ingres [12] で開発された fragment and replicate strategy (FRS)[13] は，問合せで参照されるリレーションの 1 つをノード間に分割し，その他の参照されるリレーションはその分割表を保有するノードに複製する．そのうえで，問合せ処理では構成ノードすべてで問合せを評価したうえで，その結果をまとめて返す．FRS 分割では，問合せでアクセスされるリレーションに，分割済みリレーションが含まれない場合を除いて問合せを並列に評価することができる．たとえば，FRS 分割では R_1 を 2 つのリレーション F_{11}，F_{12} に分割して，それぞれをノード 1 とノード 2 に配置する．一方，リレーション R_2 をノード 1 とノード 2 の両方に複製して配置する．こうすることで，結合処理を負荷分散し，$R_1 \bowtie_\theta R_2 = (F_{11} \bowtie_\theta R_2) \cup (F_{12} \bowtie_\theta R_2)$（ここで，$\theta$ は等結合の条件）をノード 1 とノード 2 で並列に評価することができる．

文献 11) では，FRS が 1 つのリレーションだけを分割して他のリレーションの複製を全ノードに置くのに対して，テーブル間の参照関係に基づいた被参照表からの derived-

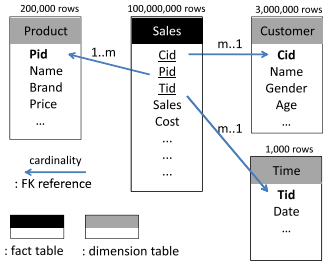


図 3　スタースキーマ構成の例
Fig. 3　An example of star schema.

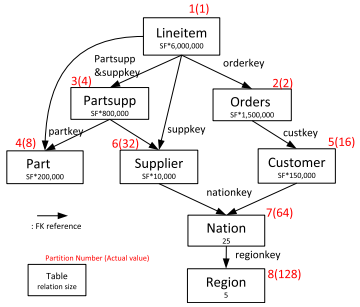


図 4　TPC-H のデータベーススキーマ（ここで SF はスケールファクタ）
Fig. 4　Database schema of TPC-H (SF: Scale Factor).

fragmentation を利用することで 1 つ以上のリレーションを分割して他のリレーションの複製を置く FRS の改良手法を提案している．誘導フラグメント化によるテーブル分割を行う際には，関数従属性のある 1 つの属性集合（dominated attributes）を選択しなければならないが，こうした単一の dominated attributes が存在するケースは限られる．たとえば，**図 3** のようにデータウェアハウスで一般的な複数のディメンション表が存在するスタースキーマ構成では FRS と同様の構成となり，並列処理できる部分が限られる．

3.1　データウェアハウスにおけるテーブル分割

データウェアハウスのスキーマは，図 3 に一例を示すように，少数（1 以上）の大きなファクト表と多数の比較的小さなディメンション表から構成される．1 つのディメンション表のエントリが複数のファクト表のエントリによって参照される one-to-many（1 : M）関係にあり，データウェアハウスの分析問合せはディメンション表とファクト表の等結合をともなうものがほとんどである．そのため，等結合演算を並列処理することが特に重要であり，相対的に大きなファクト表をできるだけ均等に分割することが必要である．

FRS 手法は，スタースキーマ構成について，ファクト表を分割してディメンション表を複製することで簡単に並列処理の恩恵を受けることができるため，既存のデータウェアハウス処理で利用されてきた[3),14)]．しかし，FRS 手法はファクト表が 1 つのものだけのスタースキーマ構成に有効に機能するが，TPC-H[15)] のように複数のファクト表が存在する複雑なスキーマや複雑な問合せには満足に対応できない．あるいは，タプルの動的再分散を必要とする．

データウェアハウスの利用では，ユーザはディメンション表の顧客の年齢に基づいて売上げの総計を求めるかもしれないし，ディメンション表の商品のブランドに基づいて売上げの平均を求めるかもしれない．そのため，ディメンション表の外部キーに基づいた誘導フラグメント化[⋆1]によりファクト表を分割する方法がより一般的である[17)]．分割の基礎とする被参照表の選択は，最もよくアクセスされる表か，あるいは結合処理の特性に基づいて行われる．

4.　Field-interleaving ハッシュ分割

本章では，3 章であげた既存のテーブル分割の問題への解として，問合せ評価時のタプル再分散を抑える Field-interleaving ハッシュ分割（Φ ハッシュ分割）を提案する．提案手法はオペレーションシステムのスナップショットであるオフラインのデータウェアハウスを適用対象とする[⋆2]．FRS 手法[13)] と異なり，対応するデータベースのスキーマをスタースキーマだけに限定しない．データウェアハウスの業界評価基準である TPC-H[15)] のデータベーススキーマ構成（**図 4**）および，その分析問合せを評価できるものとする．

4.1　基本的なアイデア

既存のテーブル分割手法では異なる属性で 3 つのリレーションが結合される場合，データ再分散なしに結合処理を並列処理することは，3 つのリレーションに 1 つの属性集合からの（推移的）関数従属性があるときを除けば不可能である．ここでは，図 4 の Lineitem 表（L），Partsupp 表（PS），Orders 表（O），Customer 表（C）の 4 つの関係を想定したと

⋆1 リレーションを対象となるリレーションに対する述部評価に基づいて分割する手法を primary horizontal fragmentation（PHF），他のリレーションに対する述部評価に基づいてリレーションを分割する手法を derived horizontal fragmentation（誘導フラグメント化）という[16)]．

⋆2 オフラインのデータウェアハウスはオペレーションシステム（たとえば，トランザクションデータベース）のスナップショットであり，分析用途に利用されるデータベースである．更新は夜間バッチ処理などで行われ，同時実行制御はトランザクション指向のデータベースのように重要ではない．

きのテーブル分割を例として，Φ ハッシュ分割の基本的なアイデアを述べる．

提案手法では，ファクト表から，つまり親リレーションよりも子リレーションから先にテーブル分割を行っていく[*1]．このテーブル分割手法は，3.1 節で説明した誘導フラグメント化手法[16] を基礎とする．ただし，誘導フラグメント化手法が基本的に単一の（親キーを持つ）被参照表の存在だけを許すのに対して，複数の被参照表が存在する構成に対応する．

Φ ハッシュ分割では，テーブルの分割にあたって次の 3 つのフラグメント化手法を組み合わせる．

(1) **primary fragmentation** 対象テーブルの主キーによってテーブル分割を行う．
(2) **derived-by-parent fragmentation** 対象テーブルの外部キー属性の値に基づいて，対象テーブルの分割を行う．
(3) **derived-by-child fragmentation** 参照整合性を保つために，参照テーブル（子テーブル）の分割に基づいて対象テーブルの分割を行う．

表それぞれを分割してできた関係の中で，元々の関係にある参照整合性制約が守られている必要がある．Lineitem 表をテーブル分割した際には，Lineitem 表の orderkey 属性と（partsupp, suppkey）属性に対応する Orders 表と Partsupp 表の該当タプルがそれぞれ存在する必要がある．一般に，誘導フラグメント化によるテーブル分割を行う際には，（推移的）関数従属性のある 1 つの属性集合を選択しなければならない．仮に関係が Lineitem 表，Partsupp 表，Customer 表に限り，問合せ結果が $L \bowtie O \bowtie C$ など custkey によって同定されるならば，Customer 表の主キーである custkey に基づいて誘導フラグメント化を行うことが考えられるが，実際には，Lineitem 表は Partsupp 表，Orders 表それぞれに従属している[*2]（$PS \rightarrow L$ および $O \rightarrow L$ と表記する）ため，Lineitem 表の誘導フラグメント化で基礎とする 1 つの属性集合を選択することができない．仮に複数の決定項（たとえば，PS と O）を用いて，従属関係にある Lineitem 表を分割したとすれば，$PS \rightarrow L$ や $O \rightarrow L$ の関係が崩れてしまうので管理が困難となる[16]．また，custkey による分割では $L \bowtie O$ や $L \bowtie PS$ に対応できない．

これに対して，提案手法では複数の決定項によりテーブル分割を行う．そのために，タプルごとにどの分割属性（集合）に基づいてテーブル分割が行われたかを示す隠し属性を加える．そのうえで，問合せ処理時に，この追加属性を利用した適切な選択演算を加えることでもともとの関係を復元する．たとえば，Lineitem 表は $PS \rightarrow L$ と $O \rightarrow L$ のそれぞれの関係によって誘導フラグメント化される．このとき，Lineitem 表のあるタプルは Partsupp 表の主キーによって分割されるかもしれないし，Orders 表の主キーによって分割されるかもしれないし，そのいずれか，あるいはその両方によって分割されるかもしれない．この Lineitem 表のそれぞれの部分を，L_{ps}，L_o，$L_{ps|o} \Leftrightarrow L_{ps} \cup L_o$，$L_{ps\&o} \Leftrightarrow L_{ps} \cap L_o$ として表記する．Φ ハッシュ分割では，この分割に利用された属性をメタデータとしてタプルごとに差し込む．この追加属性を利用することで，$L_{ps|o}$ などの部分表を選択演算により特定することができる．

primary fragmentation では，対象テーブルの主キーによってテーブル分割を行う．主キーによる分割は，外部キー制約が存在する 2 つのテーブル間の等結合処理での利用を見込むほか，複数の属性によって分割された対象テーブルを一意に特定したい場合に利用する目的で行う．

derived-by-child fragmentation は，primary fragmentation と derived-by-parent fragmentation によって分割されたリレーションに分割前のリレーションにあった参照整合性を保つために行う．たとえば，Partsupp 表を primary fragmentation（と derived-by-parent fragmentation）によって分割しただけでは，Orders 表の主キーによって分割された Lineitem の分割表 L_o は，Partsupp 表の主キーを参照する外部キー制約を満たさない可能性がある．そこで，Partsupp 表の derived-by-child fragmentation では，Lineitem 表の分割に基づいて Partsupp 表を分割して Lineitem 表の外部キー制約を保つ．

4.1.1　タプル分散の考察

図 5 に図解するように，Φ ハッシュ分割では 1 つのタプルを複数の分割キーによって 1 以上のバケットへ割り当てる．ここで，図 4 の Lineitem 表の分割を例に，N 個のハッシュバケットを用意してそれを 1 対 1 にノードに割り当てるとする．

タプル数を R として N 個のノードに理想的にタプルが分散されたとすると，1 つのバケットに R/N 個のタプルが格納される．図 5 に示すように，Φ ハッシュ分割は複数の分割キーに基づいてタプルがノードへ割り当てられる．このため，分割キーの数を P 個とすると，最悪のケースではそれぞれのバケットに $P(R/N)$ 個のタプルが格納される[*3]．

たとえば，Lineitem 表は主キーのほか，4 つの外部キー参照を持つため，4.1 節に述べた

⋆1 外部キー制約が張られた 2 つのリレーションでは，参照表が子リレーション，被参照表が親リレーションと呼ばれる[16]．図 4 の階層関係とリレーションの親子関係とは逆であることに注意されたい．
⋆2 表 PS と表 L が 1 対多の関係にあるとき，本論文では “表 L は表 PS に従属する” と表現する．
⋆3 ただし，これは子ノードからの誘導フラグメント化を考慮していない．Lineitem 表には子ノードが存在しないため，$P(R/N)$ が格納されるノード数の上限値となる．

表 1　タプルの分散
Table 1　The tuple distribution.

		region	nation	supplier	customer	part	partsupp	orders	lineitem
# of paritioning keys		1	2	2	2	1	3	2	5
# of rows		5	25	1,000,000	15,000,000	20,000,000	80,000,000	150,000,000	600,037,902
8 nodes	inserted	5	25	1,000,000	11,170,538.0	19,995,426.8	70,865,450.0	127,215,567.9	292,080,040.9
	inserted/rows (ratio)	100%	100%	100%	74.5%	100%	88.6%	84.8%	48.7%
16 nodes	inserted	5	25	1,000,000	10,597,430.9	19,712,127.1	51,029,567.2	96,437,207.9	165,409,932.8
	inserted/rows (ratio)	100%	100%	100%	70.6%	98.6%	63.8%	64.3%	27.6%
32 nodes	inserted	5	25	1,000,000	10,220,280.7	17,638,717.8	33,817,404.8	62,474,754.7	88,050,633.3
	inserted/rows (ratio)	100%	100%	100%	68.1%	88.2%	42.3%	41.6%	14.7%

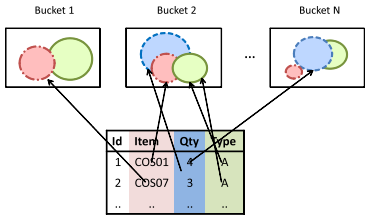


図 5　ハッシュバケットへの格納例
Fig. 5　Illustration of putting tuples into buckets.

方式により 5 つの分割キーに基づいてノードへの割当てが行われる．Lineitem 表では $P = 5$ であるため，$N = 8$ とすると，$P/N \cdot 100$ より R の 62.5%のタプルが割り当てられる．同様に，R のうち，$N = 16$ のとき 31.25%，$N = 32$ のとき 15.625%のタプルが 1 つのバケットに格納される．

実際には，図 5 に示すとおり，1 つのタプルが別の分割キーにより同一のバケットに割り当てられ衝突が発生することがある．簡単のために理想的なハッシュ関数により m 個のタプルを n 個のバケットに振り分ける balls and bins 問題[18] を考えたとき，1 つのバケットに入るタプルの数は $m(m-1)/2n$ である．これはオーダにして $O(m^2/n)$ であり，n が $O(m)$ であるとき衝突回数の期待値は $O(m)$ である．また，バケット数 $n = m^2$ では，$\binom{n}{2}$ の組が $1/m = 1/n^2$ の確率で衝突することがあるため，何らかの衝突が発生する確率 Pr は $Pr \leq \binom{n}{2}\frac{1}{n^2} = \frac{n(n-1)}{2n^2} \leq \frac{1}{2}$ で，1/2 未満である．

こうしたことから，分割キーの数 P よりもノード数 N が十分に大きいことを前提とすると，各ノードに割り当てられるタプル数を $P(R/N)$ で近似できる．ここでの我々の主張は，複数の分割キーによりタプルの割当てを行う場合でもノード数 N に応じて分散が行われるため，ノード数 N を大きくすることで分割効果がある，いい換えればノード数に対するスケーラビリティがあるということである．

実際に TPC-H の各テーブルを Φ ハッシュ分割で割当てを行ったときのタプルの分散を**表 1** に示す．表 1 で，inserted と inserted/rows とする項目は，それぞれノードあたりの割り当てられた平均タプル数と割り当てられたタプル数の R に占める割合である．表 1 で右側のテーブルほどレコード数が大きく，よく分割してしかるべきテーブルである．逆に左の小さなディメンション表については分割効果が少ないものである．Lineitem 表を例にとると，8 ノードで 48.7%，16 ノードで 27.6%，32 ノードで 14.7%となっており，前記の理論値と比較してノード数が少ないほど 1 つのタプルの分散で重複が出ていることが分かる．ノード数を 32 としたときにはほぼ理論値の 15.625%と近い値が出ていることから，TPC-H の Lineitem 表を 32 ノード以上で Φ ハッシュ分割するとき．タプル分散の計算は $P(R/N)$ で近似できる．なお，R/N は属性値偏りに依存する．属性値偏りが小さく，かつタプル数が十分に大きいときに $P(R/N)$ に近い分散となる．

このことからタプル処理数に関して最大 6.81 倍の並列データ処理の効果が期待できる．ただし，MonetDB では可変長データの辞書を持つため，レコード数の増加に対してデータベースサイズが線形に増加しない．たとえば，TPC-H の dbgen SF=100 で作成した約 107 GB のデータを MonetDB 単体にロードするとデータベースサイズは 122 GB であるのに対して，32 台構成の MonetDB/MR ではデータベースサイズは平均約 30 GB である．

4.2　テーブル分割のアルゴリズム

4.1 節では概念的に Φ ハッシュ分割のアイデアを述べたが，ここでは，より厳密にテーブル分割のアルゴリズムを疑似コードで**図 6** に示す．

図 6 では，入力として（CSV 形式の）テーブルデータをとり，各タプルごとに加工を行っ

```
Input : データベースにインポートするテーブル定義 t，およびロード対象の CSV ファイルのレコード lines
Result: 各タプルをノードにマップして，t に定義された外部キーごとに索引を構築する．
foreach line l in lines do
    fields ← l から主キーのフィールド集合を抜き出す;
    distkey ← fields を合成する;
(a) begin primary fragment mapping
        node ← distkey の担当ノードを選択する;
        mapped_node ← t の分割番号;
(b) foreach t_c in t の子テーブルの一覧 do
        fkidx ← t_c の外部キーのための索引;
        begin derived-by-child fragment mapping
            (node,pn) ← IdxLookup(fkidx,distkey);
            mapped_node ← BitOr(mapped_node,pn);
(c) foreach fk in t の外部キーの一覧 do
        begin derived-by-parent fragment mapping
            fields_fk ← l から fk に対応するフィールド集合を抜き出す;
            distkey_fk ← fields_fk を合成する;
            node_fk ← distkey_fk の担当ノードを選択する;
            pn_fk ← fk テーブルの分割番号;
            mapped_node_fk ← BitOr(mapped_node_fk,pn_fk);
(d) begin 親テーブルでの derived-by-child fragment mapping のために索引を構築する
        foreach fk in t の外部キーの一覧 do
            key ← distkey_fk;
            foreach (node, hiddenValue) in mapped do
                if node ≠ node_fk then
                    value ← (node,hiddenValue);
                    BuildIdx(fk,key,value);
    SchedInsertRecord(l,mapped);
    mapped ← ∅;
```

図 6　Φ ハッシュ分割の疑似コード
Fig. 6　Pseudocode of Φ hash partitioning.

たうえで計算ノードに割り当てる．さらに，親テーブルにおける derived-by-child fragmentation による分割のために，外部キーが定義された属性の分散を索引に記録する．なお，ここで用いる索引は，値の重複を許すものとする．外側の foreach により，次に述べるループ内の処理をタプルごとに行う．

(a) のブロックが 4.1 節で述べた primary fragmentation に相当する．ここでは，主キーのハッシュ値に基づきタプルの配置先ノードを決める．*mapped* は配置先ノードをキーとする連想配列であり，4.3 節で述べるようにノードごとにデータの分割に利用したパーティション番号のビットセットが記録される．

(b) のブロックでは，子リレーションが存在する場合に，その分割に基づいて derived-by-child fragmentation を行う．ここで用いる索引は，子リレーションのテーブル分割においてブロック (d) で構築した索引を IdxLookup 関数により参照する．(b) ブロックは参照整合性を保つ目的で，子テーブルが外部キーによって参照しているタプルを子テーブルが存在するノードに割り当てるものである．このとき BitOr 関数においては，ビット演算を用いて node に割り当てる分割番号を設定する．

(c) のブロックでは，外部キーが定義された属性集合を利用した derived-by-parent fragmentation を行う．外部キー属性ごとにノードの割当てを行い，同時に担当レコードに配置されるタプルの分割番号を設定する．

(d) のブロックは，親テーブルでの derived-by-child fragmentation のために索引を構築する．親テーブルの分割時に，(b) のブロックで子テーブルと参照整合性を保つために利用される．BuildIdx 関数により，外部キーおよび属性値をキーとして，タプルが配置されるノードと分割属性の組を記録する．

SchedInsertRecord 関数では，担当ノードとその分割属性を基にタプル配置のスケジューリングを行う．システムで設定する一定数のタプルがスケジューリングされたところで，実際にデータ送信をバックグラウンドで執り行う．

4.3　テーブル分割および問合せ処理の流れ

図 7 に，テーブル分割時および問合せ処理時のモジュール間のデータフローを示す．Φ ハッシュ分割では，それぞれのテーブルで単一の分割属性を用いる代わりに，与えられたデータベーススキーマ（より正確には，データベースカタログの参照整合性制約）から**データ分割指示部**が主キー情報とテーブル間の参照関係を考慮して分割属性候補を導出し，そこから得られる分割キーすべてを用いてタプルを 1 つ以上のノードへ割り当てる．そして，**データ配分先決定部**がタプルのノードへの割当てを行うときに，**キー付与部**がテーブルがどのような属性を用いて分割されたかという情報を分割されるタプルごとに追加属性の分割キーとして差し込む．このタプルに差し込まれたメタデータは，問合せ処理時にタプルが与えられた問合せに対して必要か否かを判断するために利用される．問合せを評価するときには，**問合せ加工部**が，与えられた問合せの期待する分割属性に基づいて分割キーを用いた選

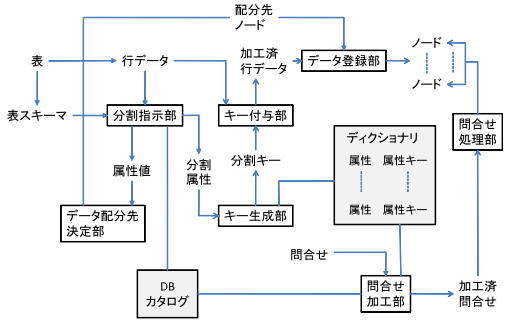


図 7　テーブル分割時および問合せ処理時のモジュール間のデータフロー
Fig. 7　Data flow among modules when partitioning tables and processing queries.

択演算を問合せに加えたうえで，加工済問合せを**問合せ処理部**が評価する．

分割キーの付与

4.1 節の各フラグメント化方法は，いずれかのテーブルの主キーに基づく．タプルに対する分割キーの付与は，**キー生成部**が誘導した主キーに対して一意なパーティション番号（partition number）を設定し，そのパーティション番号を**キー付与部**が利用することで行われる．パーティション番号は，主キーごとに 1 から昇順に連番で割り振っていく．このとき，パーティション番号の付与する主キーの選択順序は問わない．TPC-H のスキーマにパーティション番号を割り振った一例が図 4 である．

キー付与部では，パーティション番号を n とすると，4.1 節でパーティションの実際の値として 2^{n-1}（ただし $n \geq 1$）が利用される．パーティション番号をそのまま利用するのではなくビットセットを利用するのは，追加する分割属性において，分割属性値の各ビットでタプルがどの主キーに基づいて分割が行われたかを識別するためである．TPC-H のスキーマ構成ではテーブルが 8 つであるため，分割属性に必要なサイズは 1 バイト（8 ビット）である．つまり，キー付与部では比較的小さな分割情報をタプルごとにメタデータとして付与する．

5. MonetDB/MR のシステム構成

本章では，Φ ハッシュ分割を利用した無共有型並列データベースシステム MonetDB/MR の設計を述べる．MonetDB/MR の設計目標は，3 章にあげた既存のデータ分割手法の問題点をすべて解決した並列データベースシステムを実現することである．

MonetDB/MR では計算ノードごとに設置する列指向データベース MonetDB の各インスタンスを束ねる．それぞれのノードが経路表を管理し，ワーカーノードへタスクのルーティングを行う．なお，参加ノードの管理は分散ハッシュ表の構成技術の一種である Consistent Hash 法[19] による．

各データベースノードへは，テーブルの水平分割（Horizontal partitioning）によりタプルが配置される．そして，各データベースノードでは，カラムごとにデータが格納される．テーブルの水平分割と垂直分割の組合せはハイブリッド分割（Hybrid partitioning[16]）を想起させる．MonetDB/MR は，テーブルの水平分割に 4 章で述べた Φ ハッシュ分割を用いる．問合せはシステムによって分解，加工され，MapReduce 型のタスク実行を行うシステムによって各計算機で並列処理される．なお，1 つの M/R のプロセスは複数の mapper ノードと単一の reducer ノードを用いて並列処理される．MonetDB/MR への Φ ハッシュ分割の導入の狙いは，データの移動をなくして shuffle のオーバヘッド（ディスク I/O やネットワークの負荷）を下げることだけでなく，データの移動にともなうメモリからのデータ入出力を避け，各データベースで問合せを極力インメモリで処理することにある．

さらに，MapReduce の並列データベースに対する利点である耐障害性と計算機間の負荷が均等でない場合を考慮した負荷分散に対応する．負荷分散のための複製の管理は，Chained declustering[20] に基づく．

5.1　システムのアーキテクチャ

図 8 に MonetDB/MR のアーキテクチャを示す．MonetDB/MR は汎用の MapReduce に基づいたジョブ管理システムを基礎としている．SQL 問合せやテーブルのデータ分割に際しては，該当するジョブ（図 8 の SQL Job や Partitioning Job）がシステムに投入されて実行される．Job Manager は投入されたジョブをより細かい単位であるタスクに分割し，そのタスクをノードへ割り当てる．ここが MapReduce に由来するタスク処理方式に基づく．Job Manager はノード故障時へのタスクの再割当てや投機的実行といった MapReduce に特徴的な機能[1] を執り行う．

MonetDB/MR ではノードが計算機ごとに設置され，各ノードが 1 つの列指向データベース MonetDB インスタンスを管理する（Node 部）．単一障害点の問題を避けるため，MapReduce で一般的なマスタスレーブ設計ではなく，マルチマスタ構成をとる．それぞれのノードが経路表を管理し，与えられたタスクをワーカノードへルーティングする（Lookup Service 部）．MonetDB/MR のクライアントには，主に CPU 負荷，他にも I/O 負荷を考慮して，負荷の低いノードをマスタとして選んでジョブを投入する機能を設けている．

ノードに割り当てられたタスクは，担当ノードでタスク処理器（Task Processor 部）で

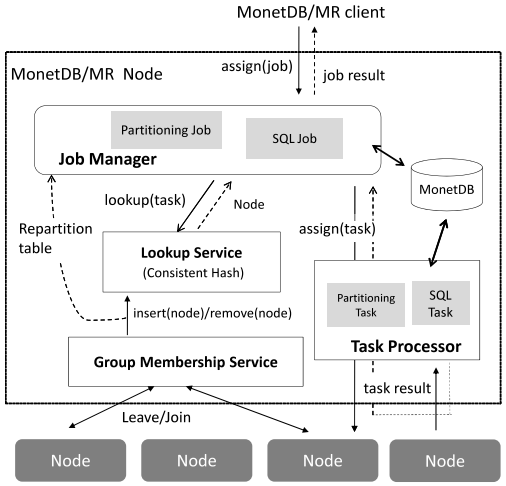


図 8　MonetDB/MR のアーキテクチャ
Fig. 8　Architecture of MonetDB/MR.

処理される．Group Membership Service 部は Lookup Service と連携して参加ノードを管理する．グループ間通信[21] で培われた技術を用いて，ノード間で通信して経路表を管理する．たとえば，ノードの参加/離脱にともなってネットワークに流れるメッセージ数は IP Multicast を利用した場合は 1 である．

なお，経路表の管理には分散ハッシュ表の構成技術の 1 つである Consistent Hash 法[19] を利用する．MapReduce 型の実行では各 map 関数の処理時間の偏りを小さくすることが肝であるが，これはタプル配置の偏りに依存する．MonetDB/MR では，Virtual processor partitioning[22] と同様に 1 つのハッシュバケットの担当領域に複数の値域を割り当てることで，バケット間のデータ配置数の偏りを低減させる．この偏り防止技術によって，6.1 節で述べる TPC-H SF=100 による実験では，タプル配置の偏りを最大で 6.6%と小さく抑えており，map 処理で生じる実行時間の偏りは平均約 13%と map タスクの投機的実行を必要としない結果を得ている．

5.1.1　並列問合せ処理

ユーザが発行した SQL 問合せは，図 7 の問合せ加工部で map 問合せと reduce 問合せに加工され，MapReduce 型のタスク実行を行うシステムによって各計算機で並列処理される．MonetDB/MR の M/R イテレーションは複数のノードで並列に map 問合せを評価し，単

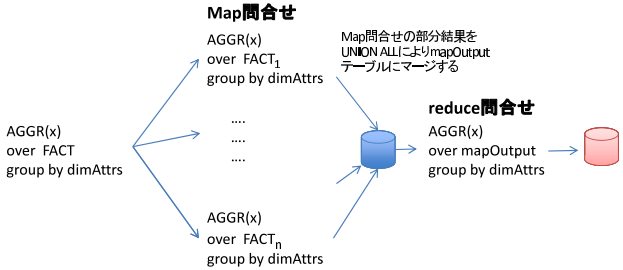


図 9　典型的な分析問合せ処理の処理例
Fig. 9　An example of a typical analytical query processing.

一の reducer ノードでそれらの問合せ結果をマージしてから最終的に reduce 問合せを実行し結果を得る．**図 9** に典型的な分析問合せの処理例を示す．この問合せは，ディメンション表の属性（dimAttrs）に基づいて FACT の集約処理 AGGR(X) を行う例である．このような並列化は AGGR が結合法則と交換法則を満たす（commutative かつ associative である）ときに可能である．SQL の COUNT 関数や SUM 関数はこれらを満たすが，AVERAGE 関数や STDDEV 関数はこれらを満たさない．しかし，AVERAGE は SUM と COUNT，STDDEV については SUM と SUM_OF_SQUARES と COUNT に置き換えることで並列に集約処理することができる．多くの集約関数は SUM，SUM_OF_SQUARES，COUNT，MAX，MIN などのプリミティブを利用して合成することができる．

さらに，MonetDB/MR は MapReduce の並列データベースに対する利点である耐障害性と計算機間の負荷が均等でない場合を考慮したタスクレベルでの負荷分散に対応する．これらは Osplay[3] と同様に，複製に基づいた負荷分散手法である Chained Declustering[20] に基づいて行われる．map ノードでの問合せ結果が設定された閾値を超えても返ってこない場合には，複製が存在するノードへ遅延している map 問合せの投機的実行が指示される．

5.1.2　問合せへの検索キーの付与方法

図 7 の問合せの加工処理は，問合せの結合グラフ[16] と分割属性情報に基づいて行われる．ここでは，TPC-H の Q3（付録 A.1 参照のこと）を例として，検索キーの map 問合せへの付与方法を述べる．

Q3 は，図 4 に示す Lineitem 表と Orders 表が orderkey によって，Orders 表と Customer 表が custkey によって結合される問合せである．このとき，Φ ハッシュ分割では Lineitem 表，Orders 表，Customer 表がそれぞれ orderkey の derived-by-parent fragmentation，pri-

mary fragmentation，derived-by-child fragmentation によって分割されていることを期待する．Lineitem 表と Orders 表を orderkey 属性により結合した中間結果表の Orders.custkey 属性は，参照整合性を満たすために対応する Customers.custkey が同一ノードにあることを期待する．

そこで，上記の意図を反映するために Q3 の WHERE 句に次の選択演算を加える．

- (lineitem._hidden & 2) = 2 -- partition by orderkey
- and (orders._hidden & 2) = 2 -- partition by orderkey
- and (customer._hidden & 2) = 2 -- partition by orderkey

ここで，_hidden は分割属性を示す追加フィールドである．検索キーには該当する分割番号値（図 4 にあるように，orderkey の分割番号値は 2）を利用する．

以上のように，問合せの結合グラフと分割属性情報より問合せ式の加工を行う．選択演算を加えることでの問合せ処理性能への影響も考えられるが，operator-at-a-time 評価戦略[23] や問合せに応じたタプル再編成と索引構築戦略[24] をとる MonetDB はこのような選択演算を得意とする．

5.2　Consistent Hash 法に基づくルーティング

MonetDB/MR では，ハッシュ分割などにおけるタプル配置先の決定，およびクラスタに参加するノードに基づく経路管理に Consistent Hash 法[19] を利用する．

N 台の計算機を利用する負荷分散における基本的なハッシュ分割手法では，計算ノードに $[0, N)$ の番号付けを行い，オブジェクト（ないしはタスク）をキー key として $hash(key)$ **mod** N 番目のノードにオブジェクトをマッピングする．

説明のための具体例として，N 台の計算機で（オブジェクト）キャッシュを構成するシステムを用いる．ここで，データの配置を決定する key はオブジェクトであり，タスクは key を鍵とするキャッシュへの操作である．

ハッシュに基づく負荷分散では，何らかの都合でマシンの追加や削除が発生し，N の値が変化するたびにすべてのオブジェクトは適切なデータ配置を求めるためにハッシュし直す必要があり，計算ノードの故障やネットワークの分断，あるいはノードの追加が発生するようなノード数が動的に変化する（計算機クラスタ）環境に適さない．

これに対して，ノード数の動的な変化に対しても有効に機能するのが Consistent Hash 法である．Consistent Hash では，まず ID 空間 S（我々の実装では 2^{64}）を想定する．計算機ノードの識別子と鍵は，S に属するものとする．Consistent Hash に参加する計算機ノードは，たとえば IP アドレスとポート番号の組を鍵にして，S 上に写像される．一方，オブジェクト α は，その鍵 K_α（$K_\alpha \in S$）から次に大きな鍵を持つノード（**successor**）に分散される[*1]．

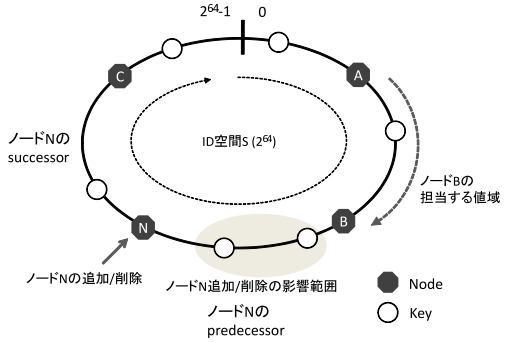


図 10　Consistent Hash 法における Key/Node マッピング
Fig. 10　Key/Node mapping of consistent hashing.

表 2　ノードの追加直後の最大キャッシュヒット率
Table 2　Cache hit ratio immediately after adding nodes.

既存台数（m）	追加台数（n）	最大キャッシュヒット率 (%)
5	1	83.3
6	2	75
7	3	70
61	3	95.3

Consistent Hash におけるオブジェクトの割当てアルゴリズムの概要を**図 10** に示す．図 10 から明らかなようにノードの参加や離脱があっても，Consistent Hash で管理するオブジェクトと格納先ノード（バケット）の関係への影響範囲は，参加・離脱対象ノードの鍵から次に小さな鍵を持つノード（**predecessor**）に限られる．つまり，計算機 m 台で構成するキャッシュに計算機を n 台追加した直後においても，理想的なキャッシュヒット率は $(1 - n/(n+m)) \times 100$ となる．**表 2** に例示するように，十分な初期台数が得られれば高いキャッシュヒット率が期待できる．

また，Consistent Hash 法では参加・離脱するホストに格納される，あるいは格納されていたデータを移動することでキャッシュヒット率を高く保つことが可能である．図 10 か

[*1] Consistent Hash で管理されるのは計算機ノードのみであり，経路表である．ハッシュに格納されるオブジェクトを管理するのはオーバレイネットワーク上の分散ハッシュ表などのルーティング層から見た上位層である．

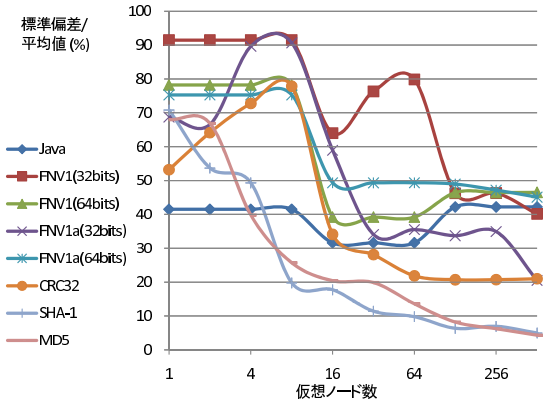
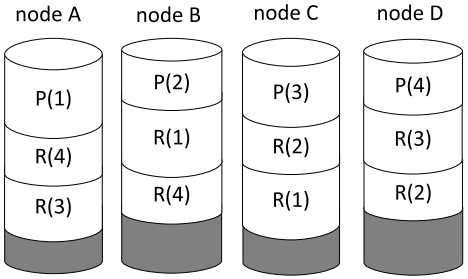


図 11　仮想ノード数とハッシュ関数の分散の関係
Fig. 11　Relationship between the number of virtual nodes and hashing functions.

ら明らかなように，ノードの追加時には predecessor から要素を受け取り，正常離脱時には successor に要素を移譲すればよい．

ハッシュ値の偏りにより各ノードに割り当てる値域に均等に（格納・計算）負荷が分布されないことも考えられるが，Consistent Hash では仮想ノードという概念を設けることで，この問題を解決している[19]．ノードを ID 空間 S の単一の点に写像するのに加えて，S 上に複数個の点を複製する．各仮想ノードが保持すべきオブジェクトは幾何分布に従うが，ハッシュ値の偏りが生じた場合に対しても各ノードが担当する値域の偏りを減少させることで負荷の集中を抑えることができる．

提案手法で用いる Consisent Hash において，仮想ノードを追加することの効果を擬態した結果を**図 11** に示す．10,000 個のオブジェクトを 10 の（キャッシュ）ノードに保存する場合について，複製数の変化とハッシュ関数の実装がもたらす影響を測る．ハッシュ関数には，一般的なハッシュ関数である FNV ハッシュ，CRC32，SHA-1，MD5，あるいは Java の標準クラスの hashCode 関数を利用し，各アルゴリズムによるハッシュ値の偏りを示す．図 11 で，横軸に示すのが仮想ノード数であり，縦軸に示すのが各ノードが担当するオブジェクト数の相対標準偏差である．この実験結果から，仮想ノード数が小さいと負荷のバランスが十分に行われないが，SHA-1 を利用して仮想ノードを 64 以上用意すれば，負荷の偏りを 10%以下に保つことができ妥当な負荷分散が可能といえる．提案手法では，このような優れた負荷分散の特性を持つ Consistent Hash 法を負荷分散の基盤として利用する．



Primary/Replica Data placement

	Data 1	Data 2	Data 3	Data 4
Primary	node A	node B	node C	node D
Replica 1	node B	node C	node D	node A
Replica 2	node C	node D	node A	node B

図 12　Chained declustering
Fig. 12　Chained declustering.

5.3　Chained declustering に基づくノード障害対応と負荷分散

MonetDB/MR では，ノード障害対応と負荷分散のために Chained declustering[20] に基づいてレプリカの管理を行う．レプリカデータベースはプライマリデータベースで map 処理が失敗したときの map 再実行や，map 処理が極端に時間がかかっているときに map 処理を投機実行するのに利用される．**図 12** に示すように，レプリカ数 2 では正/副/副の 3 つのデータベースインスタンスを 1 つの物理ノードで管理する．たとえば，ノード B が故障した際にはノード C で map 処理の再実行が行われ，ノード C の map 処理で極端に時間がかかっているときには，ノード D で map 処理の投機実行が行われる．Chained declustering では，このように複製配置チェーンを通じて負荷が伝播されることで，全体のロードが平坦化される．なお，MonetDB/MR ではノード故障や負荷の偏りによる投機実行が起きない限りは，レプリカ領域にアクセスしないため副作用は存在しない．

6.　評 価 実 験

提案したシステムの性能を MapReduce に基づくデータウェアハウスシステム Hive[25] および単体の MonetDB と比較する．Hive はオープンソースの MapReduce 実装である

表 3　実験環境のハードウェア構成
Table 3　Hardware settings of the experimental environment.

	large, 1 node	normal, 32 nodes
CPU	X5550@2.67 GHz	E5520@2.27 GHz
CPU cache	8 MB	8 MB
Socket	2	2
Core	4	4
Hyper Threading	2	2
Total threads	16	16
Memory	48 GB	24 GB
Disk	SATA 7200 rpm (Hardware RAID 1)	SATA 7200 rpm
File System	ext3 (LVM)	ext3
Ethernet	1 Gbps	1 Gbps

Hadoop 上に構築された SQL と類似の問合せ言語 HiveQL をサポートするデータウェアハウスシステムである．これら競合するデータウェアハウスソリューションとの比較により，提案システムに対してデータ数に対するスケーラビリティ，ノード数に対するスケーラビリティ，ノードの負荷を増減させた性能の観点からの評価を与える．評価のワークロードには，データウェアハウスの性能指標として広く支持されている TPC-H [15] を利用する．

実 験 環 境

実験環境に利用した計算機クラスタのハードウェア構成は**表 3** のとおりである．他の 32 台のマシン（normal インスタンス）と比べて，1 台のマシン（large インスタンス）はメモリが 48 GB と豊富で CPU のクロック数もより高いノードである．ネットワークスイッチには，48 ポートの HP ProCurve Switch 2810-48G を利用した．33 ノードは 1 つのスイッチでつながっており，スイッチング容量の理論値は 96 Gbps である．ノンブロッキングスイッチであり，表 3 の環境のスイッチとして容量に不足はない．

各ノードのソフトウェア構成は**表 4** に示すとおりである．提案システムでは，並列データベースを構成する各ノードで列指向データベースである MonetDB [8] を利用する．MonetDB は主記憶データベースとして設計されており，データは mmap システムコールを利用したメモリマップドファイル形式で OS の仮想記憶によって管理される．

競合ソフトウェアとして利用した Hadoop のパッケージは Cloudera 社が配布するもの[26]を利用し，TPC-H を評価するにあたって hadoop の設定は文献 27) で推奨される設定値に従った．このとき，MapReduce の Map 関数の出力は圧縮解凍速度に優れる LZO 方式により圧縮されてディスクに格納される．文献 27) の設定で Hive の初期状態でのデータ配置は，128 MB ごとにブロック分割されて Hadoop 分散ファイルシステム（HDFS）に格納される．なお，データの複製数は MonetDB/MR と Hive でともに 2 とした．MonetDB/MR では，5.3 節で述べたように chained-declustering [20] に基づいてデータベースインスタンスレベルで複製を管理する．

表 4　実験環境のソフトウェア構成
Table 4　Software settings of the experimental environment.

Linux	CentOS 5.4
Kernel	2.6.18/x86_64
Java	Sun JDK 1.6.0_20 (64-bit)
Hadoop	0.20.2+228-1 (CDH3)
Hive	0.5.0+20-1 (CDH3)
Lzo lib	2.03-3
MonetDB	Feb 2010 SP2

6.1　競合システムとの性能比較

ここでは，表 3 の normal インスタンス 32 ノードを実験環境として，TPC-H のスケールファクタを 100（約 107 GB のデータ）により競合システムとの比較を行う[⋆1]．MonetDB/MR ではマスタノードは不要であるが，Hadoop では分散ファイルシステム HDFS のメタデータを管理する namenode をワーカノードと別の物理ノードに用意することが推奨されているため，表 3 の large インスタンスを namenode として用い，normal インスタンス 32 ノードを Datanode および Tasktracker を実行するワーカノードとした．競合システムとしては，MapReduce に基づくデータウェアハウスシステムである Hive [25] と単一ノード構成の MonetDB（MonetDB/Single）を利用した．

Hive との比較

Hive は HiveQL という SQL と類似の問合せを 1 以上の MapReduce のジョブに分けて実行する．問合せ処理にあたって，並列データベースと同様にあらかじめ行った分割が適当でなければ，動的なハッシュ分割が行われる[28]．たとえば，SQL の group-by 処理や等結合演算が MapReduce によって処理される．動作原理として動的なテーブル再分散を行う並列データベースに近いシステムであり，提案システムとのデータ分割手法の違いが性能に影響する．

MonetDB/MR の狙いは，MonetDB/Single が 1 台のマシンで処理できない上限を超え

⋆1 単一ノード構成の MonetDB で管理可能なデータ量，そしてローカルディスク容量の制約から SF=100 を利用した．

表 5　各試行の実行時間が総実行時間に占める割合
Table 5　Percentage of each execution time in the total execution time.

	#1 try	#2 try	#3 try
MonetDB/MR	79.0%	10.5%	10.4%
Hive	33.4%	33.2%	33.4%

たときに，ノード数とデータ量に対してスケーラビリティを得ることにある．そこで本節では，Hive に加え，対象データの約半分のメモリ容量 48 GB の large インスタンス上で MonetDB/Single を運用した場合との性能比較を与えた．MonetDB は，マルチコアプロセッサを利用した主記憶上での問合せ処理性能に特に優れたシステムである[⋆1]．MonetDB/Single との比較により，スケールアップを行った全共有型（shared-everything）の単一計算機構成とスケールアウトを行った無共有型（shared-nothing）のクラスタ構成で，それぞれデータベースを構成した際の優劣を示す．

問合せには TPC-H で用意されている 22 個の問合せを利用した．問合せごとに実行時間を比較するため，問合せごとに OS のバッファ領域をクリアしてデータベースを再起動したのち，3 回連続で問合せを試行した．実際に 3 回の試行それぞれの問合せ実行時間が，その合計時間に占める割合は**表 5** のとおりである．MonetDB は mmap や malloc システムコールにより仮想記憶を活用する主記憶データベースであるため，最初の試行はすべてのデータがメモリ上に存在しない最悪の場合の性能を示すものである．

表 6 が TPC-H SF=100 を用いた MonetDB/MR，MonetDB/Single，Hive の性能比較結果である．表 5 に示したように，2 回目と 3 回目の試行間の差は軽微で計測誤差の範囲と考えられるため，MonetDB/MR（cached）には 2 回目の試行と 3 回目の試行の平均をとったうえでデータがキャッシュされた状態での性能を示した．一回目の試行は MonetDB/MR（no-cache）に示した．同様に，MonetDB/Single についてもキャッシュなしの試行（no-cache）とキャッシュされた状態での試行（cached）を示した．表 6 で MonetDB/MR（no-cache）と MonetDB/MR（cached）の項目の下添え字は，Hive の実行時間を単位時間とするもので，それぞれ Hive に対する性能比を示す．たとえば，問合せ Q1 で MonetDB/MR は，Hive に対して 8.1 倍（no-cache）〜217.8 倍（cached）の性能がある．cw Hive 行の値は，Q1–Q22 の問合せの実行時間の総計を Hive の場合を単位（=1）として示したものであり，

⋆1 アクセス対象のデータの大部分が主記憶に載って I/O バウンドにならない，かつ利用可能プロセッサ数が十分にあって問合せが CPU バウンドにならないときは，ノード数の増加による性能向上は大きく望めない．並列問合せ処理では，データ交換や最終的なマージ処理などの追加処理によって逆に性能が劣化することもある．

表 6　競合システムとの性能比較
Table 6　Performance comparison to competing systems.

TPC-H の 22 の問合せの実行時間（単位は秒）を示す．
MonetDB/MR の実行時間については Hive に対する性能比を下添え字で示す．

Query	MonetDB/MR (no-cache)	MonetDB/MR (cached)	MonetDB/Single large (no-cache)	MonetDB/Single large (cached)	Hive
Q1	159.2 $_{8.1}$	5.9 $_{217.8}$	445.8	947.2	1,283.9
Q2	41.2 $_{7.6}$	5.8 $_{53.7}$	31.5	3.0	311.6
Q3	97.6 $_{3.0}$	14.3 $_{20.7}$	372.7	21.7	295.3
Q4	78.5 $_{2.9}$	5.5 $_{41.4}$	186.4	4.1	228.5
Q5	85.6 $_{5.1}$	17.4 $_{25.3}$	399.6	9.5	439.9
Q6	111.1 $_{0.9}$	4.6 $_{22.6}$	258.8	1.8	104.1
Q7	158.7 $_{6.0}$	38.3 $_{24.8}$	317.8	16.2	950.3
Q8	141.6 $_{3.5}$	11.3 $_{43.8}$	448.4	8.3	495.5
Q9	204.2 $_{4.2}$	31.8 $_{27.0}$	386.2	25.8	858.9
Q10	679.1 $_{0.7}$	137.0 $_{3.5}$	286.8	22.1	480.6
Q11	59.4 $_{4.2}$	5.0 $_{49.2}$	37.0	1.5	248.3
Q12	118.8 $_{1.4}$	12.0 $_{13.7}$	228.1	5.6	165.5
Q13	197.7 $_{1.7}$	42.5 $_{7.8}$	260.7	108.8	330.5
Q14	79.5 $_{1.6}$	6.0 $_{21.3}$	260.9	3.6	127.5
Q15	127.2 $_{1.7}$	4.8 $_{45.3}$	274.6	6.9	216.8
Q16	36.6 $_{9.3}$	17.2 $_{19.7}$	45.7	25.4	338.9
Q17	92.8 $_{3.6}$	6.8 $_{49.2}$	236.0	24.8	336.0
Q18	188.8 $_{2.5}$	67.3 $_{7.0}$	370.0	447.8	471.8
Q19	82.2 $_{3.0}$	10.4 $_{23.8}$	527.3	138.2	249.0
Q20	135.9 $_{3.8}$	2.9 $_{182.0}$	236.7	3.4	519.5
Q21	133.1 $_{6.9}$	28.6 $_{32.1}$	227.4	21.7	916.3
Q22	92.0 $_{3.9}$	13.8 $_{25.9}$	39.4	8.5	357.4
cw Hive	3.1×	19.9×	1.7×	5.2×	

問合せ総処理時間について Hive に対する性能向上比を示すものである．

表 6 の cw Hive 行に示すように，提案システムでは MapReduce（M/R）に基づく Hive に対して，総実行時間の比で 3.1 倍〜19.9 倍の性能が得られた．この性能差は，提案システムがデータの再分散を必要とせずに，多くの場面[⋆2]で 1 つの M/R のイテレーションで問合せを評価できたことによる．

この大きな性能差から，TPC-H のようなデータウェアハウス処理に対して M/R に基づく

⋆2 Q11 と Q22 については，問合せを 1 つの M/R のイテレーションでは処理することができないため，2 つの M/R のイテレーションで問合せの評価を行う．集約演算を含む副問合せ中を第 1 の M/R のイテレーションで評価する．

システムに対する提案手法の明らかな優越を確認した．M/R に基づくシステムと並列データベースでデータをキャッシュする階層の違いがあることが性能差の要因の 1 つである．関係データベースのストレージ層は独自のページ管理アルゴリズムにより 1 度アクセスされたページをキャッシュするが，分散ファイルシステムではファイルブロックは明示的にキャッシュされない[⋆1]．また，MapReduce では各 map/reduce 関数の処理が完全にシステム側で制御されないために，利用可能なメモリ領域を最大限まで活用することが難しいという問題がある．たとえば，Hadoop は各 map/reduce プロセスを独立した JVM プロセスとして立ち上げるため，複数の map タスクを 1 度に実行するときに与えるメモリ容量は同時実行されるプロセスによって制限される．

特に Hive ではいずれの試行も実行時間の占める割合の標準偏差が，平均 0.77%最大 3.7%と小さく，いずれの試行もほぼ同等の実行時間を要していた．データウェアハウスの運用では，定期的に分析クエリが実行されるため，効率的なデータのキャッシュが重要である．メモリの大容量化と低価格化によって各ノードが 16 GB 超の大容量メモリを搭載することも一般的になってきているため，M/R によるバッチ処理を分析用途に利用するには，データのキャッシュとメモリ利用効率の改善が課題といえる．

MonetDB 単体との比較

TPC-H SF=100 について前述の 3 回の試行の平均実行時間を用い，MonetDB/MR，MonetDB/Single（normal），MonetDB/Single（large）の性能を比較した．ここで，MonetDB/Single（normal）と MonetDB/Single（large）は，それぞれ，MonetDB 単体を表 3 の normal インスタンスと large インスタンスで評価したものである．以降，特に言及しない限り，MonetDB/MR と MonetDB/Single の比較には，この 3 回の試行の平均を用いる．

図 13 は，MonetDB/MR の処理時間を単位時間（=1）として，MonetDB/Single（normal）と MonetDB/Single（large）の処理時間を log スケールで示したものである．

問合せ Q1 は，主記憶のサイズを超える 76 GB の Lineitem 表のすべてを読み出し集約演算を行う問合せであるため，ノード数を足すことによるディスク読み込みの負荷分散効果が顕著に出る問合せである．図 13 に示すように，32 台構成の MonetDB/MR は 1 台の MonetDB/Single（normal）に対して 51.89 倍の性能を示しており台数効果が線形以上に現れている．すでに表 6 に示したように，MonetDB/MR では Q1 の 1 度目の試行（no-cache）

⋆1 MonetDB の場合は必要な列だけが OS の仮想記憶で管理される．Linux では空きメモリは適応的にディスクキャッシュとして利用される．

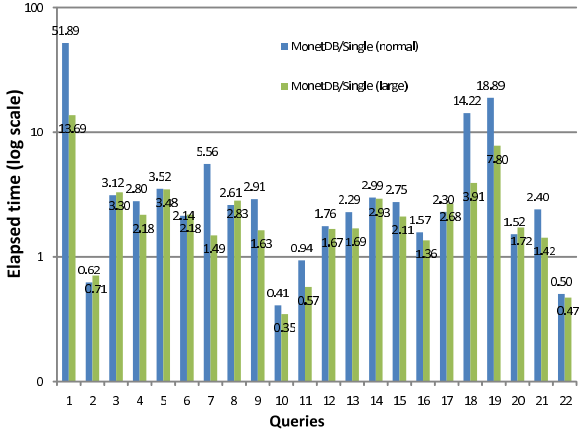


図 13　normal/large インスタンスでの TPC-H SF=100 の評価
Fig. 13　Evaluation of TPC-H SF=100 on the normal/large instance.

は 159.2 秒かかるが，2 度目以降の試行（cached）ではキャッシュ効果により 5.9 秒で処理している．1 台の MonetDB/Single（normal）構成では 2 度目の試行でも 2,585.31 秒（3 回の試行の平均は 2,956.92 秒で MonetDB/MR の平均 56.99 秒の 51.89 倍）かかる 76 GB の集約演算を 5.9 秒で処理できることは MonetDB/MR の顕著な利点である．

一方で，MonetDB/Single（normal）に対して MonetDB/MR は，図 13 の 22 種類の問合せの値を平均して 5.8 倍（最大は Q1 の 51.89 倍，最小は Q10 の 0.41 倍）の性能であり，ノード数に対して線形の性能は得られていない．台数効果が線形に現れていない理由としては，図 13 に示すように，SF=100 のデータサイズはメモリ容量 24 GB の normal インスタンスでも Q1，Q18，Q19 を除いて十分に処理可能なためである．図 13 に現れているように少なくとも TPC-H の SF=100 では，明らかに物理メモリが性能を決める主要な要素になっているのは，Q1，Q18，Q19 である．これらの問合せについては，表 6 においても MonetDB/Single（large）と MonetDB/MR の間で台数効果が確認できる．より大きいスケールファクタでは物理メモリが性能を決める主要な要素になると考えられ，MonetDB/MR の適用範囲となる．

6.1.1　MonetDB/MR の処理時間の内訳

図 14 に SF=100 での MonetDB/MR における各問合せの実行時間の内訳を示し，その性能の裏付けを行う．図 13 で Q2，Q10，Q11，Q22 についての MonetDB/MR の性能が

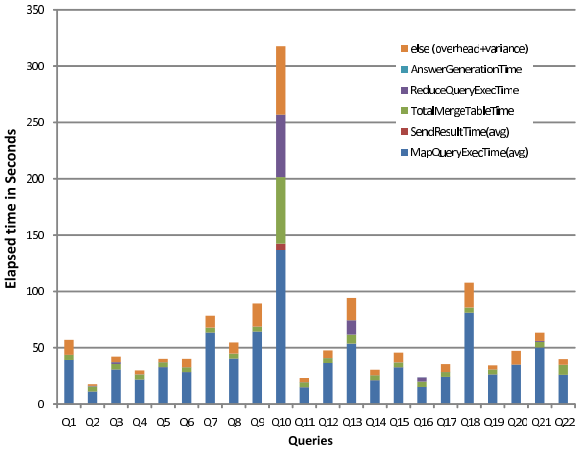


図 14　TPC-H SF=100 の処理時間の内訳（MonetDB/MR）
Fig. 14　Breakdown of the processing time of TPC-H SF=100 (MonetDB/MR).

単一ノード構成の MonetDB/Single（normal）に対して落ちる原因を述べる．

MapQueryExecTime（avg）項目は M/R の map フェーズで問合せ処理に要した時間の平均である．SendResultTime（avg）項目は map フェーズで各ノードから reducer ノードへ結果を送信するのに要した時間の平均である．Map フェーズと後続のマージ処理は並行に動作するため，MapQueryExecTime と SendResultTime には最大値ではなく平均値を利用している．TotalMergeTableTime 項目は各 mapper ノードの問合せ結果から中間結果表を作るのに要した時間である．ReduceQueryExecTime 項目は最終的な問合せ結果を生成するために，中間結果表に対して問合せを処理するのに要した時間であり，AnswerGenerationTime 項目は最終的にファイルに結果を出力するのに要した時間である．else 項目はその他の MonetDB/MR のオーバヘッドと，MapQueryExecTime と SendResultTime それぞれの最大処理時間と平均処理時間の差を含む．else 項目は主に map 関数の処理時間の偏差を示す指標である．

Q10 は map フェーズで 7 つの項目による group-by 処理を含むため map 関数の出力が大きく，そのため偏差も大きく出る問合せである．22 個の問合せ中で最大の 4,718,454 レコード（約 810 MB）のマージ処理が行われる．

Q10 で merge 処理に多くの時間を要しているのは，現在の MonetDB のバージョンの制限による．本来は merge 処理は並行処理が可能であるが，現在の MonetDB の同時実行制御にある問題を避けるために逐次的にマージ処理が行われる[⋆1]．ノード数に応じた逐次処理が行われるため，map フェーズの出力が大きい場合に merge 時間が問題となっている．各 merge に要する問合せ処理時間自体は 2–3 秒程度であるが，排他制御により最大 20 秒近く merge に要することがあった．このことが Q2，Q10，Q11，Q22 で性能が落ちる原因の 1 つとなっている．このマージ処理の逐次処理される部分のボトルネックは，物理的なマシンのディスク構成（JBOD や SSD の活用）も含めテーブルごとにディスクを分けるなどの並列書き込み処理に耐えうるテーブルスペースの設計を行ったうえで，MonetDB の同時実行制御の問題を修復することで解決できる．

表 6 の Q10 で，MonetDB/MR の Hive に対する優勢が見られなかったのは，単一の reducer を用いる現在の MonetDB/MR の実行モデルに向かなかったためである．5 章で述べたように，MonetDB/MR の M/R イテレーションは複数の mapper ノードで並列に SQL 問合せを評価し，単一の reducer ノードでそれらの問合せ結果をマージしてから最終的な SQL 問合せを実行し結果を得る．一方で Hive では，group-by でグループ化を行う項目によって，データの動的な再分散を行い reduce 処理を複数のノードで並列に行っている．現在の MonetDB/MR の単一 reducer モデルは最終的な問合せ結果や reduce 処理への入力が十分に小さいことを想定しているが，この想定に適合しない Q10 の性能を上げるためには，ノード内並列の部分集約処理（partial aggregation [29]）を行うか，複数の reducer を用いた reduce 処理を行うことが必要である．特に，MapReduce の reduce 関数はリスト $[x_1, x_2, \cdots, x_n]$ の各要素を結合的な二項演算子 $\oplus$ で結合して 1 つの値を返すような処理であるため，**reduce** $(\oplus)$ $[x_1, x_2, \cdots, x_n]$ の並列計算コストは，操作が結合則を満たすとすれば，階層的に並列実行することで $O(\log(n))$ となる．Hadoop では階層的な reduce の実行はサポートされていないが，Q10 のような reduce 処理のコストの高い問合せで，階層的な reduce 実行は性能向上のための選択肢の 1 つである．

6.1.2　Φ ハッシュ分割の効果と DB キャッシュ効果

MonetDB/MR におけるキャッシュ効果を，**図 15** に示す．no-cache は表 6 の 32 ノード構成の TPC-H SF=100 の評価の 1 回目の試行で，OS のページキャッシュの消去と DB の再起動により，問合せを超えたキャッシュの影響を排除したものである．cached は，2 回目と 3

⋆1 MonetDB はテーブルレベルに Serializable な楽観的な同時実行制御を行う．各 map 関数の出力結果はそれぞれ別のテーブルに COPY INTO コマンドによって投入され，その各 map 出力テーブルをマージする view が中間結果表として定義される．

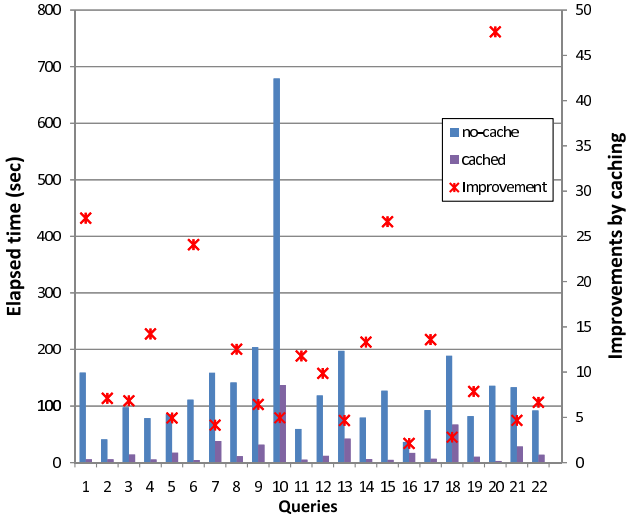


図 15　MonetDB/MR におけるキャッシュ効果
Fig. 15　Effects of caching on MonetDB/MR.

回目の試行の平均をとったものでキャッシュの効果がある場合の性能である．improvement は，cached の実行時間を no-cache の実行時間で割った値である．

一方で，MonetDB/MR における Φ ハッシュ分割自体の効果をみるため，**図 16** に Φ ハッシュ分割の効果をキャッシュの影響を排除した MonetDB/MR（no-cache）と Hive の性能比較により示す．データの再分散を必要とせずに多くの場面で 1 つの M/R のイテレーションで問合せを評価できたことによる効果をみるために，**表 7** に示す Hive における Shuffle データ量と HDFS への書き込み量を指標に用いる．これらの指標はデータの再分散，および M/R のイテレーション数にそれぞれ影響を受けた因子である．Hadoop において HDFS への書き込みは Reduce の出力において行われるため，MapReduce のイテレーション数と HDFS 書き込み量には関係がある．

図 16 の散布図が，MonetDB/MR による Hive に対する性能とこれらの 2 つの指標の相関を示すものである．散布図の各点が TPC-H の 22 種類の問合せ各々に対応する．x 軸方向は Hive に対する MonetDB/MR（no-cache）の性能比を示し，y 軸方向に Shuffle データ量と HDFS への書き込み量を利用することで性能比とそれぞれの項目の関係を示す．また，それぞれの項目について多項式近似曲線を示す．

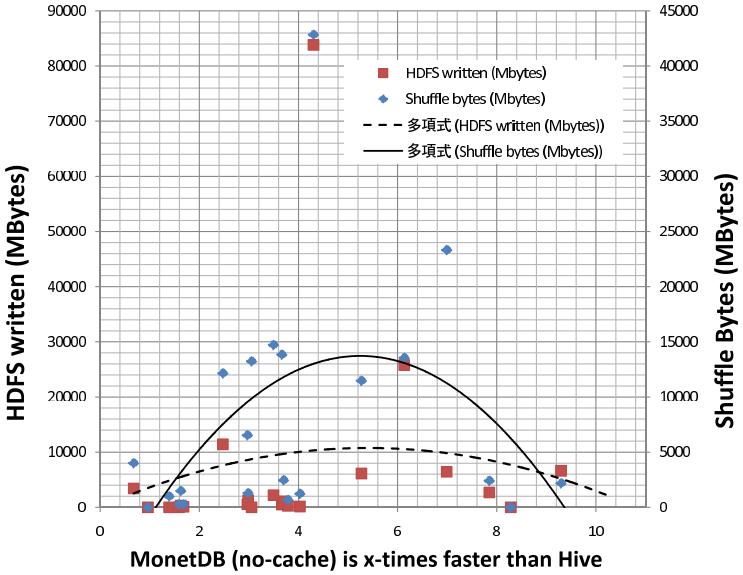


図 16　キャッシュの影響を排除した場合の MonetDB/MR と Hive の性能比較
Fig. 16　Performance comparison between MonetDB/MR and Hive under no-cache effect.

性能向上比 7 倍[⋆1]までの相関係数は，shuffle データ量で 0.56，HDFS 書き込み量で 0.31 であり，特に Shuffle データ量に関しては，7 倍程度までの性能向上の因子となっている可能性が高く，shuffle データ量が多い箇所で Φ ハッシュ分割の有効性が確認できる．図 16 で多項式近似曲線が 6 倍付近に極大値を持ち，かつ対称性を持つ理由としては，4.1.1 項で述べたように，32 ノードによる Φ ハッシュ分割によるファクト表の分割効果が最大 6.81 倍であることに関連しているとみられる．そのため 6.81 倍以上の性能向上がみられた Q1，Q2，Q16，Q21 の 4 つの問合せには，列指向データベースの効率的な I/O などの利点や問合せ内でのキャッシュ効果といった Φ ハッシュ分割以外の因子が性能向上に影響しているとみられる．

⋆1 データ項目数が少ないため，性能向上比 6.1 倍までではなく，性能向上比 6.81 倍に最も近い項目が存在する性能向上比 7 倍までのデータを利用した．

表 7　総 Shuffle 出力量と HDFS への書き込みデータ量（MB）
Table 7　Total shuffle output size and HDFS written data size (MB).

	Shuffle bytes (MB)	HDFS written (MB)
Q1	0.25	0.01
Q2	2,416.77	2,685.73
Q3	6,540.37	568.13
Q4	1,296.22	1,287.48
Q5	11,478.21	6,130.36
Q6	0.02	0.00
Q7	13,560.90	25,786.14
Q8	14,727.15	2,200.21
Q9	42,841.77	83,825.84
Q10	4,004.69	3,428.32
Q11	1,239.04	150.98
Q12	997.48	0.01
Q13	1,487.41	317.80
Q14	272.62	0.01
Q15	261.02	108.01
Q16	2,184.44	6,621.69
Q17	13,846.95	492.91
Q18	12,153.47	11,426.50
Q19	13,244.51	0.01
Q20	2,467.74	1,092.02
Q21	23,328.26	6,436.43
Q22	675.73	269.19

6.1.3　Hive における問合せ処理の流れ

選択演算と結合演算を含む問合せを処理するとき，Hive は分散ファイルシステム HDFS から読み込んだブロックデータを map 処理で選択処理して，reduce 側でソートマージ結合する．

5.1.1 項に述べた MonetDB/MR の 1 つの M/R イテレーションによる実行との違いを明らかにするため，付録 A.1 に SQL 問合せを示す TPC-H の Q3 を例に Hive の問合せ処理の流れを述べる．Hive は，Q3 を 5 つの MapReduce のジョブによって実行する[⋆1]．最初のジョブ J_1 は，map 処理で選択演算を行った後，結合条件に基づいて Customer 表と Orders 表を shuffle し，reduce 側でソートマージ結合を行い，中間結果 I_1 を HDFS に書き込む．2 番目のジョブ J_2 は I_1 と Lineitem 表を結合条件に基づいて shuffle して結合し，

⋆1 なお，Hive ではテーブルの結合順序はユーザが指定したとおりとなる．

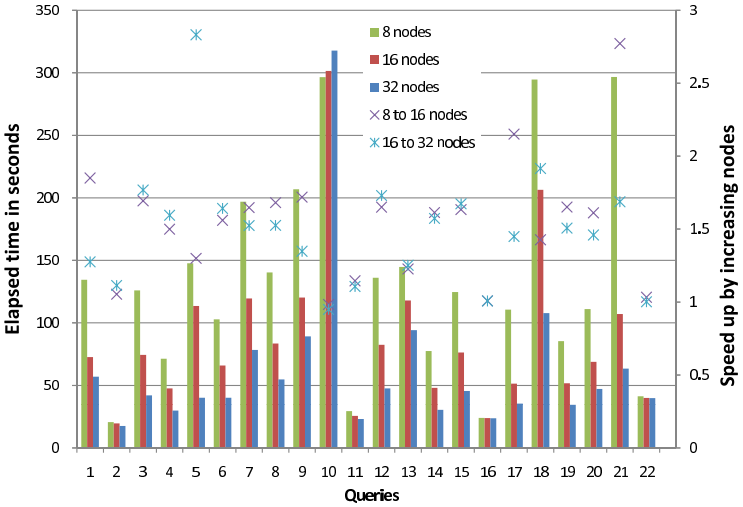


図 17　ノード数に対するスケーラビリティ（MonetDB/MR）
Fig. 17　Scalability to the number of nodes (MonetDB/MR).

その中間結果 I_2 を HDFS に書き込む．3 番目のジョブ J_3 は，I_2 をグループ化キーに基づいて shuffle して集約する．4 番目のジョブ J_4 は *revenue* と *orderdate* の降順でソートする．5 番目のジョブ J_5 は，ソート済み結果からトップ 10 件を HDFS に書き込む．

文献 30) では，Q3 については中間の結合と shuffle データ量が小さいため，中間結果の書き込みと shuffle 操作のオーバヘッドは低いとしている．Q3 の問合せ実行で Hive では 1,296.2 MB の shuffle と 1,287.5 MB の HDFS の書き込みが行われる．表 6 に示すとおり，Q3 では MonetDB/MR（no-cache）の Hive に対する性能比は 3.0 倍の向上にとどまった．一方で，図 15 に示すように，Q3 ではキャッシュ効果が 6.8 倍現れており，水平垂直分割と列指向のデータアクセスによるキャッシュ利用効率の高さが支配的であるといえる．なお，他の TPC-H 問合せの Hive における処理については文献 30) を参照されたい．

6.2　ノード数に対するスケーラビリティ

ここでは，TPC-H の SF=100 を用いて MonetDB/MR のノード数に対するスケーラビリティを示す．ノード数は 8 ノード，16 ノード，32 ノードを用いた．**図 17** にその結果を示す．図 17 の問合せのそれぞれの上部に位置するマーカは，縦軸右を軸にとり，ノード数を 2 倍としたときの処理時間の伸びの割合を示す．各問合せについて，性能向上率の平均を

表 8　ノード数の増加によって得られた性能向上
Table 8　Performance gain for increasing the number of nodes.

Speed up (x%)	Queries
$x < 0$	Q10
$0 \leq x < 20$	Q2, Q11, Q16, Q22
$20 \leq x < 50$	Q13
$x \geq 50$	Q1, Q3–Q9, Q12, Q14, Q15, Q17–Q21

グループごとにまとめたのが**表 8** である．

表 8 に示すとおり，22 個中 16 個の問合せが 50%以上の台数効果がみられており，確かな台数効果が確認できる．一方で，Q2，Q11，Q16，Q22 については台数効果があまりない問合せであり，Q10 では性能の悪化がみられた．

上記した問合せの特徴としては次のとおりである．Q10 については，6.1.1 項ですでに示したように単一の reducer の処理モデルがうまく適合しないケースである．Q11 と Q22 は 2 回の M/R のイテレーションを必要とする問合せである．ノード数を N とすると，マージ回数 $2N$ 回となり，このことが性能低下をもたらしたと考えられる．

これらの台数効果とマージされるレコードは特に相関は観察できなかった．元々の問合せが複数の map と単一の reduce の処理モデルに適合するかが性能に影響する．Merge 処理と Reduce 処理は単一の Reducer ノードで逐次的に行われる処理である．**表 9** に示すように，これらの割合が大きい問合せ（Q2，Q10，Q11，Q16，Q22）で並列処理による性能向上は限定的となる．

これは複数 mapper 単一 reducer 実行モデルだけをとる現在の MonetDB/MR の手法的な問題といえる．複数 reducer を用いるのは最終的な集約処理にかかるデータ交換などのコストを考慮すると必ずしも優れるとはいえないが，これらの reducer での処理が支配的な問合せに対しては，逐次処理部分を少なくするために，ノード内並列の集約問合せ処理（partial aggregation）を行うなどの改善が必要である．また，マージ処理に関しては，今回の計算機構成のような単一ディスク構成では直列にマージ処理を行うことは必ずしも非効率ではないが，さらなる性能向上には，物理的なマシンのディスク構成も含めてテーブルスペースを分けて設計したうえでデータを並列に投入することが有効とみられる．

6.3　データ量に対するスケーラビリティ

MonetDB/MR のデータ量に対するスケーラビリティを**図 18** に示す．縦軸右を軸とする SF100/SF50 と SF200/SF100 が示すのは，それぞれ SF50→SF100 と SF100→SF200 にデータ量を 2 倍としたときの，問合せ実行時間の伸びの割合である．実行時間の伸びが小さいときほど，基準に対して実行効率が良い．

表 9　問合せ実行時間に占める Merge/Reduce 処理の割合
（太字部分は Merge/Reduce 処理の割合が 15%以上）
Table 9　The percentage of Merge/Reduce processing on the total query execution times
(Bold portion represents that the percentage is over 15%).

	Merge (%)	Reduce (%)
Q1	7.74	0.18
Q2	**25.45**	2.64
Q3	11.65	3.80
Q4	14.89	0.19
Q5	10.59	0.14
Q6	10.76	0.07
Q7	5.56	0.10
Q8	7.84	0.16
Q9	4.78	0.10
Q10	**18.59**	**17.46**
Q11	**18.61**	0.40
Q12	9.03	0.13
Q13	8.51	13.38
Q14	14.25	0.19
Q15	9.40	0.26
Q16	**20.34**	**15.58**
Q17	11.94	0.12
Q18	4.13	0.17
Q19	12.77	0.13
Q20	0.33	0.20
Q21	7.74	1.45
Q22	**21.89**	0.42

メモリ割当ての制限により MonetDB/Single では SF=200 は処理できなかったが，図 18 より，32 ノード構成の MonetDB/MR が多くの問合せでデータ量に対してスケールすることが確認できる．ここで，データ量を 2 倍としたときに実行時間が 2 倍とならないのは，データのキャッシュ効果が依然として有効であるためである．逆に，SF100→SF200 でのデータ量の倍増により実行効率が下がった Q3，Q5，Q10，Q13，Q18 について，データ量を増加させたときにキャッシュ効果を期待するにはノード数増加によるデータ分割が必要である．

7. 関 連 研 究

Afrati らは，MapReduce 環境での多重結合演算の扱いを議論しており，単一の MapReduce/shuffle プロセス（1 回のタプル分散）で多重結合演算を扱うハッシュ分割手法を提案

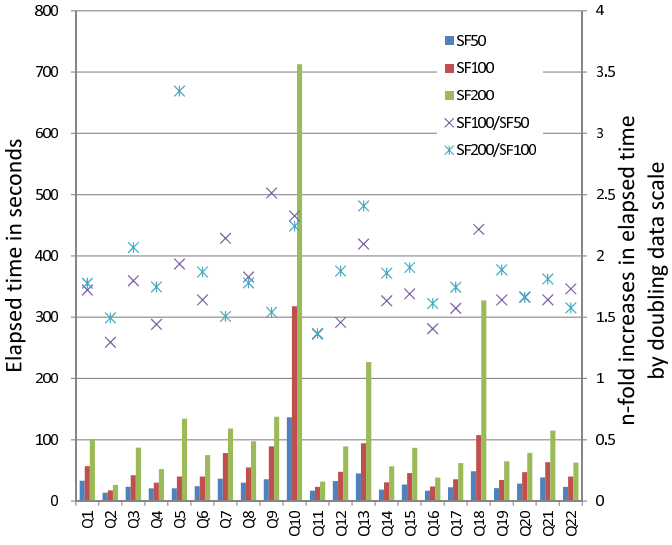


図 18　データ量に対するスケーラビリティ（MonetDB/MR）
Fig. 18　Scalability to data volumes (MonetDB/MR).

している[31]．3 つのリレーション R(a,b)，S(b,c)，T(a,c,d) が属性 b と属性 c によって結合されるとき，b と c のハッシュ値の組合せ（h(b)，h(c)）で表現される reduce プロセスにタプルを配分する．このとき，リレーション R とリレーション T は，それぞれハッシュ値 h(b) とハッシュ値 h(c) に相当するすべての reduce プロセスにタプルを配分する．この多重結合手法は結合数が大きくなると必然的にレプリケートする箇所が大きくなるという欠点が存在するため，ラグランジェの未定乗数法を用いて shuffle コストの最適解を探索し，多重結合演算のためのパーティション関数の分割キーを求める手法を提案している．そして，多重結合と Hadoop で行われている複数の MapReduce ジョブによる結合（cascade join）を組み合わせることを提案している．この手法は動的なデータ分散を行ったデータを MapReduce Job 内で閉じた中で 1 回限り使うことを前提としており，複数の問合せを考慮した静的なデータ分散には利用できない．リレーション R と T のタプルは複製が存在することがあるため，タプルの一意性を保証することが難しいといった問題があり，たとえば，R と T の属性 a による結合や R の総行数のカウントなどタプルの一意性を保証することが必要な多くの問合せを正しく処理することができない．これに対して，Φ ハッシュ分割ではデータ分割に利用した属性を追加属性としてタプルに加えることで，問合せ処理器が必要なタプルを選択することができる．文献 31) が問合せに応じて問合せ処理時に動的なハッシュ分割を行うため最低 1 回の shuffle を前提とするのに対して，Φ ハッシュ分割ではあらかじめスキーマ情報に基づいた静的なハッシュ分割を行うため，問合せ処理時の shuffle 処理に要する時間を省略できる．

Map-Join-Reduce[30] では，k 個のキーを入力とした分割関数を導入して，適宜 1 つのレコードを複数のパーティションに複製することで，結合キーの異なる 3 テーブル以上の結合処理を 1 つの（map 処理でテーブルの動的なハッシュ分割を行い，reduce 処理で多重結合演算を行う）MapReduce ジョブで処理する手法を提案している．MapReduce では Job 間の入出力がつねに（分散）ファイルシステムを介するため，HDFS への入出力が大きい場合に性能が悪化することを問題としてあげ，多重結合演算を MapReduce で行う場合に Job 数を減らすことが可能な filtering-join-aggregation プログラミングモデルを MapReduce の拡張として Hadoop に導入している．Map-Join-Reduce の多重結合演算処理手法は，文献 30) で述べられているように，本質的に文献 31) の手法を踏襲するものであるが，1) 必ずすべての結合キーを入力にとる分割関数を採用するという点，2) ラグランジェの未定乗数法の代わりに整数計画法とヒューリスティックなアプローチにより分割キーを求めるという点に違いがあるとしている．この手法には，Afrati らの手法[31] と同様の欠点が存在し，また結合数が大きくなったときの対策やリレーションの濃度が低い支配的な属性が結合属性に含まれる場合が考慮されていないという問題がある．たとえば，図 4 の Region テーブルの濃度は 5，Nation テーブルの濃度は 25 であるが，こうしたものについてユーザ側で特別に考慮して分割関数を作ることや MapReduce プログラミングを行う必要があり，文献 30) では，Nation テーブルは特別扱いをしてすべての mapper であらかじめメモリ上にロードするとしている．Φ ハッシュ分割では，関係データベースのカタログ情報より参照整合性制約を考慮することで，ユーザの介入なしにシステム側で，濃度の高い支配的な属性（dominant attribute）によって芋づる式にタプルがコピーされることを回避している．

Graefe は，partitioned b-tree と呼ぶクラスタ化 B 氏木を提案している[32]．partitioned b-tree は，複合（compound）キーを利用した一般的な B 氏木により，B 氏木の分割を行う技術である．ここで，属性をキーとしてタプルが一意に紐付けられている B 氏木を考える．partitioned b-tree は，システム側で管理対象のタプルに人工的なカラムを追加する．そして，B 氏木にタプルを追加する際のキーの第 1 項（誘導キーカラム）としてその人工的に生成した属性値を用いる．こうすることにより，人工的なキーカラムにより B 氏木を分割

する．partitioned b-tree は，ロックの競合やランダム I/O といった従来の B 氏木の問題を避け，ソート処理や索引構築，一括挿入といったバッチ的な操作に適用することを目的とした技術である．検索時にはパーティションごとに検索を行う必要があり余分な処理が発生するが，外部マージソートを利用した動的再編成を行うとしている．我々のテーブル分割手法は追加カラムによりパーティションを明示するという点において partitioned b-tree と類似する．partitioned b-tree が索引の一括処理のための技術であるのに対して，Φ ハッシュ分割は並列演算の高速化のためのテーブル分割法である．

関係データベースクラスタ索引は 1 つの次元（属性集合）から行データに効率的にアクセスできるようにする仕組みである．多次元に対して，それぞれ索引を張ることは索引のメンテナンスコストの増大を生み，現実的ではない．多面的な観点からの多次元分析が行われるデータウェアハウスのためのテーブルの分割記録手法として，Multi-Dimensional Clustering（MDC）が提案されている[33]．MDC では，キーの組合せにより多次元を定義し，スライス（slice）と呼ぶある 1 つ以上の次元に基づいてクラスタ化を行ってデータ配置を管理する．Adjoined Dimension Columns（ADC）[34] は MDC をスタースキーマに適用するうえで，グリッドの次元の切り方（グリッドのセルの作り方）を述べている．セルをまたいだアクセスではディスクのシークが行われるため，各セルを適切なサイズとすることが重要である．TPC-H のスキーマをスタースキーマ構成とした Star Schema Benchmark（SSB）[35] で，ファクト表である Lineitem 表は 4 つのディメンションを持つ．このとき，$7 \times 5 \times 25 \times 25 = 21{,}875$ でディメンションを切ることで，セルごとのサイズが $76\,\mathrm{GB}/21{,}875 = 3.5\,\mathrm{MB}$ とディスクアクセスに適したサイズになることを述べている．一般に，MDC などの多次元分割は全共有型設計あるいは高速なインターコネクトを備える共有ディスク設計で活きる技術である．無共有型設計で複数のスライスにアクセスする必要がある問合せでは，データと計算の局所性（ローカリティ）が十分に活きない．

Multi-Attribute GrId deClustering（MAGIC）[36] は複数の属性に基づいて[⋆1]ノード間にテーブルを分割する手法を提案している．しかし，MAGIC では問合せを並列処理するとき，他のノードに配置されたセルを参照することが必要である．そこで，MDC のような多次元による分割を無共有型設計に適用するうえで，通信コストや問合せによるアクセス頻度を考慮してグリッドファイル[37] の要領で切り分けるセルのサイズを調整する手法を提案している．こうした多次元分割手法では，ワークロードの変化に応じた再編成を前提としており，無共有型設計ではタプルの再分散が問題となる．これに対して提案手法は，スキーマ情報からあらかじめ考えられるすべての結合属性の組合せによりデータ分割が行われるため，タプルの再分散を必要としない．

Google の Bigtable[38] は，行キー，カラムキー，タイムスタンプの組によりテーブルを多次元に分割して管理する技術である．テーブルは行キーによって水平分割され，タブレットという単位に分割されて管理される．タブレットはさらに分割され，カラムファミリという任意の列ごとにグループ化されて Log-Structured Merge Tree[39] の一種である SSTable により索引付けて管理され，SSTable は最終的に GFS 上のファイルとして永続化される．この点で，Bigtable はグループキーによるインデックスファイル技術といえる．列指向データベースの採用する Decomposition Storage Model[40] と異なりカラム単位でのアクセスを効率的にするには，適切なカラムを行キーとして選択して索引付けを行ったうえで，行キーによる絞り込みが可能な場合に限られる．

8. むすび

本論文では，テーブルの水平垂直分割法と MapReduce とを統合するハイブリッドシステムとしての初めての試みを示した．MapReduce などと異なり，関係データベースはスキーマ情報を明示的に持つ．スキーマ情報（参照整合性制約）を利用し，再分散を必要としないことを特徴とするテーブル分割手法（Φ ハッシュ分割）を導入することで，ノード数増加に対するスケーラビリティを阻害することなしに複雑なデータ分析問合せを並列処理することができる．Φ ハッシュ分割での重要な発明の 1 つは，ハッシュパーティションに格納されるタプルごとに “そのタプルがどのような分割属性に基づいて格納されたか” というメタデータを差し込むことで，パーティションの識別を問合せ処理器に明示的にしたことである．また，Φ ハッシュ分割では，データベーススキーマがテーブル分割方法を自動的に導出するため，複雑な要因が絡むテーブル分割属性の決定にデータベース管理者（DBA）の介入を必要としない．本提案手法は列指向データベースの 1 つである MonetDB 上に実装したが，行指向データベースにも適用可能である．

評価実験では，32 ノード構成で TPC-H SF=100 で評価を与え，MapReduce に基づくデータウェアハウスシステム（Hive）に対して，我々の並列データベースが DBA の介入なしに大きな性能面での優越を示しうることを実証するとともに，我々の問合せ処理手法の現実装における有効範囲と制限にも考察を与えた．この結果は，今後の MapReduce と DBMS の統合プロジェクトの性能評価において 1 つの指標となる．

⋆1 複合キーによるパーティショニングとは異なることに留意されたい．

原理的に，Φ ハッシュ分割はノード数を増やすことでより大きなデータ量にも対応することができるが，MonetDB/MR が現在採用している複数 mapper 単一 reducer 実行モデルは，reduce 処理（あるいは merge 処理）が支配的となる場合に課題を残している．reduce 処理を単一ノードで行うか複数ノードで行うかの閾値を shuffle データ量やキャッシュ利用効率といったパラメタよりシステム側で考慮し，問合せをノード間並列あるいはノード内並列に適切に割り振ることが，MonetDB/MR の今後の課題である．

謝辞　本研究は，日本学術振興会科研費研究活動スタート支援（課題番号：22800086）の助成を受けたものである．また，本研究の一部は，日本学術振興会優秀若手研究者海外派遣事業の支援のもと，オランダ国立数学情報学研究所（CWI）で行った．本研究を進めるうえで助言を与えてくれた Martin Kersten と Peter Boncz の両氏に感謝する．

参 考 文 献

1) Dean, J. and Ghemawat, S.: MapReduce: Simplified data processing on large clusters, *Proc. OSDI*, pp.137–150 (2004).
2) Abouzeid, A., Bajda-Pawlikowski, K., Abadi, D., Rasin, A. and Silberschatz, A.: HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads, *Proc. VLDB*, Vol.2, No.1, pp.922–933 (2009).
3) Yang, C., Yen, C., Tan, C. and Madden, S.R.: Osprey: Implementing MapReduce-Style Fault Tolerance in a Shared-Nothing Distributed Database, *Proc. ICDE*, pp.657–668 (2010).
4) Kitsuregawa, M., Tanaka, H. and Moto-Oka, T.: Application of hash to data base machine and its architecture, *New Generation Computing*, Vol.1, No.1, pp.63–74 (1983).
5) Dewitt, D.J., Ghandeharizadeh, S., Schneider, D.A., Bricker, A., Hsiao, H.I. and Rasmussen, R.: The Gamma Database Machine Project, *IEEE Trans. Knowledge and Data Engineering*, Vol.2, No.1, pp.44–62 (1990).
6) Shasha, D. and Wang, T.L.: Optimizing equijoin queries in distributed databases where relations are hash partitioned, *ACM Trans. Database Syst.*, Vol.16, No.2, pp.279–308 (1991).
7) Wang, G. and Ng, T.S.E.: The Impact of Virtualization on Network Performance of Amazon EC2 Data Center, *Proc. INFOCOM*, pp.1163–1171 (2010).
8) CWI: MonetDB, available from ⟨http://monetdb.cwi.nl/⟩.
9) Dittrich, J., Quiané-Ruiz, J.A., Jindal, A., Kargin, Y., Setty, V. and Schad, J.: Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing), *Proc. VLDB*, Vol.3, No.1, pp.515–529 (2010).
10) Jiang, D., Ooi, B.C., Shi, L. and Wu, S.: The performance of MapReduce: An in-depth study, *Proc. VLDB*, Vol.3, No.1, pp.472–483 (2010).
11) Liu, C. and Chen, H.: A hash partition strategy for distributed query processing, *Proc. EDBT*, pp.371–387 (1996).
12) Epstein, R., Stonebraker, M. and Wong, E.: Distributed query processing in a relational data base system, *Proc. SIGMOD*, pp.169–180 (1978).
13) Yu, C.T., Guh, K.-C., Zhang, W., Templeton, M., Brill, D. and Chen, A.L.P.: Algorithms to Process Distributed Queries in Fast Local Networks, *IEEE Trans. Computers*, Vol.C-36, No.10, pp.1153–1164 (1987).
14) Furtado, P.: Experimental evidence on partitioning in parallel data warehouses, *Proc. DOLAP*, pp.23–30 (2004).
15) Transaction Processing Performance Council: TPC-H Benchmark Specification, available from ⟨http://www.tpc.org/tpch/⟩.
16) Ozsu, T.M. and Valduriez, P.: *Principles of Distributed Database Systems*, *2nd Ed.*, Prentice Hall (1999).
17) Bellatreche, L., Karlapalem, K., Mohania, M. and Schneider, M.: What can Partitioning do for Your Data Warehouses and Data Marts?, *Proc. IDEAS*, pp.437–445, IEEE Computer Society (2000).
18) Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein, C.: *Introduction to Algorithms*, *2nd Ed.*, The MIT Press (2001).
19) Karger, D. et al.: Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web, *Proc. STOC*, pp.654–663 (1997).
20) Golubchik, L., Lui, J.C.S. and Muntz, R.R.: Chained Declustering: Load Balancing and Robustness to Skew and Failures, *Proc. RIDE*, pp.88–95 (1992).
21) Birman, K.: *Building secure and reliable network applications*, pp.15–28, Springer (1997).
22) DeWitt, D.J., Naughton, J.F., Schneider, D.A. and Seshadri, S.: Practical Skew Handling in Parallel Joins, *Proc. VLDB*, pp.27–40 (1992).
23) Manegold, S., Kersten, M.L. and Boncz, P.: Database architecture evolution: Mammals flourished long before dinosaurs became extinct, *Proc. VLDB*, Vol.2, No.2, pp.1648–1653 (2009).
24) Idreos, S., Kersten, M. and Manegold, S.: Database Cracking, *Proc. CIDR* (2007).
25) The Apache Software Foundation: Hive, available from ⟨http://hadoop.apache.org/hive/⟩.
26) Cloudera, Inc.: Cloudera's Distribution for Hadoop (CDH), available from ⟨http://www.cloudera.com/hadoop/⟩.
27) Hive JIRA: Hive-600,

available from ⟨https://issues.apache.org/jira/browse/HIVE-600⟩.
28) Stonebraker, M., Abadi, D., DeWitt, D.J., Madden, S., Paulson, E., Pavlo, A. and Rasin, A.: MapReduce and parallel DBMSs: Friends or foes?, *Comm. ACM*, Vol.53, No.1, pp.64–71 (2010).
29) Larson, P.: Data Reduction by Partial Preaggregation, *Proc. ICDE*, pp.706–715 (2002).
30) Jiang, D., Tung, A.K.H. and Chen, G.: MAP-JOIN-REDUCE: Towards Scalable and Efficient Data Analysis on Large Clusters, *IEEE TKDE*, Vol.23, No.9, pp.1299–1311 (2011).
31) Afrati, F.N. and Ullman, J.D.: Optimizing joins in a map-reduce environment, *Proc. EDBT*, pp.99–110 (2010).
32) Graefe, G.: Sorting And Indexing With Partitioned B-Trees, *Proc. CIDR* (2003).
33) Padmanabhan, S., Bhattacharjee, B., Malkemus, T., Cranston, L. and Huras, M.: Multi-dimensional clustering: a new data layout scheme in DB2, *Proc. SIGMOD*, pp.637–641 (2003).
34) Chen, X., O'Neil, P.E. and O'Neil, E.J.: Adjoined Dimension Column Clustering to Improve Data Warehouse Query Performance, *Proc. ICDE*, pp.1409–1411 (2008).
35) O'Neil, P.E., O'Neil, E.J. and Chen, X.: The Star Schema Benchmark (SSB) (2007), available from ⟨http://www.cs.umb.edu/~poneil/StarSchemaB.pdf⟩.
36) Ghandeharizadeh, S. and DeWitt, D.J.: MAGIC: A Multiattribute Declustering Mechanism for Multiprocessor Database Machines, *IEEE Trans. Parallel Distrib. Syst.*, Vol.5, No.5, pp.509–524 (1994).
37) Nievergelt, J., Hinterberger, H. and Sevcik, K.: The grid file: An adaptable, symmetric multi-key file structure, *Trends in Information Processing Systems*, *Lecture Notes in Computer Science*, Vol.123, pp.236–251 (1981).
38) Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A. and Gruber, R.E.: Bigtable: A Distributed Storage System for Structured Data, *ACM Trans. Comput. Syst.*, Vol.26, No.2, pp.1–26 (2008).
39) O'Neil, P., Cheng, E., Gawlick, D. and O'Neil, E.: The log-structured merge-tree (LSM-tree), *Acta Inf.*, Vol.33, No.4, pp.351–385 (1996).
40) Copeland, G.P. and Khoshafian, S.N.: A decomposition storage model, *SIGMOD Rec.*, Vol.14, No.4, pp.268–279 (1985).

付　　録

A.1　TPC-H Q3

```
select
    l_orderkey,
    sum(l_extendedprice * (1 - l_discount)) as revenue,
    o_orderdate,
    o_shippriority
from
    customer, orders, lineitem
where
    c_mktsegment = 'BUILDING'
    and c_custkey = o_custkey and l_orderkey = o_orderkey
    and o_orderdate < date '1995-03-15' and l_shipdate > date '1995-03-15'
group by
    l_orderkey, o_orderdate, o_shippriority
order by
    revenue desc, o_orderdate
limit 10
```

(平成 23 年 6 月 20 日受付)
(平成 23 年 10 月 3 日採録)

(担当編集委員　鬼塚 真)

油井　　誠（正会員）

独立行政法人産業技術総合研究所情報技術研究部門サービスウェア研究グループ研究員．2003 年芝浦工業大学工学部工業経営学科卒業．同年（株）NEC 情報システムズ入社．グリッド・コンピューティングの研究開発に従事．2006 年奈良先端科学技術大学院大学情報科学研究科博士前期課程修了．2008 年日本学術振興会特別研究員 DC2．2009 年奈良先端科学技術大学院大学情報科学研究科博士後期課程修了．博士（工学）．同年日本学術振興会特別研究員 PD．この間，早稲田大学 IT 研究機構（IT バイオ研究所）客員研究員およびオランダ国立数学情報学研究所（CWI）客員研究員．2010 年独立行政法人産業技術総合研究所入所．現在に至る．高性能データベースおよび並列分散システムの研究に従事．日本データベース学会会員．

小島　　功（正会員）

昭和 59 年京都大学大学院工学研究科情報工学専攻修了，同年通産省電子技術総合研究所入所．現在，独立行政法人産業技術総合研究所情報技術研究部門サービスウェア研究グループ長．分散データベースや e-サイエンス基盤に関する研究に従事．ACM 会員．Open Grid Forum データベース統合ワーキンググループ共同議長．