情報処理学会論文誌　データベース　Vol.6　No.3　105–119 (June 2013)

PJoin: MapReduceにおける高速ジョイン処理

鬼塚 真[1,a)]　久保 類[1,b)]

受付日 2012年12月20日, 採録日 2013年4月1日

概要：MapReduce は分散処理フレームワークとして分析応用において広く利用されつつある．OLAP は分析応用の典型例の 1 つであり，OLAP 分析ではスタースキーマによる多テーブルジョインが多く利用される特徴がある．しかし，MapReduce によるジョイン処理では，ジョイン対象となるテーブルをシャッフルにより通信しなければならずコストが大きいという問題があった．本論文では，MapReduce における高速ジョイン処理 PJoin を提案する．PJoin の特徴は，ディメンジョンテーブルごとにジョイン対象となるテーブルの主キーと外部キーのペアを射影した実体化ビューを構築し，テーブルを主キーで分散配置し実体化ビューを外部キーで分散配置することで，map タスクにおいてファクトテーブルとディメンションテーブル間の多テーブルの semi-join を実行することにある．TPC-H のベンチマークによって評価した結果，PJoin により応答性能が平均的に 33.9%向上し，シャッフルによる通信量は 62.6%削減できたことを確認した．

キーワード：MapReduce，OLAP 分析，最適化

PJoin: Efficient Join Algorithm Based on MapReduce

Makoto Onizuka[1,a)]　Rui Kubo[1,b)]

Received: December 20, 2012, Accepted: April 1, 2013

Abstract: MapReduce is a distributed computation framework and is getting widely used in data intensive analytic applications. One important type of data analysis is OLAP, in which queries intensively use join operations for multi-dimensional aggregations. However, a typical join algorithm used in MapReduce, reduce-side join, shuffles the records of the join tables by the join key, which greatly increases the network overhead. We propose PJoin, an efficient join algorithm for MapReduce. The features of PJoin are 1) to materialize projection views required for semi-joins, and 2) to pre-partition them and base tables by foreign keys and primary keys, respectively. Thus, semi-joins between two tables that are connected by one-to-many relationships are to be made without shuffle. We use the TPC-H benchmark to verify the efficiency of PJoin and show that it improves the response time by 33.9% over the reduce-side join. Its key advances are a 62.6% reduction in shuffle size.

Keywords: MapReduce, OLAP, optimization

1. はじめに

Google，Facebook，Yahoo に代表される web 企業ではクリックストリームなどの膨大なデータを分析しビジネスに活用している．分散処理フレームワーク MapReduce [1] はこのような分析処理の用途に使用されており，1,000 台を超えるコンピュータのクラスタ上で動作し，高いスケーラビリティと可用性を有している．実際，Google では 1 日あたり 20 PB のデータを MapReduce で分析していて [2]，Facebook では毎日 75 TB の圧縮データを Hadoop によって処理をしている．これまで OLAP 系の分析応用はリレーショナル DBMS で実装され，並列データベースがベンダによって提供されてきている．しかし，利用者はこれらの製品は大規模なデータに対してコストに見合わないと考え

[1] NTT ソフトウェアイノベーションセンタ
NTT Software Innovation Center, Musashino, Tokyo, 180–8585, Japan
[a)] onizuka.makoto@lab.ntt.co.jp
[b)] kubo.rui@lab.ntt.co.jp

ており，実際に Hammerbacher [3] は Facebook において Oracle データベースを利用して BI 応用をスケールアップした経験について議論しており，Oracle データベースを諦めて MapReduce 上で SQL ライクなクエリを処理する Hive [4] を代替として開発して利用した．Hive はクエリを MapReduce プログラムに変換し，MapReduce 環境において実行する．

MapReduce ジョブは，ユーザが定義した map 関数を実行するフェーズと，reduce 関数を実行するフェーズの 2 つから構成される．分散ファイルシステムに格納されているデータに対して map 関数は実行され，結果はキーと値の組として出力される．こうして得られたキーと値の組はシャッフルされ同一のキーであるキーと値の組はまとめられて，reduce 関数に入力される．こうして reduce 関数はキーごとに実行され結果が分散ファイルシステムに出力される．

しかし，MapReduce にはそのアーキテクチャに根付く本質的な性能問題—スキーマの欠如，インデックスの欠如，データの偏り，シャッフルのネットワークオーバヘッド，中間データの実体化オーバヘッド—があると [5] で述べられている．特に，OLAP 分析でも最もコストが高い処理である join 処理は，MapReduce において map タスクと reduce タスク間のシャッフルに起因する性能のボトルネックがあるため，処理コストが高いという問題がある．典型的な join アルゴリズムは reduce-side join であり [6]，このアルゴリズムは join 対象の 2 テーブルの全レコードを join キーを用いてシャッフルし，reduce タスクにおいて join キーによってグループ化されたテーブル間のレコード群を join 処理する．このため，join 対象のテーブルをシャッフルする処理のコストは高い．

この課題に対して，筆者らは MapReduce での高速な join アルゴリズムである PJoin（partition-based join）を提案する．PJoin は以下の前提に基づいて設計されている．1) OLAP 分析応用では，更新操作よりも参照操作の頻度が高く，2) OLAP 分析におけるクエリの特徴は，スタースキーマのデータを対象として，多対 1 の基数を持つファクトテーブルとディメンジョン間において join 操作が多用されることである．上記の 2 点に基づいて，PJoin ではデータ更新の際に事前処理として，ファクトテーブルとディメンジョンテーブル間の semi-join に必要な射影ビューを実体化し，テーブルと実体化ビューをそれぞれ主キーと外部キーでパーティション分割する，という主な 2 つの工夫を行っている．

PJoin の 1 つ目の工夫として，semi-join に必要な外部キーをファクトテーブルから射影した射影ビューを実体化することで，join の処理コストを削減する．semi-join は（$\ltimes$ と表記），書き換え規則 $T \bowtie S = (T \ltimes S) \bowtie S$ に基づいて分散データベースにおける join（$\bowtie$ と表記）の高速化に使われる [7]．多対 1 の基数を持つテーブル間の外部キーと主キーの等価性を条件として join が行われる場合，上記の書き換え規則から $T \bowtie S = (T \bowtie S') \bowtie S$ を得ることができる．ただし，S' はテーブル S から T への外部キーを射影して得られる射影ビューである．OLAP クエリでは多対 1 の基数を持つファクトテーブルとディメンジョンテーブル間において join 操作が多用されるため，PJoin では個々のファクトテーブルとディメンジョンテーブルの組ごとに射影ビューを実体化する．

PJoin の 2 つ目の工夫として，テーブルと実体化ビューをそれぞれ主キーと外部キーでパーティション分割することで，シャッフルすることなく semi-join を map タスクで実行して join の処理コストを削減する．さらに PJoin では，ファクトテーブルの主キーの射影も射影ビューに加えることで，semi-join 結果を主キーでシャッフルし，reduce タスクにおいてこの主キーを用いて semi-join 結果（$T \bowtie S'$）と対応するテーブル（S）のパーティションを join 処理する．

その他，中間データサイズを削減する工夫として PJoin では以下の 3 点のことを行っている．1) 射影ビューの拡張：PJoin では，クエリワークロードを解析して WHERE 条件対象のカラムを射影ビューに加えることで，WHERE 条件を map タスクで実行して中間データ量を削減する．2) PJoin では，多対 1 の基数を持つテーブル間の多段の join において，1 側のテーブルから多側のテーブルへと処理を行うプランを採用する．3) 複数の semi-join をそれぞれ map タスクで並列に実行し，これら semi-join 結果を用いて reduce タスクで多テーブル join を実行する．

本論文の構成は以下のとおりである．2 章では，MapReduce における典型的な join アルゴリズム群について述べる．3 章では，PJoin の詳細について述べる．4 章では，TPC-H ベンチマークを用いた性能評価結果を報告する．5 章では関連研究について述べ，6 章で本論文についてまとめる．

2. MapReduce における join アルゴリズム

本章では，現在，広く使用されている MapReduce における join アルゴリズムについて説明する．関連研究については 5 章で説明する．

MapReduce における代表的な join アルゴリズムは 3 つあり，reduce-side join, map-side join, memory-backed join と呼ばれている [4], [6] [*1]．reduce-side join は最も一般的な手法であり，join 対象の複数テーブルのパーティションを map タスクで読み込み，join キーに従ってレコードをシャッフルし，reduce タスクにおいて join キーによってグループ化された異なるテーブルのレコード群を join 処理する（**図 1**）．このため reduce-side join はシャッフルによる

[*1] Hive [4] では memory-backed join を map side join と呼んでいる．

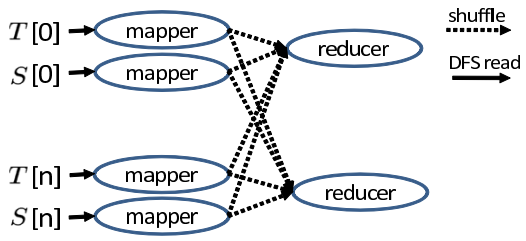


図 1　reduce-side join の概要
Fig. 1　Overview of reduce-side join.

コストが大きい．

map-side join は，join キーによって事前にソート・パーティション分割された 2 テーブルに対して join を実行する手法である．map-side join では join 処理が map タスクで完結するため，シャッフルが不要となり，reduce-side join よりも処理性能が優れている．文献 [6] によれば，MapReduce ジョブのワークフローが事前に分かっていて比較的変化しないケースでは，map-side join が実行できるように 1 つ前の MapReduce ジョブにおいて出力結果をソート・パーティション分割することができると述べている．しかし OLAP のクエリでは，ファクトテーブルは複数のディメンジョンテーブルと異なる join キー（外部キー）により join される．このため，map-side join を実行するためには，ファクトテーブルを各ディメンジョンテーブルごとの join キー（外部キー）を用いて，事前にソート・パーティション分割する必要があるが，ディメンジョンテーブル数を d[*2]としファクトテーブルサイズを $|F|$ とした場合，$d|F|$ の空間コストが必要とされるため現実的ではない．

memory-backed join は，各 map タスクにおいて join 対象となる片方のテーブルのパーティションと他方の全テーブルを読み込み，join を実行する手法である．memory-backed join でも join 処理が map タスクで完結するため，シャッフルが不要となるという特徴がある．しかし，join 対象となる片側のテーブルがメモリに載る程度に小さい場合にしか適用できないという問題がある．

対比すると，PJoin は事前にテーブルをソート・パーティション分割する点は，map-side join と同様だが，ファクトテーブルではなくコンパクトな射影ビューを各ディメンジョンテーブルごとに構築するため，空間コストが小さい．詳しくは 4 章の実験において述べるが，実体化にともなう空間のオーバヘッドは 80%程度である．

3. PJoin

3.1 概要

PJoin は MapReduce 環境における高速な join アルゴリズムである．PJoin は以下の前提に基づいて設計されている．1) OLAP 分析応用では，更新操作よりも参照操作の

[*2] TPC-H ベンチマーク [8] では，nation，region のようなサイズが小さくオンメモリに格納可能なテーブルを除けば，$d = 5$ である．

```
SELECT nation, o_year, sum(amount) AS sum_profit
FROM (SELECT n_name as nation,
       extract(year from o_orderdate) AS o_year,
       l_extendedprice * (1 - l_discount) -
       ps_supplycost * l_quantity AS amount
      FROM  part, supplier, lineitem,
            partsupp, orders, nation
      WHERE s_suppkey = l_suppkey
      AND   ps_suppkey = l_suppkey
      AND   ps_partkey = l_partkey
      AND   p_partkey = l_partkey
      AND   o_orderkey = l_orderkey
      AND   s_nationkey = n_nationkey
      AND   p_name like '%[COLOR]%' ) AS profit
GROUP BY nation, o_year,
ORDER BY nation, o_year DESC;
```

図 2　TPC-H ベンチマークにおける Q9
Fig. 2　Q9 in TPC-H benchmark.

頻度が高く，2) OLAP 分析におけるクエリの特徴は，スタースキーマのデータを対象として，多対 1 の基数を持つファクトテーブルとディメンジョン間において join 操作が多用されることである．上記の 2 点に基づいて，PJoin はクエリワークロードに対して事前にデータ処理を行う事前計算フェーズと，実際にクエリ処理を行うクエリ実行フェーズとから構成される．事前計算フェーズでは，ファクトテーブルとディメンジョンテーブル間の semi-join に必要な射影ビューを実体化し，テーブルと実体化ビューをそれぞれ主キーと外部キーでパーティション分割する．クエリ実行フェーズでは，スタースキーマの構造に従ってクエリからスター型の実行プラン（スター型プランと呼ぶ）を導出し，多対 1 の基数を持つテーブル間の多段の join において，1 側のテーブルから多側のテーブルへと処理を行うプランを採用する．このように事前にデータ処理を行うことは OLAP 分析では典型的な方法であり，OLAP の実体化ビューに関する代表的な論文の 1 つ [9] では，クエリワークロードに対して最適な実体化ビューとインデックスを構築するという課題に取り組んでいる．

図 2 は TPC-H ベンチマークのクエリ 9（Q9）であり，Q9 における join は，多対 1 の基数を持つテーブル間の外部キーと主キーの等価性を条件とした join である．**図 3** 左は Q9 がアクセスするスキーマであり，実線の四角はテーブルでありテーブル間の線は多対 1 の基数を表している．図 3 右は Q9 のスター型の実行プランであり，J1，J2，J3 は破線の四角に書かれた複数テーブルの join 処理を表している．これらの join 処理はスター型の外側から内側へ，つまり J1 から J3 への順で実行される．こうしてファクトテーブルである lineitem テーブルを最後に処理することで，中間データサイズが早い段階で大きくなることを避けている．

図 4 の例を用いて，PJoin における複数テーブルの join について説明する．この図は T と S の 2 つのテーブル間

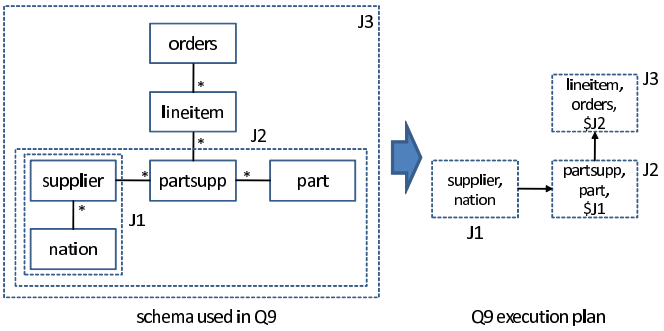


図 3　Q9 のスター型クエリプラン
Fig. 3　Star-shaped execution plan for Q9.

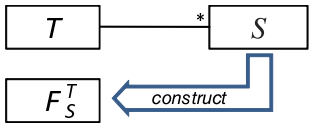


図 4　射影ビューの構築
Fig. 4　Construction of projection view.

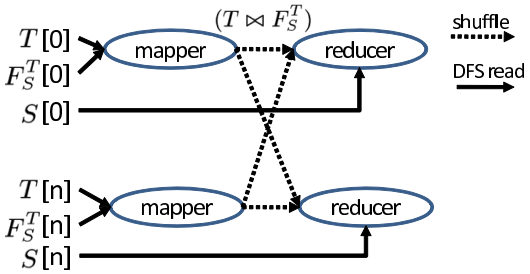


図 5　PJoin のクエリ処理の概要
Fig. 5　Overview of Pjoin.

は 1 対多の基数を持つことを表している．join の例として，T の主キーと S の外部キーの等価性を条件とした join について考えてみよう．事前計算フェーズでは，T の主キーを参照する S の外部キー（$fkey_S^T$ と表記）と，S の主キーとを射影して，実体化ビュー F_S^T を構築する．S，T，F_S^T は，それぞれ S の主キー，T の主キー，外部キー $fkey_S^T$ によってソート・パーティション分割される．

T と S の間の join は，以下の書き換え規則によって処理される．

$$T \bowtie S = (T \ltimes S) \bowtie S \quad (1)$$
$$= (T \bowtie F_S^T) \bowtie S \quad (2)$$

クエリ実行フェーズでは，式 (1) の semi-join が map task において以下のように実行される（**図 5** 参照）．

m1　T の各パーティション $T[i]$（$0 \le i \le$ n）が map task に割り当てられる．

m2　map task では $T[i]$ に対応する F_S^T のパーティション $F_S^T[i]$ を分散ファイルシステムから特定し，$T[i]$ と $F_S^T[i]$ の間で semi-join を実行する．

式 (1) における semi-join 結果と S の間の join は，reduce task において以下のように実行される．

r1　semi-join 結果 $T \bowtie F_S^T$ は S の主キーに応じて reduce task にシャッフルされる．

r2　reduce task ではシャッフルにより得られた $T \bowtie F_S^T$ のパーティション $(T \bowtie F_S^T)[j]$ を読み込み，対応する S のパーティション $S[j]$ を分散ファイルシステムから特定し，$S[j]$ と $(T \bowtie F_S^T)[j]$ の間で join を実行する．

図 1 と図 5 を比較して分かるように，PJoin では S をシャッフルしないためシャッフル量が削減される．これは事前計算フェーズにおいて，射影ビューに S の主キーを保持して，ビューの実体化とパーティション分割していることで実現されている．

3.2　事前計算フェーズ

事前計算フェーズでは，ファクトテーブルとディメンジョンテーブル間の semi-join に必要な射影ビューを実体化し，テーブルと実体化ビューをそれぞれ主キーと外部キーでパーティション分割する．射影ビューは以下のように定義される．

定義 3.1（射影ビュー）　T と S はテーブルであり，T の主キーを参照する S の外部キー（$fkey_S^T$）によって，テーブル間は 1 対多の基数を持つとする．T に対する S の射影ビュー（F_S^T と表記）の構造は（$fkey_S^T, pkey_S$）である．ただし，$pkey_S$ は S の主キーである．

通常の semi-join の手法では S の外部キーだけを射影して T と semi-join を実行するが，PJoin では射影ビューに外部キーだけではなく S の主キー $pkey_S$ を含めるよう拡張している．これは，$pkey_S$ をキーとして semi-join 結果 $(T \bowtie F_S^T)$ をシャッフルするためである．シャッフル後の処理については，3.2.1 項で後述する．

3.2.1　事前パーティション分割

シャッフルすることなく T と S のテーブル間で semi-join を実行するため，事前に T の主キー $pkey_T$ を分割キーとして T をソート・パーティション分割し，T の主キーを参照する S の外部キー $fkey_S^T$ をパーティション分割キーとして射影ビュー F_S^T をソート・パーティション分割する．この事前パーティション分割の処理は，パーティション分割キーを map 関数の出力キーとする MapReduce ジョブで容易に実現される．

定義 3.2（パーティション分割）　テーブル T は $\cup_{i=0}^{n} T[i]$ としてパーティション分割され，$i = \mathrm{hash}(pkey_T)$ として i は計算される．射影ビュー F_S^T は $\cup_{i=0}^{n} F_S^T[i]$ としてパーティション分割され，$i = \mathrm{hash}(fkey_S^T)$ として i は計算される．

T と S の semi-join 処理は，T の主キー $pkey_T$ と射影ビュー F_S^T が保持する外部キー $fkey_S^T$ との等価性を条件

として処理されるため，上記のパーティション分割の定義に従って，以下のようにパーティションを用いたsemi-join処理に書き換えることができる．

$$\begin{aligned} T \ltimes S &= T \bowtie F_S^T \\ &= \cup_{i=0}^{n} T[i] \bowtie \cup_{i=0}^{n} F_S^T[i] \\ &= \cup_{i=0}^{n} (T[i] \bowtie F_S^T[i]) \end{aligned} \tag{3}$$

3.2.2　射影ビューの拡張

テーブル間に1対多の基数を持つTとSのjoinを含むクエリであって，そのクエリがSに関するWHERE条件を有する場合，このWHERE条件はSのパーティションがreduce taskで読み込まれた後に処理される．PJoinではシャッフル量をさらに削減するため，Sに関するWHERE条件対象のカラムを射影ビューに加える．こうすることで，Sに関するWHERE条件の処理はreduce taskからmap taskにプッシュダウンされ，map taskでは実体化した射影ビューを読み込んだ際に実行される．

この射影ビューの拡張の処理は，リレーショナルデータベースにおけるインデックス構築・実体化ビュー構築[9]と同様に，クエリ実行前にクエリワークロードを解析して事前に実行する．射影ビューの拡張はストレージを消費するが，4章に述べるように，TPC-Hにおける22のクエリでは拡張された射影ビューの総サイズは，テーブルの総サイズの80.4%程度であることを確認している．

3.3　クエリ実行フェーズ

本節では，最初にクエリを構成する複数のjoinの実行順序について説明し，その後クエリ処理のコストについて説明する．

3.3.1　joinの実行順序

アルゴリズム1に示す疑似コードを用いて，主キーと外部キーの等価性を条件とするjoinが複数含まれるクエリの実行プランについて説明する．スタースキーマでは，ファクトテーブルを複数のディメンジョンテーブルが囲む構成であり，ディメンジョンテーブルはファクトテーブルあるいは別のディメンジョンテーブルとの間で1対多の基数を持つ．このスタースキーマの構造に従って，クエリQからツリー型の構造を有する実行プラン*execPlan*は次のように導出される（1行目）．実行プランを構成する中心ノードはファクトテーブルに対応して構築され，ファクトテーブルと多対1の基数を持つディメンジョンテーブルごとに，中心ノードの子ノードが構築される（図3右の例では，J2に該当する）．実行プランにおいて，中心ノードはjoinビューを表しており（図3右の例では，J3に該当する），ファクトテーブルとその周囲の複数のディメンジョンテーブル群をjoin処理する操作を表す．子ノードは，構築元であるディメンジョンテーブルとその周囲の複数のディメンジョンテーブルのjoinビューを表す．ノード構築の操作はスタースキーマにおいて，クエリが参照する最も外側のテーブルに到達するまで実行される（図3右の例では，J1に該当する）．

ツリー型の実行プランを構成する各ノードは，joinビューを表現するため以下の情報を格納する．

(1) 本ノードを構築する元となったテーブルSへの参照．
(2) クエリQのサブクエリであり，Sに関する処理（join，select，projection）をすべて含むサブクエリ．なお，Sとのjoin対象のテーブル集合を$Tset$とする．
(3) 上記サブクエリの実行結果．

各ノードにおけるSと$Tset$の多テーブルjoin処理を説明するため，まず各テーブル$T \in Tset$とSの2テーブルjoinを説明する．TはSと1対多の基数を有するため，定義3.2に従ってSおよびTに対するSの射影ビューをパーティション分割しておくことで，TとSのjoin処理は式(1)，(2)，(3)を適用して計算することができる．式(1)右辺のsemi-joinの処理は式(3)に展開され，map taskにおいて次のように処理される．$T[i]$がmap taskに入力され，map関数において$T[i]$に対応する$F_S^T[i]$を分散ファイルシステムから特定し，これらの間でsemi-joinを実行する．そして$pkey_S$をキーとしてシャッフルを実行する．semi-join結果$T \ltimes S$は$\cup_{i=0}^{n}(T \ltimes S)[i]$としてパーティション分割され，ただし$i = \mathrm{hash}(pkey_S)$として$i$は計算される．次に式(1)右辺のjoinの処理は，reduce taskにおいて次のように処理される．$(T \ltimes S)[i]$がシャッフルにより入力され，対応するSのパーティション$S[i]$を分散ファイルシステムから特定し，$(T \ltimes S)[i]$と$S[i]$のjoinを実行する[*3]．このようにして各$T \in Tset$とSの2テーブルjoin処理を実行することができる．

次に，Sと$Tset$の多テーブルjoin処理を説明する．上記のreduce taskでは，すべての$T \in Tset$とSとの多テーブルjoinがSの主キーをキーとして実行されることに注意されたい．各$T \in Tset$ごとにmap taskの処理は独立に実行しなければならないが，reduce taskはTに依存しないSの主キーをキーとしてjoin実行されるため，すべての$T \in Tset$とSとの多テーブルjoinは1つのreduce taskで実行可能である．つまり，以下の補題を得る．

補題3.3（多テーブルjoin）　$Tset$をSと1対多の基数を有するテーブル群とする．任意のテーブル$T \in Tset$とSの組について，定義3.1に従って射影ビューが実体化され，定義3.2に従って射影ビュー，T，Sがパーティション分割されているならば，$Tset$とSの多テーブルjoinは1つのMapReduceジョブで実行可能である．

[*3] もし射影ビューが$pkey_S$を含まない設計をした場合，semi-join結果サイズを縮小できるが，join対象のSのパーティションを特定できなくなり，すべてのreduce taskにブロードキャストしなければならず，マシン台数に対してスケールしないという問題が生じてしまう．

Algorithm 1 execQuery(query Q)

Ensure: *queryResult*, result of query Q
1: *execPlan* = construct star-shaped execution plan from Q;
2: *center* = get the center node in *execPlan*;
3: *level* = get the maximum radius in *execPlan*;
4: **while** *level* ≥ 1 **do**
5: **for all** $n \in$ collect nodes in *execPlan level* away from *center* **do**
6: *joinResult* = multiwayJoin(n);
7: **if** *level* == 1 **then**
8: *queryResult* = *joinResult*;
9: **return** *queryResult*;
10: **else**
11: associate *joinResult* with n;
12: **end if**
13: **end for**
14: *level* = *level* - 1;
15: **end while**

Algorithm 2 multiwayJoin(node n)

Require: *star-schema*
Ensure: *joinResult*, result of multiway join at n
1: *many-side* = get the table in *star-schema* from which n is created;
2: *join-tables* = {};
3: **for all** *one-side* in outer tables of *many-side* in the star-schema **do**
4: **if** n is a leaf node **then**
5: *temp-table* = read table of *one-side* and apply select conditions;
6: **else**
7: *temp-table* = read *join-tables* of n;
8: **end if**
9: append *temp-table* to *join-tables*;
10: **end for**
11: *joinResult* = execute multiway join between *many-side* and *join-tables*;
12: *joinResult* = execute ad-hoc join on *joinResult*;

多テーブル join 処理に関してこの補題が示すことは，reduce-side join と比較して PJoin は MapReduce のジョブ数を削減できることである．対比的に reduce-side join では，T と S の join 条件は T の主キーであるため，異なるテーブル $T \in Tset$ ごとに reduce-side join を行う MapReduce ジョブが必要である．

次に，スター型の実行プランにおける join の実行順について，PJoin ではスター型プランの外側から開始し，最後に中心のノードのビューを実行する．これは，データ量が大きいファクトテーブルの処理を最終段階で実行すること，中間データサイズが早い段階で大きくなることを避けるためである．しかし，クエリに指定された select 処理の条件によっては，ファクトテーブルから join を開始した方が性能が良い場合もありうる．このような最適化はコストベースの手法が必要になるが，これは今後の課題とする．

アルゴリズム 1 は PJoin における join の実行順序を決定する疑似コードである．スター型プラン *execPlan* において，最も外側のノードから 1 レベルずつ内側のノードへと処理を移動して（5–14 行目）．複数テーブル join（multiwayJoin）を実行する（6 行目）．処理対象のノード n が中心ノードの場合は multiwayJoin の結果を返却して処理を完了し（7–9 行目），そうでない場合は後続の処理のため処理結果をノード n と関係づける（10，11 行目）．multiwayJoin の疑似コードをアルゴリズム 2 に示す．

スター型プラン *execPlan* の処理対象のノード n がリーフノードである場合，n を構築する元となったテーブル *many-side* を取得し（1 行目），*many-side* と多対 1 の基数を持つテーブル *one-side* ごとに，select 処理を実行して（5 行目），処理結果を *join-tables* に追加する（9 行目）．テーブル *many-side* と select 処理実行後のテーブル群 *join-tables* の間で，補題 3.3 に基づいて多テーブル join を実行する（11 行目）．もし，主キーと外部キーの等価性を条件とする join 以外のアドホック join がある場合は，それを実行する（12 行目）．n がリーフノードでない場合も同様だが，この場合はクエリの処理結果 *joinResult* が計算済みであるため，それを再利用する（7 行目）．

3.3.2　一般的な join

PJoin では，join 対象の片方のテーブルがメモリに載る程度に小さい場合に，memory-backed join [4], [6] を利用する．具体的には，ChainMapper クラスを用いて memory-backed join の map 処理と semi-join の map 処理を結合して 1 つの map task で実行する．

クエリが，主キーと外部キーの等価性を条件とする join 以外のアドホック join を含む場合，この join については reduce-side join など通常の join 方式を利用する．ただし，同一のテーブルに対して，主キーと外部キーの等価性を条件とする join とアドホック join が同時に指定されている場合は，アドホック join を後に実行することとする．これは，アドホック join の方が，主キーと外部キーの等価性を条件とする join よりも一般的に高価であるためである．

3.4　例

図 2 の Q9 を用いて，PJoin アルゴリズムを説明する．TPC-H ベンチマークでは nation テーブルはレコード数が少ない設計であるため，nation テーブルの join では memory-backed join を利用する．

事前計算フェーズ：nation テーブルを除いて，PJoin では図 3 左図にある各テーブルを主キーでパーティション分割する．また，他のテーブルと多対 1 の基数を持つテーブルに関して，射影ビューを実体化・外部キーでパーティション分割する．たとえば，lineitem テーブルと多対 1 の基数を持つ order テーブルに関して，射影ビュー $F^{order}_{lineitem}$ を実体化する．$F^{order}_{lineitem}$ は，lineitem テーブルの主キー，order テーブルに対する外部キー orderkey を属性として有する．order テーブルと $F^{order}_{lineitem}$ は同一の orderkey でパーティ

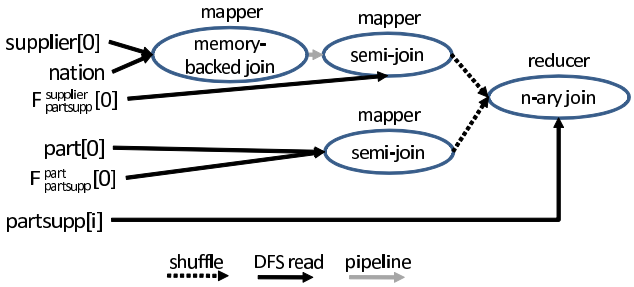


図 6　MapReduce における Q9 の join の流れ
Fig. 6　Join flow of Q9 on MapReduce.

ション分割されているため，シャッフルすることなく join を実行することができる．

クエリ実行フェーズ：Q9 はアルゴリズム 1 を用いて以下のように処理される．最初に，スタースキーマのテーブル間の構造に従って，図 3 右に示されるスター型の実行プランが得られる．lineitem はファクトテーブルであるため，join 処理は J1 = supplier ⋈ nation，J2 = partsupp ⋈ part ⋈ $J1，J3 = orders ⋈ lineitem ⋈ $J2 と割り当てられる．ただし，$J1 と $J2 はそれぞれ J1 と J2 のクエリ結果を表すこととする．nation テーブルはレコード数が少ないため，J1 は memory-backed join を用いて実現され，さらに J1 と J2 は 1 つの MapReduce ジョブに統合される．J3 は単独で 1 つの MapReduce ジョブで実現される．Q9 に含まれる GROUP BY および ORDER BY 操作は，それぞれ別の MapReduce ジョブとして実現され，結局 Q9 はトータルで 4 つの MapReduce ジョブで実行される．join の実行順序については，スター型の実行プランの外側から実行されるため，J1 と J2 の MapReduce ジョブを実行後に，J3 の MapReduce ジョブが実行される．

図 6 は J1 と J2 を実行する MapReduce ジョブの MapReduce レベルの実行フローを表している．この例は，4 つのテーブル（nation，supplier，part，partsupp）の join が 1 つの MapReduce ジョブで実現される例である．ここで，$F^{suplier}_{partsupp}$ と $F^{part}_{partsupp}$ をそれぞれ partsupp に対する supplier と part の射影ビューとする．図では 3 つの種類の map task がある．1 つ目は memory-backed join を実行する map task であり，supplier テーブルのパーティションと nation テーブル全体を join する．2 つ目の map task は，$F^{suplier}_{partsupp}$ のパーティションを用いて，1 つ目の map task の出力と partsupp の間の semi-join を実行する．3 つ目の map task は，$F^{part}_{partsupp}$ のパーティションを用いて，part と partsupp の間の semi-join を実行する．Q9 では part テーブルに対する select 処理があり（p_name like '%[COLOR]%'），part テーブルが読み込まれた後に select 処理が実行される．2 つ目と 3 つ目の map task は，両方とも partsupp テーブルの主キーをキーとしてシャッフルされ，reducer task においてシャッフル結果と partsupp のパーティションが join される．

表 1　PJoin/reduce-side join のシャッフル量/分散ファイル参照サイズ
Table 1　Shuffle and DFS read size of PJoin and reduce-side join.

	PJoin	reduce-side join
shuffle size	$\lvert\sigma_{ts}(\sigma_t(T) \bowtie \sigma_s(F^T_S))\rvert$	$\lvert\sigma_t(T)\rvert+\lvert\sigma_s(S)\rvert$
DFS read	$\lvert T\rvert + \lvert S\rvert + \lvert F^T_S\rvert$	$\lvert T\rvert + \lvert S\rvert$

3.5　コスト分析

2 テーブル join を対象として，シャッフルコストと分散ファイルシステムに対する参照コストの観点から，PJoin と reduce-side join を比較する．この分析結果を多テーブル join に拡張するのは容易であるため，ここでは省略する．3.1 節で述べた，T と S の 2 テーブルを join する例に戻ってみよう．ここで，σ_t，σ_s，σ_{ts} を，それぞれ T，S，$T \bowtie S$ に対する select 操作であるとする[*4]．reduce-side join のシャッフル量は以下の式で与えられる．

$$|\sigma_t(T)| + |\sigma_s(S)|$$

これは，T と S の select 操作の結果がシャッフルされるためである．一方，PJoin のシャッフル量は以下の式で与えられる．

$$|\sigma_{ts}(\sigma_t(T) \bowtie \sigma_s(F^T_S))|$$

これは，map task において T と S の select 操作の結果に対して semi-join が実行され，その後 σ_{ts} が実行されるためである．このため，PJoin の効果は semi-join と σ_{ts} の選択率に依存する．最悪のケースでは，semi-join と σ_{ts} がレコードをフィルタしないケースであり，こういうケースでは PJoin は reduce-side join よりシャッフル量が増加してしまう．現実的に最悪のケースが生じるのは，T と S がクエリの処理結果ではないオリジナルのテーブルの場合であり，かつ select 操作がクエリで指定されていない特殊な場合のみである．一般的には，多テーブルを join する場合は select 操作が指定されたテーブルがあるため，こういうケースでは PJoin は効果があると考えられる．

分散ファイルシステムに対する参照コストは簡単である．T と S を join する場合，reduce-side join のコストは $|T|+|S|$ であり，PJoin はさらに射影ビュー F^T_S を参照する必要があるため，コストは $|T| + |S| + |F^T_S|$ となる．ここまでの結果を**表 1** にまとめた．

4.　性能評価

本章では，TPC-H ベンチマークを用いて PJoin, MapReduce の典型的な join である reduce-side join，および文献 [10] の per-split semi-join の性能を比較する．文献 [10]

[*4] σ_{ts} は，T と T に対する条件を選言的に結合した条件などのように，T と S を join した結果に対して実行するべき select 操作である．

では 2 テーブル join の高速化手法を複数提案しているが，per-split semi-join は semi-join を利用していて，かつ事前計算と併用することで高速であることが報告されているため，評価対象とした．

ハードウェア：性能評価では 50 台のマシンから構成されるクラスタ環境を用いた．各マシンの性能は，CPU は 2.80 GHz Intel Core 2 Duo プロセッサ，OS は 64-bit CentOS 5.4（kernel version 2.6.18），メモリは 8 GB で，ハードディスクは 1 TB の SATA である．ハードディスクの性能を hdparm を用いて計測したところ，キャッシュからの読み込みは 3.7 GB/sec で，バッファからの読み込みは 97 MB/sec であった．マシンは gigabit Ethernet によって下位スイッチ（SR-S324，SR-S348）と接続されていて，下位スイッチは 20 gigabit で上位スイッチ（SR-X526）と接続されている．

ソフトウェア：Sun JDK 1.6.22 を用いて，PJoin，reduce-side join，per-split semi-join を Hadoop 0.21.0 上で実装した．Hadoop では，job tracker が MapReduce を統括し，NameNode が分散ファイルシステム全体を統括する．これらがシステム全体の性能に悪影響を及ぼさないよう，この 2 つの統括プロセス専用のマシンを用意した（残り 49 台で分散処理を実行した）．各種パラメータは標準的に用いられているチューニング方法 [11] に従って以下のように設定した．1) ソートバッファサイズは 256 MB（デフォルトは 100 MB），2) task executor の JVM のヒープサイズは 3,072 MB，job tracker の JVM の最大ヒープサイズは 4,064 MB，NameNode/DataNode/TaskTracker の JVM の最大ヒープサイズは 1,024 MB，3) Hadoop の rack awareness を有効化．加えて，各マシンはコア数が 2 であるため，map task および reduce task の起動数をマシンごとに 2 に設定した．

分散ファイルシステムとして HDFS を使用した．HDFS に関してもチューニング方法 [11] に従って，以下のように設定した．1) データのブロックサイズは 256 MB（デフォルトは 64 MB），2) レプリカ数は 1（デフォルトは 3），3) 圧縮オプションを使用しない．さらに，OLAP クエリを高速に処理するため，バイナリのカラムストア形式 [12] を利用することで，PJoin および reduce-side join における無駄なカラム参照を排除した．また，PJoin に関しては 1 パーティションが 1 コアに割り当てられるよう，テーブルのパーティション分割数を 98 とした．

4.1　ワークロード

ベンチマークは OLAP 分析の標準である TPC-H [8] を利用した．TPC-H のスキーマは，1 つのファクトテーブルと 7 個のディメンジョンテーブルから構成される．レコード数が少ない nation と region テーブルについては memory-backed join を使用し，この 2 テーブルを除いたディメンジョンテーブルに関して射影ビューを実体化した．具体的には，次の 7 個である．$F^{suplier}_{partsupp}$，$F^{part}_{partsupp}$，$F^{customer}_{orders}$，$F^{part}_{lineitem}$，$F^{supplier}_{lineitem}$，$F^{partsupp}_{lineitem}$，$F^{orders}_{lineitem}$．TPC-H のデータセットサイズを規定するスケールファクタ（SF）は 100，200，300 とし，データの統計情報および事前計算フェーズにおけるビューの構築時間を**表 2** に記載する．表の第 3 カラムに記載のとおり，スケールファクタによらず実体化した射影ビューによるオーバヘッドは 80.4%であり（つまり，射影ビューを実体化することでデータサイズが 1.8 倍になった），インデックスサイズとしては許容範囲のオーバヘッドであると考えられる．射影ビューのサイズの内訳に関して，3.2.2 項で述べた射影ビューの拡張による増分の割合はスケールファクタによらず 62.4%であった．また，ビュー構築時間に関する議論については 4.2.8 項で後述する．

表 2　データセットの統計情報およびビュー構築時間
Table 2　Statistics of datasets and time of view construction.

Scale factor	Base table size	Projection view size	Foreign key table/ Base table	View construction
100	104 GB	83 GB	80.4%	19.2 min
200	207 GB	167 GB	80.4%	33.2 min
300	311 GB	250 GB	80.4%	49.7 min

クエリについては，PJoin と reduce-side join との比較では TPC-H の 17 個のクエリを使用した．ただし，join を含まないクエリ Q1，Q6，Q22 は評価対象から排除し，入れ子クエリで多側のテーブルから 1 側のテーブルへ join 処理が必要なクエリ Q4，Q15 は，スター型の実行プランの join 実行順序が適用できないため排除した．一方，PJoin と per-split semi-join [10] との比較では，ファクトテーブルとの 2 テーブル join を行う，Q12，Q14，Q17，Q19 のクエリを使用した．これらを選択した理由は，文献 [10] で提案されている join アルゴリズム群が大規模なテーブルと小規模なテーブルの 2 テーブルを join する処理を対象として設計されているためである．

性能評価にあたって，reduce-side join については 2 種類の異なる実行順序のパターンを用意した．1 つ目は PJoin のスター型プランと同じ順序で reduce-side join を行うプラン（reduce-side (star plan)）である．ただし，3 以上のテーブルを同時に join 可能な場合は，join を 2 テーブル join に分解し，分解した join の中からテーブルサイズが小さい順に実行することで中間データサイズを削減することとした．PJoin と reduce-side join (star plan) を比較することで，PJoin の射影ビューの実体化およびパーティション分割の効果を測定する．2 つ目は人手で join の実行順序を設計したプラン [13]（reduce-side (Hive plan)）である[*5]．reduce-side join (star plan) と reduce-side join (Hive plan)

[*5] 文献 [13] では Hive を用いているが，本論文では reduce-side (star plan) と同じ実装を用いて，実行順序だけを変更した．

を比較することで，スター型の実行プランの効果を測定する．

4.2　評価結果

評価観点として，MapReduce ジョブ数，クエリ実行フェーズにおける応答時間，シャッフル量[*6]，HDFS の書き込み/参照データサイズを計測した．PJoin，reduce-side join (star plan)，reduce-side join (Hive plan)，per-split semi-join は，各クエリを複数の MapReduce ジョブを導出して実行するが，アルゴリズムの特徴を明らかにするため，これらのアルゴリズムから導出される共通の MapReduce ジョブ（GROUP BY，ORDER BY，入れ子クエリを処理するジョブなど）は計測対象から排除した．

4.2.1　MapReduce ジョブ数

クエリに関する統計情報を**表 3** に示す．ジョブ数に関しては，per-split semi-join は reduce-side join と同じであるため，表 3 での掲載を省略する．PJoin による MapReduce ジョブ数の全クエリに対する総計は，reduce-side join (star plan) より 29.4%少なく，reduce-side join (Hive plan) より 45.5%少ない．これは，PJoin では複数テーブル join を 1 つの MapReduce ジョブで実行できるのに対して，reduce-side join では 2 テーブル join で 1 つの MapReduce ジョブを必要とするためである．たとえば，PJoin は Q8 と Q9 において，それぞれ 4 テーブル join と 3 テーブル join を行うことで，reduce-side join (star plan) よりも MapReduce ジョブを 2 個削減している．また，reduce-side join (Hive plan) よりも reduce-side join (star plan) の方がジョブ数が少ない．これは，Hive plan が人手によるプランであるため，効率が良くないことが原因である．

4.2.2　PJoin と reduce-side join の応答時間

図 7 (a) に PJoin と reduce-side join のクエリ実行フェーズにおける応答時間の測定結果を示す．Q13 を除いて PJoin が 2 つの reduce-side join より高速あるいは同等であることが確認できる．全クエリの応答時間の総和を比較すると，PJoin は reduce-side join (star plan) より 33.9%高速であり，reduce-side join (Hive plan) より 42.8%高速であった．詳しく分析すると，PJoin が優れる理由は 2 点あげられる．

- PJoin では，MapReduce のジョブ数およびシャッフル量が削減されるため（後続の 4.2.4 項で詳述する），応答性能が改善されている．MapReduce のジョブ数削減によって，全体的に HDFS の書き込みデータサイズも削減されている（後続の 4.2.5 項で詳述する）．
- 射影ビューの導入の影響のため，PJoin では HDFS の参照データサイズが増加しているが，参照処理であるため性能への影響は大きくなかった（後続の 4.2.6 項で詳述する）．

[*6] シャッフル量は，Hadoop のプロファイル機能を用いて計測した．

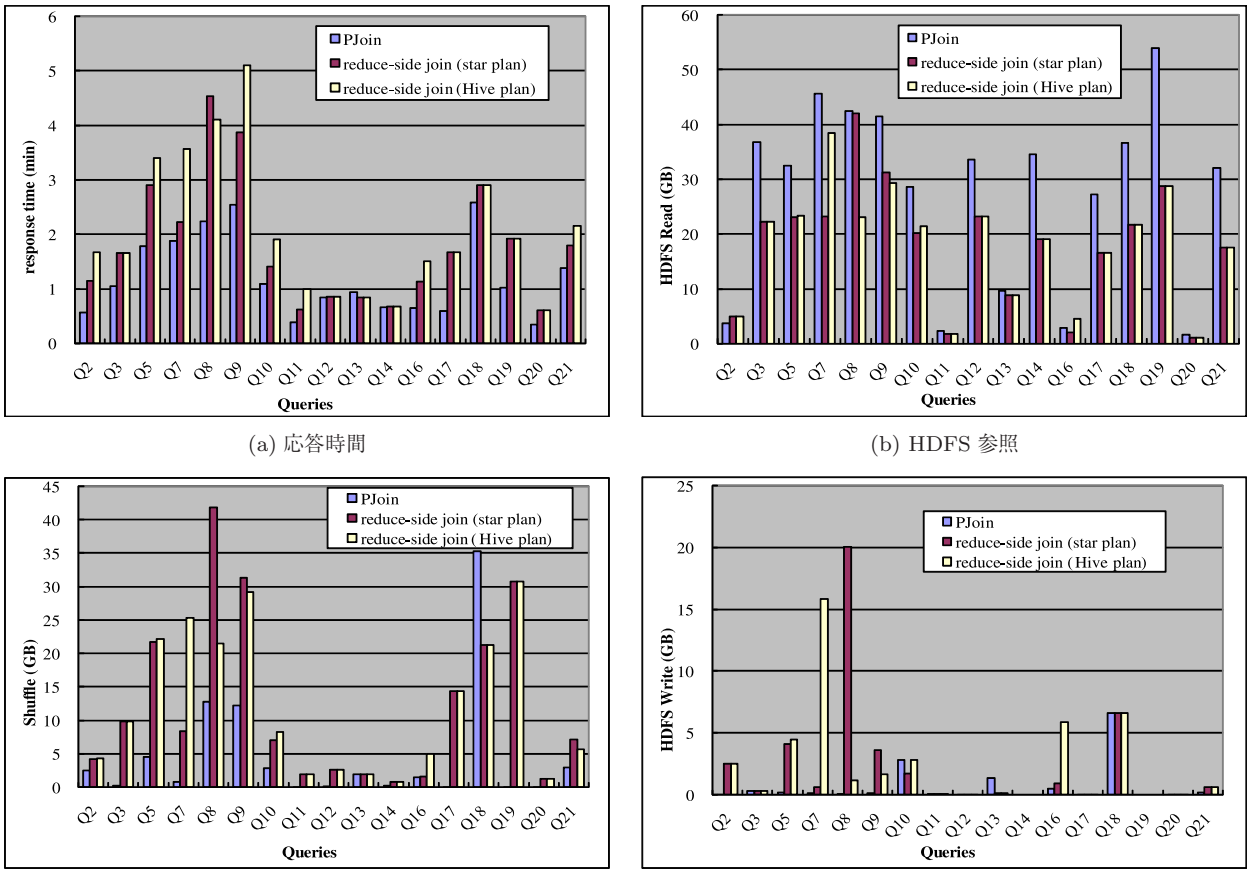
表 3　MapReduce ジョブ数（↓ は右隣りの値に対する削減を意味する）

Table 3　Number of MapReduce jobs (↓ indicates decrease to the right column).

Query	# of join tables	# of MapReduce jobs		
		PJoin	reduce-side join	
			star plan	Hive plan
Q2	4	1 (↓)	2 (↓)	3
Q3	3	2	2	2
Q5	5	2 (↓)	3 (↓)	4
Q7	5	2 (↓)	3 (↓)	4
Q8	7	2 (↓)	4 (↓)	6
Q9	6	2 (↓)	4 (↓)	5
Q10	4	2	2 (↓)	3
Q11	3	1	1 (↓)	2
Q12	2	1	1	1
Q13	2	1	1	1
Q14	2	1	1	1
Q16	3	1 (↓)	2 (↓)	3
Q17	2	1	1	1
Q18	4	2 (↓)	3	3
Q19	2	1	1	1
Q20	2	1	1	1
Q21	4	1 (↓)	2 (↓)	3
total	60	24 (↓)	34 (↓)	44

Q13 では，応答時間に関して reduce-side join よりも PJoin の方が 11.9%遅い結果になったが，この原因は 2 点あげられる．1 つ目の理由は，Q13 は 2 テーブル join のクエリであるため，PJoin の多テーブル join の効果が得られていないことがあげられる．2 つ目の理由は，Q13 の GROUP BY 操作が原因である．PJoin では多側のテーブルの主キーでシャッフルを行うが，reduce-side join による処理では join キーでシャッフルを行う特徴がある．Q13 の場合，join キーと GROUP BY キーが偶然同一であったことから，reduce-side join では GROUP BY の処理が効率的に実行できたのに対して，PJoin ではシャッフルキーと GROUP BY キーが異なったことから性能が劣化してしまった．この振舞いは，HDFS 書き込みサイズの差として表れていて，Q13 における PJoin の HDFS 書き込みデータサイズは 1,373 MB であり，reduce-side join の 135 MB と比較して 10 倍になってしまっている．

reduce-side join における star plan と Hive plan の比較：図 7 (a) が示すとおり，reduce-side join (star plan) の応答性能は，Q8 を除いて reduce-side join (Hive plan) 以上に高速な結果を得た．この結果から，多テーブルを join する場合において，最もサイズが大きいファクトテーブルをクエリ処理の最終段階で処理するべきであることが正しいことが確認できた．たとえば，Q7 は顕著な例であり，HDFS 参照/書き込みデータサイズおよびシャッフル量とも reduce-side join (star plan) の方が大きく削減できてい



(a) 応答時間　(b) HDFS 参照

(c) シャッフル　(d) HDFS 書き込み

図 7　TPC-H の評価結果（SF=100，50 nodes）
Fig. 7　Results of TPC-H (SF=100, 50 nodes).

る．以下は，star plan と Hive plan の join の実行順を表している（括弧は 1 MapReduce ジョブを表している）．

Q7 の Reduce-side join (Hive plan)

((((lineitem ⋈ orders) ⋈ customer) ⋈ supplier) ⋈ nation)

Q7 の Reduce-side join (star plan)

((nation ⋈ customer ⋈ orders) ⋈ (nation ⋈ supplier ⋈ lineitem))

Hive plan ではファクトテーブルの lineitem が最初に処理されており，この影響で後続の join において中間データサイズが大きいという問題が生じている．Hive plan は人手によって最適化されているが，特に入れ子クエリについては単純に MapReduce ジョブに変換されているため，効率が良くないケースがあるといえる．

逆に，star plan が良くない唯一のケースとして Q8 があった．Q8 ではシャッフルおよび HDFS 書き込みデータサイズに関して，star plan が Hive plan より増加してしまっていた．Q8 の star plan の join の実行順は以下のとおりであり，これは最適ではなかった．

(((lineitem ⋈ supplier) ⋈ part) ⋈ (orders ⋈ customer))

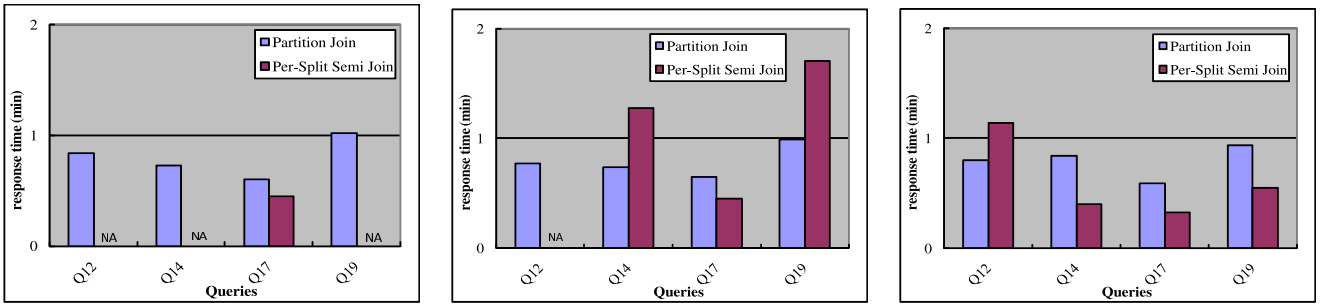
star plan ではテーブルサイズの小さい順に join を実行していたが，Q8 のケースでは part と orders テーブルに対して select 処理が指定されていたため，テーブルサイズが supplier の方が大きくなってしまったのが原因である．追加実験を行い，すべての join の実行順を実行した結果，select 処理後のテーブルサイズの小さい順に join を実行するプランが最も高速であることが確認された．このプランは以下の Hive プランと同一であった．

(((lineitem ⋈ (orders ⋈ customer)) ⋈ part) ⋈ supplier)

4.2.3　PJoin と per-split semi-join の応答時間

図 8 に PJoin と per-split sem-join のクエリ実行フェーズにおける応答時間の測定結果を示す．TPC-H のデータセットはスケールファクタ 100 としたが，per-split semi-join の文献 [10] に合わせて，join の際のファクトテーブルからディメンジョンテーブルへの参照率が 10%と 1%のデータセットも用意して評価を行った（オリジナルの TPC-H データは参照率 100%と記載した）．全体的な傾向としては，PJoin は大規模データに適した設計であり適用範囲が広い反面，小規模データの際にはオンメモリでハッシュ join を行う per-split semi-join の方が高速である結果となった．

以下，クエリごとに測定結果を分析する．Q12 は lineitem（レコード数 600 M 件）と orders（レコード数 150 M 件）を join するクエリである．実験の結果，join の際のディ



(a) ディメンジョンテーブルの参照率 100%　(b) ディメンジョンテーブルの参照率 10%　(c) ディメンジョンテーブルの参照率 1%

図 8　PJoin と per-split semi-join の応答時間（NA はメモリ枯渇により計測不能を意味する）

Fig. 8　Response time of PJoin and per-split semi-join (NA indicates failure of experiment caused by running out of memory).

メンジョンテーブルのフィルタ率が 100%，10%の際には per-split semi-join はメモリ枯渇が発生して動作せず，1%の際には Java の GC（ガベージコレクション）が悪影響して PJoin よりも 70%遅い結果となった．orders のレコード数が多いため per-split semi-join より PJoin が優れた結果となった．Q14，Q19 は lineitem と part（レコード数 20 M 件）を join するクエリであり，orders よりもレコード数が少ないためメモリ量を必要としないという特徴がある．実験の結果，join の際のディメンジョンテーブルのフィルタ率が 100%の際には per-split semi-join はメモリ枯渇が発生して動作せず，10%の際には GC が悪影響して PJoin よりも 73%（Q14，Q19）遅く，1%の際には PJoin よりも 52%（Q14），41%（Q19）高速な結果となった．Q17 は lineitem テーブルと part テーブルを join するクエリであり，part テーブルに対する WHERE 条件の絞り込みが効きやすいという特徴を有する．実験の結果，join の際のディメンジョンテーブルのフィルタ率に応じて per-split semi-join は PJoin より 25%（フィルタ率 100%），31%（フィルタ率 10%），44%（フィルタ率 1%）高速な結果となった．

以上では 2 テーブル join のクエリについて，PJoin と per-split semi-join を比較したが，多テーブル join のクエリの場合は PJoin により MapReduce のジョブ数削減の効果があるため，per-split semi-join よりも改善が見込めると考えられる．

4.2.4　シャッフル

図 7 (c) にシャッフル量の測定結果を示す．PJoin が 2 つの reduce-side join より約 1/3 と大幅にシャッフル量を削減していることが確認できる．このような結果は以下の典型的な 2 つのケースによるものである．

(1) 特にファクトテーブルを対象とする join について，PJoin では多テーブル join を 1 MapReduce ジョブで実行している，

(2) 2 テーブル join のケースであっても，PJoin では多側のテーブルのシャッフルを行わない．特にファクトテーブルのケースの効果が大きい．

以下に詳しく分析する．

最初のケースの多テーブル join は Q5，Q7，Q8，Q9，Q21 で行われていて，PJoin により実際シャッフル量が削減されている．たとえば，Q7 の PJoin プランは，以下のようになる．

Q7 の PJoin プラン

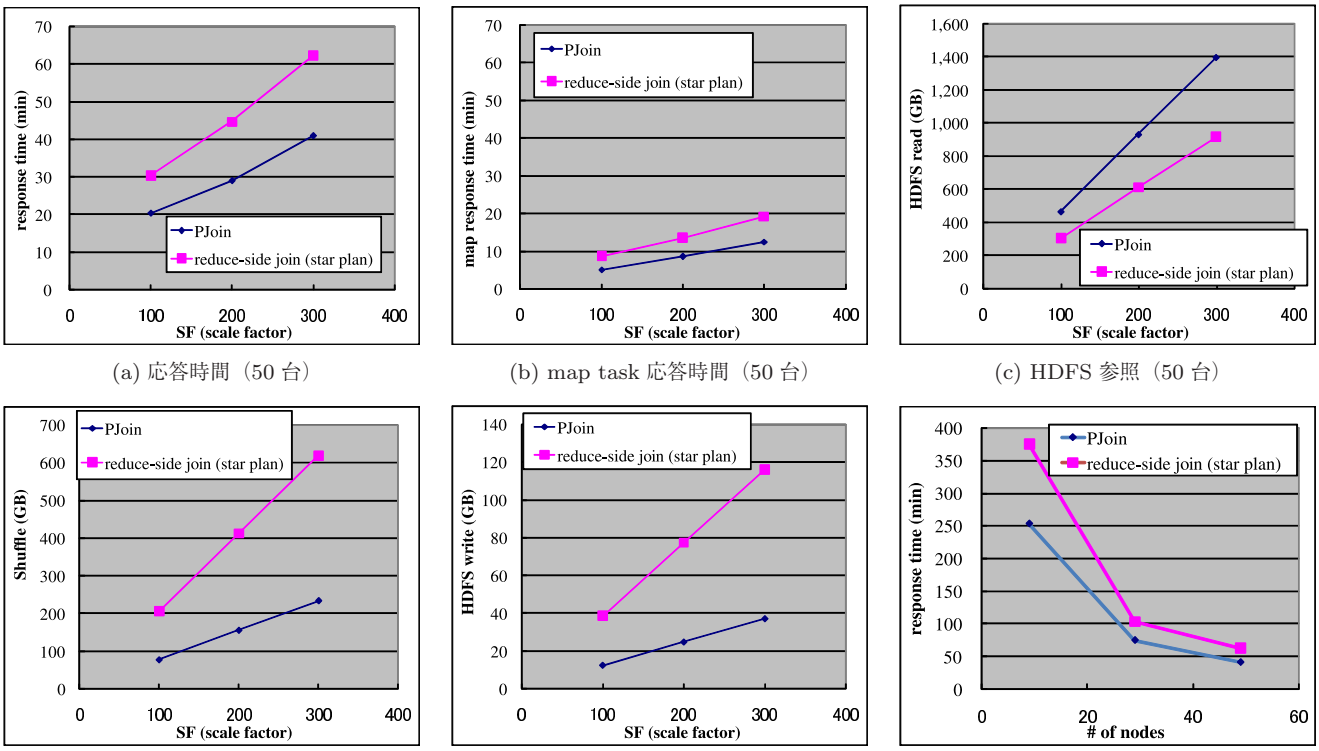
((nation ⋈ customer ⋈ orders) ⋈ nation ⋈ supplier ⋈ lineitem)

Q7 の reduce-side join (star plan) の join の実行順と比較すると，reduce-side join (star plan) では lineitem を含む多テーブル join を 2 テーブル join に分解しているのに対して（nation の memory-backed join は除く），PJoin は 3 テーブル join をクエリ処理の最終段階で実行している．2 つ目のケースのファクトテーブルに対する 2 テーブル join は Q3，Q12，Q17，Q19 で行われていて，PJoin はファクトテーブルのシャッフルを回避することで，reduce-side join よりシャッフル量が 1 桁削減されている．逆に，Q18 では PJoin の場合の方がシャッフル量が増加してしまっている．このクエリでは，PJoin は 3 テーブル join を実行しているが，ファクトテーブルを含む 2 テーブルで select 処理が指定されていない．一方，reduce-side join では select 処理を含む 2 テーブル join を先に実行しているため，シャッフル量が削減されている[*7]．ただし Q18 の場合であっても，PJoin によって MapReduce のジョブ数が削減されることで，reduce-side join よりも性能は改善されている．

4.2.5　HDFS 書き込み

図 7 (d) に HDFS の書き込みデータサイズの測定結果を示す．PJoin が 2 つの reduce-side join より約 1/3 と大幅に HDFS の書き込みデータサイズを削減していることが確認できる．また，MapReduce ジョブ数が削減された際には，HDFS の書き込みデータサイズも削減されていることが確認された．特に，Q5，Q8，Q9 では 1 桁，Q2 では 2 桁，HDFS の書き込みデータサイズが削減されている．

[*7] このような課題を解消するには，コストベースの手法が必要になるがこれは今後の課題である．



(a) 応答時間（50 台）　(b) map task 応答時間（50 台）　(c) HDFS 参照（50 台）

(d) シャッフル（50 台）　(e) HDFS 書き込み（50 台）　(f) 応答時間（SF=300）

図 9　スケーラビリティ

Fig. 9　Results of scalability.

表 3 に記載のとおり，Q2，Q5，Q8，Q9 は 4 テーブル以上の多テーブル join であり，PJoin の効果が顕著である．

一方 Q10 と Q13 については，PJoin によって HDFS 書き込みデータサイズが増加してしまっている．Q10 は 4.2.2 項で述べた Q13 と同様に，GROUP BY キーと PJoin の join キーが一致しないことが原因である．

4.2.6　HDFS 参照

図 7 (b) に HDFS の参照データサイズの測定結果を示す．ほとんどのクエリにおいて，PJoin は reduce-side join より参照データサイズが増加してしまっている．これは，PJoin では実体化した射影ビューを利用しているためである．具体的には，PJoin では reduce-side join (star plan) よりも 51.4%増加しているが，これは射影ビューのオーバヘッドである 80.4%（表 2）より少ない．この理由は，カラムストア形式によるものであり，クエリに必要なカラムのみを参照するためである．

reduce-side join に関して star plan と Hive plan を比較してみると，HDFS の参照データサイズについては 4.2.2 項で説明した Q7，Q8 を除いてほぼ同等である．

4.2.7　スケーラビリティ

入力データ規模に対するスケーラビリティ：図 9 では，マシン台数を 50 台と固定して入力データサイズのスケールファクタを 100，200，300 と変化させた際の全クエリの総和に関する (a) 応答時間，(b) map task の応答時間，(c) HDFS 参照データサイズ，(d) シャッフル量，(e) HDFS 書き込みデータサイズに関する測定結果を表している．すべての指標において，PJoin および reduce-side join (star plan) がスケールしていることが確認できる．(c)，(d)，(e) は入力データに対して指標値が線形であることが確認でき，(a)，(b) ではデータサイズが小さい場合にオーバヘッドが多少あることが確認できる．これは，MapReduce におけるタスクの起動終了コストの影響 [14] であると考えられる．

マシン台数に対するスケーラビリティ：図 9 (f) は，入力データサイズのスケールファクタを 300 と固定してマシン台数を 10，30，50 台と変化させた際の応答時間の測定結果を表している．また，応答時間にマシン台数を乗算することで処理の総コストを計算したところ，30，50 台では台数によるオーバヘッドが見られないことが確認できた．この結果，PJoin および reduce-side join (star plan) はマシン台数に対して高いスケーラビリティを示すことが確認できた．

4.2.8　ビュー構築に関する議論

事前計算フェーズにおけるビューの構築時間は表 2 に示したとおり，スケールファクタに応じてビューの構築時間も線形に増加している．スケールファクタが 100 の場合を見てみるとビュー構築に 19.2 分を要しており，ベンチマーク対象である TPC-H の 17 クエリの応答時間[*8]の総和と比

[*8] GROUP BY，ORDER BY，入れ子クエリを処理するジョブなども含めた総処理時間．

較すると，PJoin 46.4 分，reduce-side join (star plan) 56.6 分，reduce-side join (Hive plan) 65.1 分であり，ビューの構築コストがクエリの応答時間と比較して大きいことが分かる．今後の課題になるが，射影ビューに関してはリレーショナルデータベースにおける漸進的ビュー更新技術を適用することが可能である．このため，射影ビューを 1 度だけ構築すれば，以降はデータベースの更新に従ってビューを漸進的に更新することで，多くのクエリを実行すればビューを利用したクエリの高速化のメリットをより多く得ることができると考えられる．

5. 関連研究

分散・並列データベースの分野では多くの join アルゴリズムの研究が行われている．文献 [7] はサーベイであり，分散 INGRES，R*，SSD-1 などの分散データベースが記載されている．join インデックス [15] は，join 処理を高速化する実体化ビューの一種である．実体化ビューという点では PJoin と共通であるが，PJoin では MapReduce 環境においてシャッフルコストを削減するため，事前パーティション分割を行うことで複数テーブルの join を 1 MapReduce ジョブで実現している点が異なる．また，PJoin が用いる実体化ビューはテーブルの射影ビューであり，join の処理結果を実体化する過去の技術とは異なる．

文献 [6] は MapReduce 上でのプログラムの設計および大規模テキスト処理に関する応用（join，転置ファイル構築，PageRank 計算，EM アルゴリズム）に関して述べている．また，MapReduce 上での SQL 処理の高速化に関しては多くの研究があり，join アルゴリズム，クエリ最適化，格納構造の最適化の観点から以下に分類整理する．

join アルゴリズム：Hadoop++ [12] は 2 テーブルを join により関連付けてレコードを事前にパーティション分割しておくことで，シャッフルをすることなく join 処理を実現する手法である．この手法を OLAP のスキーマに適用すると，ディメンジョンテーブルごとにファクトテーブルを事前に join してパーティション分割してしまうため，ディメンジョンテーブル数のファクトテーブルのコピーが必要になってしまう問題がある．対比的に，PJoin ではコンパクトな射影ビューを用いることで，テーブルの主キーと外部キーを事前にパーティション分割しており，実体化にともなう空間のオーバヘッドは 80%程度と効率的であることを確認した．また，Hadoop++ [12] ではインデックスを利用することで 2 テーブル join の高速化を実現しており，PJoin でもインデックスを利用することでさらなる高速化が期待される．

文献 [10] では大量ログデータに対する分析処理を想定して，MapReduce における多様な join 処理（reduce-side join に相当する repartition join，memory-backed join に相当する broadcast join，semi-join，per-split semi-join）を提案している．semi-join を活用するという点において，PJoin は文献 [10] の後半 2 つの手法と共通している．これらの 2 つの手法は，共通してディメンジョンテーブルサイズが小さくオンメモリで join が可能であるデータ規模を対象に設計されているため，小規模データでは有効であるが大規模データではメモリが枯渇するという問題がある．具体的に semi-join, per-split semi-join では，ファクトテーブルの join キーをユニーク化し，ディメンジョンテーブルと semi-join を実行してレコードをフィルタし（以上が事前処理可能），フィルタ結果とファクトテーブルをオンメモリでハッシュ join する．対比的に，PJoin は事前に semi-join を実行せずに主キー・外部キーに基づく事前パーティション分割を行い，検索時にソート済みマージ法によって semi-join を実行するため，省メモリで join 処理が可能である．さらに PJoin では多テーブル join を実現することでさらなる性能改善を行っている．

Map-Join-Reduce [16] は，k パーティション分割関数を導入することで，多テーブル join を 1 MapReduce ジョブで行う手法である（ただし，k は多テーブル join における join キーの個数である）．Map-Join-Reduce のアイデアは，reduce-side join と memory-backed join を融合することにある．具体的には，入力データを k パーティション分割関数を用いて分割するのであるが，一部の join キーを持たないテーブルについて，残りの join キーで特定される複数パーティションにコピーを配布することで，MapReduce のジョブ数を削減する．このように Map-Join-Reduce ではシャッフル量の増加を犠牲にして MapReduce のジョブ数を削減しているため，マシン台数が大量になった場合には適さないと考えられる．一方，PJoin は事前処理を行うことでシャッフル量を削減すると同時に MapReduce のジョブ数を削減する手法であり，より大量のマシン環境に適した手法であるといえる．

CFA-Semi-join [17] は，Map-Reduce-Merge フレームワーク [18] 上に分散ヒストグラムを適用して，semi-join の際の負荷分散を図る技術である．

クエリ最適化：YSmart [19] は join と GROUP BY を含むクエリを対象として，join キーと GROUP BY キーが同一である場合に MapReduce ジョブを統合する最適化を行う手法である．対比的に，PJoin では join キーが異なる多テーブル join を 1 MapReduce ジョブで実行する手法である．

Comet [20] と Cheetah [21] は SQL ライクなクエリを対象とした分散環境における複数クエリの最適化手法である．代数レベルの論理的な処理共有と，scan/shuffle などの物理レベルの処理共有を行っている．複数クエリの最適化の観点から見ると，PJoin における射影ビューの実体化は，複数の join クエリから利用されるデータ構造であり，特に MapReduce における semi-join を高速化する手法で

あるといえる．

文献 [22] ではラグランジュの方程式を用いることで，reduce-side join と memory-backed join のいずれが最適であるかを判定する手法である．最適化の指標はシャッフルコストの最小化である．実測値ではなく理論的な分析結果であるが，小規模なマシン環境では memory-backed join が有利であるという結論を得ている．

格納構造の最適化：MapReduce におけるデータの格納構造に関する取り組みとして文献 [21]，[23]，[24] があげられる．PJoin では Cheetah [21] や RCFile [24] と同様のカラムストア形式を用いている．HadoopDB [23] は HDFS の代わりに PostgreSQL を用いて，Hive の代わりに PostgreSQL の最適化機構を利用する手法である．こうして HadoopDB では PostgreSQL の効率的なデータ格納構造を利用して高速なクエリ処理を実現している．

6. まとめ

クエリログやクリックログの分析などに代表される多くの分析処理では，OLAP 分析と同様に多テーブル join を行う処理で実現されている．このような処理を高速化するため，本論文では MapReduce 上で高速に多テーブル join を実現する PJoin アルゴリズムを提案した．PJoin では semi-join 技術，事前パーティション分割，ビューの実体化を組み合わせることで，スター型のクエリ実行計画を導出して高速に多テーブル join を行う特徴を有している．TPC-H ベンチマークを用いて性能評価を行った結果，reduce-side join と比較して，PJoin では応答性能が平均的に 33.9%向上し，シャッフルによる通信量が 62.6%削減されたことを確認した．

今後の課題は以下のとおりである．1) テーブルのパーティションと実体化した射影ビューのパーティションを HDFS 上で同一マシンに割り当てる配置制御．こうすることで，PJoin の semi-join 実行時に HDFS の参照時に生じる通信コストを削減できる．2) データベースの更新の際に射影ビューを再構築することを避けるため，リレーショナルデータベースにおける漸進的ビュー更新技術を導入する．3) コストベースの最適化機構を導入することで，select 処理結果後の中間テーブルサイズを考慮した最適化や，多テーブル join プランと複数の 2 テーブル join によるプランの最適化を行う．

参考文献

[1] Dean, J. and Ghemawat, S.: MapReduce: Simplified data processing on large clusters, *Proc. OSDI* (2004).

[2] Dean, J. and Ghemawat, S.: MapReduce: Simplied data processing on large clusters, *Comm. ACM*, Vol.51, No.1 (2008).

[3] J.H.I. platforms, the rise of the data scientist, *Beautiful Data*, Segaran, T. and Hammerbacher, J., O'Reilly, Sebastopol (2009).

[4] Thusoo, A., Sarma, J.S., Jain, N., Shao, Z., Chakka, P., Zhang, N., Anthony, S., Liu, H. and Murthy, R.: Hive – A petabyte scale data warehouse using Hadoop, *Proc. ICDE* (2010).

[5] DeWitt, D.J.: MapReduce: A major step backwards, available from ⟨http://databasecolumn.vertica.com/database-innovation/mapreduce-a-major-step-backwards⟩.

[6] Lin, J. and Dyer, C.: *Data intensive text processing with MapReduce*, Morgan & Claypool (2010).

[7] Özsu, M.T. and Valduriez, P.: *Principles of distributed database systems*, 2nd ed., Upper Saddle River, NJ, USA: Prentice-Hall, Inc. (1999).

[8] Transaction Processing Performance Council, "TPC-H Homepage," available from ⟨http://www.tpc.org/tpch⟩.

[9] Agrawal, S., Chaudhuri, S. and Narasayya, V.R.: Automated selection of materialized views and indexes in SQL databases, *Proc. VLDB*, pp.496–505, Morgan Kaufmann (2000).

[10] Blanas, S., Patel, J.M., Ercegovac, V., Rao, J., Shekita, E.J. and Tian, Y.: A comparison of join algorithms for log processing in mapreduce, *Proc. SIGMOD* (2010).

[11] Pavlo, A., Paulson, E., Rasin, A., Abadi, D.J., DeWitt, D.J., Madden, S. and Stonebraker, M.: A comparison of approaches to large-scale data analysis, *Proc. SIGMOD* (2009).

[12] 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), *PVLDB*, Vol.3, No.1 (2010).

[13] Jia, Y.: Running the TPC-H benchmark on Hive, available from ⟨https://issues.apache.org/jira/secure/attachment/12416257/TPC-H_on_Hive_2009-08-11.pdf⟩.

[14] Venner, J.: *Pro Hadoop*, Apress (2009).

[15] Valduriez, P.: Join indices, *ACM Trans. Database Syst.*, Vol.12, pp.218–246 (June 1987).

[16] Jiang, D., Tung, A.K.H. and Chen, G.: MAP-JOIN-REDUCE: Toward scalable and efficient data analysis on large clusters, *IEEE Trans. Knowledge and Data Engineering*, Vol.23, pp.1299–1311 (Sep. 2011).

[17] Hassan, M.A.H. and Bamha, M.: Semi-join computation on distributed file systems using map-reduce-merge model, *Proc. SAC* (2010).

[18] Yang, H.-C., Dasdan, A., Hsiao, R.-L. and Parker, D.S.: Map-reduce-merge: Simplified relational data processing on large clusters, *Proc. SIGMOD* (2007).

[19] Lee, R., Luo, T., Huai, Y., Wang, F., He, Y. and Zhang, X.: YSmart: Yet another sql-to-mapreduce translator, *Proc. ICDCS*, pp.25–36 (Online), DOI: 10.1109/ICDCS.2011.26 (2011).

[20] He, B., Yang, M., Guo, Z., Chen, R., Su, B., Lin, W. and Zhou, L.: Comet: Batched stream processing for data intensive distributed computing, *Proc. SOCC* (2010).

[21] Chen, S. and Incorporated, T.: Cheetah: A high performance, custom data warehouse on top of MapReduce, *PVLDB*, Vol.3, No.2 (2010).

[22] Afrati, F.N. and Ullman, J.D.: Optimizing joins in a map-reduce environment, *Proc. EDBT* (2010).

[23] Abouzeid, A., Bajda-Pawlikowski, K., Abadi, D., Silberschatz, A. and Rasin, A.: HadoopDB: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads, *PVLDB*, Vol.2, No.1 (2009).

[24] He, Y., Lee, R., Huai, Y., Shao, Z., Jain, N., Zhang, X. and Xu, Z.: RCFile: A fast and space-efficient data

placement structure in MapReduce-based warehouse systems, *Proc. ICDE 2011*, pp.1199–1208 (2011).

鬼塚 真　（正会員）

1991 年東京工業大学工学部情報工学科卒業．同年 NTT 入社．2000～2001 年ワシントン州立大学客員研究員．現在，日本電信電話（株）ソフトウェアイノベーションセンタ特別研究員，電気通信大学客員教授，博士（工学）．大規模グラフデータのデータ処理に関する研究開発に取り組んでいる．ACM，電子情報通信学会，日本データベース学会各会員．

久保 類

2003 年電気通信大学大学院情報システム学研究科情報システム設計学専攻修了．同年日本電信電話（株）入社．現在，同社ソフトウェアイノベーションセンタにて，仮想ネットワーク方式に関する研究開発に取り組んでいる．人工知能学会会員．

（担当編集委員　的野 晃整）