Vol.2013-DBS-158 No.6
2013/11/26

データ共有型マルチデータベースシステムにおけるクエリ効率化手法

齋藤 和広† 渡辺 泰之† 小林 亜令†

近年，データの大規模化やデータ利用の多様化に対応するために，単一のデータソースに対して特徴の異なる複数のクエリ処理エンジンを選択し利用可能となってきている．これらのクエリ処理エンジンは，それぞれデータの種類や規模に応じた最適化が施されているため，ユーザが各エンジンの特性を理解し，使い分けることでクエリ処理性能を最大限活用することができる．しかし，ユーザにデータの内容や各クエリ処理エンジンの特性理解を求められることから，ユーザの負担が大きくなるという課題がある．そこで本稿では，単一のデータソースを共有する複数のクエリ処理エンジンを仮想的に一つに見せるマルチデータベースシステムにおけるクエリ効率化手法を提案する．本方式では，ユーザが投稿したクエリから，各クエリ処理エンジンの特性に応じたクエリを自動的に分割生成し，共有しているデータソースを活用した振り分けが可能なため，ユーザの手間を低減しつつ，クエリ処理エンジンを効率的に使い分けることが可能となる．本稿では，Hive 及び Impala を備えた SQL on Hadoop システムを用いた評価実験により，提案方式の有効性を考察する．

Efficient Query Processing for Data Sharing Multi-Database System

KAZUHIRO SAITO† YASUYUKI WATANABE† AREI KOBAYASHI†

1. はじめに

様々な電子機器から生成されるセンサ情報や，SNS 等の Web 上でユーザが生成する情報，企業の経営活動における様々な情報など，莫大な量のデータが生み出されてきている．従来であれば設備の容量や計算資源の不足等から，これらのデータを蓄積して活用する動きは限定的なものであった．しかし近年では，ハードウェアの価格低下や分散処理技術の高度化等によって，一般企業が大規模データを経営やマーケティング，新サービス等に活用することが現実的になってきている．

大規模データの利活用を加速化させた代表的なソフトウェアとして Apache Hadoop[1]がある．Hadoop は Google 社が提案した分散ファイルシステム Google File System[7]と，これを利用したプログラミングモデル MapReduce[5]をオープンソース実装したもので，大規模データの保存と並列処理を容易に利用できる. Hadoop は高いスケーラビリティと信頼性を持ち，かつ安価なハードウェアで構築可能な点から一般企業等に広く普及してきている．

大規模データを容易に蓄積，分析が可能となったことから，より様々な種類のデータが収集され，より複雑な処理が求められてきている. Hadoop はオープンソースソフトウェアであり，Yahoo や Facebook 等の多くの企業に開発され様々な改良や機能追加が行われてきているが，必ずしも全てのクエリ処理が MapReduce で効率的に実行できるとは限らない．例えば，MapReduce フレームワークには Map 処理と Reduce 処理という二つの決められた様式で処理を記述しなければならない等の固有の制限があるため，MapReduce の様式に合わせることで無駄な処理が発生することがある．そこで，Hadoop の分散ファイルシステムである HDFS を共通のデータソースとして，データやクエリ処理の種類に応じて最適化された様々なクエリ処理エンジンが MapReduce の代わりとして開発されてきている．

このような構成の例として，Java によるプログラミングやビルド処理を回避するために SQL を利用して HDFS データにアクセスするための SQL-on-Hadoop システムがある．

Apache Hive[2]は，SQL ライクな言語である HiveQL を利用して MapReduce を実行することが可能な SQL-on-Hadoop システムである．Hive はユーザが投稿した HiveQL を自動で MapReduce に変換し，Hadoop 上で実行する．Metastore と呼ばれるサブシステムが，HDFS 上のデータをテーブルとして認識させるためのメタデータを保持しており，ユーザはデータベースシステムを扱うように Hadoop を操作することができる．しかし，複雑なクエリのように，複数の MapReduce ジョブが実行されることでジョブ毎のオーバヘッドが余分に発生する等，MapReduce 処理に適していない処理が存在する．

このような課題を解決するために作られた SQL-on-Hadoop システムが Cloudera Impala[3]である．これは Cloudera 社が主導で開発したオープンソースの SQL-on-Hadoop システムで，MapReduce を利用せず，HiveQL に特化した独自の分散処理エンジンを構築することで， HDFS

† (株) KDDI 研究所
KDDI R&D Laboratories Inc.

のデータに対する高速な処理が可能となった．また，メタデータに Hive の Metastore を利用しており，Hive と同じテーブルに対するクエリ処理が可能である．しかし物理メモリサイズを超えるような大規模データに対するクエリ処理は不向きであり，処理速度が大幅に低下する傾向がある．

すなわち，大規模なデータを単一の共有データソースである HDFS に保管した場合，データの種類やクエリ処理に応じてクエリ処理エンジンを使い分けることでデータを一元化しつつ，より大規模データに対する処理を効率化することが可能と考えられる．他にもカラムストア型のデータストアや，DAG 型の依存関係がある連続したクエリ処理，テキストに対する全文検索エンジン，グラフ型データに対する処理等があり，今後更なるデータの多様化と処理の複雑化が想定され，単一のデータソースを共有するクエリ処理エンジンの種類は今後も増加傾向が続くことが予想される．

しかし，このようなクエリ処理エンジンをユーザが使い分けるためには，利用するデータの内容や，各エンジンの特性を把握しなければならない．これはユーザにとって負担であり，データの規模やクエリ処理エンジンの種類が多くなるほどこの負担も大きくなる．また場合によっては，ユーザが不適切なクエリ処理エンジンを選択することで，大幅な遅延やシステムクラッシュなどの問題が発生する可能性がある．

本稿では，このような同一のデータソースを利用した複数のクエリ処理エンジンを使い分けるためのデータ共有型マルチデータベースシステムと，本システムにおける効率的なクエリ処理手法を提案し，Apache Hive と Cloudera Impala を利用した評価を行う．共有型マルチデータベースシステムでは，対象のシステムを，実データを保持するデータソースと，処理系やインタフェースを持つクエリ処理エンジンに分離して認識する物理モデルを提供する．そして，クエリ処理エンジンの特性に応じたクエリを自動的に分割生成し，共有しているデータソースを活用するクエリ振り分けを行うことで，ユーザの負荷を軽減しつつ，クエリ処理エンジンを効率的に使い分けることが可能となる．

2. 関連技術と課題

2.1 マルチデータベースシステムの課題

マルチデータベースシステム[6]はデータ仮想化システムとも呼ばれ，データベースシステムや XML データ等の，決められた構造を持つデータソースをマッピングした仮想的なテーブル（仮想スキーマ）を提供するシステムである．図 1 は，データの仮想化対象となる二つのデータベースシステム上のテーブルを，一つのテーブルにマッピングする場合の概要図である．予め定義した仮想スキーマ a には，二つのデータベースシステム S と T を示す物理モデルがあり，それぞれが持つテーブル s と t を UNION 処理するよう定義されている．クライアントが仮想スキーマ a の表すテーブル a への SQL クエリをマルチデータベースシステムに投稿すると，自動的にデータベースシステム S と T それぞれへの SQL クエリに変換する．その後，それぞれのクエリ結果をマルチデータベースシステム上に収集し，仮想スキーマに定義された処理を行った上でクライアントに結果が返る．本システムにより，ユーザはデータの配置場所を意識することなく，必要なデータに対するクエリ処理を行うことが可能となる．

しかしマルチデータベースシステムは，データソースとクエリ処理エンジンが 1 対 1 対応していることを前提としている．そのため，複数のクエリ処理エンジンが同一のデータソースを共有する場合に，各クエリ処理エンジンが同一のデータを保持せざるを得ない．また，クエリ特性に応じて適切なクエリ処理エンジンを選択することも不可能である．

この課題を回避するために，クエリ実行時に全クエリ処理エンジンにて同一クエリ処理を実行し，最短で処理完了できた結果をユーザに返すことができる．しかしこの手法では，冗長なデータ移動処理やクエリ処理が発生するため効率が悪い．

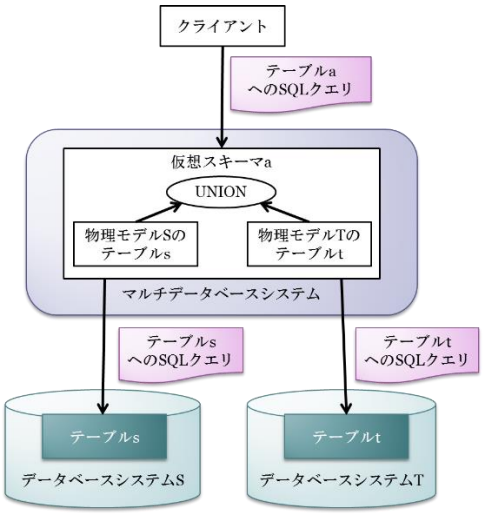

図 1　仮想スキーマの例

2.2 関連研究と課題

参考文献[8]では，マルチデータベースシステムにおいて同一のテーブルを保持するデータベースシステムが複数ある場合のクエリ処理手法を提案し，前記課題を解消している．本方式では，図 2 で示すように，仮想スキーマがなるべくデータベースシステムを跨ぐ処理とならないよう，予め適切なデータベースシステムにテーブルデータをコピーしておく．そして，複数のデータベースシステムに存在する同一のテーブルに対して一つの仮想ニックネームをつけ，クエリ実行時に一つのデータベースシステム上で処理が行われるテーブルを選択する．本方式により，同一のデータ

が複数の物理モデルに存在する状況下において，複数のデータベースシステムを跨ぐクエリ処理を回避できる物理モデルを選択することで，2.1 章で述べた冗長なクエリ処理を抑制することが期待できる．

しかし，本手法は一つのデータソースが共有されている場合を想定していないため，クエリ処理エンジンの数だけ別の物理モデルが生成されることとなり，複数のクエリ処理エンジンによる協調処理を用いた高速化が困難となる．

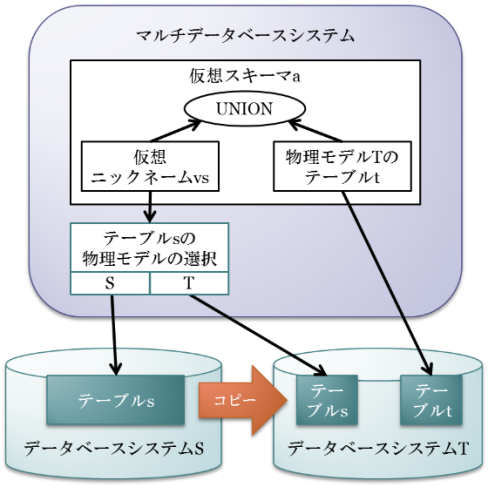

図 2　データコピーによるクエリ処理効率化

3. 提案手法

3.1 データ共有型マルチデータベースシステム

マルチデータベースシステムは複数システムを仮想的に一つに見せることで，複数システムの使い分けを行うユーザの負荷の解消や不適切なシステムの選択を回避することが可能である．しかし，2 章で述べたように，従来法では一つのデータソースを複数のクエリ処理エンジンによって共有する環境を想定していないため，データの規模や種類，クエリの内容に応じたクエリ処理エンジンの使い分けによる協調処理が困難である．

本課題を解決するために，複数のクエリ処理エンジンが一つのデータソースを共有できるデータ共有型マルチデータベースシステムを提案する．

3.2 アーキテクチャ

提案方式の構成図を図 3 に示す．本システムでは，共有対象のデータソースを一つの物理モデルとして登録する．さらに，物理モデルに対するクエリをどのクエリ処理エンジンに投稿するか判断するために，各クエリ処理エンジンの情報をマルチデータベース管理データに登録する．

このようなデータ共有型マルチデータベースシステムでは，複数のクエリ処理エンジンがデータソースを共有していることから，クエリ処理エンジンの切り替えの度にネットワーク上に中間データを転送する必要がなく，共有のデータソースを活用することで少ない遅延で切り替えが可能である．また，ユーザが投稿したクエリが複雑化するケースにおいて，クエリを効率的に分割し，処理毎に適したクエリ処理エンジンに分配することができれば，より大きな効果が期待できる．

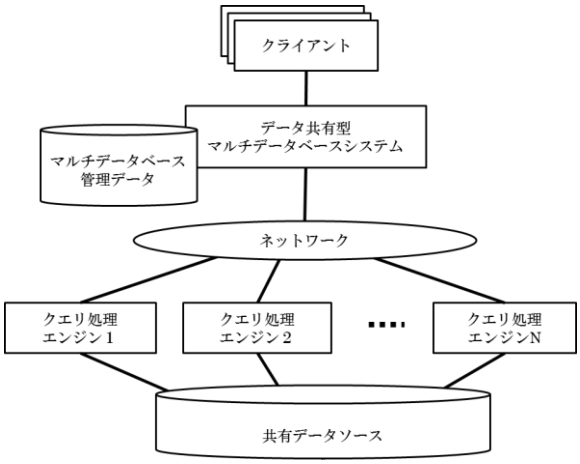

図 3　データ共有型マルチデータベースシステム

3.3 データ共有型マルチデータベースシステムにおけるクエリ効率化

本章では，前述したデータ共有型マルチデータベースシステムにおいてクエリ処理効率化を実現するためのクエリ処理制御方式について述べる．

データ共有型マルチデータベースシステムにおいては，共有しているデータソースを活用して複数のクエリ処理エンジンを使い分けることによって，効率的なクエリ処理を可能とする．このようなクエリ処理を実現する上で，必要となる要素は大きく二つある．

- 複数のクエリ処理エンジンに対する振り分けの判断と，それに従った分散クエリの生成
- 分散クエリの実行順序に従った実行制御

これに沿ったクエリ処理を示したものが図 4 である．図 4 では，まずユーザが投稿したクエリをデータ共有型マルチデータベースシステムが受け取る．このクエリを分析し，データソースやクエリ処理エンジンに関する情報を利用して，クエリに適したクエリ処理エンジンを選択し，それぞれに投稿するクエリを生成する．その後，この分割された分散クエリを元のクエリ処理の順序で実行する．図 4 の例では，三つのクエリ処理エンジンが以下の通り実行を行った例を表している．

① X に対して処理を行った結果を Temp1 として共有データソースに書き込む
② Temp1 に対して処理を行った結果を Temp2 として共有データソースに書き込む
③ Temp2 に対して処理を行った結果を最終結果 Result としてデータ共有型マルチデータベースシ

ステムに返す

本方式により，ユーザからは透過的により適したクエリ処理エンジンを選択することが可能となり，一つのクエリが部分的に複数のクエリ処理エンジンに適していた場合に，クエリを分割してそれぞれを適切なクエリ処理エンジンに振り分け，中間データを共有データソースに置くことで従来法では実現できなかった効率的なクエリ処理が可能となる．

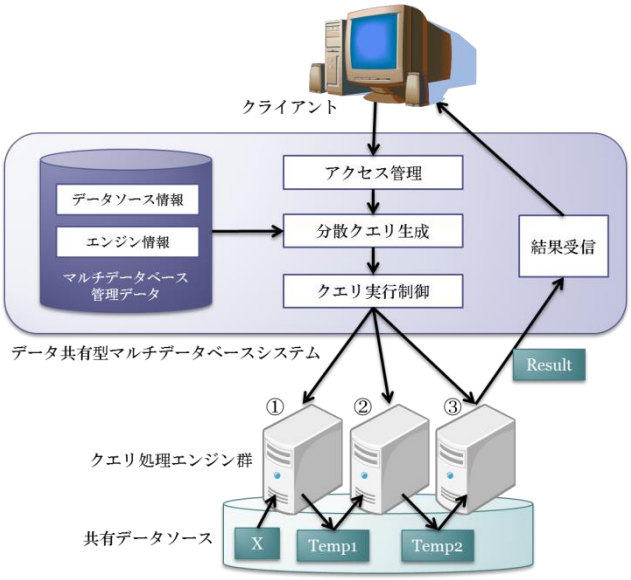

図 4　提案方式の構成例

本章のクエリ処理制御方式を，一つのデータソースを共有している複数のクエリ処理エンジンに適用してプロトタイプを実装した．以下では，クエリ処理制御の対象となるクエリ処理エンジンとその特徴及び振り分け条件について述べ，これらに対するクエリ処理制御の実装の詳細を述べる．

3.3.1 実装対象とするクエリ処理エンジン

本実装では，Hadoop HDFS を一つの共有データソースとする SQL-on-Hadoop システムのうち，HDFS に変更を加えずに導入でき，かつオープンソースである Apache Hive[2]と Cloudera Impala[3]を利用した．

Hive は，MapReduce で分散処理することで大規模なデータを高速に実行することが可能であるが，MapReduce 自体がバッチ処理向けであるため，インタラクティブな処理に向いていない．例えば，MapReduce はジョブ毎の初期化処理や，Map と Reduce の間のディスク利用等の大規模データ処理向けのオーバヘッドによって，最低数十秒の処理時間がかかってしまう．特にサブクエリを含むような複雑なクエリになると，複数回の MapReduce ジョブが実行されることとなり，そのようなオーバヘッドがより大きくなる．

一方 Impala は MapReduce を使わず，Hive のようなオーバヘッドを削減し，より高速なクエリ処理が可能となった．しかし高速化のために全てのデータを常にメモリ上に展開して処理を行うため，処理対象のテーブルサイズが，クラスタ全体で利用できる物理メモリサイズの合計を超えると，スワップが発生し処理速度が大幅に低下する（場合によってはアウトオブメモリで異常終了する）．そのため，Impala を有効活用するためには，対象のテーブルサイズとクエリ処理上で生成される中間データのサイズが，物理メモリサイズの合計以下であることをユーザが把握している必要がある．

以上のことから，これらのクエリ処理エンジンの振り分けとして，処理対象のデータサイズを条件にし，Impala は「クラスタ全体の物理メモリサイズの合計を超えない場合」に，Hive は「クラスタ全体の物理メモリサイズの合計を超える場合」にクエリを実行することとして実装した．このデータサイズには，HiveQL におけるサブクエリ単位の入出力テーブルの合計サイズを利用している．これにより，例えば一部のサブクエリだけが物理メモリサイズを超えている場合，その部分を Hive が実行し，結果を中間データとして共有データソースである HDFS に書き込み，Impala がその中間データを利用して残りのクエリを実行することが可能となる．

3.3.2 分散クエリ生成の実装

ここで，クエリ処理エンジンへ振り分けるクエリの生成手法とその実装について詳細を述べる．図 5 はこのフローを示す．

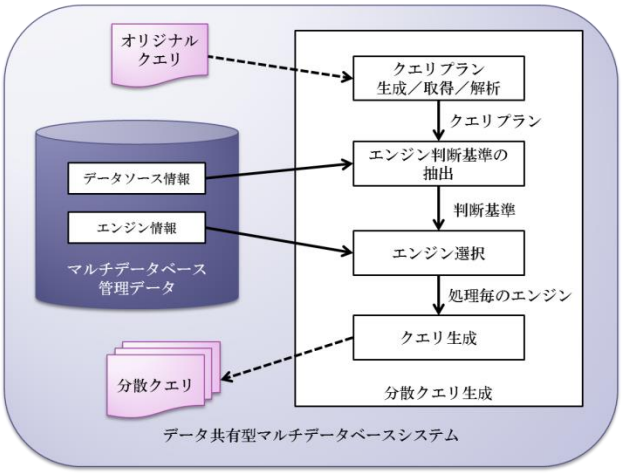

図 5　分散クエリ生成フロー

はじめに，クライアントが投稿したクエリを基にクエリの実行計画を示すクエリプランを生成する．本実装ではユーザの投稿したクエリをクエリ振り分け対象である Impala に Explain 句で投稿することで，Impala が生成するクエリプランを取得し，これを流用している．

次に，実行するクエリ処理エンジンを決めるために必要な要素を，クエリプランとデータソース情報から抽出する．本実装では，利用するテーブルのサイズと，サブクエリご

との処理結果のテーブルサイズを抽出している．これらを計算するために，データソース情報としてテーブルごとのサイズとレコード数，中間データの計算に必要な条件ごとの選択率，属性情報を格納している．

上記で生成したクエリに関する情報と，別途用意したクエリエンジンの特徴や制限に関する情報を利用して，クエリプラン上の処理単位で実行するクエリ処理エンジンを決定する．本実装では，計算したテーブルサイズと中間データサイズを利用し，各クエリ処理エンジンの上限となるデータサイズを基にクエリ処理エンジンを選択する．これをサブクエリ毎に行うことで, 全体では Hive のみの実行となるクエリでも，Impala で実行可能なサブクエリを部分的に抽出し, Impala を利用して高速化を行うことが可能となる．

最後に，各クエリ処理エンジンの実行形態やクエリの文法に従って，クエリプラン通りのクエリを生成する．Hive 及び Impala は HiveQL をインタフェースとしていることから，この文法に従った HiveQL クエリを生成する．このとき，複数のクエリ処理エンジンに跨がるクエリは，各クエリの中間データの書き出し処理と，利用する中間データの読み出し処理，利用した中間データの削除処理を追加する必要がある．ここでクエリプラン上の実行順序も併せて記録することで，後のクエリ実行制御を可能とする．また，Impala ではテーブル作成やデータ更新後に，Impala が持つメタデータのキャッシュの更新が必要となるため，Hive でテーブルを作成・削除・変更した場合は，Impala で「INVALIDATE METADATA」クエリが必要となり，また同様に Hive でレコードの更新を行った後は「REFRESH <テーブル名>」クエリの実行が必要となる[3]．そのため，Impala の中間テーブル作成前に「INVALIDATE METADATA」クエリを，Hive 実行後の Impala 実行前に「REFRESH <テーブル名>」クエリを挿入している．このような分散クエリの生成例として，図 6 のクエリを Hive 及び Impala に分割したときの分散クエリを図 7 に示す．左端の数字はクエリの実行する順序を表しており，その横の括弧が実行されるクエリ処理エンジンである．コロンを挟んだ右側が実行するクエリを示している．

```
SELECT * FROM a JOIN (
  SELECT id, name FROM b WHERE name = “xxx”
) AS j ON a.id = j.id;
```

図 6　元のクエリ

```
1(Impala) : INVALIDATE METADATA;
2(Impala) : CREATE TABLE hive_temp (id int, name string);
3(Hive)   : INSERT INTO TABLE hive_temp SELECT id, name
                FROM b WHERE name = “xxx”;
4(Impala) : REFRESH hive_temp;
5(Impala) : SELECT * FROM a JOIN hive_tmp as j on a.id = j.id;
6(Impala) : DROP TABLE hive_tmp;
```

図 7　本手法適用後の分散クエリ

3.3.3 実行制御の実装

クエリ処理エンジンに投稿するクエリが一つの場合，実行時に順序を意識する必要がない．しかし複数のクエリがある場合，各クエリは中間データを必要とするため，必要な中間データを生成するクエリが終了するまで実行することができない．ここでは，このような依存関係を考慮したクエリの実行を制御する．

3.3.2 で生成した分散クエリを，クエリ生成時に記録した実行順序を基に実行する．一つクエリを実行すると，中間データの生成完了であるクエリ完了が通知されるまで待ち，これが届き次第，次のクエリが実行される．これを全ての分散クエリが実行完了するまで行われ，最終的な出力データが返ってきた段階で分散クエリの実行は完了となる．

4. 評価実験

本章では，これまでに述べた提案手法の実現性について評価した結果を述べる. 提案手法のプロトタイプを開発し，二つのクエリ処理エンジン Hive と Impala を利用して TPC-H ベンチマークのテストクエリに対する振り分けを行い，その実行結果と性能の測定を行った．

4.1 実験環境

実験環境に利用するソフトウェアを表 1 に示す．なお Hadoop 及び Hive は Cloudera 社が提供しているディストリビューション（CDH）を利用している．

システム構成は Data Node, Task Tracker, Impalad が稼働するスレーブノードが 3 台，Name Node, Job Tracker, Impala Statestore が稼働するマスタノードが 1 台, Hive Server, Hive Metastore, 提案手法を導入したデータ共有型マルチデータベースシステムとマルチデータベース管理データが稼働するクライアントが 1 台の計 5 台構成となり，これらのハードウェア仕様を表 2 に示す．これらを図 8 で示す構成で 1GbEthernet の同一ネットワーク上に構築した. ここで共通のデータソースに該当するシステムが HDFS（Name Node, Data Node）であり，二つのクエリ処理エンジンが， Hive（Hive Server, Job Tracker, Task Tracker）と Impala（Impalad）となる．なお，Hive の Metastore と提案手法で利用するマルチデータベース管理データは, 同じ PostgreSQL 上に構築した．

マルチデータベース管理データ上のエンジン情報には Impala 及び Hive の情報として，Hive は実行上限となるデータサイズがないと想定した設定にし, Impala は 4GB を設定した．つまり，各サブクエリの入出力テーブルサイズの合計が 4GB 以下の場合に Impala へ振り分けることとなる．

ベンチマークには TPC-H[4]で提供されているテストクエリのうち，Q4，Q5，Q6，Q8 を利用し，テーブルサイズを表す Scale は 10（全 10GB）と 5（全 5GB）の 2 種類を利用した．Q4 は二つのクエリが実行され，最初に大容量テーブルに対する処理が行われ，二個目のクエリは比較的に小

容量のテーブルに対する処理が行われる. Q5 はサブクエリが四つあり，比較的複雑なクエリ処理となる．Q6 は比較的小規模なサブクエリのないクエリである. Q8 はサブクエリが七つあり，Q5 よりも複雑さが増しているクエリである．

表 1　ソフトウェア構成

OS	CentOS release 6.3 (Final)
Java	OpenJDK 1.6.0_24 (rhel-1.57.1.11.9.el6_4-x86_64)
Hadoop	CDH 4.4.0 (MapReduce1)
Hive	0.10.0 (CDH4.4.0)
Impala	1.1.1
PostgreSQL	8.4.13

表 2　ハードウェア構成

	スレーブノード	マスタノード	クライアント
CPU	Xeon 5160 3.0GHz Dual-Core x 2	Dual-Core AMD Opteron 1222 3.0GHz	Xeon E5410 2.33GHz Quad-Core
メモリ	4GB x 8	2GB x 4	4GB
HDD	1.5TB	1TB x 2	1TB x 4

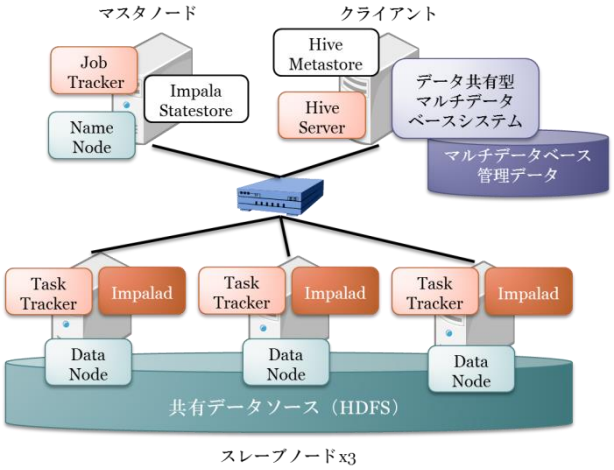

図 8　実験環境のシステム構成図

4.2 実験結果

4.1 で示した環境とベンチマークを利用して, 提案手法のプロトタイプを実行した．前述の通り，クラスタの物理メモリの合計を 4GB と設定していることから, 本実験のベンチマークの対象データ（5GB, 10GB）は，ユーザから見ると Impala での実行は出来ない．そこで，提案手法の比較対象として，全て Hive で実行した場合の実行時間と比較した．図 9 はその結果であり，縦軸が Hive の実行時間に対する提案手法の実行時間の比を, 横軸が TPC-H のクエリを表していて，四つのクエリで容量を変えて実行したときの性能差を示している．

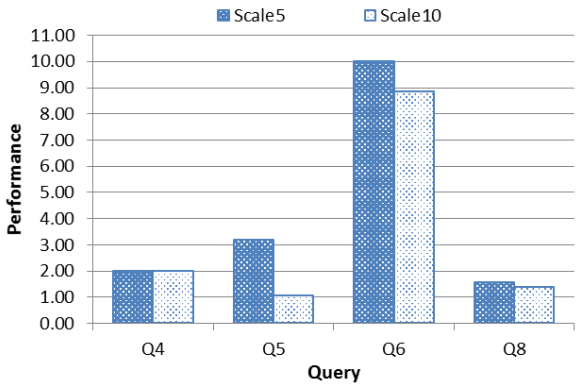

図 9　TPC-H における提案手法と Hive 単体の性能比

クエリ Q4 では，Scale5 と Scale10 共に提案手法が Hive と比較して 2 倍高速化した. クエリ Q5 では, Scale 5 が 3.18 倍と大幅に高速化しているが，Scale 10 では 1.03 倍であり Hive の実行時間とあまり変わらない結果であった．クエリ Q6 は Scale 5，Scale 10 それぞれが 10 倍，8.5 倍と大幅な速度向上が見られている．クエリ Q8 では，提案手法が Hive と比較して Scale 5 で 1.59 倍，Scale 10 で 1.38 倍性能が向上した．

4.3 考察

本実験では, 提案手法を実装したプロトタイプを用いて, 二つの異なる特徴をもつクエリ処理エンジン Hive と Impala に対して TPC-H のクエリを振り分ける実験を行った．結果としてクエリごとに四つの異なる特徴を持つ結果が得られた．本章では，クエリの種類と，データサイズの変化という二つの観点で実験結果を考察する．

4.3.1 クエリの種類

本実験では四つのクエリの種類に応じて，それぞれ異なる特徴の実行結果が得られた．

Q4 と Q6 では，通常であれば Hive で実行されるテーブルサイズのクエリであっても，提案手法でクエリが実際に物理メモリを利用するデータサイズを計算することで，より高速に処理できる Impala に振り分けることができ，高速化を実現することが可能であることを示している. Q4 の一つ目のクエリは Hive で実行されたが，二つ目のクエリが Impala に振り分けて実行されたことによって，二つ目のクエリの実行時間が大幅に短縮されたことがわかった．一方 Q6 は，全てを Impala に振り分けたことによって得られた速度向上である．表 3 は，Q6 を Impala 単体で実行した場合と提案手法を利用した場合の実行時間を示している．Scale5 の場合は約 0.4 秒，Scale10 の場合は約 0.3 秒提案方式の方が遅い．これは，提案方式における分散クエリの生成処理を行ったことによるオーバヘッドと考えられる．提案手法は，クエリのサブクエリ単位でクエリの振り分けに必要な情報を抽出することから，クエリの複雑さに比例してオーバヘッドが大きくなるが，クエリ処理対象のデータサイズには非依存である．つまり，処理対象データサイズ

が大規模な場合では，クエリ全体の実行時間が十分に長くなり，オーバヘッドの影響は小さくなると考えられる．

表 3　Q6 における Impala と提案手法の実行時間（秒）

	Impala	提案方式
Scale5	3.96	4.36
Scale10	7.33	7.63

Q5 と Q8 から，サブクエリ等の分割可能なクエリにおいて適しているクエリ処理エンジンが異なる場合に，共有するデータソースを有効活用したクエリ分割を行うことで，それぞれの適したクエリ処理エンジンに振り分けることが有効であると言える．

ここで，Q5 を例に，分散クエリがどのような順序で実行され，どの程度のオーバヘッドが発生しているかを分析する．図 10 が Q5 における分散クエリを示しており，図 7 と同様である．この分散クエリのクエリ実行順序に対応したそれぞれの実行時間が表 4 であり，手順 4（Impala）と手順 5（Hive）が提案方式により分割されたユーザクエリであり，残りの手順 1,2,3,6 が提案手法により付与されたオーバヘッドに相当するクエリである．今回の実験ではこれらのオーバヘッドは大きな影響を与えず，提案手法が Hive と比較して高速に実行されているが，Impala を利用することによって得られる高速化が小さいクエリが存在すると，無視できなくなる可能性があると考えられる．

```
1 (Impala) : INVALIDATE METADATA
2 (Impala) : CREATE TABLE temp_table1 ( -省略-)
3 (Impala) : REFRESH temp_table1
4 (Impala) : INSERT INTO TABLE temp_table1 select  - 省略-
5 (Hive) :  - 省略- from customer c join temp_table1 o1 on - 省略-
6 (Impala) : DROP TABLE temp_table1
```

図 10　Q5 の分散クエリ

表 4　図 10 における各クエリの実行時間（Scale5）

実行順序	Engine	Time(s)
手順 1	Impala	5.19
手順 2	Impala	0.05
手順 3	Impala	0.05
手順 4	Impala	10.10
手順 5	Hive	56.70
手順 6	Impala	1.56

4.3.2 データサイズの変化

本実験では，クエリの種類だけでなく，クエリ処理対象のテーブルサイズも変えて実験を行った．以下で，振分条件を処理対象のデータサイズとする複数のクエリ処理エンジンにおける，クエリ振り分けの性能と処理データサイズの関係を考察する．

Q5 では，Scale 10 において，Impala で処理されたクエリのテーブルサイズ容量の割合が Hive に比べてごく小さい容量のテーブルであったため，あまり性能向上が見られなかったと考えられる．一方で，Scale 5 においては大幅な速度向上が見られているが，これはテーブルサイズが Scale 10 に比べて小さくなったことにより，Impala の実行範囲の全体に占める割合が大きくなったためと考えられる．図 11 は Scale 5 において Impala が実行したクエリであり，図 12 は Scale 10 にて同様に Impala が実行したクエリを示していて，これらを見ると Scale 5 における Impala が実行したクエリの範囲が広いことがわかる．Q5 はサブクエリが全部で四つあり，Scale 5 では Impala がこれらサブクエリの全てを実行しているが，Scale 10 では二つのみの実行となっている．このことから，クエリの分割時は，処理対象のテーブルサイズが大きくなるほど Impala が処理するクエリの範囲が少なくなるため，Hive の性能に近づくことがわかる．

```
INSERT INTO TABLE temp_table1
 select n_name, l_extendedprice, l_discount, s_nationkey, o_custkey from orders o join
    ( select n_name, l_extendedprice, l_discount, l_orderkey, s_nationkey from lineitem l join
     ( select n_name, s_suppkey, s_nationkey from supplier s join
      ( select n_name, n_nationkey
        from nation n join region r
        on n.n_regionkey = r.r_regionkey and r.r_name = 'ASIA'
      ) n1 on s.s_nationkey = n1.n_nationkey
     ) s1 on l.l_suppkey = s1.s_suppkey
    ) l1 on l1.l_orderkey = o.o_orderkey and o.o_orderdate >= '1994-01-01'
        and o.o_orderdate < '1995-01-01'
```

図 11　Q5 の Scale 5 における Impala 実行クエリ

```
INSERT INTO TABLE temp_table1
select n_name, s_suppkey, s_nationkey from supplier s join
 ( select n_name, n_nationkey
   from nation n join region r
   on n.n_regionkey = r.r_regionkey and r.r_name = 'ASIA'
 ) n1 on s.s_nationkey = n1.n_nationkey
```

図 12　Q5 の Scale 10 における Impala 実行クエリ

Q8 はクエリの形が Q5 と類似しており，Q5 と同様に処理対象のテーブルサイズ（Scale）が小さくなることで，速度向上がみられる．しかし，実際の分割されたクエリを分析すると，Scale 5 と Scale 10 において Impala で実行しているクエリが同じことがわかった．この性能差が，どのようにして現れたかを分析するために，Impala 単体で Hive 及び提案手法と同様に TPC-H の Scale 5 及び Scale 10 の Q8 を実行した．その実行時間と性能比をグラフ化したものが図 13 である．棒グラフが実行時間であり，線グラフが性能比を表している．Impala と Hive の性能比は，処理対象のデータ容量が増えると低くなる傾向にあることがわかる．これは Q6 でも同様であり，図 9 の結果を見ても，提案手法（実際は Impala で実行）と Hive の性能比が低下していることがわかる．つまり，Impala は処理対象のデータサイズが大きくなった場合の速度低下が，Hive よりも大きくなる傾向にある．その結果，クエリ Q8 において提案手法に

おけるクエリの分割に変化がなかったにも関わらず，Scale 5 の方が Scale 10 よりも速度向上が大きくなっている．

Q5 の結果から，クエリ処理エンジンが処理するクエリの範囲が提案手法の性能に大きく影響すると分析したが，一方で Q8 のようにこの範囲が変化しなかったとしても，対象のデータサイズに変化が起きることで，複数のクエリ処理エンジンそれぞれのもつ特徴に応じた性能変化が起こり，同様の現象が起こることがわかった．よって今後，分割されたクエリの実行範囲と，各クエリ処理エンジンが持つデータサイズと処理性能の相関を利用することで，提案方式のデータサイズ変化に伴う実行時間の推移を予想できると考えられる．

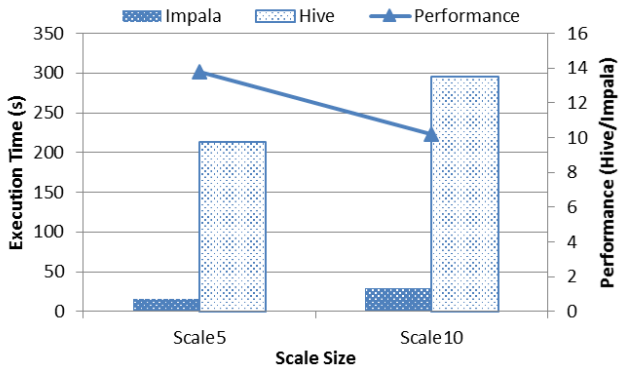

図 13　Q8 における Impala と Hive の性能比

5. おわりに

本稿では，データ共有型マルチデータベースシステム及び，効率的なクエリ処理手法を提案した．実験の結果，処理対象のデータサイズを用いて，より高速なクエリ処理エンジンの自動選択を可能であることを示した．今後は，Hive と Impala 以外のデータソースを共有するシステムで評価を行い，データや処理の種類に着目した振り分けに関する検証を行いたい．

参考文献

[1] Apache Hadoop, http://hadoop.apache.org/
[2] Apache Hive, http://hive.apache.org/
[3] Cloudera Impala, http://www.cloudera.com/content/cloudera/en/products/cdh/impala.html
[4] TPC-H, http://www.tpc.org/tpch/
[5] J. Dean and S. Ghemawat: MapReduce: Simplified Data Processing on Large Clusters, OSDI'04, pp.137-150, 2004.
[6] M. T. Özusu, P. Valduriez: Principles of Distributed Database Systems, Third Edition, Springer, 2011
[7] S. Ghemawat, H. Gobioff, and S. Leung: The Google File System, SOSP'03, 2003.
[8] 大川 昌弘，黒澤 亮二，福田 剛志: 分散データベース環境における複製データの仮想化による SQL 処理の最適化手法, DEWS2008