情報処理学会論文誌　データベース　Vol.8　No.3　24–39 (Sep. 2015)

適切なクエリ処理エンジンを自動選択するマルチデータベースシステム

齋藤 和広[1,a)]　渡辺 泰之[1]　村松 茂樹[1]　小林 亜令[1]

受付日 2015年3月21日, 採録日 2015年7月8日

概要：近年，共有ストレージ上の同一データに対して異なる特徴を持つ複数のクエリ処理エンジン（QE）が利用可能な環境が開発されている．代表例が Hadoop とそのエコシステムである．従来のマルチデータベースシステムは，このような同一データを持つ複数の QE を自動的に選択することができない．そのため，使い分けによりユーザの負荷が増加し，さらにはユーザの不適切な選択により，大幅な遅延やシステムクラッシュ等の問題が発生する．そこで本論文では，同一データを持つ複数の QE から，適切な QE を自動選択するマルチデータベースシステムを提案する．提案システムは，同一データを持つ複数の QE を単一のデータソースとしてユーザが認識可能な仮想スキーマを提供することで，適切な QE の自動選択を可能とする．さらに，クエリ内のオペレータ単位で適切な QE を選択実行するクエリ分割実行方式を備えることを特徴とする．共有ストレージとして Hadoop HDFS，QE として SQL-on-Hadoop の Hive と Impala および全文検索エンジンの Solr を対象としたプロトタイプを実装した．Hive と Impala に対して TPC-H ベンチマークを用いて評価した結果，Impala 単体で実行が失敗するクエリに対しても，適切な QE を自動選択することで実行可能とし，さらに分割実行により Hive 単体と比較して最大 2.5 倍高速化できることを確認した．また Solr と SQL-on-Hadoop の組合せで Twitter データを用いて評価した結果，テキスト検索において Solr を活用することで，SQL-on-Hadoop 単体と比較して最大 9.4 倍高速化し，またデータサイズが大きくなることでより高速化することを確認した．

キーワード：マルチデータベースシステム，仮想スキーマ，クエリ分割実行，クエリ処理エンジン，Hadoop

A Multidatabase System with Efficient Selection of Query Engines

Kazuhiro Saito[1,a)]　Yasuyuki Watanabe[1]　Shigeki Muramatsu[1]　Arei Kobayashi[1]

Received: March 21, 2015, Accepted: July 8, 2015

Abstract: Recently, new architecture of the data source has been developed, which can use multiple query engines on a shared storage. These query engines share same data, but have different characteristics each other. It includes the Hadoop and its eco-systems as a typical example. However, a multidatabase system cannot automatically select a QE to post a query to these QEs sharing same data. As a result, the users must use different QEs in spite of a logical single database system. This leads additional burdens of users. And in worst case, it occurs a problem that it leads system crash by the inappropriate selection of a QE. In this paper, we propose a multidatabase system which can automatically select an appropriate QE from these QEs sharing same data. The proposal system provides a virtual schema for users to recognize multiple QEs sharing same data as a single data source. When users execute a query to this virtual schema, the proposal system selects an appropriate QE automatically. Furthermore, the proposal system also provides split query execution method which select an appropriate QE for each operator of user query. We implemented the prototype system to evaluate the proposal system targeting Hadoop HDFS as a shared storage and two SQL-on-Hadoop QEs (Hive and Impala) and a full-text search engine Solr as multiple QEs sharing same data. In the result of evaluation for Hive and Impala by this prototype system using TPC-H benchmark, its efficient selection of query engines achieved to execute a query that Impala fails to execute, and we observed 2.5 times higher performance at a maximum than Hive only execution by its split query execution. Furthermore, in the result of evaluation for SQL-on-Hadoop and Solr by this prototype system using twitter data, we observed 9.4 times higher performance at a maximum than SQL-on-Hadoop only execution by using Solr at text search operator.

Keywords: multidatabase system, virtual schema, split query execution, query engine, Hadoop

1. はじめに

電子機器から生成されるログデータや，Web 上のサービスに関する情報等，様々なデータが生成されている．これらのデータは 1 つのデータソース（DS）に集約して活用することが理想的だが，実際には用途別に DS が増え続けている．よって，DS が複数あることを前提として，迅速なデータ連携の重要性が高まっている．対応する手段としてマルチデータベースシステム（またはデータ仮想化システム）があり，複数の DS を論理的に 1 つのデータベースシステムとして利用できる [1]．これにより，既存の DS に変更を加えることなく迅速なデータ活用を実現できる．

一般的には，たとえばデータベースシステムのように，ストレージと DBMS 等のクエリ処理エンジン（QE）がセットで 1 つの DS を構成する．一方で，ストレージと QE を分離して，共有ストレージ上のデータに対して異なる特徴を持つ複数の QE を利用可能な新しいアーキテクチャが存在する．それぞれの QE は同一のデータを共有し，データや処理の種類に応じた最適化が施されている．しかし，従来のマルチデータベースシステムは，対象の DS に，上記のような同一データを複数の QE が持つ環境を想定していない．同じデータを持つ DS（ストレージと QE のセット）が QE 数だけ存在すると認識する．したがって，DS を仮想的に隠蔽することができず，使い分ける必要が生じ，ユーザの負荷が発生する．さらにはユーザが不適切な QE を選択することで，大幅な遅延やシステムクラッシュ等の問題が発生する．

そこで本論文では，同一データを持つ複数の QE から，適切な QE を自動選択するマルチデータベースシステムを提案する．提案システムは，同一データを持つ複数の QE を単一の DS として認識可能とする仮想スキーマを提供する．この仮想スキーマにより適切な QE の自動選択を可能とする．さらに，クエリ内のオペレータ単位で適切な QE を選択実行するクエリ分割実行方式を備えることを特徴とする．QE の特性に応じて，QE 間連携を実現するクエリを自動的に生成し，振り分けることで，ユーザの負荷を軽減しつつ，QE の効率的な使い分けを提供する．また連携する QE の同一データが同じストレージで共有されている場合，そのストレージを活用した効率的な QE 間連携を可能とする．

代表的なシステム構成のユースケースとして**図 1** のように Apache Hadoop [2] がある．Hadoop はファイル形式でデータを保存でき，様々な規模の，かつ様々な形式のデータを 1 つのシステムに統合することができる．このようなデータに対応したクエリ処理を実現するために，HDFS を共通のストレージとして，特徴の異なる QE が数多く開発されている．データの規模に特徴がある QE の例としては，SQL-on-Hadoop がある．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 Apache Hive [3] は，大規模データに対する処理に適切である．一方で MapReduce を利用せず SQL 処理に特化してオンメモリで処理する Impala [4]，Apache Drill [5]，Presto [6] 等は，メモリに収まる小規模なデータ処理を高速に処理可能である．またデータや処理の種類に特化した QE の例として，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，グラフ型データに対するグラフ処理等，様々な QE が存在する．このような QE が複数存在する環境で提案システムを利用することで，ユーザは各 QE の特性を意識することなく，適切な QE を選択できる．

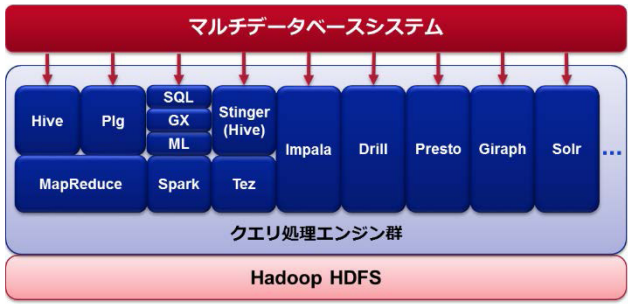


図 1　提案システムのユースケース
Fig. 1　A use case of our proposal system.

提案システムを利用した具体的な分析のユースケースとして，データ規模の異なる複数データが混在する環境における分析と，特性の異なる処理が混在する分析の 2 通りが考えられる．前者は，たとえば購入予測のように，大規模データである購入履歴と顧客情報との紐付けによる顧客ごとの趣味嗜好の予測を大規模データ向け QE（Hive 等）で実行して，顧客ごとの趣味嗜好と関連する商品を商品マスタから紐付けする処理を小規模データ向け QE（Impala 等）で行うといった分析処理を 1 つのクエリで行うことができる．また後者の分析では，たとえば Twitter を用いた分析において，全文検索の QE によって特定のフレーズ（たとえば「電波」「悪い」等）を発言しているユーザを抽出し，ユーザごとのその結果の数とフォロワ数を，集計機能を持つ QE（Impala 等）で集計することで，そのフレーズの Twitter における影響度を算出するといった分析を 1 つのクエリで行うことができる．提案システムはこのようにして，1 つのクエリに特性の異なる複数のデータや処理が混同している場合に，その特性に最適な QE をデータごとや処理ごとに自動選択してクエリ実行できる．これにより，ユーザの負担を軽減しつつ，単体の QE で実行するよりも高速化することが期待できる．

[1] 株式会社 KDDI 研究所
KDDI R&D Laboratories Inc., Fujimino, Saitama 356–8502, Japan
[a)] ku-saitou@kddilabs.jp

本論文では，提案システムを構成する仮想スキーマとクエリ分割実行方式を解説する．また，共有ストレージとして Hadoop HDFS，複数の QE として SQL-on-Hadoop の Hive と Impala，および全文検索エンジンを対象としたプロトタイプを実装し，提案システムの有効性を評価する．本論文の構成は以下のとおりである．2 章で従来研究の課題を述べ，3 章で提案システムの詳細を述べる．4 章で提案システムのプロトタイプ実装について述べ，5 章でこのプロトタイプによる評価を述べる．最後に 6 章で本論文の結論と今後の課題を述べる．

2. 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の研究は古くから行われており，RDBMS を対象とした多くのプロトタイプシステム（HERMES [7]，Pegasus [8]，TSIMMIS [9]，Garlic [10] 等）が開発されている [11], [12]．近年ではさらに，RDBMS だけでなく様々な DS を対象に統合スキーマを提供するデータ仮想化システム（GReIC [13]，OASIS [14]，ViDa [15]，Teiid [16] 等）が提案されている．これら従来のマルチデータベースシステムは，GAV 方式 [17] によって接続先の DS に対して 1 対 1 で仮想スキーマを構成する．しかし同一データを持つ複数の QE がある環境では，**図 2** に示すとおり，同一データに対して接続先ごとに同じスキーマの仮想スキーマが作成される．この環境でユーザがクエリを実行するためには，複数の仮想スキーマから適切な QE の仮想スキーマを手動で選択する必要がある．その結果，それぞれの QE の特性理解と，実行するクエリと適切な QE との特性のマッチングがユーザに求められる．また，クエリ単位による QE の選択では，1 つのクエリに特性の異なる処理が複数ある場合でもいずれかの QE への振り分けとなり，特性の異なる複数の QE がある環境の性能を最大限引き出すことができない．以上のことから，同一データを持つ複数の QE がある環境において，マルチデータベースシステムが解決すべき課題は以下のとおりである．

- 同一データに対する複数の仮想スキーマ
- 適切な QE の手動選択
- クエリ単位の QE 選択

2.2 関連研究

HadoopDB/Hadapt [18] は，Hadoop のデータノードとして各ノード上に RDBMS（PostgreSQL）を配置し，SQL クエリを MapReduce に変換することで分散処理を行い，各ノードでの処理を SQL に再変換して実行する．これは Hadoop を介してマルチデータベースシステムの機能を提供することで，同一データをレプリケーションとして扱い，仮想スキーマの課題を解決している．しかし，異なる特徴の QE が複数ある環境を想定していないため，適切な QE を事前に手動で選択する必要がある．

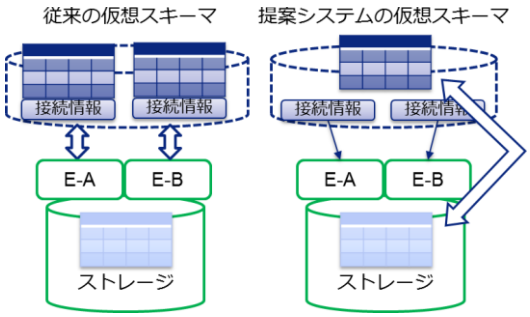


図 2　従来システムと提案システムの仮想スキーマ
Fig. 2　Virtual schema: existing system, proposal system.

Polybase [19] は Hadoop と RDBMS に対してクエリ分割を行うシステムで，SQL クエリ内の HDFS データに対する処理は MapReduce に変換され実行される．これにより，Hadoop を 1 つのデータベースシステムとして活用することを可能とした．しかし，Hadoop に同一データを扱う複数の QE があった場合には，従来のマルチデータベースシステムと同様の課題が発生する．

DBMS+ [20] および HFMS [21] は，対象となる複数のシステムを QE とストレージで分離して，クエリに応じてより適した QE やストレージを選択するシステムのコンセプトを提案している．DBMS+ は Hadoop 上の複数のストリーミングシステムを QE として，ユーザの要求に応じて CQL（Continuous Query Language）クエリを適切な QE で実行する研究である．HFMS は複数の QE をまたぐ複数クエリのワークフローの最適化に関する研究である．両者ともに適切な QE の自動選択を可能としているが，クエリ単位で QE を選択する課題を解決していない．

3. 提案システム

3.1 システムの概要

提案システムは，従来のマルチデータベースシステムの 3 つの課題を解決するために，以下の機能を提供する．

- 同一データに対する単一の仮想スキーマ
- 適切な QE の自動選択
- クエリの分割によるオペレータ単位の QE 選択

これにより，ユーザの負荷を軽減しつつ，クエリ実行の高速化を実現する．なおオペレータの定義は 3.2 節で述べる．

提案システムを利用して同一データを持つ複数の QE 上で分割クエリを実行するためのシステムアーキテクチャを**図 3** に示す．クライアントは提案システムに対して，単一の仮想スキーマによって，単体のデータベースシステムと同様にクエリを投稿する．提案システムは，クエリ分割実行方式により，投稿されたクエリに適切な QE を選択し，クエリを実行する．クエリ内のオペレータごとに異なる QE に適していると判断した場合，分割クエリを生成して適切な QE で分割クエリを実行する．これを実現するため

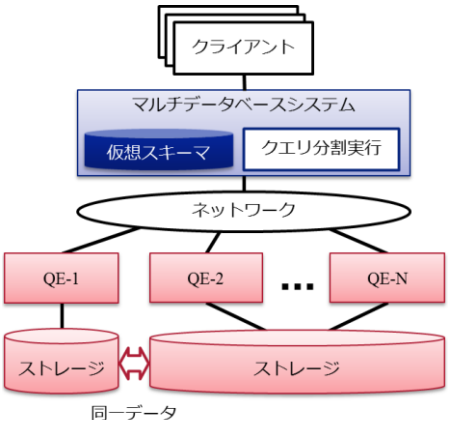


図 3　システムアーキテクチャ
Fig. 3　System architecture.

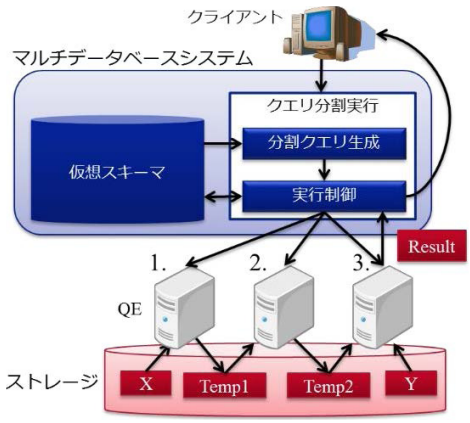


図 4　提案システムの構成とクエリ実行例
Fig. 4　System components and Execution example of a split query.

に，クエリ分割実行方式は，クエリをオペレータ単位で分解してそれぞれに適切な QE を割り当てる分割クエリ生成と，生成した分割クエリを適切な順序で実行する実行制御で構成される．このように，提案システムを利用することでユーザには透過的に適切な QE を自動選択することが可能となり，かつ分割クエリを生成してそれぞれの処理を適切な QE に振り分け，効率的なクエリ処理が可能となる．

このアーキテクチャでは，複数の QE が同一のデータをストレージ上で共有している場合に，QE の切替えのたびに中間データをネットワーク上に転送する必要がなく，低遅延での切替えが可能となる．このような共有ストレージを活用した QE 切替えを実現するためには，対象が共有ストレージであることを認識する仮想スキーマと，複数の QE に振り分ける際に共有ストレージを活用する分割クエリ生成および実行制御が必要となる．図 4 にこの例を示す．この例では，3 つの QE が以下のとおり実行した例を示している．

① X に対して処理を行った結果を Temp1 としてストレージに書き込む．
② Temp1 に対して処理を行った結果を Temp2 としてストレージに書き込む．
③ Temp2 と Y に対して処理を行った結果を最終結果 Result としてマルチデータベースシステムに返す．

以下の節では，提案システムの前提条件とともに対象とするクエリと QE について述べ，提案システムを実現するための仮想スキーマおよびクエリ分割実行方式について述べる．

3.2　対象範囲

提案システムのクエリ分割はクエリプランにおけるオペレータ単位で行われる．たとえば SQL では，射影演算や結合演算等の演算子や，集計関数や UDF 等のそれ以上分解不可能なモジュールがオペレータとなる．なおオペレータ自体は分割せず，同様に処理対象のデータも分割しない．また分割適用前後でクエリ処理結果の一貫性を保証するために分割したクエリの実行順序を制御する．

提案システムのユーザインタフェースであるクエリ言語は，クエリ分割のためにクエリプランによってオペレータが明確になればよい．したがって，特定のクエリ言語に依存せず，クエリプランを生成できるクエリパーサが実装されていれば，独自のクエリ言語であってもクエリ分割が可能である．

対象となる QE は，提案システムの実装で定義したオペレータを，SQL やコマンド等の事前に決められたクエリで実行可能な QE である．したがって，提案システムのオペレータと，対象の QE のクエリの変換ルールを事前に設定する必要がある．また，対象の QE 群は同一のデータを共有している必要があり，ある QE が更新したデータが，他の QE でも即時反映されることが前提となる．

QE の組合せパターンとして，リソース利用量によって実行の可否や処理性能が異なる QE の組合せと，特定のデータやオペレータにおいて実行の可否や処理性能が異なる QE の組合せがある．前者に関しては，ある範囲のリソース利用量において QE どうしに実行可否の違いや性能の優劣がある QE の組合せにおいて適用可能である．リソース利用量の例として，あるオペレータにおける利用メモリサイズや，CPU コアの利用数，ネットワーク負荷等が該当する．後者に関しては，ある特定のオペレータに特化して他の QE より高速な QE の組合せにおいて適用可能である．またこれらを組み合わせて，ある一定のリソース環境下で，特定のオペレータの性能が低下する QE でも同様に適用可能である．

3.3　仮想スキーマ

提案システムは，ユーザに接続先の QE を意識させないために，QE 数に依存せず，ストレージ上の同一のデータであるスキーマと仮想スキーマを 1 対 1 で紐付ける．そのために，図 2 のように仮想スキーマの情報をスキーマと接続

情報で分離する．これをそれぞれ，図 3 の仮想スキーマにおけるデータソース情報とエンジン情報として管理する．データソース情報は，ユーザに見せるスキーマとして提供される．また，対象のデータの統計情報（データサイズや最大/最小値等）もあわせて保存される．エンジン情報は，提案システムと QE の物理的な接続情報が保存される．さらに，QE の処理ごとの性能や仕様（実行不可クエリ，処理アルゴリズム等）が保存され，処理を実行する QE を判断するうえで利用される．この仮想スキーマにより，同一データを持つ複数の QE に対して，複数の仮想スキーマが生成されなくなり，ユーザによる仮想スキーマの選択が不要となる．またこの仮想スキーマを利用することで，以降で述べるクエリ分割実行方式による QE の自動選択が可能になる．

3.4　クエリ分割実行方式

3.4.1　分割クエリ生成

図 4 のクエリ処理方式を実現するための分割クエリの生成について述べる．**図 5** に処理フローを示す．

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し，実行する QE を決めるために必要なクエリの情報を，クエリプランとデータソース情報から抽出する．ここでのクエリプランは処理コストがより小さくなるように生成できればよく，従来のマルチデータベースシステムや Hadoop におけるワークフローの最適化手法（Garlic [10]，Starfish [22]，Hyracks [23] 等）が適用可能である．ここで生成したクエリプランに対して，実行する QE の判断基準となるリソース利用量の情報を付与する．この情報はクエリプランのオペレータごとに，データソース情報にある統計情報等を利用して計算される．そしてこのクエリプランに付与された QE の判断基準となる情報とエンジン情報を利用して，オペレータごとに実行する QE を決定する．なお QE の判断基準については 3.4.4 項で述べる．最後に，各 QE の実行形態や文法に従って，実行可能なクエリを生成する．またあわせて，QE の切替えに必要な管理クエリも生成する．

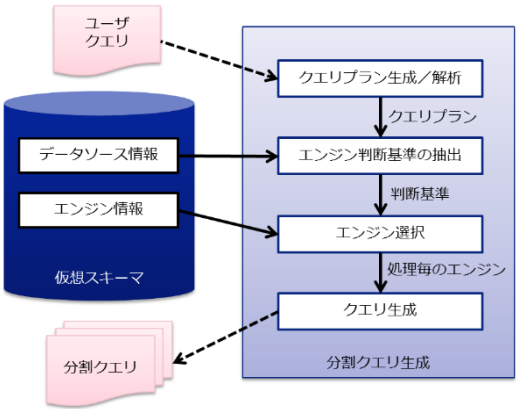


図 5　分割クエリ生成の処理フロー
Fig. 5　Process flow of split query generation.

複数の QE を利用する分割クエリでは，各 QE で行われるクエリの解釈・最適化処理や，QE 切替え時の中間データの読み書き等のオーバヘッドによる分割損が発生する．分割クエリを生成することで，この分割損により元のクエリの性能を下回るのであれば，これを回避する必要がある．提案システムの分割クエリの生成では，後述する分割クエリのパフォーマンスモデルを導入し，分割損による性能低下を回避する．

3.4.2　実行制御

分割クエリの実行時，各クエリは中間データを必要とするため，中間データを生成するクエリが終了するまで次のクエリを実行することができない．クエリ実行制御では，この依存関係を考慮して分割クエリの実行を制御する．

まず生成した分割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．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つ．これが届き次第，次のクエリを実行する．この処理をすべての分割クエリが実行完了するまで行い，最終的なクエリ結果のデータを取得した段階で分割クエリの実行は完了となる．また，クエリ実行時のデータを分析し，件数やサイズ等を取得して，投稿時のクエリのオペレータとあわせて分析することで，統計情報としてオペレータごとの選択率等のデータソース情報を収集する．この情報は，分割クエリ生成時に，オペレータごとのリソース利用量の計算や，QE の判断基準に利用される．

ここで QE におけるクエリ実行失敗時について述べる．クエリ実行が失敗する理由は，以下の 4 つが考えられる．

① オペレータごとのリソース利用量の予測の誤差
② QE の判断基準の誤り
③ システム障害
④ 他ユーザによるリソース利用可能量の動的な変化

クエリの実行失敗時の対処は，それぞれの事象によって異なる．①の予測誤差および②の判断基準の誤りに関しては，仮想スキーマの情報（予測パラメータや，利用可能なリソースの上限等）を更新することで，再度分割クエリの生成を試みることが可能である．①および②で更新内容が不明な場合は，分割クエリの生成を行わず，元のユーザクエリを確実に実行可能な QE 単体で実行する必要がある．なお実行可能な QE がない場合はエラー終了となる．③の場合は，クエリ実行が不可能なため，エラー終了となる．これは，事前に疎通試験を行っておくことで，クエリ失敗までのロスを排除することが可能となる．一方④が発生するマルチユーザ環境では，①および②との切り分けが困難になり，①および②も含めて，クエリ実行失敗時はエラー終了となる．これは，失敗原因が分割クエリ生成上の問題なのか，リソース利用可能量の変化によるものなのか，を切り

分けができないためである．なお QE が Linux の cgroups 等を利用して，ユーザごとに利用可能なリソースの上限が変化しない場合には④は発生しない．

3.4.3　パフォーマンスモデル

分割クエリの生成による分割損を考慮するために，分割クエリのパフォーマンスモデルを定義する．パフォーマンスモデルを表現するうえで，$|E|$ 個の QE の集合 E は同一のデータを保持し，各 QE $E_x(1 \leq x \leq |E|)$ は提案システムに接続されている環境を想定する．このとき，クライアントからクエリ Q が投稿されると，提案システムは $|q|$ 個のクエリの集合 q に分割し，各クエリ $q_i(1 \leq i \leq |q|)$ に対して適切な QE E_{x_i} を対応付ける．最終的に生成される分割クエリ $splitQ$ は，分割されたクエリの集合 q と，中間データの保存場所作成等の管理クエリの集合 $mqList$ を含む．このとき，分割する前のクエリ Q を QE E_x で実行した場合と，提案システムを用いて生成した分割クエリ $splitQ$ を実行した場合の性能比 $Perf(\cdot)$ は以下のように定式化できる．

$$Perf(Q, E_x, splitQ) = \frac{Cost(E_x, Q)}{C(mqList) + \sum_{i=1}^{|q|} Cost(E_{x_i}, q_i)} \quad (1)$$

ここで，$C(mqList)$ は管理クエリ $mqList$ にかかるコストを表し，$Cost(E_x, Q)$ はクエリ Q を QE E_x で実行した場合のコストを表す．式 (1) の右辺の分母は，提案システムが生成した分割クエリの総実行コストを表し，分子は分割せずに 1 つの QE で実行した場合のコストを表す．

提案システムでは，式 (1) を用いて分割クエリのパフォーマンス判定を行う．$Perf(Q, E_x, splitQ) > 1$ であれば，クエリ Q を単体の QE E_x における実行時より，提案システムで生成した分割クエリ実行時の性能が高い．一方で，$0 < Perf(Q, E_x, spiltQ) < 1$ であった場合は，分割クエリの実行による性能向上は見込めない．

$Cost(E_x, Q)$ は QE の実装に依存している．これには大きく 2 つの要素があり，クエリ最適化等の初期化にかかるコスト $Init(E_x, Q)$，およびクエリ処理アルゴリズムやデータアクセス手法によって決まるデータ処理コスト Exec (E_x, Q) に分解することが可能である．これを利用して，性能が向上する条件 $Perf(Q, E_x, splitQ) > 1$ を展開すると，以下のようになる．

$$Exec(E_x, Q) - \sum_{i=1}^{|q|} Exec(E_{x_i}, q_i) > C(mqList) - Init(E_x, Q) + \sum_{i=1}^{|q|} Init(E_{x_i}, q_i) \quad (2)$$

ここで左辺はクエリ分割によるクエリ処理の性能向上であり，右辺が分割によって発生するオーバヘッドである．

利用している各コストを事前に計測もしくは推定することで，式 (2) の評価により提案システムのクエリ分割による性能劣化を回避することが可能となる．

パフォーマンスモデルの適用例を以下で述べる．あるクエリ Q を同一のデータを持つ 2 つの QE A および B に対して振り分けた場合の計算例を示す．まず性能向上するパターンとして，クエリ Q を q1，q2，q3 の 3 つのクエリに分割し，管理クエリ 3split を生成した場合の例をあげる．この分割クエリは，q1 を B で，q2 を A で，q3 を B で実行する．なお，各 QE の初期化処理はクエリにかかわらず同一とする．このときの式 (2) における各コストが**表 1** であったとすると，式 (2) の左辺と右辺はそれぞれ以下のようになる．

$$Exec(A, Q) - \sum_{i=1}^{3} Exec(E_{x_i}, q_i) = 297.34 \quad (3)$$

$$C(3split) - Init(A, Q) + \sum_{i=1}^{3} Init(E_{x_i}, q_i) = 4.55 \quad (4)$$

これらの式から，左辺 > 右辺が成り立ち，クエリ Q を 3 分割することで性能向上が見込めることが分かる．次に，性能低下するパターンとして，クエリ Q を A で実行する q4 と B で実行する q5 の 2 つに分割し，管理クエリ 2split を生成した場合の例をあげる．これらのコストが同様に表 1 であったとすると，式 (2) の左辺と右辺はそれぞれ以下のようになる．

$$Exec(A, Q) - \sum_{i=4}^{5} Exec(E_{x_i}, q_i) = 2.18 \quad (5)$$

$$C(2split) - Init(A, Q) + \sum_{i=4}^{5} Init(E_{x_i}, q_i) = 3.23 \quad (6)$$

この分割パターンでは，左辺 > 右辺が成り立たず，クエリ Q を 2 分割することで性能劣化すると判断できる．

3.4.4　QE の判断基準

分割クエリの生成においては，オペレータごとの実行対象の QE を決める判断基準が重要となる．この判断基準の基本的な考え方は 2 通りあり，(1) 実行の可否と，(2) QE 間の性能差である．(1) 実行の可否は，オペレータの種類とリソースの制限から決まる．オペレータの種類による判断は，対象の QE のクエリの文法上の対応可否が基準となる．リソースの制限による判断は，対象 QE の CPU コア

表 1　クエリ Q における各コスト
Table 1　Costs in a query Q.

Var.	Time (s)	Var.	Time (s)
$C(3split)$	4.13	$Exec(B, q_1)$	177.35
$C(2split)$	3.02	$Exec(A, q_2)$	2978.14
$Init(A, Q)$	11.81	$Exec(B, q_3)$	1.97
$Init(B, Q)$	0.21	$Exec(A, q_4)$	3450.59
$Exec(A, Q)$	3454.80	$Exec(B, q_5)$	2.03

Algorithm 1:　Selecting Query Engine (selectEngine)

```
    Input: planTree: Query plan,
           engineInfoList: Query engine list
    Output: planTree: Query plan binding query engine
 1  preselectEngine = null;
 2  for each operator op in post order from planTree
 3  |  op.engine = null;
 4  |  for each engine from engineInfoList
 5  |  |  level = evaluateEngine(op, engine);
 6  |  |  if level is 1
 7  |  |  |  op.engine = engine
 8  |  |  else if level is 0 and
 9  |  |  |          preselectedEngine equals to engine
10  |  |  |  op.engine = engine
11  |  |  |
12  |  |
13  |  If op.engine = null
14  |  |     error("Can't execute this query");
15  |  |
16  |  op.calcCostCriteria();
17  |  preselectedEngine = op.getEngine();
18  |
19  If checkPerformance(planTree) is false
20  |  engine = engineInfoList.getSafeEngine();
21  |  selectEngine(planTree, engine);
22  |
```

Algorithm 2:　Evaluating query engine (evaluateEngine)

```
    Input: op: an operator of the original query,
           engine: Evaluated query engine
    Output: appropriateLevel: result of evaluation
 1  if engine.executable(op) is true
 2  |  if op.engine is null
 3  |  |  appropriateLevel = 1;
 4  |  else if engine.getPerf(op)
 5  |  |          > op.engine.getPerf(op)
 6  |  |  appropriateLevel = 1;
 7  |  else if engine.getPerf(op)
 8  |  |          < op.engine.getPerf(op)
 9  |  |  appropriateLevel = -1;
10  |  else // same performance
11  |  |  appropriateLevel = 0;
12  |  |
13  else
14  |  appropriateLevel = -1;
15  |
```

数や，メモリ，ストレージ，ネットワーク等のリソースの利用可能な量を条件とし，オペレータのリソース利用量が基準となる．(2) QE 間の性能差は，対象のオペレータにおける定量的な処理性能の差が判断基準となり，性能が高い QE を選択する．

上記の判断基準は，事前に仮想スキーマのエンジン情報に設定する必要がある．提案システムが受け入れ可能なオペレータすべてに対する実行の可否および性能と，リソースの利用可能上限を設定する．オペレータに対する性能は，QE がオペレータを処理する際の性能を表し，QE ごとに全オペレータに対して設定される．これは複数の QE 間での性能差を判断するうえで利用され，定数値として QE 間の性能の優劣が判定できるように設定する．さらにこれらの判断基準を組み合わせて設定することで，たとえばある特定のオペレータで一定のメモリ利用量を超える場合に限り特定の QE を利用し，これを超えない場合や他のオペレータでは他の QE を利用する，という動作が可能になる．

3.4.5　QE の選択アルゴリズム

クエリプランの全オペレータの適切な QE を決定する手順を Algorithm 1 に示す．

初めに，クエリの一連のオペレータを持つクエリプラン *planTree* から全オペレータの適切な QE を選択する．オペレータごとに全 QE から適切な QE を evaluateEngine(*op, engine*) を利用して評価し，適応度を取得する．適応度の判定は，後述するアルゴリズムによって行う．適応度が 1 の場合は評価対象の QE *engine* が適切であり，その *engine* をオペレータ *op* の *op.engine* に設定する．0 の場合は差がなく，QE の切替えによるオーバヘッドをできる限り減らすために，対象の *op* の直前の *op* で選択された QE *preselectedEngine* と同じものを選択する．オペレータ *op* に対して全 QE の適応度を評価した段階で，3.4.3 項で述べたパフォーマンスモデルの適用に必要なコスト情報を CalcCostCriteria() で計算する．以上を *planTree* の全オペレータ分繰り返すことで QE 選択を完了する．

最後に，オペレータごとのコスト情報を用いて式 (2) を評価する関数 checkPerformance(*planTree*) で分割クエリのパフォーマンスを評価する．このとき，性能劣化すると判断した場合には分割をやめ，確実に実行可能な QE 単体での実行に切り替える．

適応度の判定を行う evaluateEngine() のアルゴリズムを Algorithm 2 に示す．ここでは，3.4.4 項で述べたように，仮想スキーマのエンジン情報にある QE *engine* の判断基準を利用して行われる．まず対象のオペレータ *op* の実行可否を executable(*op*) で判断する．実行可能である場合，すでに *op* に登録済みの QE *op.engine* と判定対象の *engine* における *op* の性能を比較する．これにより，2 つ以上適切な QE がある場合に，より適切な QE を選択することができる．engine.getPerf(*op*) は，事前に設定したオペレータ op における QE engine の性能の定数値を取得する．適応度として，性能が高いまたは *op.engine* が null の場合は 1 を，処理できないまたは性能が低い場合は −1 を，差がない場合は 0 を返す．

3.5　システム管理

提案システムを導入してクエリ分割実行を運用するうえで，事前に接続先 QE の判断基準をオペレータの種類の数

だけ登録する必要がある．そのためにシステム管理者は，QE の特性として各 QE のリソースの利用可能上限と，オペレータごとの相対的な性能差を算出する必要がある．QE が個別で動作している場合と比較して，ユーザの QE 選択によるコストは削減されるが，システム管理者は導入時に上記のコストが発生する．また同様にシステム導入時に，QE の切替えで発生する中間データの容量だけ，QE のストレージの容量を確保する必要がある．これは一時的な利用にすぎないが，QE のストレージの空き容量がない場合は中間データを QE 間でやりとりできないため，クエリ処理が失敗する．この容量は，実行するクエリによって異なるため，事前に投稿されるクエリを把握して容量を確保するか，事前に決めた容量を超える場合はクエリ実行を失敗する前提の運用となる．

4. プロトタイプ実装

4.1 対象システム

プロトタイプでは，Hadoop HDFS を共有ストレージとする SQL-on-Hadoop システムおよび全文検索エンジンを対象とした．SQL-on-Hadoop の QE には，Hive [3] と Impala [4] を利用した．これら QE はスキーマ情報を保持する Metastore を共有するため，同一データに対してクエリ処理を行う．全文検索エンジンには Solr [24] を利用した．Solr は SQL-on-Hadoop と同一のデータを利用し，そのインデックスは Cloudera Search を利用して Hadoop の HDFS 上に配置した．

Hadoop のディストリビューションは CDH5.0 を利用した．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，Zookeeper が稼働するマスタノードが 1 台，Hive Server，Hive Metastore Server，Solr Server，プロトタイプが稼働するクライアントが 1 台の計 5 台とした．Metastore とプロトタイプの仮想スキーマの保存は PostgreSQL 8.4.13 を利用した．すべてのサーバの OS は CentOS 6.4 である．ハードウェア環境を**表 2** に示す．これらのサーバは 1GbEthernet の同一ネットワーク上に構築している．

表 2　プロトタイプのハードウェア環境
Table 2　Hardware environments of our prototype system.

	スレーブノード	マスタノード	クライアント
サーバ	Dell PowerEdge 1950	Sun Fire X2100 M2	Dell PowerEdge 2950
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

4.2 プロトタイプにおける QE の判断基準

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．しかし，MapReduce 自体がバッチ処理向けであり，インタラクティブな処理には向いていない．たとえば，MapReduce はジョブごとの初期化処理や，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が発生する．特にサブクエリを含む複雑なクエリでは，複数回のジョブが実行され，上記のオーバヘッドがより大きくなる．

一方 Impala は MapReduce を使わないことで Hive におけるオーバヘッドを削減し，より高速なクエリ処理が可能である．しかし高速化のためにすべてのデータをつね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．さらに場合によってはアウトオブメモリで異常終了する．よって 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必要がある．

Hive および Impala のインタフェースは SQL ライクな言語である HiveQL であり，実装のバージョンによってはどちらかが利用できないオペレータがある．しかし，本実装ではどちらも実行可能なクエリのみを対象とし，オペレータの違いによる振り分けは行わない．

以上のことから，Hive および Impala の判断基準を処理対象のデータサイズとし，Impala には「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 には「クラスタ全体の物理メモリサイズの合計を超える場合」に振り分ける．プロトタイプの仮想スキーマのエンジン情報には Impala および Hive の適応度判定 evaluateEngine() で利用する情報として，リソースの制限であるメモリの利用可能上限と，処理性能の差を格納した．メモリの利用可能上限に関して，Hive は上限となるデータサイズがない設定とし，適応度判定における executable(*op*) で必ず true となる設定とした．Impala はスレーブノード 3 台の物理メモリサイズの合計の 70%である 67.2 GB を設定した．処理性能の差は，全オペレータで Impala の方が Hive より優れているとした．なお，本プロトタイプでは，Hive が UNION 演算に対応していないことから，UNION 演算と，同様の処理である WHERE 句の OR 演算に未対応である．また Hive が対応していない WHERE 句内サブクエリも同様に未対応である．

Solr は全文検索向けのインデキシングが可能であり，テキストデータに対する部分一致検索に関して Hive および Impala より優っていることから，判断基準はテキスト検索クエリに関する処理性能となる．したがって，WHERE 句における LIKE 演算の性能が Hive と Impala よりも優れている設定とした．またこれらの演算以外は実行不可とし，

メモリの利用可能上限はない設定とした．

4.3　プロトタイプの実装

プロトタイプのクエリプランは，Impala を利用して生成している．具体的には，ユーザクエリに explain 句を付与して Impala に投稿し，クエリプランを取得している．このクエリプランを字句解析し，プロトタイプのクエリプランとして利用する．

仮想スキーマの情報として，データソース情報に Impala，Hive，Solr が共有しているテーブルのメタデータ情報とその統計情報を，エンジン情報に各 QE への接続情報を内部の PostgreSQL に保持する．ユーザにはこのデータソース情報を提供することで，単一の仮想スキーマを表現している．

プロトタイプでは，実装の簡易化のために Hive および Impala の選択をサブクエリ単位で行う．まずオペレータごとに中間データサイズを推定し，入力データサイズと中間データサイズの合計が最も大きいオペレータをサブクエリごとに抽出する．これをサブクエリにおける最大利用メモリサイズとして，QE の選択時の適応度判定に利用する．適応度判定では，サブクエリ単位で最大利用メモリサイズと各 QE のメモリの利用可能上限を比較する．サブクエリの最大利用メモリサイズが Impala の利用可能上限である 67.2 GB 以下の場合は，処理性能の差から，対象のサブクエリに Impala を振り分ける．一方 Impala の実行可能上限を上回った場合，サブクエリに Hive を振り分ける．Solr に関しては，WHERE 句に LIKE 演算があった場合に，当該オペレータのみを抽出して Solr 向けのクエリを生成する．

オペレータごとの中間データサイズは，入力データサイズに対する中間データサイズの比率である選択率（Selectivity Factor）を利用して，オペレータごとの入力データサイズと選択率の積から推定する．そのために，推定対象のオペレータの選択率を，処理対象のテーブルと処理の条件（JOIN の結合条件等）ごとに事前に計測してデータソース情報に保存している．また出力後の属性も考慮し，射影演算により変化する属性のサイズの比も利用して中間データサイズを計算する．

分割後のクエリは，Hive および Impala に対して HiveQL クエリとして生成し，Solr に対しては HTTP リクエストの URL のクエリを生成する．異なる QE を跨がるクエリを生成する場合，QE 間の中間データ受け渡しに共有データソースである HDFS を利用する．Impala は Hive のメタデータを内部にキャッシュして利用しているため，Hive で更新されたメタデータを明示的に取得する必要がある．そのため，Impala を利用する場合，分割クエリの最初に「INVALIDATE METADATA」クエリを，中間データ挿入後に「REFRESH ＜テーブル名＞」クエリを挿入している．このような分割クエリの生成例として，**図 6** のクエリを

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

図 6　サンプルクエリ
Fig. 6　Sample query.

```
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　プロトタイプにおけるサンプルクエリの分割例
Fig. 7　Sample queries split by our prototype system.

Hive と Impala 向けに分割したクエリを**図 7** に示す．左端の数字はクエリを実行する順序を表しており，その横の括弧が実行する QE である．また，Solr はクエリ結果を直接 HDFS に挿入することができないため，Solr へのクエリ実行後に，クエリ結果を HDFS に書き込み，LOAD 句を利用して Hive もしくは Impala へロードする．

5.　評価

提案システムは，オペレータの実行可否や性能差から適切な QE を選択してクエリ実行することで，ユーザの手間を軽減しつつ高速化を実現するシステムである．この有効性を評価するために，5.1 節で処理サイズの違いによるクエリ分割，5.2 節でオペレータの違いによるクエリ分割を，プロトタイプを用いて評価する．処理サイズの違いによるクエリ分割の評価によって，リソースの利用量でオペレータの実行可否や処理性能が決定する QE の組合せにおいて，提案システムが有効であることを示す．さらにオペレータの違いによるクエリ分割の評価によって，特定のデータやオペレータにおいて実行可否や処理性能が異なる QE の組合せにおいても，提案システムが有効であることを示す．

5.1　システム評価（Hive/Impala）

本節では，処理サイズの違いによるクエリ分割に関して，Hive および Impala を用いて，プロトタイプの機能，性能，および中間データサイズ推定を評価する．機能評価では，提案システムの提供する 3 つの機能が課題を解消することを確認する．性能評価では，上記課題の解消によって得られたプロトタイプの性能を定量的に評価する．中間データサイズ推定では，QE の判断基準である中間データサイズ推定の精度を評価する．

5.1.1　評価環境

本評価では 4 章で述べたプロトタイプを利用するが，Solr への振り分けはここでは行わない．Hive および Impala の単体の実行時間に関しても同環境を利用した．このテストクエリには TPC-H [25] を利用し，テーブルサイズを表す

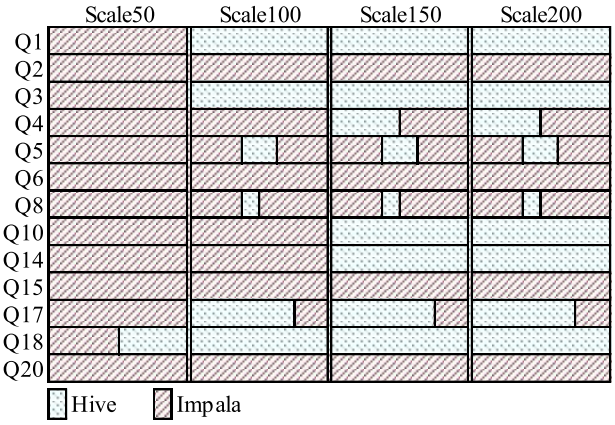


図 8　各テストクエリにおいて QE が実行したサブクエリの範囲
Fig. 8　Areas of subqueries that query engine executes in each test query.

Scale は 50（全 50 GB），100（全 100 GB），150（全 150 GB），200（全 200 GB）の 4 種類とした．ただし，TPC-H のクエリはそのままでは Hive，Impala で実行できないため，HiveQL 仕様に対応したクエリを利用している [26]．主な変更点は以下の 2 点である．(1) Impala が未対応である EXIST 句や IN 句によるサブクエリを 2 つのクエリに分割して，別々のクエリとした．(2) FROM 句におけるカンマ区切りの暗黙的な JOIN を，明示的にサブクエリによる JOIN に変更した．利用した TPC-H のクエリは，プロトタイプで未対応の UNION 演算と OR 演算，WHERE 句内サブクエリを含むクエリを除外した．また，Q9 および Q21 は，Impala から取得するクエリプランの処理順序と，元のサブクエリの順序が異なるため，サブクエリ単位で処理するプロトタイプの仕様では対応できず対象外とした．

プロトタイプを利用せずに Hive か Impala を選ぶ場合，ユーザはどちらが実行可能かを判断する必要がある．本環境では，Scale 50 を実行する場合であっても，JOIN 等を考慮すると Impala の最大利用メモリサイズがメモリ利用可能上限 67.2 GB を超える可能性があり，Impala で実行可能と判断することができない．したがって，確実に実行可能な Hive を選択すると想定し，Hive と比較することでプロトタイプの性能を評価する．

5.1.2　機能評価

図 8 は，TPC-H の 13 個のテストクエリをプロトタイプで実行し，Hive および Impala が実行した範囲を，分割単位であるサブクエリ（ベースクエリ含む）単位で Scale 別に示している．Q1，Q3，Q6，Q10，Q14 はサブクエリがなくクエリ単体のため，Scale の変化に応じて Hive か Impala のどちらかに振り分けている．これにより，提案システムによりユーザには透過的に適切な QE を自動選択できていることが分かる．上記以外のクエリ Q2，Q4，Q5，Q8，Q15，Q17，Q18，Q20 は，複数のサブクエリもしくは複数のクエリで構成されている．その結果のうち，Q5，Q8，Q17 においては，1 つのクエリの中で，より適切な QE をサブクエリ単位で選択してクエリ分割を行っている．このことから，提案システムによってオペレータ単位による QE の自動選択が可能となることがいえる．

5.1.3　性能評価

5.1.2 項と同じ TPC-H の 13 個のテストクエリを，プロトタイプと Hive で各 3 回実行し，その平均実行時間を計測した．**図 9** がその結果であり，各グラフは TPC-H データセットの Scale 別に示している．図 9 (a) が Scale 50，図 9 (b) が Scale 100，図 9 (c) が Scale 150，図 9 (d) が Scale 200 である．

5.1.2 項で述べたように，Q1，Q3，Q6，Q10，Q14 は Hive か Impala のどちらかに振り分けている．これらの性能評価結果として，Hive に振り分けたクエリ・Scale（Q1 の 100 等）では，図 9 で Hive とほぼ同じ結果となった．一方，Impala で実行したクエリ・Scale（Q1 の 50 等）は大きく性能向上した．上記以外のクエリは，上記同様，すべてのサブクエリを Impala に振り分けたクエリ・Scale では，Hive と比較して高速化し，すべてのサブクエリを Hive に振り分けたクエリ・Scale は Hive とほぼ同じ結果であった．また，提案システムにおけるクエリ分割実行方式によって，一部のサブクエリを Impala に振り分けているクエリ・Scale（Scale 50 の Q18 等）では，そのすべてで Hive 単体よりも高速に実行できることが確認できた．

次に，パフォーマンスモデルでは考慮ができない提案システムのオーバヘッドがクエリ分割実行によってどの程度発生しているかを評価した．本評価を行うにあたり，評価対象のオーバヘッドを大きく 2 つに分類した．1 つが，仮想スキーマと各種データソースへの接続処理，分割クエリ生成処理を含む初期化処理であり，もう 1 つが接続した仮想スキーマおよび各種データソースへの接続解除の終了処理である．計測対象のクエリは，5.1.2 項で実行したクエリのうち，特徴的なクエリ Q4，Q5，Q8 を選択した．これらのクエリは，ベースクエリを含むサブクエリがそれぞれ 2 個，5 個，8 個と処理の複雑さが異なっている．これら 3 つのクエリの初期化処理時間および終了処理時間を計測した結果が**図 10** である．なお Q4 は Q4-1 と Q4-2 に分けてグラフ化している．これは EXIST 句に対応するために 2 つのクエリに分けて実行していて，それぞれで初期化および終了処理が発生しているためである．Q4-1 は，Q4 のうち EXIST 句内のサブクエリであり，Q4-2 は Q4-1 の結果と EXIST 処理対象だったテーブルとの JOIN を行っている．

図 10 (a) における初期化処理に関しては，Q4-1 の Scale 150 と 200 を除き，サブクエリの数に比例して処理時間が長い．図 10 (b) における終了処理に関してもほぼ同様の結果であるが，時間のオーダが 1 桁異なり，0.01〜0.05 秒程度の処理時間であった．これらのオーバヘッドは，クエリ実行時間の 1%未満であり，プロトタイプに対して影響が

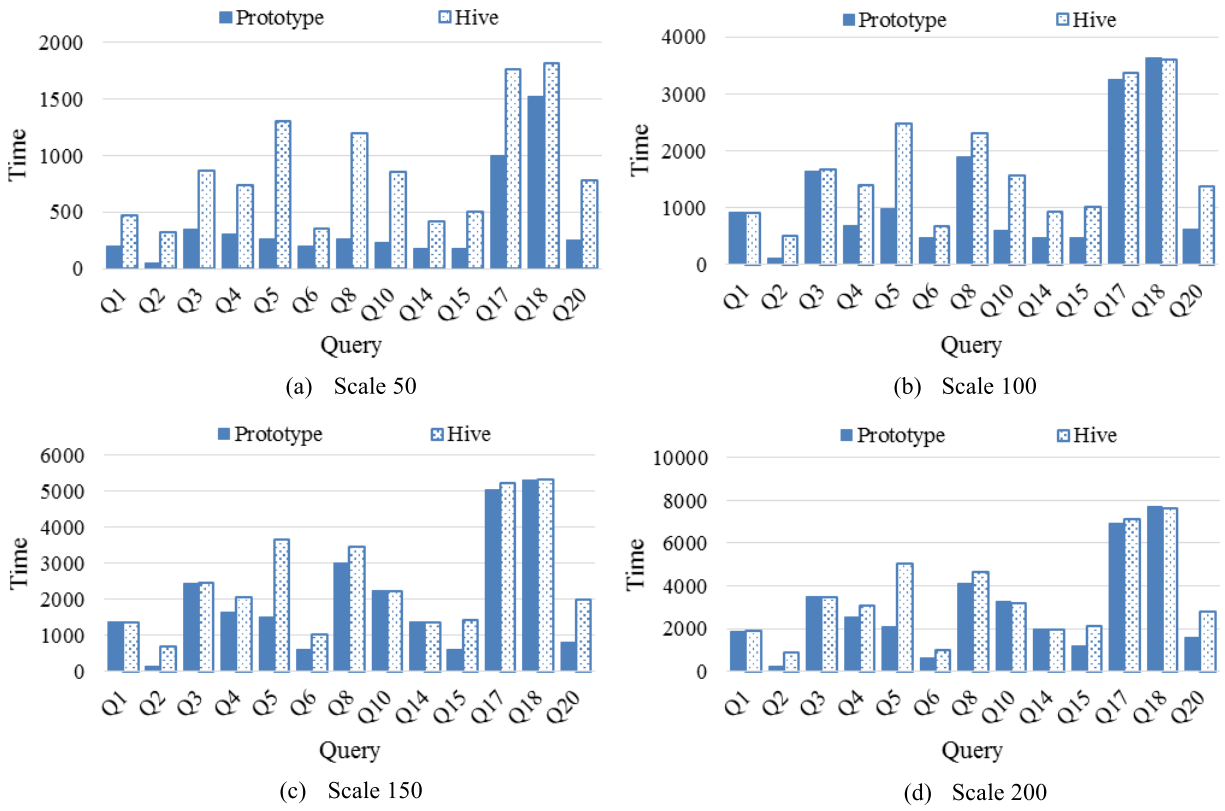


図 9　SQL-on-Hadoop における性能評価結果

Fig. 9　Results of evaluations for performance on SQL-on-Hadoops.

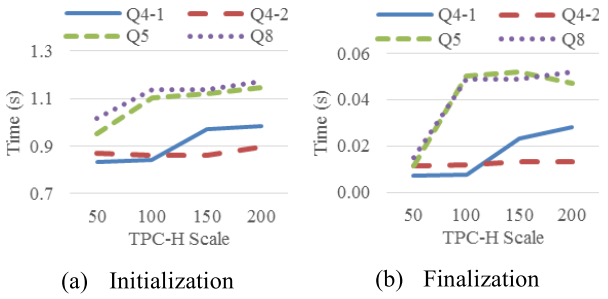


図 10　オーバヘッド評価結果

Fig. 10　Results of evaluations for overhead.

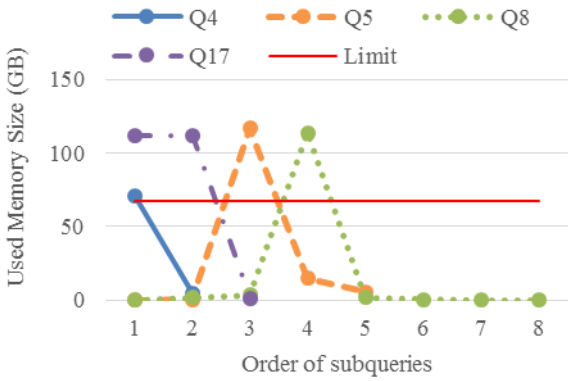


図 11　Scale 150 における推定したサブクエリごとの最大利用メモリサイズ

Fig. 11　Predicted peak used memory sizes of each subquery on scale 150.

ないといえる．

以上の性能評価から，リソースの利用量でオペレータの実行可否や処理性能が決定する QE の組合せにおいて，提案システムが有効であることを示した．

5.1.4　中間データサイズ推定評価

プロトタイプにおける選択率による中間データサイズ推定は，実行する QE を判断する重要な要素であり，提案システムの課題解決に対して大きな影響を与えることから，この精度について評価した．ここでは，Hive および Impala 両方に分割実行している Scale 150 の Q4，Q5，Q8，Q17 を用いて，プロトタイプがサブクエリごとに予測した最大利用メモリサイズを計測した．その結果が図 **11** である．横軸がサブクエリの実行順序で，縦軸が最大利用メモリサイズである．マーカのない線グラフは Impala のメモリの利用可能上限を表し，これを超えている場合に，当該サブクエリを Hive で実行している．

また，同じクエリで実際の中間データサイズを計測し，サブクエリごとに最大利用メモリサイズを計算した．この実際の最大利用メモリサイズに対する推定による最大利用メモリサイズの比を誤差率として図 **12** に示す．誤差率の 1 は誤差がなく，1 より大きい場合は最大利用メモリサイズを大きく，1 より小さい場合は小さく推定していることを表す．なお，TPC-H のデータセットは Scale の変化によってデータの中身の特徴が変化しないため，誤差率も同様に

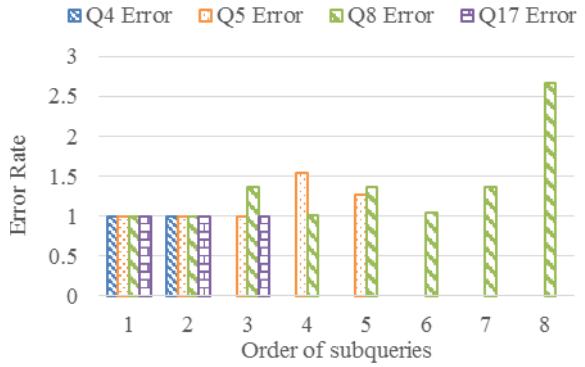


図 12　Scale 150 における最大利用メモリサイズの誤差率
Fig. 12　Error rates of predicted peak used memory size.

Scale の変化に影響せず，どの Scale でも同じ値であったことから，Scale 150 のみを図示する．

この結果から，各クエリの前半のサブクエリではほぼ誤差は発生していないが，後半で誤差が発生していることが分かる．しかし，誤差が発生しているサブクエリの実際の最大利用メモリサイズは，利用可能上限に対して十分に小さく，本評価では QE の選択に誤りが発生するほどの誤差は発生していないことが分かる．

5.2　システム評価（SQL-on-Hadoop/Solr）

本節では，オペレータの違いによるクエリ分割に関して，SQL-on-Hadoop（Hive または Impala）と全文検索エンジン Solr を用いてプロトタイプの性能を評価し，特定の処理に特化した QE においても提案方式が有効であることを評価する．

5.2.1　評価環境

システムは 5.1 節と同様に 4 章で述べたプロトタイプを利用する．なおここでは SQL-on-Hadoop における Hive および Impala のクエリ分割は行わない．評価データは，Twitter Search API で取得したひらがなを含むツイートの 1 日分（約 4.5 GB），5 日分（約 22.5 GB），10 日分（約 45.3 GB）の 3 種類のデータを用いた．レコード数は 1 日分で約 700 万ツイートである．このツイートデータは HDFS に CSV データで保存し，Hive および Impala にはそのままデータ登録し，Solr にはツイートのテキストに対して bi-gram を用いたインデックスを作成した．評価に利用したクエリは，「電波」と「悪い」の両方が含まれるツイートをしたユーザを抽出し，その発言回数が最も多いユーザを上位 10 件出力するクエリである．プロトタイプに投稿する評価用の SQL クエリを**図 13** に示す．

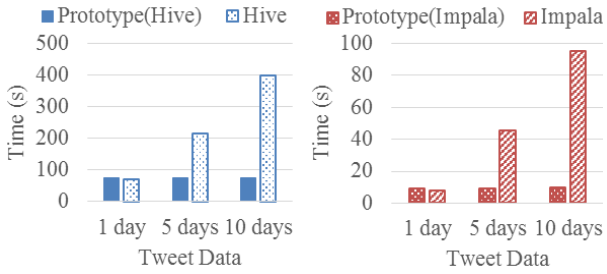
5.2.2　性能評価

評価用クエリを SQL-on-Hadoop 単体で実行した場合と，プロトタイプで Solr と SQL-on-Hadoop に分割実行した場合の実行時間をツイートデータのサイズ別で**図 14** に示す．図 14 (a) は振り分け先の SQL-on-Hadoop として Hive を対象にした場合の実行結果であり，図 14 (b) は Impala を対象にした場合の実行結果である．**図 15** はプロトタイプが生成した分割クエリ（管理クエリ除く）であり，Solr へ分割実行した際の Solr クエリおよび HiveQL クエリを示している．1 日分のツイートデータでは，Hive および Impala 単体と比較して，プロトタイプによる分割実行によって同程度もしくはプロトタイプの方が少し性能劣化している．一方で，5 日分および 10 日分のツイートデータでは，プロトタイプの実行時間が Hive および Impala 両方の場合で，データサイズに比例して単体実行より大幅に高速化する結果となった．Hive の場合は，5 日分で 3.0 倍，10 日分で 5.3 倍高速化し，Impala の場合は，5 日分で 4.8 倍，10 日分で 9.4 倍高速化した．

以上の性能評価から，特定のデータやオペレータにおいて実行可否や処理性能が異なる QE の組合せにおいて，提案システムが有効であることを示した．

```
SELECT from_user, COUNT(1) AS count FROM tweets
  WHERE text LIKE '%電波%' AND text LIKE '%悪い%'
  GROUP BY from_user
  ORDER BY count DESC
  LIMIT 10;
```

図 13　ツイートデータ分析における評価用クエリ
Fig. 13　Evaluation query for tweet data analysis.



(a)　Solr + Hive　　(b)　Solr + Impala

図 14　Solr と SQL-on-Hadoop における性能計測結果
Fig. 14　Results of evaluations for performance on Solr and SQL-on-Hadoop.

```
<Solr>
http://solr-server:8983/solr/tweets_s_shard1_replica1/select?
q=text%3A%22%E9%9B%BB%E6%B3%A2%22+AND+text
%3A%22%E6%82%AA%E3%81%84%22&fl=from_user&wt=
csv&csv.header=false

<SQL-on-Hadoop>
SELECT from_user, COUNT(1) AS count FROM tmp1
  GROUP BY from_user ORDER BY count DESC LIMIT 10;
```

図 15　Solr および SQL-on-Hadoop への分割後のクエリ
Fig. 15　Split query for Solr and SQL-on-Hadoop.

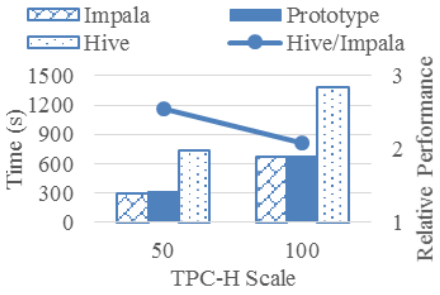


図 16　Impala，プロトタイプ，Hive それぞれの Q4 における実行時間と，Hive および Impala の相対性能比

Fig. 16　Execution time of Q4 by Impala, our prototype and Hive, and relative performance of Impala against Hive.

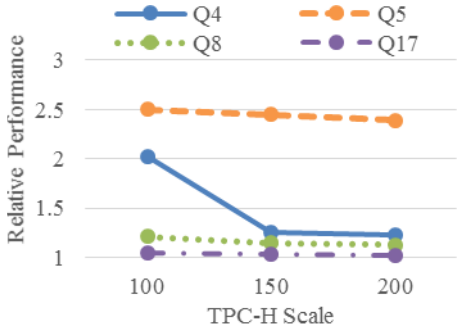


図 17　Hive に対するプロトタイプの相対性能比

Fig. 17　Relative performance of our prototype against Hive.

5.3　考察

5.3.1　性能の考察

性能評価した TPC-H クエリのうち一部の Scale では，すべてのサブクエリを Impala で実行しており，Impala 本来の性能とほぼ等しい結果となっている．これを示すために，Q4 を Impala 単体で実行した．**図 16** はこの結果であり，棒グラフが Impala，プロトタイプおよび Hive における各実行時間を示す．この結果からプロトタイプが Impala 単体の実行時間とほぼ等しいこと分かる．図 10 で示したとおり，プロトタイプの分割クエリ生成等のオーバヘッドの影響で遅延が発生しているが，Impala の実行時間との差分からもほぼ無視できることが分かる．

処理の複雑さおよびデータサイズの変化によって性能がどのように変化するかを分析するために，Scale 100 以上で Hive および Impala 両方に分割実行している Q4，Q5，Q8，Q17 を用いて，Hive に対する速度比を考察する．プロトタイプが Hive に対して何倍高速化したかを示すグラフが**図 17** である．なおこれらのクエリで Impala が実行した Scale50 を除いている．この結果から，Impala 単体では実行できないクエリでも，Hive と Impala を組み合わせることで，Hive 単体で実行するより最大 2.5 倍高速化したことが分かる．

また図 8 から，Q4 では 150 から 200 にかけて，Q5，Q8 および Q17 では 100 から 200 にかけて，各サブクエリを実行する QE の範囲が変化していないことが分かる．しかし，図 17 ではこれらクエリの相対性能比が Scale の増加とともに低下傾向にある．本提案システムの特徴は，部分クエリを高速な QE で実行することにあり，性能向上も同様にこの高速な QE と相対的に低速な QE の速度比に依存する．よって，分割クエリに変化がないにもかかわらず，Hive に対するプロトタイプの相対性能比が低下した原因は，図 16 の線グラフが示すように，Scale の増加によって Hive に対する Impala の相対性能比が低下したためである．

図 17 において，Q5，Q4，Q8，Q17 の順で相対性能比が大きい．このようなクエリの違いによるプロトタイプの性能向上の違いが発生する原因を分析する．図 11 から，Q5 において Impala で処理されている 4 番目および 5 番目のサブクエリの最大利用メモリサイズが，他の Impala で処理されたサイズより比較的大きいことが分かる．これは，実際に計測した最大利用メモリサイズでも同様である．実際に計測した全オペレータの中間データサイズの合計に対する Impala で実行した処理の中間データサイズの合計の比は，Q5 が 11.0%あるのに対し，Q4 は 6.4%，Q8 は 6.1%と 2 倍近く差があった．また，Q4 と Q8 に関しても，差は小さいものの，この中間データサイズの比が影響しており，Q4 の性能向上が Q8 よりも少し大きい．Q17 に関しては Impala の実行分のメモリサイズが全体の 0.3%と非常に小さく，結果として性能向上が小さい．この結果から，単体の QE を利用した場合に対する提案システムのクエリ分割実行による性能差を決める要素は，処理対象のデータサイズも重要であることが分かる．

図 10 におけるオーバヘッドは，初期化処理および終了処理ともに接続している QE の数に比例していると考えられる．これは，Q5 および Q8 において Scale が 50 から 100 に変化するときにオーバヘッドが上昇している点から推定できる．これは Scale 100 になってクエリが分割されたことで，Scale 50 では Impala のみであった実行 QE が，Hive と Impala の両方で実行されたためである．同様に Q4-1 において Scale 100 から 150 に変化した場合においても同じ動きが見られる．しかしこれは Impala で実行していたクエリが Hive に変更されただけであり，この増加は QE の違いによるコネクション確立・切断にかかる時間の変化が影響していると考えられる．また，同じ数のコネクション処理を同じ QE に対して行っている Scale 50 や，Q5 および Q8 の Scale 100～200 において，各クエリのオーバヘッドに差がある．この差は各クエリの複雑さと比例しており，クエリの構文解析処理や，取得する統計情報の量が増えることによる影響と考えられる．

SQL-on-Hadoop と Solr におけるプロトタイプの分割クエリでは，1 日分のツイートデータを除いて，Hive および Impala 両方の場合で高速化を実現した．この要因を分析するために，管理クエリを含んだ分割クエリにおけるクエリ別の実行時間を**図 18** に示す．図 18 (a) は Solr と Hive

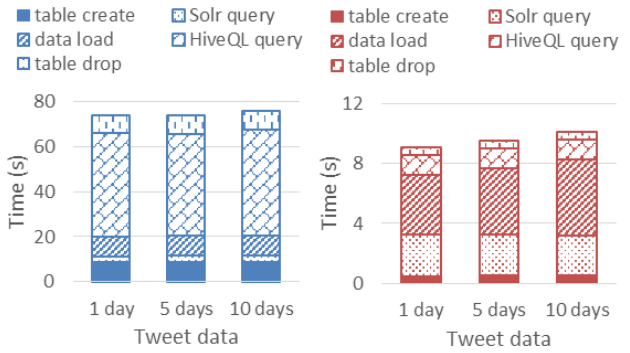


図 18　プロトタイプにおける分割クエリの実行時間内訳
Fig. 18　Breakdowns of execution times on prototype system.

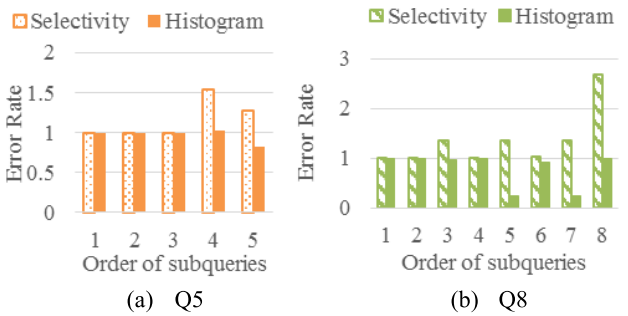


図 19　選択率方式とヒストグラム方式の誤差率比較
Fig. 19　Comparisons of error rates of selectivity method and histogram method.

に対する分割クエリの実行時間の内訳であり，図 18 (b) は Solr と Impala に対する分割クエリの実行時間の内訳である．実行時間の各要素は，下から順に，一時テーブル作成，Solr のクエリ実行，一時テーブルへのデータロード，Hive のクエリ実行，一時テーブル削除である．これらの図から，Solr の実行時間は検索対象のデータサイズによらず，ほぼ一定の時間で検索可能であると分かる．よって，サイズ規模が大きくなるほど，全文検索用のインデックスを作れない Hive および Impala よりも高速に検索が可能となるといえる．一方で 1 日分のような小さいサイズでは，Solr による高速化よりも，QE の切替えによる処理時間の割合が大きくなり，プロトタイプが SQL-on-Hadoop 単体よりも遅くなる．ここでは評価のために性能低下の振り分け回避を行わなかったが，3.4.3 項で述べたパフォーマンスモデルによってこのような分割クエリ生成の回避が可能である．

5.3.2　中間データサイズ推定の考察

プロトタイプのデータサイズ推定は処理前後のサイズ比率である選択率を利用しているため，データの特徴（分散等）が変化しない限り，予測の精度は変わらない．しかし，同じテーブルにおけるデータの更新や，複数の処理が連続することによってデータの特徴が変化すると，誤差は大きくなる．そのため，図 12 において処理の後半になるにつれ，誤差が大きくなる傾向にある．サブクエリの数が少ない Q4 および Q17 では誤差はほぼ発生していない．一方でサブクエリが多い Q5 および Q8 はクエリの後半で最大 2.5 倍の誤差が発生している．このことから，テーブルデータの更新が頻繁に起こる場合や，クエリの後半になるにつれてデータサイズが増加するようなクエリでは，データの特徴を考慮したデータサイズ推定が必要になると考えられる．

そこで，このようなデータの特徴の変化に対応可能なサイズ推定方式として，ヒストグラムを利用した中間データサイズ推定方式 [27] を実装し，同様に予測データサイズを評価した．本方式のヒストグラムは，属性ごとの統計情報である．属性の値の数を，値の種類（ドメイン）別に集計する．このドメインを一定の範囲（バケット）の集計に簡易化して統計情報として利用する．これにより，属性ごとのドメインの数の偏り（分散）を把握することができる．サイズ推定では，全テーブルの全属性で事前にヒストグラムを作成し，クエリ実行時にオペレータごとの中間ヒストグラムを作成することで，利用メモリサイズを推定する．本実装では，1 つの属性で最大 100 個のバケットを作成した．各バケットはドメインの数が等しくなるように作成した．

図 19 (a) は Q5 における選択率方式とヒストグラム方式の実際に対する誤差率を比較したグラフであり，図 19 (b) は同様に Q8 における誤差率の比較である．Q5 に関しては，選択演算や結合演算におけるデータの特徴変化による誤差を排除することができ，全体的に誤差が少ない結果となった．Q8 に関しては，3 番目および 8 番目のサブクエリは改善された．しかし，5 番目および 7 番目が大きく誤差が発生しており，メモリサイズを小さく予測している．これはヒストグラム方式の結合演算の予測の問題であり，バケットの中で集計されているドメインが連続値でない場合に，結合処理で少なく見積もる傾向にあるためである．より正確な利用メモリサイズ推定を実現するためには，さらなる改善が必要であるといえる．

また，中間データサイズ推定の誤差は避けられないため，誤差によるクエリの再実行が発生する可能性がある．これをできる限り回避する手法として，実行時に動的に QE を判断する方式 [28] が考えられる．これにより実行時にサイズ推定の誤差を修正しながら，実行する QE を動的に切り替えることが可能となる．さらに，マルチユーザによる利用によってリソースの利用可能上限が動的に変更する場合にも有効である．しかし分割クエリをオペレータごとに生成する必要があり，クエリの分割損が発生するため，そのトレードオフを考慮する必要がある．また分割クエリ実行中のリアルタイムな統計情報取得方法も課題である．

6.　おわりに

本論文では，同一データを持つ複数の QE から，適切な QE を自動選択するマルチデータベースシステムを提案した．SQL-on-Hadoop と全文検索エンジンを対象とする

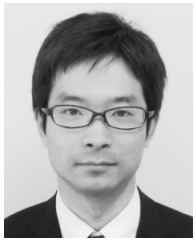
提案システムのプロトタイプを実装し，Hive と Impala を対象として TPC-H ベンチマークを用いて評価した結果，Impala 単体で実行が失敗するクエリに対しても，適切な QE を自動選択することで実行可能とし，さらに分割実行により Hive 単体と比較して最大 2.5 倍高速化できることを確認した．また SQL-on-Hadoop と Solr を対象に Twitter のデータを利用して評価した結果，テキスト検索において Solr を活用することで，最大 9.4 倍高速化し，またデータサイズが大きくなることでより高速化することを確認した．本論文における貢献は以下のとおりである．

- 同一データを持つ複数の QE に対して単一の仮想スキーマを提供することで，複数の仮想スキーマが作成される課題を解決した．
- 上記仮想スキーマを用いたクエリ分割実行方式により，同一データを持つ複数の QE から適切な QE を自動選択することが可能となり，ユーザの負荷軽減を実現した．
- クエリ分割実行方式によって，クエリのオペレータ単位に適切な QE を選択した分割クエリの生成が可能となり，複数の QE を活用することによってクエリ高速化を実現した．

提案システムは様々な QE を活用することで，より高い効果を発揮することが期待できる．そこで，プロトタイプにおける対象の QE を増やし，より様々な QE の特徴に対応した分割基準のモデルを作成したい．

参考文献

[1] Özusu, M.T. and Valduriez, P.: *Principles of Distributed Database Systems, 3rd Edition*, Springer (2011).
[2] Apache Hadoop, available from ⟨http://hadoop.apache.org/⟩.
[3] Apache Hive, available from ⟨http://hive.apache.org/⟩.
[4] Impala, available from ⟨http://impala.io/⟩.
[5] Apache Drill, available from ⟨http://drill.apache.org/⟩.
[6] Presto, available from ⟨http://prestodb.io/⟩.
[7] Subrahmanjan, V.S. et al.: HERMES: A Heterogeneous Reasoning and Mediator System (1995), available from ⟨http://www.cs.umd.edu/projects/hermes/overview/paper⟩.
[8] Ahmed, R. et al.: The Pegasus Heterogeneous Multidatabase System, *IEEE Computer*, Vol.24, No.12, pp.19–27 (1991).
[9] Chawathe, S. et al.: The TSIMMIS Project: Integration of Heterogeneous Information Sources, *Journals of Intelligent Information System*, Vol.8, No.2, pp.117–132 (1994).
[10] Carey, M.J. et al.: Towards Heterogeneous Multimedia Information Systems: The Garlic Approach, *Proc. RIDE-DOM*, pp.124–131 (1995).
[11] Litwin, W. et al.: Interoperability of Multiple Autonomous Databases, *ACM Computing Surveys*, Vol.22, No.3, pp.267–293 (1990).
[12] Sheth, A.P. and Larson, J.A.: Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases, *ACM Computing Surveys*, Vol.22, No.3, pp.183–236 (1990).
[13] Fiore, S. et al.: Data Virtualization in Grid Environments through the GReIC Data Access and Integration Service, *Proc. ICITST*, pp.1–6 (2009).
[14] Salloum, M. et al.: Online Ordering of Overlapping Data Sources, *PVLDB*, Vol.7, No.3, pp.133–144 (2013).
[15] Karpathiotakis, M. et al.: Just-In-Time Data Virtualization: Lightweight Data Management with ViDa, *Proc. CIDR*, pp.1–11 (2015).
[16] Teiid: JBoss Project, available from ⟨http://teiid.jboss.org/⟩.
[17] Lenzerini, M.: Data Integration: A Theoretical Perspective, *Proc. PODS*, pp.233–246 (2002).
[18] Abouzeid, A. and Bajda-pawlikowski, K.: HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads, *PVLDB*, Vol.2, No.1, pp.922–933 (2009).
[19] DeWitt, D.J. et al.: Split Query Processing in Polybase, *Proc. SIGMOD*, pp.1255–1266 (2013).
[20] Lim, H. et al.: How to Fit when No One Size Fits, *CIDR* (2013).
[21] Simitsis, A. et al.: HFMS: Managing the lifecycle and complexity of hybrid analytic data flows, *Proc. ICDE*, pp.1174–1185 (2013).
[22] Herodotou, H. et al.: Starfish: A Self-tuning System for Big Data Analytics, *CIDR*, pp.261–272 (2011).
[23] Borkar, V. et al.: Hyracks: A Flexible and Extensible Foundation for Data-Intensive Computing, *ICDE*, pp.1151–1162 (2011).
[24] Apache Solr, available from ⟨http://lucene.apache.org/solr/⟩.
[25] TPC-H, available from ⟨http://www.tpc.org/tpch/⟩.
[26] Impala 対応 TPC-H, 入手先 ⟨https://github.com/kj-ki/tpc-h-impala⟩.
[27] Bruno, N. and Chaudhuri, S.: Exploiting Statistics on Query Expressions for Optimization, *Proc. SIGMOD*, pp.263–274 (2002).
[28] Kabra, N. and DeWitt, D.: Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans, *Proc. SIGMOD*, pp.106–117 (1998).



齋藤 和広　（正会員）

株式会社 KDDI 研究所．2010 年成蹊大学大学院工学研究科博士課程前期修了．同年 KDDI 株式会社入社．2012 年より株式会社 KDDI 研究所に勤務．分散システム，データベースシステムの研究開発に従事．日本データベース学会会員．

渡辺 泰之　（正会員）

株式会社 KDDI 研究所．1995 年東京理科大学大学院理工学研究科情報科学専攻修士課程修了．同年国際電信電話株式会社（現，KDDI 株式会社）入社．現在，株式会社 KDDI 研究所クラウドプラットフォームグループ開発マネージャー．分散システム，データベースシステム等の研究開発に従事．日本データベース学会会員．

村松 茂樹　（正会員）

株式会社 KDDI 研究所．1999 年東京大学大学院工学系研究科電子情報工学専攻修士課程修了．同年 KDD 株式会社（現，KDDI 株式会社）入社．現在，株式会社 KDDI 研究所データマイニング応用グループ研究主査．位置推定，行動認識，データベースシステム等の研究開発に従事．本会 2011 年度論文賞受賞．

小林 亜令　（正会員）

株式会社 KDDI 研究所．1998 年北海道大学大学院工学研究科修士課程修了．同年 KDD 株式会社（現，KDDI 株式会社）入社．現在，株式会社 KDDI 研究所クラウドプラットフォームグループリーダー．これまで XML，SVG，ITS，通信放送融合技術，センサデータマイニング等の研究開発に従事．2003 年 FIT2003 船井記念ベストペーパー賞受賞，2008 年 ARIB 電波功績賞受賞．本会 UBI 研究会運営委員．

（担当編集委員　鷹野 孝典）