データ仮想化におけるクエリ分割実行の実装と評価

齋藤 和広† 米田 信之† 村松 茂樹† 渡辺 泰之† 小林 亜令†

†株式会社 KDDI 研究所 〒102-8460 東京都千代田区飯田橋 3-10-10 ガーデンエアタワー

E-mail: †{ku-saitou, no-maita, mura, yasuyuki, kobayasi}@kddilabs.jp

あらまし 複数のデータソース(DS)にあるデータを複合的に分析するために，DS の統合が求められている．これを実現する手段として，複数の DS を論理的に統合するデータ仮想化がある．データ仮想化は，スキーマ情報のみを統合し，クエリ実行時に必要なデータを DS から取得する．そのため，DS から取得するデータ量が物理メモリサイズを超えるほど大規模であると，クエリ処理性能が劇的に劣化する．更に，最悪の場合，クエリ実行が失敗する課題がある．本研究では，データを分割取得してクエリ処理することで，上記課題を解消する手法を提案する．また，提案手法をオープンソースソフトウェアの Presto に実装し，TPC-H ベンチマークを用いて評価する．

キーワード データ仮想化，データベース，問合せ処理，Presto

1. はじめに

企業が持つデータの多種多様化により，データの用途や目的に応じて異なるデータソース（DS）が複数構築されている．またビッグデータ活用の普及によりデータ分析が複雑化し，様々なデータを複合的に利用する事例が増えている．企業の競争力向上にはデータ分析の加速が必須であり,データ分析を効率化するため，複数存在する DS の統合が求められている．しかし DS の物理的な統合は，移設に伴う既存のアプリケーションへの影響やコストの観点から容易に実現できない．

そこで筆者らは，複数 DS の論理的統合を実現するデータ仮想化技術[1]に着目している．データ仮想化はスキーマ情報のみを統合し，実際のデータを統合しない．データ仮想化を実現するデータ仮想化システム（DVS）が，クエリ実行時に必要なデータを DS から取得する．これにより複数の DS を参照するクエリの実行を実現している．しかし，DS から取得するデータや，DVS におけるクエリ処理の中間結果サイズが物理メモリサイズを超えるほど大規模になると，クエリ処理性能が劇的に劣化し，最悪の場合にはクエリ実行が失敗する．

本課題を解消すべく，DS からデータを分割取得し，DVS においてクエリ処理を分割実行する手法を提案する．提案手法は，DVS で確実に実行可能な利用メモリサイズとなる分割クエリの生成と演算の分割実行で構成される．本手法をオープンソースソフトウェアの分散 SQL クエリエンジンである Presto[2]に実装し，TPC-H を用いて有効性を評価する．

本論文の構成は以下の通りである．2 節でデータ仮想化技術の概要と課題を述べ，3 節で提案手法の詳細を，4 節で Presto への実装を述べる． 5 節で提案手法を評価し，最後に 6 節で本論文の結論と今後の課題を述べる．

2. データ仮想化

2.1. 概要

データ仮想化は，DVS に接続した DS 上のデータをテーブル形式のスキーマ情報にマッピングすることで複数の DS を論理的に一つのデータベースシステム（DBS）にできる．これらのスキーマ情報は仮想スキーマと呼ばれ, DVS 上に登録される．仮想スキーマは，対象データの部分データを抽出する条件や,複数の DS を跨ぐデータを結合するビュー等を表現することができる．また，DBS のようにスキーマ定義されているデータだけでなく，CSV や JSON 等のファイルでも仮想スキーマとして定義可能である．

データ仮想化の対象となる DS には, DBS に限らず，ファイルシステムや Web サーバなどのネットワーク経由でデータ取得可能なシステムも含まれる．そのための仕組みとして，DVS は DS への接続プロトコルやデータファイルの種類に応じたコネクタを持ち，クエリ実行時は適切なコネクタで接続先からデータを取得する．

図 1 にデータ仮想化の概要図を示す．ユーザは，仮想スキーマを処理対象のテーブルとする SQL クエリを DVS に投稿することができる．DVS はユーザが投稿した SQL クエリから仮想スキーマの情報に従って独自の演算木を生成し，処理対象のデータがある DS を特定する．次に DVS は DS のインタフェースに合わせたクエリを生成し，演算木に従ってその DS に投稿する．投稿したクエリの結果が DS から返ると，DVS はその結果を仮想スキーマのデータ形式に変換する．更に DVS は，DS で処理しなかった残りの演算を処理し，ユーザにその結果を返す．

2.2. データ仮想化の課題

データ仮想化は，ユーザが投稿したクエリを実行するために，利用するデータを DS 又は DVS の物理メモリに展開する．そのため，DS 又は DVS の物理メモリ

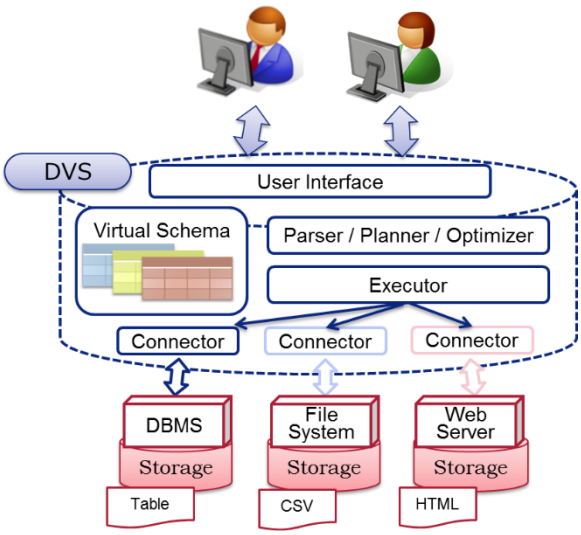


図 1　データ仮想化の概要図

サイズを超えるほど大規模なデータを扱う場合に，クエリ処理性能が劇的に劣化し，最悪の場合にはクエリの実行が失敗する課題がある．この課題は，DS 及び DVS 間のデータ通信時と，DVS におけるクエリ処理時の二箇所で発生する．以下で課題の詳細を述べる．

2.2.1. データ通信時

DVS はネットワーク経由で DS にクエリを投稿してデータを取得する．ここで DVS の処理待ちが発生すると，取得するデータを DS 又は DVS の物理メモリで保持しておく必要がある．そのデータサイズが物理メモリを超えるとストレージアクセスが頻発し，場合によってはメモリ不足で強制終了する．例えば DVS が JDBC で DS からデータを取得する場合，DVS が DS に投稿したクエリの結果を全て取得して，DVS の演算処理が始まるまで DVS の物理メモリに保持する．また JDBC のカーソルを利用した場合，DVS の演算処理が開始されるまで DS の物理メモリでデータを保持する．このように，データの取得方法に応じて，DVS と DS のどちらかでメモリ不足が発生する可能性がある．

これを回避するため，DS で出来る限りデータサイズを小さくする手法が考えられる．その手段の一つがクエリプッシュダウンであり，DS へ関係演算処理（射影や選択，結合など）を委譲することで，データサイズを削減する[3]．また別の手段として，異なる DS のデータに対する結合演算を含むクエリにおいて，片方の DS のデータをもう一方の DS に再配置することでデータを削減する手法がある[4]．

しかしこれらのデータ削減の手法を用いたとしても，データ通信のサイズを物理メモリサイズ以下にできない場合がある．例えばクエリプッシュダウンした結果が大規模である場合や，結合するデータがどちらも大規模である場合である．またデータ仮想化は，既に運用されている DS を対象とできるが，その DS にとっては構築当初に想定していない用途のため，従来の用途や運用上の制約によって DS のリソースを利用できず，前述のデータ削減の手法が使えない場合がある．

従って，DS 又は DVS の物理メモリのサイズを超える大規模なデータでも，確実に DS からデータ取得できる手法が必要となる．

2.2.2. DVS におけるクエリ処理時

DVS は DS から取得したデータに対してクエリ処理を実行する．DVS におけるクエリ処理方式は，従来の DBS のクエリ処理方式が適用される．これには，演算毎にデータを全て実体化して処理する Materialized 方式[5]，演算木のルートから順に下位の演算に対して 1 レコードずつ要求して処理する Volcano 方式[6]，ブロック単位で取得して並列処理する Morsel 方式[7]が存在する．

上記の方式はいずれも，一度読み込んだデータをメモリに保持したまま全ての演算を処理する．そのため，入力データが大規模の場合や，また多対多の結合演算や直積演算，キャストによる型の変換，列数の増加等によって，演算結果の出力データのサイズが入力データよりも大きくなる場合に，物理メモリを超えることがある．その結果，メモリ不足が発生し，演算処理でストレージアクセスの頻発による性能の劣化や強制終了する場合がある．

従って，物理メモリのサイズを超える大規模なデータを対象とする演算においても，DVS が確実に演算処理できる手法が必要となる．

2.3. 既存の DVS

データ仮想化に関する研究として，関係データベースと XML データベースを対象とする GRelC[8]と，DS の統計情報を活用して DS の処理順序を最適化する OASIS[9]がある．両者ともに，グリッド環境上に散らばる複数の DS を対象に，仮想的な一つのデータストレージを提供しており，ユーザはデータの場所や各 DS のインタフェースを意識することなくアクセスすることが可能となる．一方，DVS 上ではクエリ処理を行わず，クライアントがデータ取得後に独自で処理を実行することが前提である．また大規模データの取得に関する考慮もしていない．

DVS の一種である MIND[10]や，オープンソースソフトウェアとして DVS の実装を行っている Teiid[11]は，複数の DS を仮想的に一つのストレージとしてユーザに提供するだけでなく，通常の DBS と同様に SQL クエリ処理が可能である．複数の DS を跨ぐテーブル同士の結合処理も可能であり，複数の DS の結果を DVS に集約し，DVS 上のクエリエンジンで結合処理を

実現している．しかし大規模データの処理を前提としていないため，データ通信時及び DVS のクエリ処理時にメモリ不足が発生する可能性がある．

オープンソースソフトウェアの Presto[2]及び Apache Drill[12]は，分散システムの Apache Hadoop で管理されるデータを SQL インタフェースで操作可能な分散クエリエンジンである．これらは，Apache Hadoop 以外の DS も対象に処理でき，データ仮想化を実現できる．更に，DVS におけるクエリ処理を分散処理することができ，データ規模に応じて物理サーバを増やすことで大規模データに対応できる．しかし，構築した物理サーバの物理メモリを超える規模のデータに対してはクエリ処理することができない．

3. 提案手法

前節で述べたデータ仮想化の課題を解消するために，DS からデータを分割取得するクエリ分割手法と，ストレージを利用した DVS の分割実行手法を提案する．

3.1 ではクエリ分割手法について述べ，3.2 で DVS の分割実行手法を述べる．3.3 ではこれらの手法で共通して必要な分割条件について述べる．3.4 では提案手法を含む DVS のシステム構成について述べる．

3.1. クエリ分割手法

本手法では，DS からデータを取得するクエリを複数に分割した分割クエリを作成し，一つのクエリで取得するデータサイズを物理メモリサイズ以下にする．これにより，データ通信時に発生する課題を解消する．

分割クエリは，本来一つであったクエリに対して，3.3 で述べる分割条件を利用して，取得するデータの範囲を指定して複数に分割したクエリである．この分割クエリを全て実行して，一つのデータを取得する．例えば DS のインタフェースが SQL の場合，SQL クエリの WHERE 句で，特定の列の実値に対する取得範囲の条件式を追加する．そしてその特定の列における全ての範囲を網羅するように条件式を変更し，SQL クエリを複数個生成する．

生成した分割クエリは，DVS のクエリ実行時に順次 DS に投稿される．一つの分割クエリによって部分データを取得後，そのデータが DVS の次の演算処理で利用されてメモリが解放された後に，次の分割クエリを投稿する．なお，DVS の演算処理も物理メモリを利用するため，分割クエリの分割条件は 3.2 で述べる分割実行を考慮して設定される．

3.2. DVS の分割実行手法

本手法は，DVS における演算処理で物理メモリサイズを超える場合を考慮し，DVS 上の一時ストレージを利用して演算処理を分割実行する．これにより DVS のクエリ処理における課題を解消する．

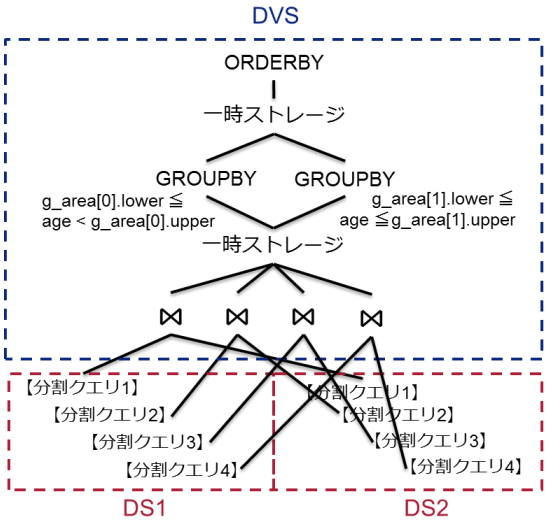


図 2　分割実行の演算木の例

分割実行は DVS が作成した演算木に従って，演算単位で行われる．ある演算で分割実行が必要と判断された場合，Materialized 方式と同様にその演算のみが処理されるよう制御する．その演算には，クエリ分割手法と同様に分割条件が作成され，分割条件に従ったデータのみを入力データとして処理する．その入力データは DS か一時ストレージに格納されている必要がある．そのため，分割実行される演算の直前の演算は，分割の要否に関わらず，一時ストレージに出力する必要がある．分割実行した演算の出力データは一時ストレージに退避する．

分割実行の演算木の例を図 2 に示す．この演算木では，各 DS への分割クエリの投稿と結合演算処理を順次繰り返し，結果を一時ストレージに格納する．その結果を集約演算（GROUP BY）の分割条件に従って 2 分割で一時ストレージ内の指定の分割範囲のデータに対して処理し，一時ストレージに格納する．最後に整列演算（ORDER BY）を一時ストレージにあるデータに対して実行し，クエリ処理完了となる．

3.3. 分割条件

クエリ分割手法及び DVS の分割実行手法では，分割するための条件を生成する．その分割条件は以下の三つの要素で構成される．

- 分割数
- 分割基準属性（分割の基準となる属性）
- 分割範囲（分割して処理するデータの範囲）

分割数は，分割対象のクエリや演算で利用するメモリサイズを，物理メモリサイズで割って小数点以下を切り上げることで算出する．分割基準属性は処理対象の数と等しく選択され，単項演算と分割クエリは一つ，二項演算は二つのデータのそれぞれから一つずつ選択される．分割範囲は分割されるデータの範囲であり，

分割後のデータサイズが物理メモリサイズ以下となるように，分割基準属性が持つ実値から決定される．これは分割数分作成される．

分割範囲の計算には，演算毎の中間結果サイズの予測と，更にデータを分割するために列の実値が必要となる．分割範囲を計算する最も単純な方法は，属性毎の最大値と最小値を統計情報として作成し，分割数で最大値から最小値を均等に分割する方法である．しかし実値の分布が偏ることで分割範囲内のサイズに偏りが生じ，利用可能な物理メモリサイズを超える場合がある．そこで提案手法では，統計情報として属性毎の実値の分布を表すヒストグラムを利用することで，偏りを考慮した分割範囲の計算を行う[13]. また, 分割の要否の判定に利用する演算毎の出力データサイズ（中間結果）も同様にヒストグラムを利用して計算する[14].

以下で単項演算と二項演算，分割クエリの分割条件の生成方法を述べる．

3.3.1. 単項演算の分割条件

単項演算における分割条件は，分割実行された結果同士で追加処理が発生しないように生成することで，余計な I/O や DS へのクエリ投稿を抑制する．そのために，分割基準属性を演算のキー属性にし，その属性に対する分割範囲で分割実行する．例えば集約演算の場合，GROUP BY 句で指定される属性を分割基準属性に選択することで，一回の分割実行内で演算処理を完結することができる．なお，キー属性がない演算の場合や，もしくは複数のキー属性がある場合には，入力データを取得する際に検索が高速な属性(索引付き等)を優先する選択ポリシーとなる．

例外として，GROUP BY 句がない場合の集計処理（COUNT(*)等）において，分割実行後に，それぞれの結果に対して更に集計処理を実行する必要がある．また，整列演算に関しても最終的に分割実行した結果のそれぞれで順序の整合性をとる必要があるが，提案手法では分割範囲の指定をソートの順序と合せて分割実行した順序通りに結果を並べることで，整列演算の処理を完了することができる．

3.3.2. 二項演算の分割条件

二項演算における分割実行は，単項演算の場合と同様に，分割実行単位で演算が完結する分割条件を生成する．結合演算に関しては，結合条件で利用される二つの属性をそれぞれ分割基準属性に選択し，これらの分割範囲を各分割実行で共通の値にする. これにより，Block Nested Loop Join のように同じデータを複数回呼び出す必要がなくなり，ブロック単位の Merge Join の要領で実行することができる．ただし，直積演算に関しては複数回の呼び出しが必須であり，二つのデータの分割基準属性は，それぞれが単項演算におけるキー属性がない場合の選択ポリシーに従う．集合演算（UNION）に関しても結合条件が存在しないため，キー属性がない場合の選択ポリシーに従うが，分割実行単位で演算を完結させるために，二つのデータで同じ分割条件を生成する．

3.3.3. 分割クエリの分割条件

DS から分割してデータを取得する分割クエリの分割条件は，演算木における直上の演算の分割条件を利用する．しかし，DS と DVS で利用できる物理メモリサイズが異なる場合がある．もし DS の方が小さく，直上の演算の分割条件ではメモリに保持できない場合, DS のメモリ溢れによる遅延や強制終了が発生する. そこで以下の二つの方法でこれを回避する．一つは，取得するデータを DVS に蓄積する全取得手法である. もう一つは，分割条件を再計算し，直上の演算の分割範囲を更に分割する Scan 分割手法である．また，演算木の直上に分割実行する演算がない場合も同様の方法で回避する. 分割条件を再計算する場合は, DS に投稿するクエリの出力データサイズを用いて，分割の要否の判断と分割条件の計算を行う．

3.4. 提案手法のシステム構成

これまでに述べた提案方手法を含む DVS のシステム構成図を図 3 に示す. DVS はまず, クライアントから投稿されたクエリを評価し, 従来の DVS と同様に仮想スキーマ情報を基に演算木を生成する．この演算木から演算毎の利用メモリサイズを算出し，演算毎に一定サイズを超える演算の分割条件を生成する. 次に DS からデータを取得するためのクエリを生成する．データが大規模な場合，分割条件に従ってクエリを分割して生成する．そして演算木に従って，対象の DS に順次クエリを投稿し, その結果に対して DVS 上での演算を処理する. DVS 上での演算処理に分割実行が必要な場合，その分割された演算処理の結果を一時ストレージに退避して次の分割処理をする．演算木上の全演算が完了した段階で，結果をクライアントに転送する．

4. 実装

4.1. Presto

提案手法を, オープンソースソフトウェアで DVS の一種である Presto に実装した．バージョンは 0.100 を利用した．なお，Presto は DS が SQL クエリを処理可能であったとしてもデータを取得するだけであり，数値型の範囲選択演算と射影演算以外をクエリプッシュダウンしない. そこで, 提案手法の実装に先立ち, JDBC コネクタを対象に, 数値型以外の選択演算, 結合演算, 集約演算，整列演算，集合演算のクエリプッシュダウンを実装した．

Presto は, クライアントから SQL クエリを受け付け,

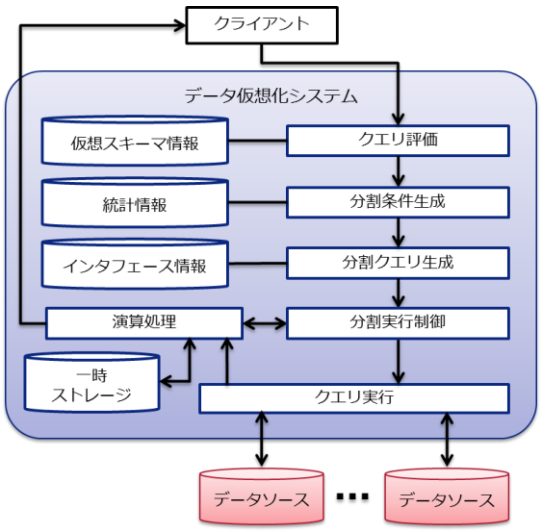


図 3　提案手法のシステム構成図

実行計画を作成する Coordinator と，DS からデータを取得して演算処理を実行する Worker から構成される．

Coordinator は実行計画として SQL クエリから Left-deep な演算木を作成する．また演算木には，DS からデータを取得する Scan 演算があり，DS の接続情報とクエリプッシュダウンする演算の情報が含まれている．

Worker は Coordinator から実行計画を受け取ると，処理対象のデータを取得するために DS のインタフェースに合わせたクエリを生成する．以降で述べる提案手法の実装では DS に DBS を想定しているため，生成するクエリはクエリプッシュダウンする演算を含む SQL クエリとなる．SQL クエリ実行後，Worker にて演算木に従った演算処理を実施する．

Presto のクエリ処理方式は Volcano 方式をベースに，最大 1MB のページ単位でデータ要求して演算処理する方式を採用している．DS からのデータ取得は JDBC のカーソルを採用し，演算の要求に対して都度 DS からデータを取得する．なおカーソルのフェッチサイズは 1000 レコードに設定している．

4.2. 提案手法の実装

提案手法であるクエリ分割手法及び DVS の分割実行手法の Presto における実装を述べる．提案手法のシステム構成である図 3 を Presto のシステム構成に適用すると図 4 となる．色塗りのボックスが新規に作成したモジュールで，点線のボックスが Presto で既存のモジュールに変更を加えたものである．Coordinator で SQL クエリから生成される実行計画に対して分割条件を付与し，Worker に転送する．Worker は演算木に従って演算を処理し，分割条件に従って一時ストレージを利用して分割実行する．DBS のデータは，分割条件がある場合に分割クエリを生成して取得する．なお統計情報と一時ストレージは，それぞれ管理用の DBS として PostgreSQL[15]を利用して構築する．

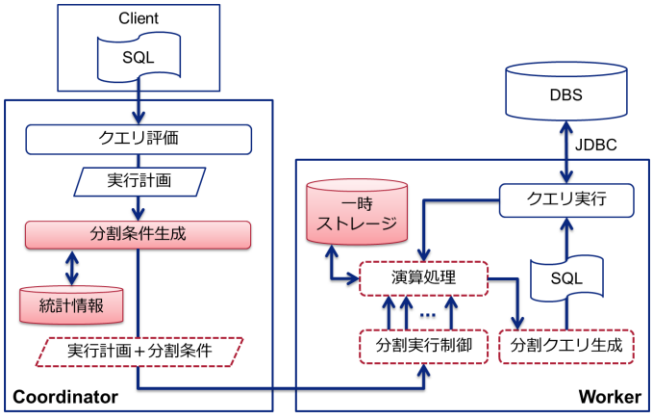


図 4　Presto における提案手法の実装

以下で，実装のシステム構成に従い，分割条件の実装，DVS の分割実行手法の実装，並びにクエリ分割手法の実装の詳細を述べる．

4.2.1. 分割条件の実装

分割条件の計算に必要な処理対象テーブルの統計情報（テーブル名，テーブルサイズ，レコード数，属性名，型名，型のサイズ，属性のヒストグラム情報）は，Coordinator 専用の管理用データベースから取得する．この統計情報は，ユーザがクエリを実行した際に，該当のテーブルがある DBS から取得して管理用データベースに格納する．一度 DBS から取得すると，テーブルのデータが変更されるまでは再度取得しない．

分割条件生成のモジュールは，まず実行計画の演算木と統計情報を利用して演算毎の出力データのヒストグラム（中間ヒストグラム）と出力データサイズを計算する．演算の入出力データの合計サイズを，Presto の利用可能なメモリサイズと比較することで分割の要否を判断する．分割を要する演算は，入出力データの中間ヒストグラムから分割条件を生成し，その演算に紐付けた形で実行計画に付与する．

Scan 演算は演算木上の直上の演算の分割条件を引き継ぐ．Scan 演算の実行時，Presto はデータ取得に JDBC のカーソルを利用する．そのため，DBS の利用可能メモリサイズが Presto より小さい場合に，分割範囲内で取得するデータサイズが DBS の利用可能なメモリサイズを超える可能性がある．これを回避するために，Presto に全取得手法と Scan 分割手法の二つを実装する．全取得手法は，JDBC のカーソルをオフにして全データを DVS に蓄積することで実現し，分割条件の生成時にそのためのフラグを設定する．もう一つの Scan 分割手法は，直上の演算の分割範囲内において，DBS の利用可能なメモリサイズを用いて更に分割範囲を計算する．なお直上の演算が分割実行しない場合は，Scan 演算の出力データサイズと DBS の利用可能

なメモリサイズから分割要否判定と分割条件の生成を行う．

4.2.2. DVS の分割実行手法の実装

Coordinator で分割条件の生成が完了した後，分割条件の有無から演算毎の入出力場所（DBS，メモリ，一時ストレージ）を決定し，各演算に付与する．入力場所として設定される「DBS」は，演算の直下が Scan 演算の場合に設定される．

Worker は Coordinator から実行計画を受け取って演算木のルートから各演算の処理を順次開始する．Presto のクエリ処理方式は Volcano 方式がベースのため，分割条件がある演算が Materialized 方式の実行となるように，本実装では直下の演算が完了するまで処理を開始しない．その直上の演算も，その分割実行が全て完了後に処理を開始する．そのため，複数の演算で分割実行がある場合，最も深い演算から処理を開始する．二項演算の分割実行に関しても同様で，それ以下の演算に分割実行がある場合は，左右の両方の演算が完了するまで開始しない．待ち状態の演算において入力場所が「DBS」の場合は，該当の Scan 演算の実行（クエリの投稿）も行わない．これにより分割実行の演算以外でメモリを消費せず処理できる．

分割実行時の入出力場所である一時ストレージには，Worker 専用の管理用データベースを利用する．入力場所が「一時ストレージ」の演算は，SQL クエリを利用して入力データを取得する．分割実行の場合は分割条件を付与して分割実行に必要なデータのみを取得する．入力データが格納されていたテーブルは，入力データを全て取得した後に削除する．出力場所が「一時ストレージ」の演算は，最初に演算処理結果の出力先テーブルを作成する．挿入には INSERT クエリを発行する．

4.2.3. クエリ分割手法の実装

分割条件が付与された Scan 演算は，Worker にて分割クエリを生成する.直上の演算が分割実行の場合は，一回の分割実行につき，一つの分割クエリを実行し，クエリ結果をその演算に渡す．その演算の分割実行が完了後に，次の分割クエリを実行する．

JDBC のカーソルオフのフラグが立っている場合は全取得手法であり，JDBC のフェッチサイズを 0 に設定変更して分割クエリを実行する．一方で，直上の演算の分割条件以外に更に Scan 独自の分割範囲がある場合は Scan 分割手法であり，DBS から取得するクエリを更にその範囲で分割して実行する．この場合，直上の演算は，その分割範囲内にある複数の分割クエリが実行完了してから，演算処理を開始する．通常の分割クエリは，直上の演算の分割実行完了後に次の分割クエリを実行するが，Scan 独自の分割クエリに限り，それぞれ実行完了後にすぐ次の分割クエリを実行する．また，Scan 演算に分割条件があり，直上の演算が分割実行ではない場合も同様に，一つの分割クエリが実行完了後，すぐに次の分割クエリを実行する．

5. 評価

5.1. 評価目的

本評価では，提案手法を実装した Presto を利用して提案手法の有効性を評価する．提案手法は，従来の Presto では実行できない規模のクエリの実行が可能となる．そこで，従来の Presto がクエリ実行を失敗する環境において，そのクエリ実行時の動作を評価する．また，Scan 演算における二つの分割クエリの生成手法である全取得手法と Scan 分割手法のそれぞれの実装の違いを明確にする．更に，利用可能なメモリサイズを超えた場合とそうでない場合との動作の違いとして，一時ストレージの利用における性能の影響を評価し，提案手法の性能を分析する．

5.2. 評価環境

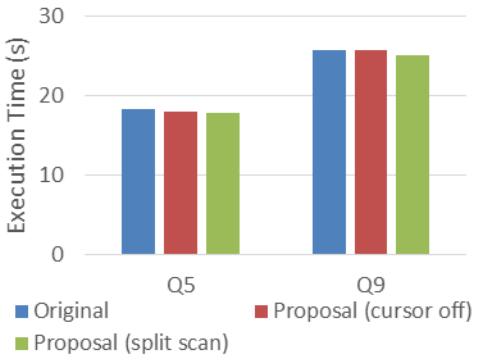
評価のために，クエリプッシュダウンを実装した従来の Presto（Original）と提案手法を実装した Presto（Proposal）を 1 台のサーバ（DVS）に構築した．なお，両 Presto の Worker 及び Coordinator は同一のサーバに構築している．データ仮想化の対象 DS は 2 台構成（PG1, PG2）とし,それぞれのサーバに PostgreSQL（9.4.0）[15]を構築した．3 台のサーバは Dell PowerEdge R410 であり，仕様は CPU: Xeon X5675 (3.06GHz , 6 Cores) x 2, Memory: 96GB, SAS-HDD: 1TB x 4 (RAID 1), NIC: 1000Base-T である．3 台のネットワークは 1GbitEthernet で構築した．導入ソフトウェアは 3 台共に CentOS 6.6 と Java 1.8.0 である．

各サーバの利用可能なメモリサイズは，評価の簡易化のために物理メモリサイズより小さくし，Presto のあるサーバを 2GB に，各 DS を 1GB に設定した．

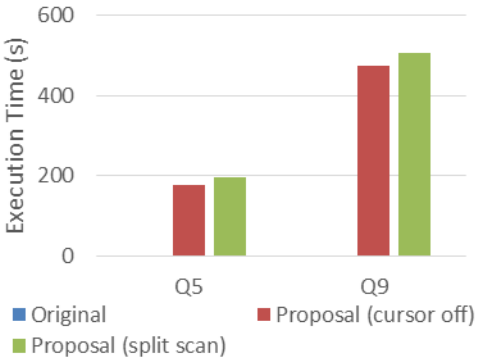
評価用のクエリとデータは TPC-H[16]を利用した．本評価で利用したクエリは，TPC-H で提供されているサンプルクエリの内，特に利用メモリサイズが大きい Q5 と Q9 を利用した．評価用データのサイズは Scale 1（約 1GB）と Scale10（約 10GB）である．データ配置は，Lineitem テーブルを PG1 に配置し，それ以外のテーブルを PG2 に配置した．

5.3. 評価結果

Original と Proposal にて，TPC-H のクエリ Q5 と Q9 を実行した．図 5 (a)は TPC-H の Scale 1 のデータに対してクエリを 3 回実行した結果の平均実行時間であり，図 5 (b)は Scale 10 のデータで同様に実行した結果である．なお，Proposal は，分割クエリの生成手法別に全取得手法（cursor off）と Scan 分割手法（split scan）でそれぞれ実行した．



(a) Scale 1



(b) Scale 10

図 5 TPC-H による従来手法と提案手法の実行時間比較

Scale 1 では，Original でも実行可能なデータサイズであり，Proposal においても分割実行が発生していない．そのため，少し誤差はあるものの，各手法が横並びの結果となった．

Scale 10 では，累計 10GB のデータであり，Original では実行できず，Proposal は提案手法によって実行可能となることを確認した．Q5 に関しては，演算の分割実行は発生していないが，PG1 で実行する Lineitem 取得のクエリが 1GB を超えた．そのため，全取得手法ではカーソルをオフにし，Scan 分割手法では分割クエリを 2 つ生成した．Q9 は，Lineitem と Part の結合演算が 2 分割され，更に PG1 への Lineitem 取得のクエリが 1GB を超えて Q5 と同様の挙動となった．従って Scan 分割手法では，Lineitem 取得の分割クエリが 3 つ，Part 取得の分割クエリが 2 つ生成された．

5.4. 考察

図 5 (b)における二つの分割クエリ手法を比較すると，性能面で全取得手法が優れていることがわかる．この二つの手法の大きな違いは分割クエリの数であり，この影響が性能の差を生んでいると考えられる．

これを検証するために，Scale 10 の Q9 における実行時間の内訳を図 6 に示す．split query は分割クエリの

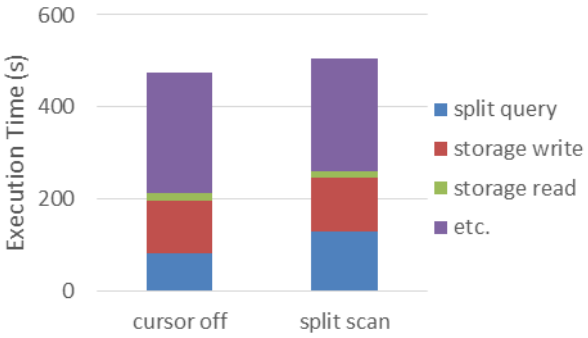


図 6 Scale 10 の Q9 における実行時間の内訳

実行待ち時間で，storage write は演算の分割実行における一時ストレージへの書き込み，storage read は同様に一時ストレージからの読み込み，etc.はそれらを除く従来手法の処理（分割クエリでないクエリの実行待ちや，Presto 上の演算処理等）である．この図から，Scan 分割手法が split query で多くの時間を要していることがわかる．分割クエリは分割基準属性にインデックスのない属性を選択すると全件探索するため，このように分割クエリの数の差が顕著となる．

上記のことから，性能面ではカーソルをオフにした全取得手法が優れていると言える．一方でリソース利用量の観点では，Scan 演算においてページ単位のデータを保持する Scan 分割手法と比較すると，全取得手法は一度全てのデータを DVS で保持する必要があり，メモリの利用効率が悪い．提案手法では全取得手法を前提に演算の入出力サイズの合計から分割条件を計算しているが，Scan 分割手法を前提に計算式を再考することで，DVS 上でより多くのデータを演算処理可能となり，分割数を削減できると考えられる．

次に，図 6 の実行時間の内訳のうち，提案手法の分割実行によって生じる処理を分析する．etc.以外の三つの部分は全て分割実行によるオーバヘッドである．ここでは，一時ストレージの読み書きに着目する．読み込み（storage read）に関しては，全体の約 3%と小さいが，書き込み（storage write）が全体の約 25%と大きく，ボトルネックとなっていることがわかる．つまり提案手法の高速化のためには，一時ストレージへの書き込みのチューニングが重要となる．現状，一時ストレージとして PostgreSQL を利用し，挿入に INSERT コマンドを利用している．本実装の性能向上という観点では，PostgreSQL のチューニングや，高速な挿入ツール（pg_bulkload 等）の利用等によってボトルネックを軽減できると考えられる．

6. おわりに

本論文では，データ仮想化において，物理メモリサイズを超える大規模データのクエリ処理性能が劇的に

劣化し，最悪の場合にはクエリ実行が失敗する課題を解決するために，クエリ分割手法及び DVS の分割実行手法を提案した．クエリ分割手法により，DS からデータを分割取得することが可能となり，DVS と DS のデータ通信時の課題を解決した．更に，DVS の分割実行手法によって DVS のストレージを利用して，演算木の各演算を分割実行することが可能となり，DVS のクエリ処理時の課題を解決した．

提案手法の有効性を確認するために，提案手法を Presto へ追加実装し，TPC-H を用いて評価を実施した．その結果，従来手法では実行できない規模のクエリに対して有効に働き，クエリを確実に実行可能であることを示した．

今後の課題は，物理メモリを超える大規模データに対して評価実験することで提案手法の有効性を検証することである．また，本論文における提案手法の実装は Volcano 方式ベースであり，Morsel 方式のようにオペレータが並列実行される場合においても同様に有効であることを検証したい．

参　考　文　献

[1] Rick F. van der Lans, Data Virtualization for Business Intelligence Systems, Morgan Kaufmann (2012).

[2] Presto, https://prestodb.io/.

[3] 齋藤和広，米田信之，渡辺泰之，村松茂樹，小林亜令，データ仮想化システムにおける効率的なクエリプッシュダウンの実装と評価，情報処理学会研究報告, Vol. 2015-DBS-162, No.25, pp.1-6 (2015).

[4] 米田信之，米川慧，齋藤和広，渡辺泰之，村松茂樹，小林亜令，データ仮想化における効率的なクエリ処理の実装と評価, DEIM2016.

[5] S. Idreos, et al., MonetDB: Two Decades of Research in Column-oriented Database Architectures, IEEE Data Engineering Bulletin, vol. 35, No. 1, pp. 40-45 (2012).

[6] G. Graefe, Volcano-an extensible and parallel query evaluation system, *IEEE Trans. Knowledge Data Engineering*, vol. 6, no. 1, pp. 120–135 (1994).

[7] V. Leis, et al., Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age, *In Proceedings of* SIGMOD, pp. 743-754 (2014).

[8] S. Fiore, et al., Data Virtualization in Grid Environments through the GRelC Data Access and Integration Service, *In Proceedings of ICITST*, pp. 1–6 (2009).

[9] M. Salloum, et al., Online Ordering of Overlapping Data Sources, *PVLDB*, Vol. 7, No. 3, pp. 133-144 (2013).

[10] S. Nural, et al., Query Decomposition and Processing in Multidatabase Systems, *In Proceedings of* Object Oriented Database Symposium of the 3rd European Joint Conference on Engineering Systems Design and Analysis, pp. 41-52 (1996).

[11] Teiid: JBoss Project, http://teiid.jboss.org/

[12] Apache Drill, https://drill.apache.org/.

[13] 齋藤和広，渡辺泰之，村松茂樹，小林亜令，ヒストグラムを利用した SQL クエリの分割範囲計算手法の提案，情報処理学会第 77 回全国大会大会講演論文集第 1 分冊，pp.497-498 (2015).

[14] N. Bruno and S. Chaudhuri, Exploiting Statistics on Query Expressions for Optimization, *in Proceedings of* SIGMOD, pp.263-274 (2002).

[15] PostgreSQL, http://www.postgresql.org/

[16] TPC-H, http://www.tpc.org/tpch/