Vol.2015-DBS-162 No.25
2015/11/26

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

齋藤和広[†1]　米田信之[†1]　渡辺泰之[†1]　村松茂樹[†1]　小林亜令[†1]

概要：データ分析の複雑化により，用途や目的に応じて構築された複数の異なるデータベースシステム(DBS)へのアクセスが必要な事例が増えている．より効率的にデータ分析を行うために，そのような複数の DBS の統合が求められている．この統合を実現する手段の一つとして，データ仮想化システム(DVS)を用いた複数 DBS の論理的統合がある．DVS はユーザが投稿したクエリを解釈し，処理に必要なデータを DBS から取得してクエリ処理を行う．DVS の性能は，DBS からネットワークを介して取得するデータ量に大きく依存する．そのため，DBS 単体で処理可能な演算を対象 DBS に委譲するクエリプッシュダウンを行うことで，DBS からの取得データ量を削減し性能を向上することが可能となる．しかしクエリによっては，クエリプッシュダウンによって取得データ量が増加し，性能低下する課題がある．そこで本稿では，クエリプッシュダウンを行う演算の選択と演算の順序を制御して課題を解決する手法を提案する．また，本手法をオープンソースソフトウェアの Presto に実装し，TPC-H を用いて有効性を評価する．

キーワード：データ仮想化システム，データベースシステム，クエリ最適化，クエリプッシュダウン，Presto

Implementation and Evaluation of An Efficient Query Pushdown Method for Data Virtualization Systems

KAZUHIRO SAITO[†1]　NOBUYUKI MAITA[†1]　YASUYUKI WATANABE[†1]
SHIGEKI MURAMATSU[†1]　AREI KOBAYASHI[†1]

Abstract: A complexity of the data analysis has led to the necessity to leverage multiple database systems (DBSs) separately constructed for their own use cases and purposes. In order to conduct the complex data analysis more efficiently, integration of such multiple DBSs is required. One of techniques to achieve this is logical integration of multiple DBSs using a data virtualization system (DVS). The DVS parses a query posted by a user, and processes it by retrieving required data from DBSs. A performance of the DVS depends on a size of data received from DBS through the network. Therefore, by conducting query pushdown which pushes operators included in their query into proper DBSs, it is possible to reduce the size of received data and to improve the performance. However, query pushdown has a problem that it sometimes leads to performance degradation by picking up an inappropriate set of operators in a user query for pushdown. In this paper, we propose a method to avoid this problem by controlling the selection and the processing order of sets of pushdown operators. Additionally, we implements our method to open source software, Presto, and evaluates its availability with TPC-H benchmark.

Keywords: data virtualization system, database system, query optimization, query pushdown, 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ベースシステム（DBS）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 DBS の統合が求められている．しかし DBS 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 DBS 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．データ仮想化環境下では，データ仮想化システム（DVS）が対象の DBS とネットワークを介して接続する．DVS はユーザが投稿したクエリを解釈し，必要なデータを DBS から取得してクエリを処理する．

DVS のクエリ処理性能は, DBS からネットワークを介して取得するデータ量に大きく依存する．このデータ量は，DBS 単体で処理可能な演算を対象 DBS に委譲するクエリプッシュダウン 1)によって削減可能である，しかし，複数の DBS が対象となるクエリでは, クエリプッシュダウンが却って性能劣化を引き起こす課題がある．

そこで本稿では，クエリプッシュダウンを行う演算の選択と演算の順序を制御することで性能劣化する課題を解決し，複数の DBS をより効率的に活用する手法を提案する．本手法では，転送データ量が増加する結合演算を分解し，別のサブクエリとして DBS にクエリプッシュダウンする．更に, DVS がクエリプッシュダウンして取得するデータの結合処理の順序をクエリプッシュダウンのコストをベースに制御して，クエリ処理における DVS と DBS の並列性を

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

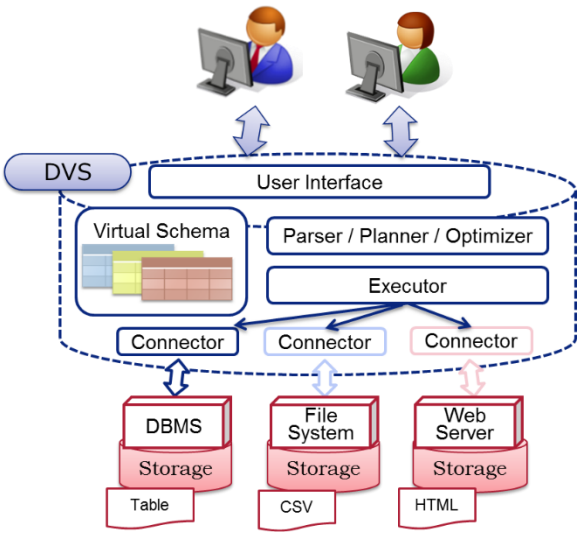


図 1　データ仮想化システムの概要図

高めることで性能を改善する．また，オープンソースソフトウェアの分散 SQL エンジンである Presto に本手法を実装し，TPC-H ベンチマークによって有効性を評価する．

2. 従来のクエリプッシュダウン

DVS のデータソースとなる対象システムは, 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 はデータソース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)．代表的な実装例が Presto 2)である．

本稿におけるデータソースは DBS を対象とし，DVS 及び DBS のインタフェースに SQL を想定する．DVS は，取得するデータ量を削減するために，DBS に対して SQL の関係演算処理（射影や選択，結合など）を出来る限りクエリプッシュダウン 1)することができる．DVS の一種である MIND 3)やオープンソース実装の Teiid 4)では，同一の DBS に存在するテーブルを一つのサブクエリにまとめてクエリプッシュダウンし, 異なる DBS のクエリ結果同士の結合処理を DVS 上で実行する．

しかしながら, 複数の DBS に対してクエリプッシュダウンすることで，直積や多対多の結合演算が発生し，取得するデータ量が肥大化する可能性がある．図 2 は TPC-H の Q5 5)の Join graph 6)であり，ノードはテーブル，エッジは結合演算を表す．エッジに丸のついたテーブル S: supplier と C: customer の結合が多対多である．ここで，テーブル L: lineitem とその他のテーブルは別の DBS にあると仮定する．DBS にクエリプッシュダウンする場合, 図 2 のように多対多の結合演算を含むサブグラフ Subquery 2 の結果が肥大化する．Scale 1 におけるレコード数は，Subquery 2 の各テーブルをクエリプッシュダウンせずに個別に取得すると合計

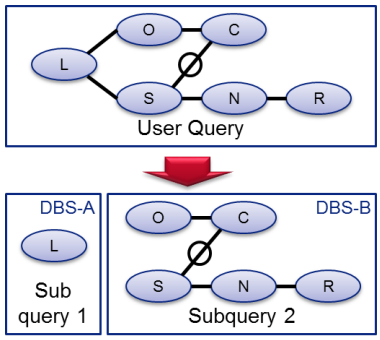


図 2　TPC-H Q5 における従来の Join graph 分解

で約 166 万レコードだが，クエリプッシュダウンすると約 6000 万レコードと 40 倍になる. 結果として, DBS から DVS へのデータ転送がボトルネックとなり，クエリプッシュダウンによって性能劣化が発生する．この中間結果が肥大化して性能劣化する課題は直積に関しても同様である．なお Teiid の手法では直積を回避しているが. 多対多の結合演算では同様の現象が発生する．

また，クエリプッシュダウン後の結合順序に関して，MIND は実行時に最も早く受信したサブクエリ結果から順に結合処理を行う動的な手法を用いている．これは事前計算によるクエリプッシュダウンのコストの予測ミスを回避することが目的である．しかし，結合演算対象の二つのデータを取得するまで結合処理が開始できない．そのため，各データを逐次的に処理する結合アルゴリズムの場合，図 3 のように片方の処理が完了するまで待ち時間が発生し，クエリプッシュダウンによる DBS のクエリ実行と DVS におけるクエリ実行が並列実行されない課題がある. 一方で, Teiid は DVS における結合処理の順序をクエリ実行前にサブクエリのコストベースで決定する．その順序は従来の DBS と同様であり，left-deep な演算木を前提に DVS における結合演算のコストが最小となる順序を選択する．そのため, クエリプッシュダウンによる DBS のクエリ処理コストの考慮がなく，DBS のクエリ結果が小さい場合でも，そのクエリ実行時間が長いことで DVS における結合処理の待ち時間が発生し, 結果として図 3 のように MIND と同様に DVS と DBS のクエリ処理が並列実行されない．

3. 提案手法

DVS において，直積や多対多の結合演算のクエリプッシュダウンを回避し，更に DVS と DBS の並列性を考慮した結合順序により，前述の課題を解決する．3.1 では，直積や多対多となる結合演算部分を Join graph から分離する手法を述べる．3.2 では，DVS 上のクエリ処理とクエリプッシュダウンによる DBS のクエリ処理の並列性を向上するコストベースの演算木作成とコスト計算手法を述べる．

3.1 Join graph の分解

DBS から取得するデータ量を削減するために，クエリプ

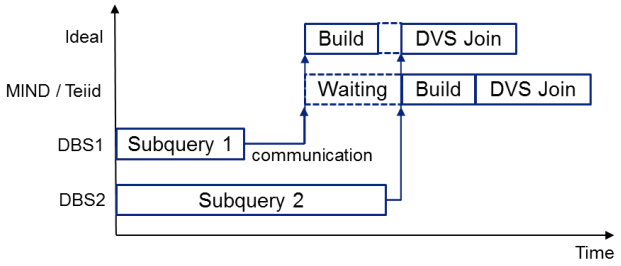


図 3　クエリ実行時における DVS（MIND，Teiid，理想）と DBS の時系列推移の比較

ッシュダウンによる直積や多対多の結合演算を回避する．始めにユーザクエリから Join graph を生成する．次にクエリプッシュダウンの単位で Join graph を分解するために，エッジで繋がるテーブルのペアを取得して，Join graph 分解の必要性を確認する．エッジ単位で確認することで，クエリプッシュダウンによる直積の発生を防ぐ．ここで以下の二つの条件のどちらかを満たした場合に，エッジを削除し，テーブルのペアとエッジを DVS 上の処理として別の Join graph に記録する．

- それぞれのテーブルがある DBS が異なる
- 結合演算が多対多

エッジ削除後の Join graph において，確認した二つのテーブルが，他のエッジによって繋がっていない場合は，これらのサブグラフを別の Join graph として分離する．以上を全エッジ分繰り返すことで Join graph の分解が完了し，DBS に投稿するサブクエリが完成する． TPC-H の Q5 に対して，図 2 と同様の環境下で上記を行うと，図 4 で示すサブクエリが完成する．

結合演算が多対多である条件は，結合後のレコード数が，大きい方のテーブルのレコード数を超えることである．結合後のレコード数が不明な場合は，小さい方のテーブルの結合条件の属性がユニークでない場合に，多対多であると判断できる．

3.2 クエリプッシュダウン結果の結合順序効率化

Join graph の分解が完了した後に，DVS の結合演算の順序を決定し，演算木を作成する．提案手法では，結合演算の処理アルゴリズムとして Hash Join 方式を前提とする．この前提では，DVS がクエリプッシュダウンの結果を取得したあとに，ハッシュテーブル生成完了までのハッシュ突合の待ち時間を最小化することでクエリプッシュダウンによる DBS のクエリ処理との並列性を向上できる．また，多くの DBS では left-deep な演算木を構成するが，DVS のアーキテクチャでは bushy な演算木によってより多くのクエリプッシュダウンの並列実行が可能となり性能が向上する 7).

そこで提案手法では，ハッシュテーブルの生成対象（Building テーブル）と，ハッシュテーブルへの突合対象

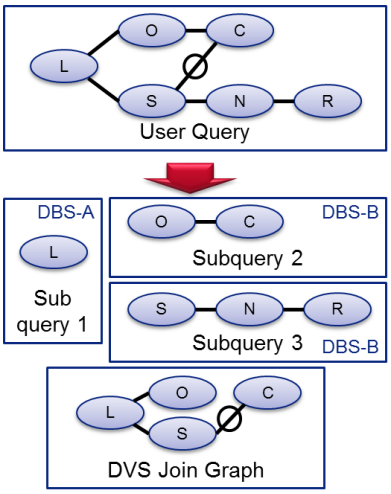


図 4　提案手法の Join graph 分解

（Probing テーブル）をサブクエリのコストを基に選択し，クエリプッシュダウンによる並列性を確保する．3.1 で作成した分解されたサブクエリ（Join graph）のうち，コストが最も大きいサブクエリを選択する．このサブクエリと結合可能なエッジを DVS Join graph から選択し，その結合先のサブクエリのうち，結合条件が多対多でなく，かつコストが最も小さいサブクエリとの結合を選択する．このサブクエリ同士の結合を一つの DVS 上の Join graph として保存する．これを繰り返して DVS 上の Join graph とサブクエリを全て繋いで演算木を作成することで，結合順序の効率化が完了する．ここで 2 回目以降，一度 DVS 上の Join Graph で繋がった結合演算は，サブクエリと同様にコストを再計算し，残るサブクエリとの結合対象となる．そしてサブクエリよりも DVS 上の結合処理後のコストの方が低いと判断された場合に DVS 上の結合演算も含めて bushy な演算木となる．以上により，DVS と DBS のクエリ処理の並列性を最大限利用した実行を可能とする．

TPC-H の Q5 において図 4 の分解後に適用すると，図 5 で示すサブクエリを繋いだグラフが作成される．この図では，始めにテーブル L のサブクエリ 1 と，テーブル S, N, R を含むサブクエリ 3 が結合し，次にその結合結果とテーブル O, C を含むサブクエリ 2 を結合する順序となる．これを演算木に適用すると図 6 になる．

結合演算の順序決定に利用されるコストは，従来の DBS では対象テーブルのサイズがコストとして利用され，分散 DB では通信コストが利用される 6). 一方で DVS のアーキテクチャにおけるコストは，DBS におけるサブクエリ実行に関わるコストであり，通信コストが小さくとも，元のデータ量が大規模であれば，通信開始までに多くの時間を要する．これらを考慮して結合演算の処理時における待ち時間を最小化して DVS と DBS の並列性を高めるために，Building テーブルと Probing テーブルの選択条件に DVS における処理コストと DBS における処理コストを両方考慮

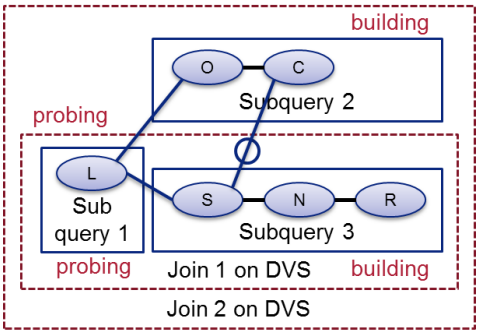


図 5　提案手法を適用後の結合順序

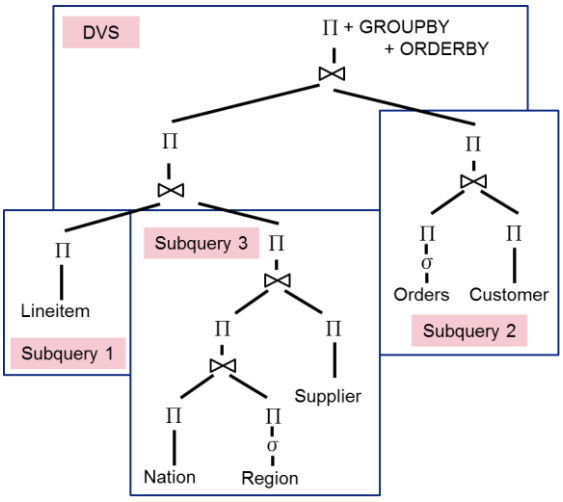


図 6　図 4 の演算木

する必要がある．そこで提案手法におけるサブクエリのコストは，読み込むテーブルの合計サイズと演算結果のデータサイズを合計したサイズをコストとする．

4. 実装

4.1 Presto

Presto 2)は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り，Hadoop に対して独自の Java プログラムを介さずにインタラクティブなインメモリ処理を提供するシステムである．Presto は汎用的なコネクタを持ち，Hadoop だけでなく様々なデータソースからデータを取得する SQL クエリを処理できる．この仕組みは DVS と同様であることから，本稿では Presto に提案手法を実装する．バージョンは 0.100 を利用する．

また Presto は，データソースが 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，従来のクエリプッシュダウン手法として，数値型以外の選択演算，結合演算，集約演算（Group by），整列演算（Order by），集合演算（Union）のクエリプッシュダウンを実装した．

4.2 提案手法の実装

前述の追加実装を施した Presto に対して提案手法を実装した．Presto の内部構造に出来る限り変更を加えないよう，最適化が完了した演算木に対して提案手法を適用し，演算木を変更する実装とした．

まず演算木からテーブル情報と結合演算情報を抽出して Join graph を作成する．これに提案手法を適用することで Join graph を分解し，更に結合演算の順序を決定する．そして元の演算木に対して，作成した結合順序を適用する．

Presto の演算木は多くの DBS 実装と同様に left-deep に構成されていて，結合演算を示すノードの左側に Probing テーブルもしくは結合演算ノードが繋がり，右側に Building テーブルが繋がるシンプルなデータ構造となっている．一方で提案手法では bushy な構造となる．まずサブクエリの部分木をこの演算木で構成するために，サブクエリに紐づくテーブルを一つの Probing テーブルもしくは Building テーブルに集約する．そしてサブクエリ単位に集約した後，同様にサブクエリの部分木単位で入れ替えて，提案手法で作成した結合順序の演算木に変更する．なお必要に応じて，結合演算の条件式や射影演算を修正している．

コストの計算に必要な Join graph の結合演算と演算木における各演算の中間結果サイズはヒストグラム方式 8)を利用してデータサイズ推定した．

5. 評価

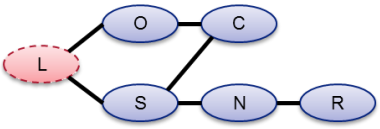
5.1 評価環境

提案手法の有効性を評価するために，4 節で述べた Presto を利用する．評価のために，従来の Presto（Original），従来のクエリプッシュダウンを実装した Presto（Pushdown），提案手法を実装した Presto（Proposal）の計 3 システムを 1 台の同一のサーバ（DVS）に構築した．なお，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 DBS は 2 台構成（PG1, PG2）とし,それぞれのサーバに PostgreSQL（9.4.0）9)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 5), 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．

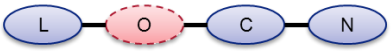
5.2 評価クエリ

評価には TPC-H ベンチマークの Scale 10（全テーブルの合計が約 10GB）のデータセットを利用する．利用するクエリは TPC-H の Q3, Q5, Q10 である．これらの Join graph と評価時の配置場所を図 7 に示す．Q3 は図 7 (a)の通り，三つのテーブルの結合演算を含み，テーブル L: lineitem を PG1, それ以外を PG2 に配置して評価する．この評価では，クエリプッシュダウンの有無による評価を行い，Original に対する，Pushdown 及び Proposal の効果を示す．Q5 は図 7

(a) TPC-H Q3, O(lineitem): PG1, others: PG2



(b) TPC-H Q5, L(lineitem): PG1, others: PG2



(c) TPC-H Q10, L(orders): PG1, others: PG2

図 7 評価用クエリの Join graph と配置場所

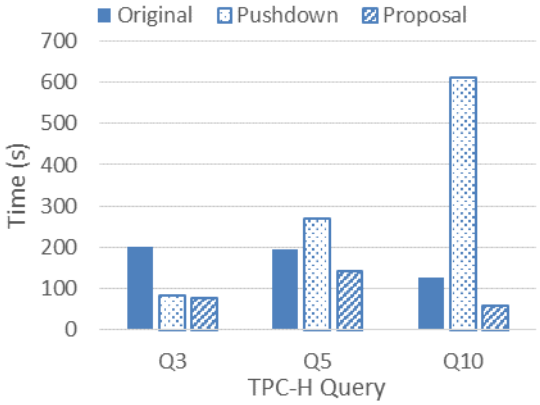


図 8 TPC-H による提案手法の評価結果

(b)の通り，2 節及び 3 節の例で述べた環境と同様であり，3 環境における実行時間を比較して評価する．Q10 は図 7 (c)で示す通り四つのテーブルの結合演算を含み，テーブル O: orders を PG1 に，それ以外を PG2 に配置して評価する．この環境では Pushdown のシステムで直積が発生し，その場合の実行時間を評価する．

5.3 評価結果

5.2 で述べた三つのクエリを用いて，Original, Pushdown, Proposal の 3 環境で実行した．図 8 がこの結果である．縦軸が実行時間，横軸でクエリ毎のシステムの実行結果を比較している．

Q3 を実行した結果から，従来の Presto である Original に対して Pushdown 及び Proposal がクエリプッシュダウンによって性能向上していることがわかる．演算木を変えずにクエリプッシュダウンしている Pushdown は 2.42 倍，提案方式により結合演算の順序を効率化している Proposal は 2.67 倍性能が向上した．この Pushdown と Proposal の差は，ハッシュ結合とクエリプッシュダウンの並列性の向上によるものであると言える．Pushdown では，クエリプッシュダウンしたテーブル O: orders とテーブル C: customer の結合処理のサブクエリ結果を Probing テーブルとし，テーブル L: lineitem を Building テーブルに結合演算を処理した．一方で，両者のコストを計算するとテーブル L のコストが大きいことから，Proposal ではテーブル L を Probing テーブルとし，サブクエリ結果を Building テーブルとしている．その結果，1.1 倍の性能差が発生し，クエリプッシュダウンの性能向上を実現している．

次に Q5 を実行した結果は，2 節で述べたように，従来のクエリプッシュダウンによって多対多の結合演算が発生し，Pushdown が Original よりも 1.40 倍遅延した．一方で提案手法を実装した Proposal では従来の二つのシステムより高速化し，Original より 1.37 倍，Pushdown より 1.91 倍高速化した．本結果より，提案手法によりクエリプッシュダウンによる性能劣化を回避しつつ，クエリプッシュダウンの効果が発揮されていることを示した．

Q10 では Pushdown の従来のクエリプッシュダウンによって直積が発生している．その結果，Original と比較して 4.8 倍もの遅延が発生している．一方で提案手法は，クエリプッシュダウンのない Original より 2.16 倍高速化した．また従来のクエリプッシュダウンである Pushdown より 10.48 倍高速化した．

以上のことから，多対多及び直積においてクエリプッシュダウンによる高速化を実現した．また，DVS の結合演算の順序を DBS と DVS の両方のコストによって並び替えることにより，従来手法と比較してより効率的なクエリプッシュダウンが可能となった．

6. おわりに

本稿では，クエリプッシュダウンを効率的に実行するために，直積や多対多の結合演算を回避する Join graph の分解と，クエリプッシュダウン後の結合演算順序をコストベースで制御して DVS と DBS の並列性を高める手法を提案した．その結果，従来手法ではクエリプッシュダウンによって直積や多対多の結合演算により性能劣化していたクエリを，提案手法によって高速化できることを示した．また従来のクエリプッシュダウンにおいても性能向上するクエリに関して，結合演算の順序効率化によって DVS と DBS の並列性を高め，性能向上できることを示した．

今後，提案手法で作成した結合演算の順序に関して，DBS や DVS の演算処理コストを含めたコストモデルを作成したい．また，実用に近い形で評価するために，実際の業務アプリケーションで利用されるクエリと，実際のデータ配置を考慮した環境で評価したい．

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) S. Nural, P. Koksal, F. Ozcan and A Dogac, “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.
4) Teiid, http://teiid.jboss.org.
5) TPC-H, http://www.tpc.org/tpch/.
6) M. TamerOzsu. and P. Valduriez., “Principles of Distributed Database Systems Third Edition”, Springer Science, 2011
7) W. Du, M. Shan and U. Dayal, “Reducing Multidatabase Query Response Time By Tree Balancing”, *in Proceedings of* the 1995 ACM SIGMOD international conference on Management of data, pp. 293-303, 1995.
8) N. Bruno and S. Chaudhuri, “Exploiting Statistics on Query Expressions for Optimization”, *in Proceedings of* 2002 ACM SIGMOD international conference on Management of data, pp.263-274, 2002.
9) PostgreSQL, http://www.postgresql.org/