高速なデータ仮想化を実現するSparkクエリアクセラレーション

三浦　貴司†　　中村　　実†

† 株式会社富士通研究所　〒211-8588 神奈川県川崎市上小田中 4-1-1
E-mail: †{miura_takashi,nminoru}@jp.fujitsu.com

あらまし　企業は大量のデータを蓄積しているが，これらのデータは部署ごとにあるいはデータ種類ごとに複数のデータベースに管理するのが一般的である．しかし変化の激しい今日のビジネス環境においては，これらの分断されたデータを横断する分析をリアルタイムに行う必要に迫られている．このためにネットワーク上に散らばる複数のデータベースを物理的なデータ移動を行わずに 1 台のデータベースの仮想的なテーブルとして登録し，仮想テーブル間を分析するクエリを実行可能にするデータ仮想化技術が注目されている．しかしデータ仮想化は個々のデータベース製品の内部情報 (インデックス，統計情報) を知ることができないままクエリを実行するため，1 つのデータベースに全てのデータを集約した場合に比べて分析の処理速度が低下する．そこで本論文では，データ仮想化システムのクエリ処理を Apache Spark によって分散実行することで高速化する手法を提案する．Spark はパーティション分割のための列名といくつかの変数をユーザが選択しそれに基づいてノード間にデータを分散させるが，我々の提案手法ではデータ仮想化対象のデータソースのデータ量を元にパーティション分割のための列名と変数を合成することで効率的な分散実行を実現する．我々の提案手法を PostgreSQL ベースで実装し，実機評価によりその効果を示した．

キーワード　データ仮想化，PostgreSQL，Apache Spark

1. はじめに

今日様々な種類のデータが大量に生み出され，それに対応した多種多様なデータマネジメントシステムが生み出されている．リレーショナルデータベース (RDB)，Key Value Store (KVS)，Document DB など存在し，それらの製品は 300 を超える [1]．スキーマを事前設計する構造データ以外に，スキーマの事前設計を行わない半構造データ，非構造データ，ストリームデータの利用が拡大している．さらに，これら膨大なデータに対する統計分析，機械学習，AI の活用の重要性も増している．

一方，日々生み出される大量のデータは同一の企業内でも管理する部署・データの種類・アプリケーションごとに別々のデータベースに格納され「データソースのサイロ化」が生じている．また企業内で運用されるデータベースの数も固定ではなく，日々新しいデータベースが追加される．しかし現在のビジネス環境で求められているのは，このような分断され点在するデータベース間のデータを横断的に分析し，効果的に経営に反映させることである．さらに企業内に閉じたデータ以外に，地理情報や道路交通情報などのパブリックデータが Web API や Linked Open Data (LOD) などで多数公開されており，これをローカルなデータと組み合わせて分析したいという要求も高まっている．

このような要求に対して，従来は各データベースの内容を ETL(Extract/Transform/Load) ツールを介してフォーマットを変換しつつ物理的に一ヶ所に集め，データウェアハウス (data warehouse; DWH) を作成してから分析していた．しかしこのようなデータ統合はデータのコピーの過程で多くの時間を消費するため，現在のデータを使用したリアルタイム分析は不可能であった．

このような問題を解決する手段がデータ仮想化 (Data Virtualization) である．データ仮想化システムは，仮想的なデータベースを作成し，個別のデータベースの中にあるテーブルやテーブルに類似するデータ構造（例えば MongoDB のコレクションなど）をその中に仮想テーブルとして登録する．仮想データベースの設定後は普通のデータベースのように SQL クエリを実行することができる．クエリが実行されて仮想テーブルに対してスキャンが行われた時に初めて，データ仮想化システムは個別データベースからデータをリアルタイムに取り出すことになる．複数の仮想テーブルをテーブル結合 (JOIN) するようなクエリであれば，実際には異なるデータベース間のデータを統合して分析することになる．

しかし複雑な分析クエリを実行する場合，データ仮想化は DWH のような物理統合した場合に比べて処理速度が遅くなる傾向にある．データを DWH に格納する場合，データを列ごとに格納するカラム指向格納，インデックス，統計情報の採取などが行われ，クエリはこれらの情報を利用しながら最も効率の良い実行プランを選択して実行する．一方，データ仮想化は配下に置いたデータベースの統計情報などを把握しておらず，効率のよい実行プランを生成することができないために処理時間が伸びる場合がある．この性能差はクエリの中で結合するテーブルの数が増えれば増えるほど顕著となる．

このデータ仮想化における問題を解決するために，本論文ではデータ仮想化クエリ処理の分散実行を提案する．従来の Cisco Data Virtualization [2] などは，単一サーバ内のマルチコアを利用した並列実行は可能だったが，複数台のサーバを利用した分散実行はできなかった．我々の提案手法は Apache Spark [3]

を用いて仮想化クエリの分散実行を実現する．

本論文は以下の通り構成されている．2 章では関連技術について述べる．3 章では前提とするデータ仮想化システムについて述べる．4 章では提案手法について詳しく述べる．5 章では今回の評価実験について述べる．6 章では実験結果と考察を示す．7 章では今回の提案と実験を総括する．

2. 関連技術

Spark を活用する試みはいくつか存在する．Spark は，統計や機械学習の処理には SparkR や ML (MLlib)，(準) ストリーミング処理に対しては Spark Streaming や Structured Streaming，グラフ分析に対しては GraphX，RDB のような SQL による処理に対しては SparkSQL が用意されており，統一的なインターフェイスで複数種類の処理を組み合わせて実行可能である．このような特徴をうまく活用する試みとして，HPE Vertica は Spark と接続するためのコネクターを提供しており，Spark の ETL ツールとしての活用や Spark ML を利用した高速な分析環境の実現のために使用している [4]．SAP HANA Vora は Spark と連携することで Hadoop 環境下のビックデータと SAP HANA の基幹データを組み合わせながら SQL で分析するニーズに対応している [5]．IBM Wildfire は Spark プラットホーム上で OLTP と OLAP を両立させる試みを行っている [6]．SnappyData は Spark をベースとしてストリーム，OLTP，OLAP の 3 つをオールインワンで実現するインメモリデータベースを実現している [7]．

3. データ仮想化システム

我々の提案手法を適用するデータ仮想化システムは，OSS のデータベースである PostgreSQL 9.5 [8] をベースとして開発を進めている．図 1 はこのシステムの概略図である．これは我々が PostgreSQL をベースとした製品開発を行っており，過去に PowerGres Plus [9] や FUJITSU Software Enterprise Postgres [10] に出荷しているためである．我々の提案手法は PostgreSQL にデータ仮想化機能をアドインする運用を想定している．

PostgreSQL をベースとすることは以下のようなメリットがある．

- データ仮想化システムのために必要な SQL 文の解析ルーチンや実行プランの最適化ルーチン，他のデータベースとの接続機能を PostgreSQL から流用することが可能である．
- PostgreSQL の持つセキュリティ機能 (Role-base security，Row-level security) をデータ仮想化の中に取り込み，データソースに因らないアクセス制限を可能にする．
- PostgreSQL 自身が保持するデータはデータ仮想化機能から見てローカルなデータであり，このデータに対してはインデックスや統計情報を使った高速なアクセスが可能になる．

我々のデータ仮想化システムでは，PostgreSQL が外部データを取り込む仕組みである Foreign Data Wrapper(FDW) を拡張した専用 FDW を設けて仮想化対象のデータベースとの接続を開発中である．PostgreSQL には Oracle DB や MySQL など各種データベース製品との接続のための FDW が存在しているが，実現している機能はまちまちであった．我々のデータ仮想化システムの FDW では，PostgreSQL の SQL から別のデータベース製品の SQL への変換の枠組みを実装し，各データベース製品への対応は SQL 方言の情報とそのデータベース製品への接続部分だけを実装することで統一的に行う狙いがある．

しかし PostgreSQL FDW をベースとしたデータ仮想化システムで TPC-H の分析クエリを使って予備測定の結果，長大な時間がかかることが判明している．TPC-H [11] のスケール 100 のテーブルを外部 PostgreSQL に格納して FDW を介して我々のデータ仮想化システムに接続し，クエリを実行した場合は表 1 のように処理時間が 1 時間を超えるものも存在する．1 台の PostgreSQL サーバ上で同じクエリを実行した場合には数分で実行可能であることから，実用を目指すのであればこの処理時間と同等の時間で処理可能にするための大幅な性能向上を行う必要となる．

そこで我々はデータ仮想化システムでのクエリ処理を Spark にオフロードすることで，分散処理によるクエリ処理速度の向上をはかる．

表 1　FDW を介して外部データソース（PostgreSQL）をデータ仮想化システムに接続して TPC-H (SF100) のクエリを処理するのに要する時間（秒）．詳細は 5 章を参照されたい．

	new	ageing
Q1	5362	5430
Q3	2011	2133
Q5	3233	3412
Q6	196	225
Q10	1051	1123
Q17	3017	3203

4. 提案手法

4.1 Apache Spark

Spark へクエリ処理をオフロードすることで処理の高速化を実現するために，Spark SQL [12] を利用してクエリの処理を行う．Spark SQL は Spark 上で SQL クエリの処理を担っており，JDBC などデータソースとのコネクターを通してデータソースからデータを取得し，RDB のテーブルのようなスキーマを持つ DataFrame の形でデータが保持される．

Spark の持つ分散並列実行能力を活かすためには，従来はユーザ自身が SQL クエリを発行する前に適切にパーティション設定を行う必要があった．パーティション設定とは 1 つの DataFrame を複数に分割して並列処理させるための設定である．図 2 は TPC-H の lineitem に対する DataFrame のパーティション設定の例である．パーティション設定には，データソースの url，ユーザ名とパスワード，テーブル名 dbtable を指定した上で，さらにパーティション分割のための基準となる列名 partitionColumn （以下ではパーティション列と呼ぶ.），パーティション列の要素値の最小値 lowerBound と最大

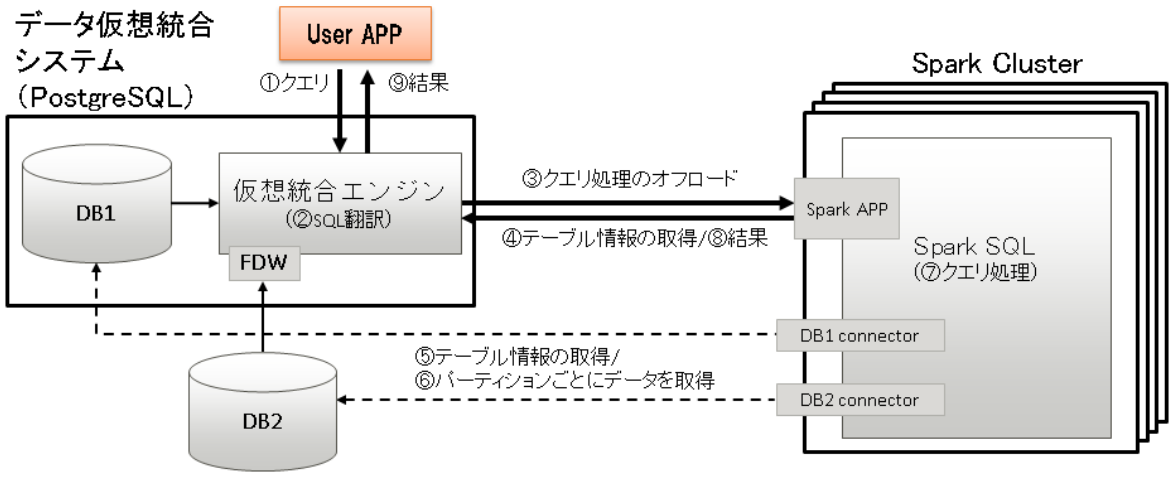

図 1　提案するデータ仮想化システムと Spark クラスターとの連携.

```
val myDF = spark.read.format("jdbc")
              .options(Map("url" -> "jdbc:postgresql://host:port/databaseName"
                                    + "?user=user&password=password",
                           "dbtable" -> "lineitem",
                           "partitionColumn" -> "l_orderkey",
                           "lowerBound" -> "1", "upperBound" -> "600000000",
                           "numPartitions" -> "6000",
                           "fetchsize" -> "100000"))
              .load()
```

図 2　TPC-H の lineitem に対する DataFreme 作成時のパーティション変数の設定の例

値 upperBound，パーティション数 numPartitions，JDBC コネクターを使用する場合には 1 度にデータ転送する行数を指定する変数 fetchsize を設定する必要がある．この時，データソースのテーブルデータを漏れなく転送するためには，パーティション列を主キー列のような NULL を許さない整数型の列に指定する必要がある．図 2 の例では lineitem の l_orderkey をパーティション列として 1~600000000 の間の値を 6000 個の領域に等分割し，各パーティションへのデータ転送を 1 度に 100000 行単位で行う設定である．

このようなユーザによるパーティション設定をもとに，ユーザが SQL クエリを発行したタイミングで，Spark SQL はデータソースと JDBC コネクターを通して

SELECT ∗
FROM *lineitem*
WHERE *l_orderkey* >= 100001
AND *l_orderkey* < 200000 ;

のような領域分割の WHERE 句を含むクエリ発行し，データソースはこの WHERE 句に基づいて各パーティションにデータを転送する．

しかし，図 2 のパーティション設定のための変数をユーザがクエリ発行時に一々設定していたのではアドホックなクエリによるリアルタイムな分析の障害となる．

4.2　効率的なパーティション設定の自動化

そこで今回我々は，ユーザがデータ仮想化システムに対してアドホックなクエリを発行し Spark へクエリ処理をオフロードして効率良く処理するために，クエリ処理実行時に自動的にパーティション設定を行う方式を採用する．さらにこの時，従来のパーティション列の要素値に基づくパーティション設定ではなく，データ量に基づく効率の良いパーティション設定の方法を提案する．

この提案方式を採用することで次の 2 つの場合に効果があると考える．

- 追記型データベース導入時に順番に整列していた主キー列の要素値が，長年稼働している間に大量の UPDATE が実行されることで徐々にバラバラに配置されてしまう場合
- テーブル内に整数型の主キー列に相当する列が元々存在しない場合

1 つ目は長年稼働している PostgreSQL のような追記型のデータベースを使用する場合である．このようなデータベースのテーブルに対して主キー列をパーティション列に指定すると，

要素値がバラバラに配置されていることでデータ読み取り時のランダムアクセスが大量に発生してクエリの処理速度の低下が起こる．このようなクエリ処理速度の低下を回避するために，データ量でのパーティション設定を行うことで，データ読み取り時のランダムアクセスを生じさせずに効率の良いクエリ処理が可能であると考える．2つ目は従来手法でのパーティション設定において十分に並列実行を行えない場合である．テーブル内に整数型の主キー列に相当する列が存在しない場合には代わりとなるパーティション列を採用する必要があるが，この場合パーティション設定を行っても特定のパーティションにデータが集中してしまいバランスの良い並列実行が実現できなかったり，代わりとなるパーティション列が存在しないためにパーティションが1つになってしまう．このためにSparkへクエリ処理をオフロードしても全く並列実行を活かした高速な処理を行えない．このような場合であっても提案手法であるデータ量に基づくパーティション設定は有効であり，Sparkの並列実行能力を十分に生かした高速な処理が可能であると考える．

今回はデータソースとしてPostgreSQLを使用することを前提とし，データ量に基づくパーティション分割の自動化を実現するために以下の開発を行った．

- データ仮想化システムであるPostgreSQLのSQL文からSpark SQLのSQL文への翻訳処理
- データソースのurl，ユーザ名とパスワード，テーブル情報の取得によるパーティションの自動設定
- データ量でのパーティション設定のためのデータソースであるPostgreSQLのctid列に対する高速なカスタムスキャン
- Spark SQLのctid列対応

1つ目にSpark SQLのSQL文への翻訳処理の開発である．データ仮想化システムに投入されるSQL文はPostgreSQLのSQL方言に基づくものであり，このままSparkへオフロードしてもSpark SQLはSQL文字列を認識できない．そこで，投入されたクエリのSQL文字列を1度パースし，1つ1つの句の内容をSpark SQLのSQL方言に置き換え，SQL文を再構築する翻訳処理をデータ仮想化システムでの処理として開発した．

2つ目にデータソースのテーブル情報等の自動集処理の開発である．Sparkへクエリ処理がオフロードされた後は自動的に必要な情報を取得してパーティション設定を行う必要がある．まずテーブル名をもとにデータ仮想化システムの情報テーブルpg_foreign_server，pg_foreign_data_wrapper，pg_user_mapping，pg_foreign_table，pg_classの5つのテーブルに問い合わせを行い各データソースのurl，ユーザ名とパスワードを取得する．次にこれらの情報からデータソースに直接と合わせを行い図2のようなパーティション設定のための情報を取得し，これをもとに各パーティションにほぼ均等のデータ量が割り当たるようにパーティション設定のための変数を算出する．この時ユーザが指定する変数は1つのパーティション割り当てる行数とフェッチサイズの2つのみに限定し，テーブルの行数に応じて自動的にパーティション数を設定するように実装した．この一連の自動化処理をSparkのアプリケーションとして開発した．

3つ目にデータソースであるPostgreSQLのctid列の高速なカスタムスキャンの開発である．ctid列の要素にはテーブルデータを8kiB単位に分割した際の各ブロックの識別番号が含まれているため，データ量単位でのパーティション設定にこのctid列を使用する．ただし，ctid列はHeap Only Tuple(HOT)機能で使用しているためインデックス作成することは好ましくない．そこで無駄なタプルのスキャンをスキップ可能なようにカスタムスキャンを開発した．このようなデータ量単位を識別するctid列はPostgreSQLに限ったものではなく，例えばOracleであればROWIDが存在する．PostgreSQL以外のデータソースを使用する場合には，このようなデータ量単位を識別する列に対する高速なスキャンを利用する，もしくは開発する必要がある．

4つ目にSpark SQLのctid列対応の開発を行った．Spark SQLのデフォルトのソースコードはctid列を認識できないため，JDBCRDD.scalaとJDBCRelation.scalaの2つのソースコードを追加・修正することでctid列を認識可能にした．また結果を出力する際にはこのctid列は不要であるため，クエリ処理結果のテーブルからctid列を自動的に検出して消去するようにした．これらの処理はそれぞれSparkのソースコードの修正とアプリケーションの実装で実現した．

これにより，提案するデータ仮想化システムがSparkへクエリ処理をオフロードする際に，自動的に効率の良いパーティション設定を行うことが可能となった．

4.3 クエリ処理の流れ

以下が我々の提案手法でのクエリ処理の流れである．下記の番号は図1と対応するものである．ここでDB1はデータ仮想化システムのローカルなデータベースであり，DB2は外部データソースである．

① ユーザがデータ仮想化システムに対してクエリを発行する．

② データ仮想化システムは発行されたクエリをSpark SQLのSQL文に翻訳処理し，クエリ処理で使用される全てのテーブル名を抽出する．

③ 翻訳されたSQL文とクエリ処理で使用するテーブル名を引数としてspark-submitでSparkアプリケーションを実行する．

④ Sparkアプリケーションはデータ仮想化システムに対して引数として渡されたテーブル名に紐付いたurl，ユーザ名とパスワードをDB1から取得する．

⑤ ④で取得したurl，ユーザ名とパスワードを使用してDB1やDB2のデータソースから直接テーブル情報（主キー列とその要素値の最小値と最大値，全テーブルブロック数と全行数）を取得する．

⑥ 設定変数の1パーティション当たりに割り当てる行数と各テーブルの全行数からテーブルごとのパーティション数を算出してパーティション設定を行い，DB1やDB2のデータソースからデータを取得する．

⑦ 取得したデータをもとにクエリ処理を実行する．

⑧ クエリ処理結果をデータ仮想統合システムの専用一時

テーブルに書き込む.

⑨ 専用一時テーブルから結果を読み出してユーザに結果を渡し，その後この専用一時テーブルを消去する.

これが提案する Spark オフロードによるクエリ処理の流れである．次章でこのクエリの処理速度について評価する.

5. 評価実験

5.1 評価内容

我々の提案するデータ量に基づくパーティション設定が，パーティション列の要素値に影響を受けないクエリ処理に有効であるかを評価する．評価実験のためのデータとして商品管理データベースに対するレポート作成を想定したベンチマークである TPC-H のスケール 100 のデータを使用する．従来のパーティション設定と提案手法である ctid 列に基づくパーティション設定を行った場合での処理速度の違いを検証するために，以下の 2 通りのデータソースを使用する.

- new: TPC-H (SF100) の各データを主キーの要素値の昇順で INSERT したテーブルのデータソース
- ageing: TPC-H (SF100) の注文リスト orders と注文別アイテムリスト lineitem の 2 つのデータの行をすべてランダムに入れ替え，主キーの要素値がランダムに並んだ状態で INSERT したテーブルのデータソース

PostgreSQL のような追記型のデータベースを長年稼働している場合，データベース導入時には順番に整列していた主キー列が，大量の UPDATE が実行されていることで徐々にバラバラに配置されてしまう．このようなテーブルの状態を擬似的に再現したものが ageing である．この ageing におけるデータの入れ替えは TPC-H (SF100) の全行数の 87%，全データ量の 84%の入れ替えに相当する.

この 2 種類のデータソースを使用した実験結果を比較することで，パーティション列の要素値がバラバラに配置されていることによるデータ読み取り時のランダムアクセスの発生がクエリの処理速度の低下にどれだけ影響するのかを評価する．測定は new と ageing のデータソースそれぞれ使用し，パーティション列にはインデックスの効果を最大限利用できる主キー列（複合キーの場合は 1 番目のキー列）を使用した場合と提案手法である ctid 列を使用した場合に分けて実施する．測定する際はディスク I/O の影響を取り除くため，データソース側でテーブルデータをメモリ上に載せた状態から計測を行う.

また，外部データソースの PostgreSQL との FDW を介したクエリ処理速度の比較，PostgreSQL 上のローカルテーブルに対するクエリ処理速度の比較と，Spark クラスターに使用するノード数を変えた場合でのクエリ処理速度の向上率を評価する.

今回の測定では，PostgreSQL の SQL クエリから Spark SQL の SQL クエリへの翻訳処理の対応が完了している TPC-H のクエリ Q1，Q3，Q5，Q6，Q10，Q17 を使用する．Q3，Q5，Q10 では lineitem と orders の両方が使用されており，Q1，Q6，Q17 では lineitem が使用されている.

5.2 実験環境

今回の実験では以下に挙げるサーバ上にシステムを構成し，各サーバ間のネットワークに InfiniBand を使用して実験を行う．データ仮想化システムと Spark クラスターの master は同一サーバ上に設定する．Spark クラスターの slave は同一仕様のサーバ 4 台を使用し，データ仮想化システムやデータソースとは別のサーバ上に構成する．データソースには PostgreSQL を使用する．1 台のサーバ上にデータソースを構築するが，データ仮想化システムに対して FDW 使用して外部テーブル登録を行う際に登録サーバ名をテーブルごとに設定することで，各テーブルが異なるデータソース上に存在する形式にする．表 2 は実験に使用するノード数の異なる Spark クラスターの構成と設定である．今回の Spark クラスターの設定では，各ノードの CPU コア数とメモリ量を全て処理に使用するのではなく，使用上限をリソースの半分に設定する．これは実際の現場で Spark クラスターを使用する際に，新たなサーバを導入するのではなく既存のサーバを間借りしてクラスター構築することを念頭に置いたものである.

データ仮想化システム

- 使用サーバ:
  PRIMERGY RX2540 M1
  CentOS release 6.8 (Final)
  Intel Xeon CPU E5-2660 v3 @ 2.60GHz (20 コア)
  Memory: 576GB
- PostgreSQL 9.5.4
  gcc -O2 でコンパイル
  shared buffers: 250GB
  worker memory: 8GB

Spark クラスター

- master 使用サーバ:
  データ仮想化システムと同一サーバ
- slave 使用サーバ（4 台）:
  PRIMERGY RX300 S7
  RHEL 6.3 (Santiago)
  Intel Xeon CPU E5-2680 0 @ 2.70GHz (16 コア)
  Memory: 128GB
- Spark 2.0.1
  spark.driver.memory: 24GB
  spark.executor.memory: 64GB

データソース

- 使用サーバ:
  PRIMERGY RX300 S8
  CentOS release 6.8 (Final)
  Intel Xeon CPU E5-2697 v2 @ 2.70GHz (48 コア)
  Memory: 768GB
- PostgreSQL 9.5.4
  gcc -O2 でコンパイル
  shared buffers: 250GB
  worker memory: 8GB
  SAS HDD 上にデータ格納

表 2　Spark クラスターの構成と，使用 CPU コア数と使用メモリ量の設定．() 内の値は各ノードでの使用コア数．

ノード数	使用 CPU コア数	使用可能メモリ量
1 node	8 cores	64 GB
2 nodes	16 cores (8 cores)	128 GB (64 GB)
4 nodes	32 cores (8 cores)	256 GB (64 GB)

6. 実験結果と考察

今回の評価実験では，測定の基準をそろえるためにデータソースを new として Spark クラスターに直接クエリ Q1，Q2，Q3 を発行して，クエリ処理が最も速い付近のパーティション変数を事前に調査した．その結果をもとに，主キー列を利用したパーティション設定の場合には 1 パーティション当たり 10 万行，フェッチサイズを 10 万行に設定し，ctid 列を利用したパーティション設定の場合には 1 パーティション当たり 100 万行，フェッチサイズを 10 万行に設定して実験を行った．

実験結果は表 3，表 4 と図 3，図 4，図 5，図 6 に示す通りである．表 5 には比較用のデータとして，外部データソースの PostgreSQL のテーブルに対して FDW を介してクエリ処理を行った場合と PostgreSQL 上のローカルテーブルに対してクエリ処理を行った場合での処理時間を掲載する．

表 3　Spark オフロードにおけるパーティション設定に主キー列を利用した場合でのデータソースの違いと Spark クラスターに使用するノード数ごとのクエリの処理時間．

	new（秒）			ageing（秒）		
	1 node	2 nodes	4 nodes	1 node	2 nodes	4 nodes
Q1	574	298	169	1244	641	382
Q3	289	162	100	1060	578	340
Q5	452	265	175	1252	699	442
Q6	76	43	30	405	226	138
Q10	244	155	89	1003	558	327
Q17	601	331	203	1910	1041	619

表 4　Spark オフロードにおけるパーティション設定に ctid 列を利用した場合でのデータソースの違いと Spark クラスターに使用するノード数ごとのクエリの処理時間．

	new（秒）			ageing（秒）		
	1 node	2 nodes	4 nodes	1 node	2 nodes	4 nodes
Q1	534	275	155	546	276	159
Q3	205	116	81	215	123	86
Q5	364	219	155	383	222	162
Q6	50	29	22	50	30	23
Q10	172	100	67	173	104	69
Q17	469	265	172	468	264	175

表 5　外部データソースの PostgreSQL のテーブルに対して FDW を介してクエリ処理を行った場合 (FDW) と PostgreSQL のローカルテーブルに対してクエリ処理を行った場合 (local) の処理時間．

	FDW（秒）		local（秒）	
	new	ageing	new	ageing
Q1	5362	5430	782	753
Q3	2011	2133	268	348
Q5	3233	3412	215	299
Q6	196	225	116	114
Q10	1051	1123	255	264
Q17	3017	3203	201	20

6.1　ランダムアクセスの影響評価

従来手法である主キーをパーティション列に使用したパーティション設定を行った場合と提案手法である ctid 列を使用したデータ量でのパーティション設定を行った場合で，データ読み取り時のランダムアクセスがクエリ処理時間にどの程度影響するかを評価した結果が以下である．

図 3 は Spark オフロードにおけるパーティション設定に主キー列を利用した場合と ctid 列を利用した場合のクエリの処理時間をグラフ化したものである．ここではデータソースに new と ageing を使用して Spark クラスターを 4 ノードで構成した場合のみを掲載した．

主キー列を使用したクエリ処理は従来手法においてインデックスの効果を最大限利用することの可能な処理速度が最も速い理想的な状況である．一方で，従来手法での ageing に対する処理時間は何れのクエリにおいても最も長い結果になった．この時、Q6 ではデータソースの違いで最大 4.6 倍もの処理時間の伸びが生じた．これはインデックスの効果が最大限利用できる主キー列をパーティション列に使用しても，パーティション列の要素値がバラバラに配置されていることでデータ読み取り時にランダムアクセスが発生し，処理速度が低下したものと考えられる．

次に ctid 列を使用したデータ量でのパーティション設定の場合では，new に対する処理時間は従来手法の主キー列を使用した場合と比べて若干処理速度が速く，インデックスの効果を最大限利用した場合と遜色無い処理速度が得られた．さらに提案手法は new と ageing の違いで処理速度が低下せず，Q6 では従来手法よりも最大 6 倍速いことが分かった．これはデータ量に基づくパーティション設定を行った効果が現れたものであると考える．

これにより，我々の提案するデータ量に基づくパーティション設定を採用することで，パーティション列の要素値がバラバラに配置されている場合でもランダムアクセスが発生せず，クエリの処理速度の低下を引き起こさないことが示された．さらに，従来手法で new に対してインデックスの効果を最大限利用した主キー列を使用した場合の処理速度が提案手法では ageing に対しても実現可能であることが分かった．

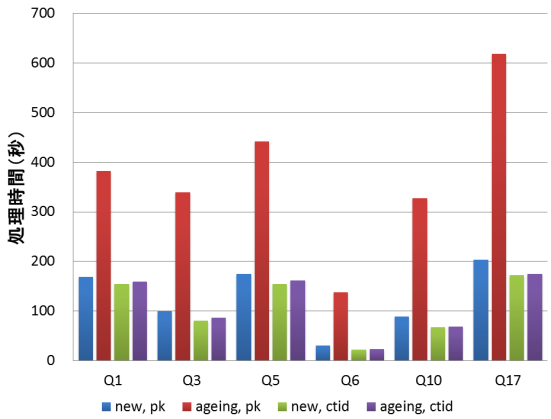

図 3　Spark オフロードにおけるパーティション分割に主キー列を利用した場合（pk と表記）と ctid 列を利用した場合（ctid と表記）のデータソース new と ageing ののでのクエリの処理時間．Spark クラスターは 4 ノード構成．

6.2　Spark クラスターのノード数変化とクエリ処理時間

図 4 は Spark オフロード時のパーティション設定に提案手法である ctid 列を利用した場合の Spark クラスターに使用するノード数の違いでのクエリ処理速度の比較図である．提案手法では new と ageing によるクエリ処理速度に違いが無いことから，ageing を使用した場合のみを掲載した．ここで 1 ノードでのクエリ処理速度を 1 としている．

提案手法を採用した場合，Q1 ではノード数増加に伴って最大約 3.4 倍クエリ処理速度が向上した．ただし，ノード数に伴うクエリ処理速度のスケールが線型ではないことが明らかになった．従来手法においても同様の傾向が確認できており，我々が主眼としているアドホックなクエリ処理に対しては特定のクエリに合わせた調整を行えないために Spark クラスターの十分な並列性能が引き出せていない可能性がある．

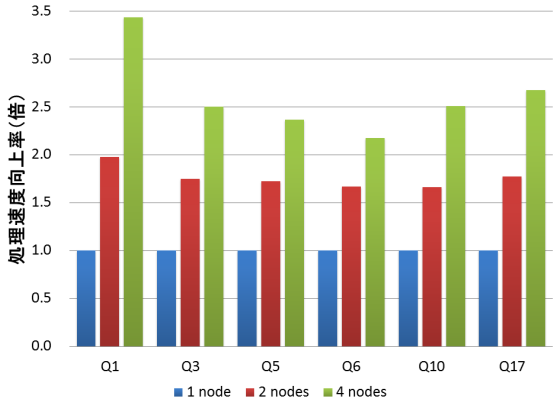

図 4　Spark オフロードにおけるパーティション設定に ctid 列を利用した場合での Spark クラスターに使用するノード数の違いでのクエリ処理速度の比較．データソースには ageing を使用．ここでは 1 ノードでのクエリ処理速度を 1 としている．

6.3　FDW を介したクエリ処理速度との比較

図 5 は外部データソースの PostgreSQL のテーブルに対して FDW を介してのクエリ処理速度と提案手法である ctid 列を使用したデータ量でのパーティション設定を行った Spark オフロードでのクエリ処理速度の比較図である．ここで FDW でのクエリ処理速度を 1 としている．データソースには ageing を使用し，Spark クラスターに 4 ノードを使用した場合を掲載してる．この時，Q1 では最大約 34 倍高速にクエリ処理可能であることが分かった．

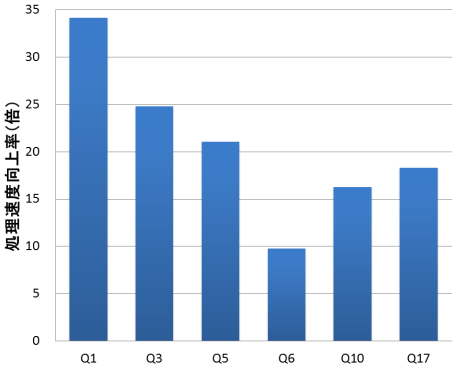

図 5　外部データソースの PostgreSQL のテーブルに対して FDW を介してのクエリ処理速度と提案手法である ctid 列を利用した場合でのクエリ処理速度の比較．ここで FDW を介したクエリ処理速度を 1 としている．データソースには ageing を使用し，Spark クラスターは 4 ノード構成である．

6.4　ローカルテーブルに対する処理速度との比較

図 6 は PostgreSQL 上のローカルテーブルに対するクエリ処理と提案手法である ctid 列を使用したデータ量でのパーティション設定による Spark オフロードでのクエリ処理速度の比較図である．ここで PostgreSQL のローカルテーブルに対するクエリ処理速度を 1 としている．データソースには ageing を使用し，Spark クラスターに 4 ノードを使用した場合を掲載している．

この時，Q6 では最大約 5 倍高速にクエリ処理可能であることが分かった．また，lineitem や orders を含む複数のテーブルに対する JOIN 操作を行う Q3，Q5，Q10 においても PostgreSQL のローカルテーブルに対するクエリ処理速度を上回る結果が得られた．ただし Q17 に関しては，new を使用した際には 4 ノードの Spark クラスターでのクエリ処理が PostgreSQL より速かったが，ageing を使用した際にクエリ処理速度が PostgreSQL の 5 分の 1 となった．この Q17 のデータソースの違いによるクエリ処理時間の大きな差の原因は PostgreSQL が選択した実行プランの違いによるものであった．表 6 は各クエリの実行プランの概要である．データソースの違いで実行プランが異なっていたのは Q17 のみであった．Q17 の実行プランでは 3 度のテーブルスキャンを行っており，そのうち TPC-H のテーブルで最大行数の lineitem を 2 度スキャンする．データソースが new の場合は 1 つはインデックススキャン，もう 1 つはシーケンシャルスキャンが計画された実行プランで，クエリに使用したテーブルの総行数の 51%に対してシーケンシャルスキャンを実行していた．一方，データソースが ageing の場合では 2 つともインデックススキャンが計画された実行プランであった．この時のクエリに使用したテーブルの総行数に対するシーケンシャル

スキャンを実行の割合は僅か 2%であった．このインデックススキャンの選択がクエリ処理速度の違いに現れたものと考える．

これらのことから，実行プランの違いに因る部分はあるものの，多くのクエリの場合で我々の提案手法が PostgreSQL のローカルテーブルに対するクエリ処理速度を上回ることが示された．

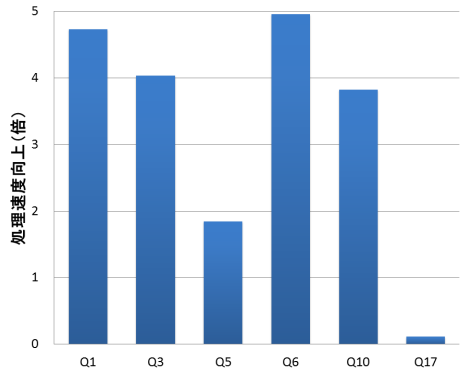

図 6　PostgreSQL のローカルテーブルに対するクエリ処理速度と提案手法である ctid 列を利用した場合でのクエリ処理速度の比較．ここでローカルテーブルに対するクエリ処理速度を 1 としている．データソースには ageing を使用し，Spark クラスターは 4 ノード構成である．

表 6　PostgreSQL のローカルテーブルに対するクエリ処理を行った際の実行プランの概要．TPC-H の natin と region の 2 つのテーブルは相対的に極めて小さいテーブルであるため実行プラン内での操作をカウントしない．Q17 は new，ageing に対してそれぞれ Q17n，Q17a と分けて表記する．

	使用テーブル数	Join 数	Index scan 数	Seq scan 数	Seq scan の割合
Q1	1	0	0	1	100%
Q3	3	2	3	0	0%
Q5	4	3	1	3	20%
Q6	1	0	1	0	0%
Q10	3	2	1	2	80%
Q17n	3	1	1	2	51%
Q17a	3	1	2	1	2%

7. おわりに

本論文では，データ仮想化システムのクエリ処理を Apache Spark によって分散実行することで高速化する手法を提案した．従来は Spark のパーティション設定をユーザが行うことでノード間にデータを分散させていたが，我々の提案手法ではデータ仮想化対象のデータソースのデータ量を元に自動的にパーティション変数を合成することで効率的な分散実行を実現した．実際に我々の提案手法の効果を実証するために PostgreSQL をベースとして実装を行い，評価実験では実際に長年稼働し大量の UPDATE が実行されているデータソースを想定したパーティション列の要素値がバラバラに配置されたデータソースを作成して実験を行った．

この結果，データソースの違いで従来手法であるパーティション列の要素値に基づくパーティション設定の場合では最大 4.6 倍のクエリ処理の速度低下が生じたが，我々の提案手法であるデータ量に基づくパーティション設定の場合ではクエリ処理の速度低下が生じず，提案手法の効果が示された．さらに，従来手法で主キー列を使用したインデックスの効果を最大限利用可能な処理速度が最も速い状況が，提案手法では常に実現可能であることが分かった．この時従来方式よりも処理速度が最大 6 倍速いことが明らかとなった．また，提案手法においても従来手法と同程度に Spark クラスターのノード数の増加に合わせてクエリ処理速度が向上することが明らかとなった．外部データソースの PostgreSQL のテーブルに対して FDW を介してのクエリ処理速度と提案手法でのクエリ処理速度を比較した場合には，提案手法が最大約 34 倍高速にクエリ処理可能であることが明らかとなった．PostgreSQL のローカルテーブルに対するクエリ処理速度と提案手法でのクエリ処理速度を比較した場合には，実行プランの違いに因る部分はあるものの，多くのクエリの場合で我々の提案手法が PostgreSQL のローカルテーブルに対するクエリ処理速度を上回ることが示された．

今回の実験では TPC-H の 6 つのクエリに限って実験を行ったが，今後は TPC-H の 22 種類のクエリすべてが実行可能にするため Spark SQL 翻訳処理を強化する予定である．

文　献

[1] “DB-ENGINES” を参照のこと．
http://db-engines.com/en/ranking

[2] Cisco Data Virtualization, http://www.cisco.com/c/ja_jp/products/analytics-automation-software/data-virtualization/index.html

[3] Apache Spark, https://spark.apache.org/

[4] J. LeFevre, R. Liu, C. Inigo, L. Paz, E. Ma, M. Castellanos, and M. Hsu, “Building the Enterprise Fabric for Big Data with Vertica and Spark Integration,” *SIGMOD ’16*, Proceedings of the 2016 International Conference on Management of Data Pages 63-75.

[5] SAP HANA Vora, http://www.sap.com/product/data-mgmt/hana-vora-hadoop.html

[6] R. Barber, M. Huras, G. Lohman, C. Mohan, R. Mueller, F. Özcan, H. Pirahesh, V. Raman, R. Sidle, O. Sidorkin, A. Storm, Y. Tian, and P. Tözun, “Wildfire: Concurrent Blazing Data Ingest and Analytics,” *SIGMOD’16*, Proceedings of the 2016 International Conference on Management of Data Pages 2077-2080.

[7] J. Ramnarayan, B. Mozafari, S. Wale, S. Menon, N. Kumar, H. Bhanawat, S. Chakraborty, Y. Mahajan, R. Mishra, and K. Bachhav, “SnappyData: Streaming, Transactions, and Interactive Analytics in a Unified Engine,” *SIGMOD’16* (Demo Paper).

[8] PostgreSQL, https://www.postgresql.org/

[9] PowerGres Plus, http://www.fujitsu.com/jp/products/software/partners/partners/powergresplus/

[10] FUJITSU Software Enterprise Postgres, http://www.fujitsu.com/jp/products/software/middleware/database/enterprisepost gres/

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

[12] M. Armbrust, R.S. Xin, C. Lian, Y. Huai, D. Liu, J.K. Bradley, X. Meng, T. Kaftan, M.J. Franklin, A. Ghodsi, and M. Zaharia, “Spark SQL: Relational Data Processing in Spark,” *SIGMOD’15*, Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data Pages 1383-1394.