DBSJ Japanese Journal
Vol. 16-J, Article No. 14, March 2018

NoSQL データソースに対する SQL クエリ処理実行の効率化

Efficient Execution of SQL Queries over NoSQL Data Sources

加藤 千裕♡ 中村 実♢ 原田 リリアン♠

Chihiro KATO Minoru NAKAMURA Lilian HARADA

様々なデータソースに大量のデータが格納される昨今，データソースを跨いだ解析は重要な課題となっており，構造化データや半構造化データ，非構造化データに跨る処理をあたかも 1 つのデータソース上の処理に見せるデータ仮想化技術が注目されている．我々はその中でも，機能性に優れ複雑なデータ処理が可能なリレーショナルデータベースシステム（**RDBMS**）を処理エンジンとした仮想化に着目した．しかし，**RDBMS** による仮想化のデータソースに半構造化データが含まれる場合は，データアクセスにおいて二つの問題が生じる．一つはスキーマが事前に定義できないこと，もう一つは仮想化によるオーバーヘッドである．そこで本研究では，特にシリアライズされたバイナリ形式の **JSON** データについて注目し，これに対する読み込み前にスキーマ定義を必要としない **SQL** の入力を実現するとともに，実行計画を変更し二段階の型変換を内部的に一段階として処理することによって，仮想化のオーバーヘッドを削減し処理を効率化する方法を提案する．オープンソースの **RDBMS** である **PostgreSQL** を処理エンジンとして用い，ドキュメントデータベースの **MongoDB** に格納されたデータに対し，提案手法を適用し実機による効果検証を行った．

Nowadays, huge amounts of data are stored in various data sources, and analysis spread over those multiple data sources has become a big issue. Therefore, data virtualization that allows the seamless access of different data stores, as if there was a single data source, has attracted much attention recently. In this work we use a relational database management system (RDBMS) as the processing engine for data virtualization because of its rich set of functionalities and complex processing capabilities. Virtual access of data sources containing semi-structured data through RDBMS presents two major problems. First, data schema cannot be pre-defined for semi-structured data; second, data virtualization can cause significant data conversion overhead. Here we address these problems for serialized binary JSON data. Our proposal realizes SQL processing without predefined schema, and reduces the virtualization overhead by modifying the query execution plan to replace the conventional two-stage type conversion with an internal one-stage type conversion. We implemented the proposed virtualization approach using PostgreSQL as the processing engine to access a MongoDB document database as the external data source, and present some evaluation results to confirm its effectiveness.

♡ 正会員　株式会社富士通研究所
kato-chihiro@jp.fujitsu.com
♢ 正会員　株式会社富士通研究所
nminoru@jp.fujitsu.com
♠ 正会員　株式会社富士通研究所
harada.lilian@jp.fujitsu.com

1. はじめに

様々なデータが分析に用いられ格納される昨今においては，その利活用に重要な役割を果たすデータベースシステム技術が重要視され，多様なデータベースが登場，発展してきた．いち早く台頭したリレーショナルデータベースシステム（RDBMS）は，SQL という言語を用いてデータの処理を記述するもので，構造を一意に定める必要があるが複雑な分析処理を得意とする．これに対するものとして，キーバリューストアやドキュメントデータベースと呼ばれるデータベースシステムも近年盛んに開発されており，複雑な処理に関する速度や機能面では RDBMS に劣るものの，柔軟な非構造化データおよび半構造化データを許して格納することができるため，注目を集めている．これらのデータベースシステムは NoSQL と呼ばれ，中には SQL に近い言語が使える場合もあるが，多くは SQL を用いない記法により処理を記述する．

このようなデータベースシステムの多様化に従って，これらの使い分けは大きな課題となりつつある．管理の方法によっては様々なデータベースシステムの中にデータが散逸してしまい，分析が困難になる場合がある．そのため，構造化データや半構造化データ，非構造化データに跨る分析をあたかも一つのデータソース上の処理に見せる，データ仮想化は重要な技術の一つと言える．その中でも RDBMS の処理エンジンを利用した仮想化は，RDBMS の複雑なデータ処理機能などの機能性を十分に生かし細かい分析を行うことができる上，RDBMS のユーザにとってはこれまで培ってきたデータベース管理のノウハウをそのまま流用できるという利点があるため，本論文においてはデータ仮想化の手段として着目した．

しかし RDBMS は，本来はスキーマが静的に決定された構造化データを扱うデータベースである．そのため，RDBMS による仮想化のデータソースに半構造化データを扱うデータソースが含まれる場合は，データアクセスにおいて二つの問題が生じる．一つは半構造化データのスキーマが事前に定義できないこと，もう一つは仮想化によるオーバーヘッドである．そこで本研究では，まず RDBMS 上で SQL 入力前にスキーマを定義しない問い合わせを行うため，データソースからの読み込みを一度文字列型データに変換して RDBMS に認識させる．そのうえで仮想化のオーバーヘッドに対し，特にシリアライズされたバイナリ形式の JSON データを扱う場合に着目し，最終的に必要な型への変換処理の削減を提案する．具体的には，RDBMS によって生成された実行計画の，文字列を経由する型変換を連続した処理に書き換えて纏め，文字列型への変換とその解析を内部的にスキップし直接バイナリから目的の型へと変換することでオーバーヘッドを削減する．

この提案手法の有効性を確かめるため，検証用としてオープンソースの RDBMS である PostgreSQL をクエリ処理エンジンに用い，BSON 形式 [1] のデータを取り扱う MongoDB をデータソースとし，実際にデータ仮想化を行いクエリを実行して検証する．なお BSON は Binary JSON の略で，シリアライズされたバイナリ形式の JSON データであり，キーとフィールドの組み合わせに加えて型情報（整数型，浮動小数点型，文字列型，論理型など）を保持している MongoDB 独自の型を指す．これらの検証

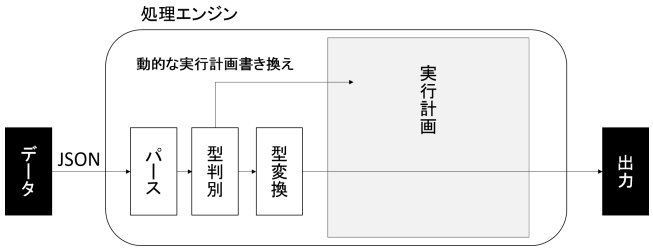

図 1: 動的な計画変更を行う JSON データ処理機構

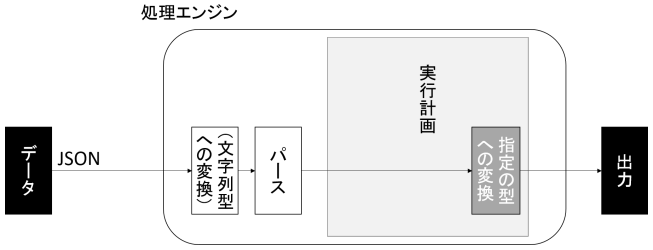

図 2: スキーマを事前に決定しない JSON データ処理機構 (RDBMS)

用データベースを用い，まず提案手法を適用した場合と適用しなかった場合について計測し，提案手法の性能向上を確認する．そのうえで，MongoDB における直接実行との比較を行い，データ仮想化の有用性を示す．最後に既存の MongoDB のデータ仮想化機構と性能を比較し，本提案を適用したデータ仮想化機構の実用性について検証する．

本論文の構成は次のとおりである．まず第章において関連研究について紹介する．第章で提案手法について述べ，第章にて検証環境と検証方法を示し，第章において実際のクエリを用いた検証を行い，この結果を考察する．最後に第章に本論文のまとめと今後の課題を示す．

2. 関連研究

2.1 半構造化データのスキーマと型決定

従来の RDBMS では，半構造化データを扱う際には事前にスキーマを定義する必要があり，これは以前から様々な研究が行われてきた [2][3][4]．しかし半構造化データは元々一定のスキーマを持たないものであり，形式に反しない限りはどのようなデータであっても格納・検索できることが求められる．クエリ実行前にスキーマを定義することは，半構造化データの読み込み方法として最善であるとは言い難い．本来は問い合わせを実行する際に格納するデータを読むことで初めてそのスキーマを決定する方法を用いるのが適切である．このため半構造化データを扱う場合にはこの要件を満たす NoSQL データベースに格納し処理を行うのが一般的であった．

近年は，SQL ライクな言語を用いながらもクエリ実行前にスキーマを定義せずにデータを読むことができる処理機構が多く登場している．一つは Norikra[5] と呼ばれるストリーム処理エンジンである．これは最初の入力を読み込んだ時にそのデータを見てスキーマやデータ型を判定し，動的に計画を変更する．そして以降のデータに関しては同じスキーマであると仮定して処理を進める．また，Apache Drill[6] は全てのデータに関してスキーマやデータ型の判定を行い，動的に対応している．これらの方法は図 1 のように表せる．

一方 RDBMS にも事前にデータ内部のスキーマを定めない状態での半構造化データへのアクセスをサポートする動きがある．Oracle Database 12c は JSON が一つ入っているテーブルを定義させることで，JSON 内部のスキーマを定義しない状態でのアクセスを実現している [7]．しかし返り値は文字列であり，ユーザが必要に応じて型変換を指定することが求められる．これは，RDBMS のプランナーは実行計画やその中で使われる詳細な型までを実行前に一意に決定しなければならず，実行開始してから動的に変更することが出来ない，という制約の影響である．JSON 内部のスキーマを事前に定めない場合，実行前に読み込むデータの型を決定することは難しいため，最初に文字列のような一定の形式でデータを読み込む必要がある．この方法は図 2 のように表され，入力の JSON がシリアライズされたバイナリ形式の場合にはバイナリを文字列に変換した JSON データを更に解析しパース及び型変換する必要があるため，バイナリ形式の持つ情報は用いられずオーバーヘッドが大きくなる．この JSON データのパースに関する研究は近年も盛んにおこなわれているが [8][9]，本論文においては，JSON データのパースなしにバイナリから直接型変換する方法を考える．

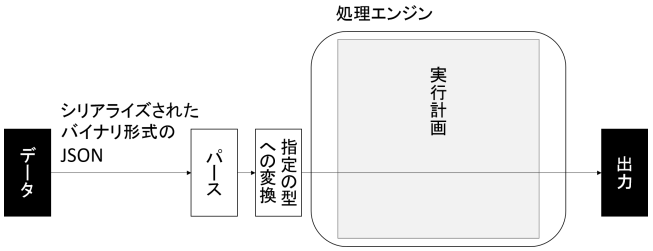

図 3: 本提案の JSON データ処理機構

具体的には，シリアライズされたバイナリ形式の JSON データの読み込みを行う際，動的な実行計画の変更は行わず，見かけ上は文字列に変換して読み込みを行いながらも，ユーザの型変換指定があった場合に二段階になってしまう型変換を内部的に纏める．そしてバイナリの情報を利用してパースし目的の型へと一段階に変換することで，図 3 のように実行方法を書き換えて JSON 解析の処理を省き高速化を図る．

2.2 データソース仮想化

NoSQL なデータベースは様々なデータに柔軟に対応しうるが，一方で分析の機能面では RDBMS に劣る点が存在する．また，大量のデータが用いられる現在では，単一のデータベースに全てのデータが入っているという状況は少なくなっており，複数のデータソースからデータを読み出し，複雑な分析を行う場合がある．そのため，RDBMS から SQL ライクな方法で NoSQL のデータを読み，分析することは重要な課題の一つであると言える．

この課題は過去にも様々な研究がなされている．2013 年にはドキュメント DB の一つである MongoDB と PostgreSQL を接続し，データを読むだけでなく，MongoDB へのプッシュダウンや一時テーブルの作成による最適化を行った論文が発表されている [10]．また，2014 年には，様々なデータソースを繋ぎ，クエリを解析し，特に Join のプッシュダウンに力を入れて速度の最適化を試みたクエリ実行エンジンが発表されている [11]．オープンソースのプロダクトとして，Teiid というデータ仮想統合ソフトも存在し，製品化もなされている [12]．

しかし，このような既存の研究は RDBMS 上で JSON 内部のスキーマを決定するかどうかはあまり考慮されていない．そのためプッシュダウンを行って NoSQL のデータベースをスキャンする際，読むフィールドの限定は行っているが，その型については特に触れていない．実際にはスキーマを事前に定めずにシリアライズされたバイナリ形式の JSON 型を読む場合は，バイナリから目的の型へ直接変換することでバイナリの情報を利用して読み込みデータへの処理を減らすことができ，オーバーヘッドが削減できると考えられる．そこで本論文では型変換を一括化した読み込み方法を提案する．

```
CREATE FOREIGN TABLE t1 { doc json };
```

図 4: JSON 内部の構造を事前決定しないスキーマ定義例

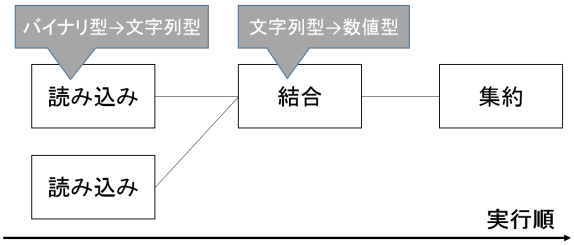

図 5: 従来の実行計画における型変換

3. 提案手法

3.1 半構造化データの内部構造を決定しないスキーマ定義

半構造化データは一定のスキーマを持たず，様々なフィールドを持つ．これに対応するために，まず RDBMS 上で外部のデータに対し，事前に JSON 内部のスキーマを定義せずにデータアクセスを行う必要がある．本提案においては，シリアライズされたバイナリ形式の JSON データを一度文字列型として読み込み，これを RDBMS 側において解析し，欲しいフィールドの情報を読む，単純な機構を実装した．これにより，ユーザは事前にスキーマを定めずとも，図 4 のような記法で「JSON 形式の文字列データが一つ入ったテーブル」という定義をすることで中の情報を自由に読み込むことができる．なお本論文は仮想化によるデータ解析を目的にしているため，挿入や更新，削除クエリに関しては考慮していない．

この方法は一度文字列の形に変換しなくてはならないため，図 2 の例と同様にバイナリから文字列，文字列から JSON 解析による目的のデータ抽出と型変換，という二度の変換の手間を踏まなくてはならない．そこでこの問題を解消すべく，クエリ実行前に JSON 内部のスキーマの定義を必要としないという前提は保持したままに，文字列への変換を無くす方法を次項において提案する．

3.2 実行計画の変更による内部的な型変換の一括処理

RDBMS における計画は木構造の形で表される場合が多い．それぞれのノードが読み込みや集約，ソート，表結合などの動作を表す．外部データソースからの読み込みは木構造の末端にあり，節の方法を用いた場合，まずこのノードにはバイナリを文字列に変換したデータが読み込まれ，その後実行計画の中でユーザの求める型への変換処理が行われる．このユーザ指定の型変換は読み込みノードで必ず行われるわけではなく，文字列型のまま次のノードへ渡されてソートや絞り込みが行われ，結合の条件比較など，型変換が必要になった場面で初めて型変換が実行される．そこで我々は別々のノードにある型変換を一つに纏めるために，RDBMS によって計画の木構造が作られた後，それを解析し，外部データソース読み込み処理ノードの中で型変換が行われるよう計画を変更した．その上で，二段階の型変換の代わりにバイナリから目的の型への直接の型変換を行い，次のノードへと渡す．但し書き換えが難しい一部のノードに関しては今回は非対応としている．

以上の流れを図として表したのが図 5 と図 6 である．本来は図 5 のように結合処理の条件式などのために必要になったときに初めて行われる文字列から特定の型への変換処理を，全て読み込み時に行われるよう計画を変更し，図 6 に示すように，計画内における二段階の型変換を一か所に集める．そして実際の読み込み時にはこの二段階の型変換について，データソースからの入力型と最終的な出力型のみを参照し，図 3 のように文字列への変換を経ずに型変換を行う．これにより，ユーザが定義したスキーマ上は一度文字列として読み込んだ JSON データを解析し，必要なデータを取り出して指定の型へキャストしているように見えるが，実際には文字列に変換する段階をスキップし，バイナリ形式の情報

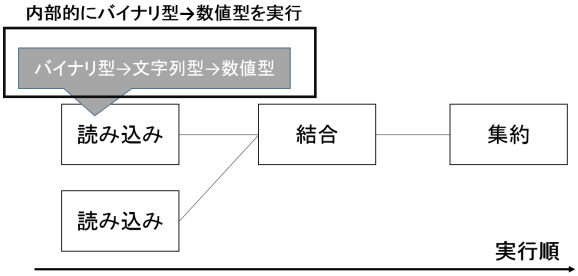

図 6: 型変換のオーバーヘッド削減のため変更された実行計画

表 1: 実験環境

サーバ	Primergy RX300 S6
プロセッサ	Xeon X5675 3.06GHz 6-core x2
メモリ	DDR3 16GB (1066 RDIMM) x12 (192GB)
ストレージ	HDD 900GB x4 (RAID 0, Partition 1)
OS	CentOS Linux release 7.2 (64bit)
データベースシステム	PostgreSQL 9.5.4 (buffer size 24GB) MongoDB 3.2
ワークロード	TPC-H SF10

を利用したパースと一回の型変換によって処理を完了することになる．よって，クエリ実行前に JSON 内部のスキーマの定義を必要としないまま読み込みのオーバーヘッドを削減することができると考えられる．

4. 検証

4.1 実験環境

マシンや OS などの設定を表 1 に示す．今回，比較に用いたワークロードは TPC-H であり，そのスケールファクターは 10 とした．この実際のサイズはおよそ 10GB ほどである．シリアライズされたバイナリ形式の JSON データソースとして MongoDB を用い，クエリ処理エンジンとして PostgreSQL を用いた．MongoDB はチューニング等は行っておらず，TPC-H の表一つ一つをそれぞれ別々のコレクションに挿入する形でデータを格納し，主キーにのみインデックスを張った．PostgreSQL には実データは格納していない．PostgreSQL はバッファサイズを 24GB とした．これはスケールファクター 10 のデータを全て載せることができる大きさである．また，PostgreSQL から MongoDB への接続と提案手法の適用は全て Foreign Data Wrapper（FDW）として実装した．このため PostgreSQL と MongoDB のソースコードには手を加えておらず，PostgreSQL のエクステンションの形で再コンパイルなしに適用し，使うことができる．

4.2 使用した手法

本検証では，以下の方法において実行，計測し，性能比較を行った．

まず一つは節に記載した単純な機構を実装した FDW である．これは BSON 形式のデータを全て文字列として取得し，処理を行う．以下この FDW は SchemaOnReadFDW と記載する．また，既存手法として，必要なデータのみを取ってくるようにプロジェクションと単純な WHERE 句のフィルタのプッシュダウンを行う手法を SchemaOnReadFDW に適用したものについても計測を行う．以下この FDW は SchemaOnReadFDW+PP と記載する．この SchemaOnReadFDW+PP に本提案手法を適用したものを ProposeFDW とし，前述の二つの FDW との比較，検証を行う．

更に Enterprise 社がオープンソースとして公開している MongoDB Foreign Data Wrapper[13] を用いて PostgreSQL においてクエリを実行する方法（MongoFDW）についての計測も行っ

```
CREATE FOREIGN TABLE lineitem {
  li json
} SERVER xxx;

SELECT
  (li->>'l_orderkey')::integer,
  sum((li->>'l_quantity')::real) AS sum_qty
WHERE
  (li->>'l_shipdate')::date <= date'1998-12-01'
GROUP BY (li->>'l_orderkey')::integer
```

図 7: JSON 内部を事前定義しない場合のクエリ例

```
CREATE FOREIGN TABLE lineitem {
  l_orderkey integer,
  l_linenumber integer,
  l_quantity real,
  ...
} SERVER xxx;

SELECT
  l_orderkey,
  sum(l_quantity) AS sum_qty
WHERE l_shipdate <= date'1998-12-01'
GROUP BY l_orderkey
```

図 8: 事前にスキーマを定義するクエリ例

た．この MongoFDW は MongoDB へのプッシュダウンは単純な WHERE 句にのみ適用されており，必要なフィールドに関わらず条件に合うデータ全てをロードする．スキーマに関しては事前に CREATE FOREIGN TABLE の段階でデータ型含め細かく指定する必要があるため，フィールドの増減に応じてテーブル定義からやり直さなければならないが，代わりに事前に指定したスキーマの情報を用いて BSON 型からの直接変換を行うことができる．

また，比較対象として，MongoDB に直接クエリを投げた場合についても計測を行った．MongoDB から PostgreSQL へのデータ転送がないため，その部分のオーバーヘッドはゼロになるが，代わりに集約と結合を MongoDB の処理エンジンで行うため低速になる可能性がある．

なお，PostgreSQL から ProposeFDW 及び SchemaOnReadFDW，SchemaOnReadFDW-PP を介して実行するクエリは，図 7 のように文字列以外の型全てについてユーザが型を指定する形で問い合わせを行っている．一方 MongoFDW を介したクエリは，図 8 のように全てのスキーマを最初に定義したのち，問い合わせを行う．MongoDB に直接投げるクエリは MongoDB の記法に従う．

本実験では，まず提案手法の効果を確認するため，ProposeFDW と，SchemaOnReadFDW+PP，SchemaOnReadFDW の三つについて比較を行う．そして ProposeFDW と MongoDB による直接実行と提案手法を比較し，RDBMS によるデータ仮想化が複雑なクエリ処理において直接実行より有効であるかを確認する．最後に，既存の MongoFDW と性能比較を行い，考察する．

5. 検証結果・考察

5.1 JSON 内部を事前定義しない読み込みの性能比較

表 2 に TPC-H ワークロードのクエリ 1 から 10 に対して PostgreSQL から JSON 内部スキーマを事前定義せずに読み込みを行う三つの方式を実行したときの実行時間を示す．但しクエリ 2 は長時間停止しなかったため，未計測とした．なお，スペースの関係上 SchemaOnRead を SOR と記載する．また，図 9 は，表 2 の SchemaOnReadFDW の速度を 1 として三つの方式の実行速

表 2: TPC-H のクエリ実行時間

クエリ	SORFDW (s)	SORFDW-PP (s)	ProposeFDW (s)
1	6,547.6	1,937.7	339.5
3	4,026.2	1,231.3	259.0
4	5,541.6	757.4	178.9
5	4,682.2	1,129.5	235.0
6	2,554.6	175.1	82.3
7	6,934.7	1,487.5	288.1
8	9,176.8	1,489.3	1,060.2
9	10,000.9	2,685.6	1,124.0
10	3,289.4	479.4	126.8

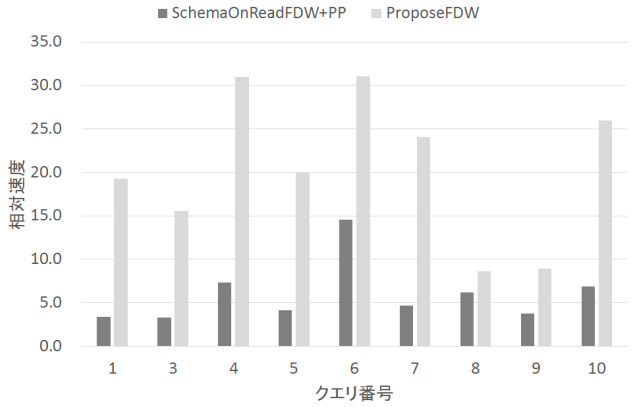

図 9: SchemaOnReadFDW を基準とした ProposeFDW の相対速度

度を比較したグラフである．横軸はクエリ番号，縦軸は相対的な速度を表す．

何も手を加えていない SchemaOnReadFDW と比較すると，既存手法であるプロジェクションとフィルタのプッシュダウンのみを適用した SchemaOnReadFDW+PP も大きく実行速度が上がっていることが確認できる．更にこれに加えて型変換の一括化を行った ProposeFDW は最低でも 8.7 倍，幾何平均 18.7 倍の性能向上が見られた．

また，表 2 を元に SchemaOnReadFDW+PP の速度を 1 として ProposeFDW と比較したのが図 10 である．クエリ 1，3，4，5，7，10 は 4 倍近くかそれ以上に性能が上がっている．型変換を一括処理したことで，特に文字列を JSON として PostgreSQL が解析して目的の型に変換する処理がなくなったことによるオーバーヘッドの減少が効いた結果，このように性能が向上したと考えられる．

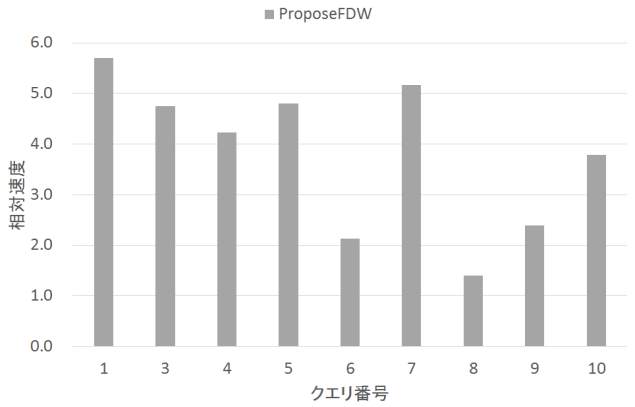

図 10: SchemaOnReadFDW+PP を基準とした ProposeFDW の相対速度

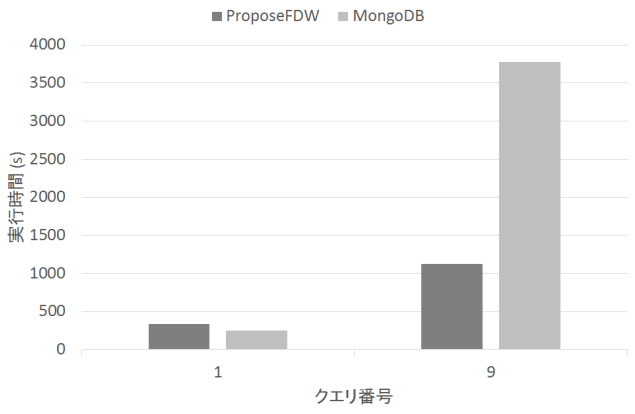

図 11: TPC-H の MongoDB による直接実行結果

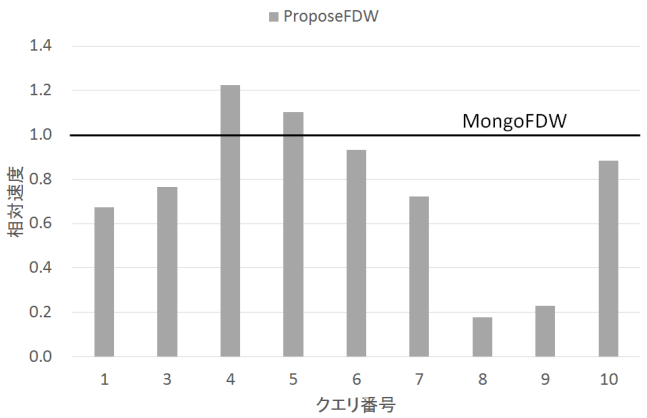

図 12: MongoFDW を基準とした ProposeFDW の相対速度

一方クエリ 6，8，9 は 2 倍弱から 2 倍未満の速度上昇に留まった．クエリ 8，9 に関しては ProposeFDW を介した場合でも，MongoDB からのデータ送信が止まった後に PostgreSQL のみが動いている時間が見られ，PostgreSQL 側の処理が追いついていないことが観測された．この原因としては，Analyze を行っていないため，PostgreSQL 側が生成した実行計画そのものが処理が重い計画であった可能性があり，その結果，PostgreSQL 側の処理が大きなオーバーヘッドとして残ってしまったことが考えられる．クエリ 8，9 は大小多数の表結合を行うクエリであり，結合方法や順序によっては PostgreSQL 側の処理が非常に重くなるため，このような結果が生じたと思われる．また，ProposeFDW を介したクエリ 6 の実行中には，MongoDB 側の CPU 使用率が 75%まで上昇していることを観測した．他のクエリは最大で 50%弱であったことから，クエリ 6 は MongoDB 側のデータ読み込みがオーバーヘッドとなった可能性が考えられる．そのため PostgreSQL 側の処理を提案手法によって軽減しても大きな性能向上が得られなかったと思われる．クエリ 6 は一つの表全体を読み込み，集計を行うという単純なクエリであるため，PostgreSQL 側の処理がもともと軽かったことが，このような MongoDB 側のオーバーヘッド発生につながったと考えられる．

これらの結果から，クエリによって差はあるものの，型変換の効率化は実行性能を高めるために有用であると言える．Analyze に対応し実行計画そのものを改善することにより，更に性能向上を見込める可能性があるため，今後の課題としたい．

5.2 MongoDB における直接実行との比較

図 11 は提案手法を用いた ProposeFDW と MongoDB における直接実行の実行時間の比較である．横軸は TPC-H のクエリ番号，縦軸は実行にかかった時間を秒数で表す．なお，MongoDB において複雑なクエリの実行は非常に難しく，全ての TPC-H のクエリを変換することが困難であるため，代表として二つのクエリについての結果を記載する．クエリ 1 の場合，MongoDB と比較すると提案手法の速度は 0.72 倍と，MongoDB の方が優位な性能を出している．しかし一方で，クエリ 9 に関しては ProposeFDW は 1124 秒で止まったのに対し，MongoDB での実行は 3776 秒と，ProposeFDW が 3.6 倍早いという結果になった．これは，MongoDB に実装されている表結合方式の問題であると考えられる．MongoDB は表結合方式としてネステッドループ結合のみをサポートしており，ハッシュ結合やマージ結合を行うことができない．そのため，これらが最適と考えられるクエリに関しては著しく性能が低下する．クエリ 1 は表結合を必要としないクエリであったため，MongoDB による直接実行は PostgreSQL にデータを送らなくてよいという利点が発揮され，ProposeFDW よりもよい性能を出すことができたと考えられる．しかしクエリ 9 は逆に 6 個の表を結合するクエリであり，PostgreSQL の EXPLAIN コマンドを用いると，その全てにおいてハッシュ結合かマージ結合が選択されている．このようなクエリにおいては，ハッシュ結合とマージ結合が選択できずネステッドループ結合を行わなければならない MongoDB での実行は効率が悪く，性能が大きく低下する原因になったと考えられる．この結果により，ProposeFDW によってデータを読み込み PostgreSQL のクエリ処理エンジンで処理することは，MongoDB に格納された BSON 形式のデータに対して複雑な結合処理を行う場合には有用であるということが示された．

5.3 既存の MongoFDW との比較

図 12 に，事前にスキーマを細かく定める既存の MongoFDW と，本提案手法を採用した ProposeFDW の実行速度の比較を示す．横軸は TPC-H のクエリ番号，縦軸は既存の MongoFDW の実行速度を 1 としたときの ProposeFDW の相対的な実行速度を表す．結果に示されるように，速度はクエリにより大きく違う．クエリ 8 や 9 は 0.2 倍程度の性能しか出ていない一方，クエリ 4 や 5 は既存の MongoFDW を上回る性能を記録している．但し節と同様，クエリ 2 に関しては未計測である．

この原因は MongoFDW と ProposeFDW の計画が違うことに起因すると考えられる．提案手法は RDBMS がクエリを解析し，実行計画を得た後，それを読み計画の変更を行うものである．しかし，そもそも MongoFDW を用いた時と ProposeFDW を用いた時は，PostgreSQL のオプティマイザーが出力する実行計画に差異が生じていた．図 7 と図 8 を見比べると明らかであるように，ProposeFDW を用いて実行するクエリは，MongoFDW を用いて実行するクエリと比較してキャスト演算子などが追加されている上，PostgreSQL から見えるテーブル定義も全く違うものである．これらの差異が原因となって実行計画に差が生じ，その結果，単純に比較が難しい結果になったと考えられる．同一の実行計画による性能比較は今後の課題である．

但し MongoFDW を上回る性能を発揮したクエリがあったことから，提案手法でも十分な性能を得られる可能性があることは示唆されている．

6. おわりに

様々なデータソースにデータが散逸する昨今において，データの仮想化と一括管理は重要な課題である．本論文は，この解決方法として既存の RDBMS をクエリ処理エンジンとして用いるデータ仮想化に注目した．そして RDBMS が半構造化データを用いているデータソースからデータを読み込む場合，実行前に JSON 内部のスキーマを定義しないことが好ましいと述べ，シリアライズされたバイナリ形式 JSON をこの方式で読み込む際に生じる二段階の型変換と冗長な解析について，RDBMS によって作成された計画を変更し，内部的に一段階の型変換に纏めて JSON 文字列の解析をスキップし，オーバーヘッドを削減する方法を提案した．オープンソースの RDBMS である PostgreSQL と，BSON 形式

のデータを扱う MongoDB を用いて検証を行った結果，提案手法により大きく性能向上するという結果を得た．また，MongoDB による直接実行と比較した場合には，結合を多用するような複雑なクエリにおいて提案手法が明らかに優位であるとの結果を得た．既存の MongoFDW との比較においては，クエリやスキーマの違いから同一の実行計画を得られず十分な比較とは言えないが，MongoFDW を上回る性能を発揮するクエリが存在することを確認した．

今後は，Analyze への対応，集約関数のプッシュダウンなどを行いさらなる性能向上を図るとともに，半構造化データ向けのベンチマークによる測定，挿入・削除・更新への対応，同一の実行計画による MongoFDW との比較などを行っていく．

[文献]

[1] “BSON – Binary JSON”. http://bsonspec.org/.

[2] Alin Deutsch, Mary Fernandez and Dan Suciu. “Storing semistructured data with STORED”. In *ACM SIGMOD Record*, Vol. 28, pp. 431–442. ACM, 1999.

[3] Daniela Florescu and Donald Kossmann. *“A performance evaluation of alternative mapping schemes for storing XML data in a relational database”*. PhD thesis, INRIA, 1999.

[4] Philip Bohannon, Juliana Freire, Prasan Roy and Jérôme Siméon. “From XML schema to relations: A cost-based approach to XML storage”. In *Proceedings of the 18th International Conference on Data Engineering*, pp. 64–75. IEEE, 2002.

[5] “Norikra: Stream processing with SQL for everybody”. http://norikra.github.io/index.html.

[6] Michael Hausenblas and Jacques Nadeau. “Apache drill: interactive ad-hoc analysis at scale”. *Big Data*, Vol. 1, No. 2, pp. 100–104, 2013.

[7] Zhen Hua Liu, Beda Hammerschmidt and Doug McMahon. “JSON data management: supporting schema-less development in RDBMS”. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pp. 1247–1258. ACM, 2014.

[8] Daniele Bonetta and Matthias Brantner. “FAD. js: fast JSON data access using JIT-based speculative optimizations”. *Proceedings of the VLDB Endowment*, Vol. 10, No. 12, pp. 1778–1789, 2017.

[9] Yinan Li, Nikos R Katsipoulakis, Badrish Chandramouli, Jonathan Goldstein and Donald Kossmann. “Mison: a fast JSON parser for data analytics”. *Proceedings of the VLDB Endowment*, Vol. 10, No. 10, pp. 1118–1129, 2017.

[10] John Roijackers and George HL Fletcher. “On bridging relational and document-centric data stores”. In *British National Conference on Databases*, pp. 135–148. Springer, 2013.

[11] Ramon Lawrence. “Integration and virtualization of relational SQL and NoSQL systems including MySQL and MongoDB”. In *Computational Science and Computational Intelligence (CSCI), 2014 International Conference on*, Vol. 1, pp. 285–290. IEEE, 2014.

[12] “Teiid Homepage. Teiid”. http://teiid.jboss.org/.

[13] “PostgreSQL foreign data wrapper for MongoDB”. https://github.com/EnterpriseDB/mongo_fdw.

加藤 千裕 Chihiro KATO
株式会社富士通研究所．2016 東京大学大学院情報理工学系研究科電子情報学専攻修士課程修了．データベースシステムの研究に従事．日本データベース学会会員．

中村 実 Minoru NAKAMURA
株式会社富士通研究所．2001 東京大学大学院工学研究科情報工学科修士課程修了．コンパイラ，仮想マシン，データベースシステムの研究に従事．情報処理学会，日本データベース学会会員．

原田 リリアン Lilian HARADA
株式会社富士通研究所．1990 東京大学大学院工学研究科電子工学専攻博士課程修了，工学博士．データベースシステム，データマイニング，データ工学の研究に従事．情報処理学会，日本データベース学会会員．