DEIM Forum 2017 H4-2

RTA: 公開型テーブルへの直接問い合わせ機構の提案

村上 柊† 小坂 祐介† 五嶋 研人† 遠山元道††

† 慶應義塾大学理工学部情報工学科 〒 223–8522 神奈川県横浜市港北区日吉 3–14–1

E-mail: †{shu,kosaka,goto}@db.ics.keio.ac.jp, ††toyama@ics.keio.ac.jp

あらまし 現在、気象情報、郵便番号，その他統計データ等の様々なオープンデータが各機関から提供されており，それらの提供方法として CSV，XML などのダウンロード形式、Web サービスの API による提供，LOD に基づく RDF による提供などがある．しかし，利用者側が自身の持つ RDB と合わせてそれらのデータを二次利用する際には，利便性が低いのが現状である．そこで本論文では，そのようなオープンデータに対して加工せずに，RDB のまま利用できるアーキテクチャ（RTA: Remote Table Access）を提案する．通常であれば 1 週間に 1 回，1ヶ月に 1 回等しか更新されないオープンデータなどであっても，RDB のまま利用可能になることによって常に最新のデータにアクセスすることが可能になる．またこのアーキテクチャを利用する事によって，自身の持つ RDB 内のテーブル，複数の公開型テーブルなどと合わせて 1 つの SQL 内で処理を行う事も可能になる．また，本論文は研究の初期段階の提案論文である．

キーワード federated database，分散データベース，オープンデータ，SQL

1. はじめに

現在、気象情報、郵便番号，株価その他統計データ等の様々なオープンデータが各機関から提供されており，それらの提供方法として CSV，XML などのダウンロード形式、Web サービスの API による提供，LOD に基づく RDF による提供などがある．しかし，利用者側が自身の持つ RDB と合わせてそれらのデータを二次利用する際には，利便性が低いのが現状である．

そこで本論文では，そのようなオープンデータに対して加工せずに，RDB のまま利用できるアーキテクチャ（RTA: Remote Table Access）を提案する．通常であれば 1 週間に 1 回，1ヶ月に 1 回等しか更新されないオープンデータなどであっても，RDB のまま利用可能になることによって常に最新のデータにアクセスすることが可能になる．またこのアーキテクチャを利用する事によって，自身の持つ RDB 内のテーブル，複数の公開型テーブルなどと合わせて 1 つの SQL 内で処理を行う事も可能になる．また，本論文は研究の初期段階の提案論文である．

以下，本稿の構成を示す．2 章では現在のオープンデータの概要，問題点について述べ，3 章では関連技術・関連研究について述べる．4 章では RTA の概要，利用例を述べ，5 章では RTA の具体的なアーキテクチャ，クエリ分解について述べる．そして 6 章では評価について述べ，7 章で結論を述べる．

2. オープンデータ

2.1 オープンデータの概要

この章では，オープンデータの概要と現在のオープンデータ利用の問題点について述べる．オープンデータとは，政府，民間企業，個人などがそれぞれの保持するデータを，原則として二次利用を妨げないライセンスを基本として全ての人が利用できるように公開されたデータのことである．具体的には気象情報，郵便番号，株価など多岐に渡って公開されている．

ユーザーがオープンデータを二次利用する際には主に 3 つの方法がある．1 つ目は，ダウンロード形式であればそのファイルを一度ダウンロードし，自身の DBMS にテーブルを作成しデータを挿入してから利用する方法．2 つ目は．提供機関の提供する API を利用する方法．3 つ目は，第 3. 章で述べる，既存リモートアクセス技術を用いる方法である．

2.2 オープンデータ利用の問題点

オープンデータの現在の問題点として，前節で述べた 3 つの方法それぞれに以下のような問題点がある．

ダウンロード利用

- 一度 CSV 等をダウンロードし、自分のデータベースに挿入しなければいけない手間.
- 頻繁に更新されるデータ（毎日の気象情報、株価データ等）では、データが更新されるたびに挿入しなければいけないという手間.

API 利用

- 追加でプログラムを記述する手間.
- 予め提供者側の想定する形でしかデータが取得できないため，柔軟性に欠ける．

既存リモートアクセス技術利用

- 利用者側のテーブルと JOIN 出来ないものがある，異種 DBMS 間で利用できない等の問題点（詳細は第 3. 章，第 6. 章で後述）．

このような問題点を解決するために，本研究では RTA システムを開発した．RTA ではデータ提供者側が Web アプリケーションを通じて公開データを RDB のまま登録，利用者側はそれをまるで自身の DBMS 内に存在するかのように SQL を記述することによって，上記の問題点を解決して利用者側，提供者側双方にとって円滑なオープンデータ利用を可能にしている．

2.3 IoT への応用

近年，IoT 技術の発展によって，様々な分野の莫大な量のストリームデータが日々蓄積されている．それらのデータは通常 NoSQL データベースに格納されている．そして集積したデー

タを利用する際には，そのストリームデータから有用なデータのみを選択して関係データベースに格納してから利用するのが一般的である．この段階で RTA を用いることによって，データ集積者のみではなくより幅広い人々がそれらの有益なデータを利用することが出来るようになるため，近年の需要に合った機構であると言える．

3. 関連技術・関連研究

3.1 関連技術

リモートのデータソースに対するアクセス手法は様々な企業などによって研究，開発が行なわれている．

RDA とは，リモートデータベースへのデータベース操作の送信，操作結果のクライアントへの送信等について定めた ISO の国際標準規格である．現在，Microsoft などによって RDA 規格に基づき実装が行なわれている [10]（SQL Server に実装が行なわれているが，これは将来のアップデートでは廃止予定となっている）．

MySQL では，リモートの MySQL データベースへのアクセスを可能にする FEDERATED ストレージエンジンを提供している [11]．これは，予めリモートテーブルと同じテーブル定義のテーブルを作成し，接続情報を登録しリンクしておくことで，ローカルの MySQL にクエリを発行するだけでリモートテーブルにアクセスすることが出来るようになるというものである．また，PosgreSQL の postgres_fdw [12] も同様の技術である．

3.2 関連研究

Dennis Heimbigner らは，多様化する情報アクセスの手段に対する解決策として，Federated Database の概念ついての初期段階の論文を発表した [1]．この論文中で Federated Database とは，相互接続して互いのデータを利用できるような自立したデータベースの集合であると定義されている．

また，Amit P. Sheth らは，Federated Database を，自立性があり不均一な協調システムの集合であると定義している [2]．この論文では，Federated Database System ににおいて重要な要素は自立性，不均一性，分散であると述べている．

Laszlo Dobos らは，自身の持つデータベースとリモートデータベースを 1 箇所のリモートの SQL Server 上で管理することで，それらの相互利用を容易にしようとする Graywulf Project という研究を行っている [3]．Graywulf では，リモートサーバーからコピーするデータ量を最小化するために，実行されたクエリを解析し，必要なカラムのみをコピーするようにしている．

Youzhong Ma らは，IoT 時代に大量に増え続けるデータを効率的に処理するために，更新とクエリの効率的なインデックスフレームワークである update and query efficient index framework（UQE-index）[4] を提案した．これは効率的な多次元クエリを同時に提供できる Key-Value 型のデータストアである．

Jeff Shute（Google）らは，Bigtable のような NoSQL システムのスケーラビリティと，従来の関係データベースの一貫性と使いやすさを兼ね備えたハイブリッド・データベースである F1 [5] を開発した．F1 は同 Google 社の Spanner [6] 上に構築されており，クロスデータセンターレプリケーション（XDCR）と強力な一貫性を提供する．

4. RTA の概要

この章では，RTA システムの概要について述べる．RTA は，まず各機関がオープンデータを RDB の形のまま公開型テーブルとして登録を行う．注意として本論文では，提供されるオープンデータは元々RDB の形式で保存されたものであるとする．そしてデータ利用者側は読み取り専用ユーザーとして，登録された公開型テーブルに SQL によって直接アクセスすることが出来るようになるというものである．また，先述した関連技術では同一の DBMS 間でのデータアクセスのみが可能であったが，RTA では MySQL，PostgreSQL，Oracle 等の一般的に広く普及している RDBMS であればそれらの相互利用可能な点が大きな特徴である．関連研究で述べた Federated Database とは，他のデータベースのデータの利用を可能にするという点では共通しているが，RTA ではネットワーク上での相互接続は想定しておらず，公開されたデータに対して利用者側が一方的にアクセスするという点で異なる．

RTA によって実現可能になることとして，データ利用者側，データ公開者側それぞれに以下のようなものがある．

4.1 データ利用者側のメリット

- 公開情報を関係データベースとして扱えるため，他の形式よりデータが管理しやすくなる．
- 公開データベース（複数でも可）とローカルのデータベースとの間で 1 つの SQL で直接 JOIN 操作などを行うことが出来るようになり，データを利用しやすくなる
- 常に最新のデータを取得することができる

4.2 データ公開者側のメリット

- データを公開し，利用してもらうために専用の WEB サービスを作る必要が無くなる．
- 定期的に公開データの更新をする必要が無くなる．

4.3 RTA の具体的な利用例

RTA においてリモートテーブルを利用する際には，**RTA クエリ**と呼ばれるクエリを用いる．RTA クエリとは，以下のクエリ 1 のように，FROM 句にテーブル名を記述する際に，リモートテーブルであることを示す識別子#を付けた SQL のことである．

次に RTA の具体的な使用例について述べる．今回は例として、現在オープンデータとして公開されている東証の株価データ [8] を利用して，ローカルの users テーブル（表 1）、ユーザーの所有している株式の証券コードのみが保存されている user_stocks テーブル（表 2）と JOIN する以下の操作を挙げる．

クエリ 1

```
SELECT u.id, u.name,
SUM (us.number * s.ending_price) as sum
FROM users u, user_stocks us, #stocks s
WHERE u.id = us.user_id AND us.code = s.code
GROUP BY u.id
```

表 1　利用者の持つテーブル (users)

id	name
1	村上
2	佐藤
3	中山

表 2　利用者の持つテーブル (user_stocks)

user_id	code	number
1	7203-T	100
1	6753-T	2000
1	4661-T	300

表 3　公開テーブル (stocks)

code	brand	ending_price
7203-T	トヨタ	7131
6753-T	シャープ	237
4661-T	OLC	6708

クエリ 1 を実行することで，表 4 のような実行結果が得られる．

表 4　実 行 結 果

id	name	sum
1	村上	3199500
2	佐藤	1715100
3	中山	5910000

また今回はローカルテーブルとリモートテーブルの JOIN を例に挙げたが，ただリモートテーブルのデータを取得してくること，複数のリモートテーブル同士の JOIN を行うことも出来る．

4.4　RTA Library

この節では，公開者側が自身の公開したいテーブルを登録する際に利用する Web アプリケーションである，RTA Library について説明する．

RTA Library とは，PHP 言語により記述された，公開テーブル登録用 Web アプリケーションである．

公開者側は，図 1 のように「データベース登録」から公開したいテーブルの公開情報を登録する．すると 2 のように登録した情報が公開リストに追加される．そして，詳細ボタンを押すと 3 のように公開テーブルの詳細が表示される．登録を行った段階で自動的にそのテーブルに存在するカラム名，型が検索，追加され，その後それぞれのカラムについての詳細説明を編集することが出来る．

利用者側は,「データベース一覧」から利用したいデータを探し,「アクセス名」に記述されているアクセス名をクエリ内に記述するだけでそのデータが自由に利用可能となる．

図 1　公開テーブル登録画面

アクセス名	説明	詳細
#postal_code	東京都の郵便番号データです	詳細
#stocks	12/20の東証株価データです	詳細
#stations	駅データです	詳細
#lines	路線データです	詳細
#station_joins	接続駅データです	詳細
#prefectures	都道府県データです	詳細

図 2　公開テーブルリスト

カラム名	型	説明
code	varchar	証券コード
brand	varchar	銘柄名称
market	varchar	市場名
opening_price	numeric	始値
high_price	numeric	高値
low_price	numeric	低値
ending_price	numeric	終値
turnover	int4	出来高
trading_value	int8	売買代金

図 3　公開テーブル詳細

5.　RTA 機構と実装

この章では，RTA の具体的なアーキテクチャ，実装について述べていく．図 4 が RTA 全体のアーキテクチャである．

5.1　RTA Client

図 4 の左辺は，データ利用者側を示している．利用する DBMS の接続情報などは設定ファイルに記述を行い，実行の際に同一 DBMS 上に，データ利用者の指定した名前の実行用データベースを自動生成する（例：tmpDB）．

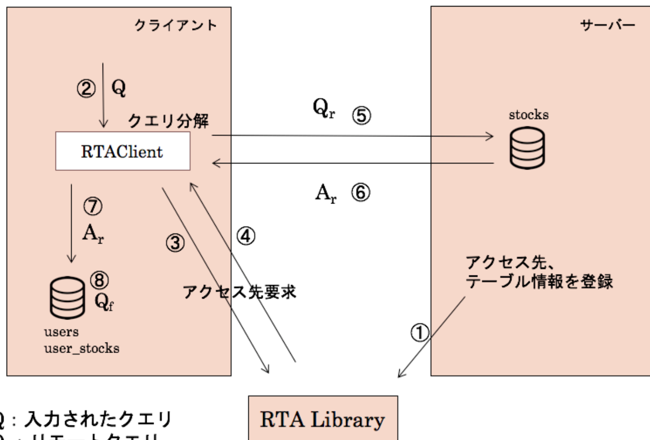


図 4　RTA アーキテクチャ

5.2　RTA Server

図 4 の右辺は，データ提供者側を示している．提供するデータベース情報は，読み取り専用ユーザーを別途作成し，先述した RTA Library より登録を行う．

5.3　処理の流れ

この節で図 4 に従って詳しい処理手順について述べていく．今回は説明の簡易化のため，リモートテーブルは 1 つであると仮定する．

（1）　先述した RTA Library を利用して，公開テーブルの登録を行う（①）．

（2）　利用者からクエリ Q が RTA プログラムへ送信されると，クエリの構文解析，リモートテーブルそれぞれへ問い合わせるための，Q_l，Q_r へのクエリ分割（第 5.5 節）が行われる（②）．

（3）　構文解析の行われたクエリ中の#付きのアクセス名から，そのアクセス名に紐づけられた接続先情報を取得する（③〜④）．

（4）　Q_l をローカルデータベースに問い合わせ A_l を取得する（⑤〜⑥）．

（5）　Q_r をローカルデータベースに問い合わせ A_r を取得する（⑦〜⑩）．

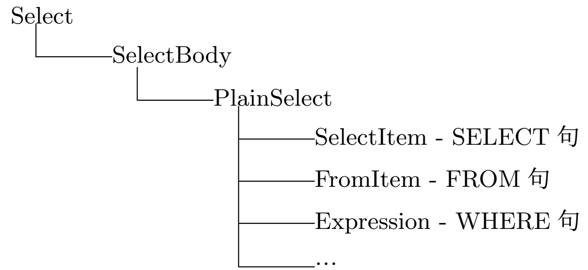
（6）　得られた A_l，A_r を実行用データベースに挿入し，そこで元のクエリ（から#を抜いたもの）を実行する（⑪）．

これが RTA の処理の流れの概略となるが，実際の現在の実装では同一 DBMS を使っているため,（4）部分でローカルのデータをコピーする必要がないため，リモートから取得したデータのみを実行用 DB に格納している．

5.4　JSQLParser

上の②の手順で述べたクエリの構文解析，分解について述べていく．元のクエリの構文解析には，Java によって記述された JSQLParser [9] を用いた．JSQLParser とは，入力された SQL を java のクラス階層構造に変換を行う java プログラムである．一つの DBMS のみではなく，Oracle，SQLServer，MySQL，PostrgreSQL など様々な DBMS に対応しており，Oracle の+を用いた JOIN 操作，PostgreSQL の::を用いたキャスト操作にも対応しているのが特徴である．

今回使用するのは SELECT 文のみであるため，SELECT 文についての JSLparser のクラス階層を簡単に述べる．



説明の簡潔さのために必要最小限のクラス階層のみを記述したが，Expression 以下には Limit，Offset，OrderBy 等のクラスが続いていくことになる．このクラス階層の SelectItem，FromItem，Expression にそれぞれのインスタンスを加えていくことで，SQL を再構築することが出来るというものである．

5.5　クエリの構文解析

この手順での目標は，元のクエリ Q を Q_l，Q_r に分割することである．そのアルゴリズムが以下の Algorithm である．

クエリ Q

```
SELECT u.id, u.name,
SUM (us.number * s.ending_price) as sum
FROM users u, user_stocks us, #stocks s
WHERE u.id = us.user_id AND us.code = s.code
GROUP BY u.id, u.name
```

クエリ Q_l

```
SELECT u.id, u.name, us.number, us.user_id, us.code
FROM users u, user_stocks us
```

クエリ Q_r

```
SELECT s.ending_price, s.code
FROM stocks s
```

このアルゴリズムについて，先述したクエリ Q，Q_l，Q_r の例で説明する．アルゴリズム中に現れる remoteConnector，localConnector とは，自身で作成した TableConnector クラスのインスタンスであり，それぞれの接続先情報と FromItem，SelectItem 等を持つ．まず，3 行目からのループで#postal_code というアクセス名を発見し，RTA Library に問い合わせを行い，登録されていれば接続先情報を取得してアクセス名と共に connector へ格納する．次に 10 行目で，Q から#を抜き最終的に実行するクエリに変換する（Q_f）．

Algorithm クエリ分割アルゴリズム

```
1: Input: 元の SQL（Q）
2: Output: 分割された SQL（Q_l，Q_r）
3: for all FROM 句に存在する#付きアクセス名 do
4:   if そのアクセス名が RTA Library に登録されている then
5:     remoteConnector[n] ⇐ 取得した接続先情報，アクセス名
6:   else
7:     return ' そのアクセス名は利用できません'
8:   end if
9: end for
10: もとの SQL から#を除く
11: JSQLParser によって SQL を Parse
12: for all FROM 句のテーブルのリスト do
13:   for all remoteConnector do
14:     if そのテーブルが remoteConnector[n] に存在する then
15:       remoteConnector[n].fromItem ⇐ テーブル
16:     end if
17:   end for
18:   if どの remoteConnector にもテーブル名が存在しない then
19:     localConnector.fromItem ⇐ テーブル
20:   end if
21: end for
22: for all SELECT 句のカラムのリスト do
23:   for all remoteConnector do
24:     if そのカラムのテーブルが remoteConnector[n] に存在する
        then
25:       remoteConnector[n].selectItem ⇐ カラム
26:     end if
27:   end for
28:   if どの remoteConnector にもそのカラムのテーブルが存在し
      ない then
29:     localConnector.selectItem ⇐ カラム
30:   end if
31: end for
32: for all WHERE 句の以降に現れるカラムのリスト do
33:   for all remoteConnector do
34:     if そのカラムのテーブルが remoteConnector[n] に存在す
        る，かつまだ selectItem に挿入されていないカラムである
        then
35:       remoteConnector[n].selectItem ⇐ カラム
36:     end if
37:   end for
38:   if どの remoteConnector にもそのカラムのテーブルが存在し
      ない，かつまだ selectItem に挿入されていないカラムである
      then
39:     localConnector.selectItem ⇐ カラム
40:   end if
41: end for
42: for all remoteConnector do
43:   その接続先についての SQL を再構築
44: end for
45: ローカルに問い合わせる SQL を再構築
```

クエリ Q_f

```
SELECT u.id, u.name,
SUM (us.number * s.ending_price) as sum
FROM users u, user_stocks us, stocks s
WHERE u.id = us.user_id AND us.code = s.code
GROUP BY u.id, u.name
```

次に 11 行目で Q_f を JSQLParser によって構文解析を行い，SelectItem，FromItem，Expression などを抽出する．12 行目からのループで，FromItem のテーブル名と remoteConnector へ格納したアクセス名が一致すれば FromItem に追加，どこの remoteConnecor にも見つからなかったら localConector の FromItem へ追加を行う．

21 行目終了時点での connector

```
remoteConnector {
アクセス名：stocks
FromItem：stocks s
SelectItem：なし
}
localConnector {
アクセス名：なし
FromItem：users u, user_stocks us
SelectItem：なし
}
```

そして 22 行目からのループで SelectItem のカラムをそのカラムの存在するテーブルごと振り分ける．具体的には，u.id, u.name などは users u のエイリアス u から同じテーブルであると判断出来るため localConector へ振り分ける．テーブルのエイリアスが付いていない場合には，RTA Library に問い合わせをし，カラム名が存在すればその remoteConnector へ振り分ける．

30 行目終了時点での connector

```
remoteConnector {
アクセス名：stocks
FromItem：stocks s
SelectItem：s.ending_price
}
localConnector {
アクセス名：なし
FromItem：users u, user_stocks us
SelectItem：u.id， u.name, us._number
}
```

そして最後に，32 行目からのループでそれ以降にのみ現れるカラムの振り分けを行う．この例では，u.postal_code が SELECT 句では現れておらず Q_f を実行した際にエラーになってしまうため，追加で localConnector へ振り分ける必要がある．

37 行目終了時点での connector

```
remoteConnector {
アクセス名：stocks
FromItem：stocks s
SelectItem：s.ending_price, s.code
}
localConnector {
アクセス名：なし
FromItem：users u, user_stocks us
SelectItem：u.id, u.name, us.number, u.user_id, us.code
}
```

以上の処理によりカラムの振り分けが完了したので，最後に 43 行目，45 行目で JSQLParser の機能により SQL を再構築し，その SQL を connector へ格納して完了となる．このようにして，リモートテーブルのアクセス名を含んだ SQL の構文解析，分解が実現される．

6. 評　　価

6.1 既存リモートアクセス技術との機能比較

RTA の性能を測るために，先述した既存技術・既存研究である RDA，FEDERATED DATABASE（FD）との機能比較を行った．

本研究で提案した RTA と RDA，FD の機能比較を表 5 に示す．この表から，既存の技術に比べて，アクセスしたいリモートテーブルの情報をほとんど知らなくてもオープンデータが利用可能であるということが分かる．公開されているオープンテーブルの種類，そのテーブルのカラム情報などは全て先述した RTA Library 上に記述されているため，それを見れば一目で理解することができる．また，そのアクセス名をクエリ内に記述するだけで自動的にアクセスしてデータを取得するため，パスワード等のデータベースへの接続情報を利用者側が知っている必要も無い．

また，RDA，FD はそれぞれ同種の DBMS 間でのみ利用可能な技術である．しかし，TA は例えば公開側の DBMS が PostgreSQL，利用者側の DBMS が MySQL 等である場合も，データを取得してきた際に自動的に利用者側の DBMS に合わせたデータ型にデータを変換し，テーブル作成，データ挿入等を行うため，利用者側が DBMS の違いを意識することなく自由にデータを利用することができるという優位性がある．

表 5　各リモートアクセス技術の機能比較表

機能	RDA	FD	RTA
互いの接続先情報が不要	〇	×	〇
自身で保存用テーブル作成不要	×	×	〇
テーブル定義，カラム情報が不要	×	×	〇
ローカルテーブルとの JOIN 操作	×	〇	〇
異なる DBMS 間でも利用可能	×	×	〇

6.2 段階的実装における速度評価

今後の実装の目安として，段階的評価における速度評価を行った．実装段階として，以下のものがある．

段階 1：リモートテーブルの全件取得
現在実装済みの段階．リモートクエリを WHERE 句で絞り込むことなく必要なカラムを全件取得．リモートデータのサイズが大きくなると非常に無駄が多くなる．
例）SELECT s.ending_price, s.code FROM stocks s

段階 2：WHERE 句による絞り込み
現在不完全に実装済みの段階．リモートクエリを WHERE 句で絞り込み，取得する件数を減らす．
例）7 から始まる証券コードの行のみを取得
SELECT s.ending_price, s.code FROM stocks s WHERE code LIKE '%7'

段階 3：Semi-JOIN による絞り込み
ローカルテーブルも考慮したクエリ最適化を行い，段階 2 より更に取得する件数を必要最小限にする．
例）ローカルテーブルである user_stocks（表 6）中に存在する証券コード（code）の部分のみをリモートの stocks テーブルから取得する．

表 6　user_stocks テーブル

user_id	code	number
1	7203-T	100
1	6753-T	2000
1	4661-T	300

6.2.1 実 験 環 境

ローカル側，リモート側の実験環境はそれぞれ表 7 の通りである．

表 7　実 験 環 境

	ローカル	リモート
OS	OS X 10.11	CentOS 7.2
CPU	Intel Core i5 2.7GHz	Nehalem-C 2.4GHz
RAM	16GB	30GB
DBMS	MySQL 14.14	MySQL 14.14

6.2.2 実 験 方 法

クエリ
各ユーザーの保有している東証 2 部銘柄の株式総資産額を求める以下の RTA クエリで実験を行った．

評価クエリ

```
SELECT u.id, u.name, SUM (us.number * s.ending_price)
FROM users u, user_stocks us, #stocks s
WHERE u.id = us.user_id AND us.code = s.code
AND s.market = ' 東証 2 部' GROUP BY u.id, u.name
```

測定対象
以下の条件でそれぞれ 5 回ずつ実行した平均時間

条件（カッコ内は実際にリモートから取得した行数）
条件１：ローカルのみ（0 行）
条件２：全件取得（4001 行）
条件３：WHERE 句の絞り込み（534 行）
条件４：Semi-JOIN（10 行）- 擬似的実装（WHERE code IN で記述）

6.2.3 実験結果

表 8 実験結果（msec）

	条件１	条件２	条件３	条件４
実行速度	0.5	2730	1618	837

表 8 より，全てローカルで実行した際に非常に速い速度で実行が行われているのは当然であるが，条件２〜条件４と段階的に実装を進めていくことによって，初期段階では 3 秒近くかかっていた実行が 1 秒を切るまで速度を速めることが出来るということが分かる．条件４における 837 ミリ秒という数字だが，そのほとんどが先述した RTA クエリの構文解析，クエリ分解に掛かっている時間であり，その時間を減らすのは現段階では非常に困難であると考えられる．そこで当面の目標は，あらゆるクエリを 150 ミリ秒以内で実行完了させることとする．

7. おわりに

7.1 結論

本研究では，主にオープンデータの効率の良い二次利用という観点から，リモートデータへ RDB の形式のままアクセスすることが可能な RTA アーキテクチャの提案，実装を行った．RDB のまま利用可能という形式を取ることで，データ利用者がデータをダウンロードし，自身の DB に挿入するといった負担を軽減することが出来る．また，RTA Library という Web アプリケーションを見ることで利用可能なデータの種類，カラム名，データ型などを容易に確認することが出来るので，利用者側がそれらを判断するといった負担を軽減することも出来る．

また，既存のリモートアクセス技術と比べても，ローカルの DB 内にテーブルを作成する必要が無い，接続先情報を登録する必要が無い，異なる DBMS 間でも利用可能であるという点で優位性があるといえる．

7.2 今後の課題

RTA は初期段階の研究であるため，今後の課題が多く存在する．まず現在の実装では，リモートテーブルを全て取得しているため，公開されるデータの量が大きくなると非常に非効率である．そのためクエリの最適化を行い，必要なデータのみを取得するような実装を行う必要がある．次に，現在では Java アプリケーション上のみからしか RTA を実行出来ないため，これは非常に実用性に欠ける．そのため，今後は MySQL, PostrgreSQL 等の標準的な DBMS の拡張機能として RTA を提供する必要があると考えている．

参考文献

[1] Dennis Heimbigner, Boulder Dennis McLeod: “A federated architecture for information management”, Journal of ACM Transactions on Information Systems (TOIS), 1985
[2] Amit P. Sheth, James A. Larson: “Federated database systems for managing distributed, heterogeneous, and autonomous databases”, Journal of ACM Computing Surveys (CSUR), 1990
[3] Laszlo Dobos, Istvan Csabai: “Graywulf: A platform for federated scientific databases and services”, Proceedings of the 25th International Conference on Scientific and Statistical Database Management Article (SSDBM) No.30, 2013
[4] Youzhong Ma, Jia Rao, Weisong Hu, Xiaofeng Meng, Xu Han, Yu Zhang, Yunpeng Chai, Chunqiu Liu: “An Efficient Index for Massive IOT Data in Cloud Environment”, Proceedings of the 21st ACM international conference on Information and knowledge management（CIKM）2012
[5] Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littleeld, David Menestrina, Stephan Ellner, John Cieslewicz, Ian Rae, Traian Stancescu, Himani Apte: “F1: A Distributed SQL Database That Scales”, Proceedings of the VLDB Endowment 2013
[6] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, Dale Woodford: “Spanner: Google’s Globally Distributed Database”, ACM Transactions on Computer Systems 2013
[7] 郵便番号データ: http://www.post.japanpost.jp/zipcode/download.html
[8] 株価データ: http://k-db.com/
[9] JSQLParser: https://github.com/JSQLParser/JSqlParser
[10] Microsoft RDA: https://msdn.microsoft.com/ja-jp/library/cc414853.aspx
[11] FEDERATED ストレージエンジン: https://dev.mysql.com/doc/refman/5.6/ja/federated-storage-engine.html
[12] postgres_fdw: https://www.postgresql.jp/document/9.3/html/postgres-fdw.html