RTA: A Framework for the Integration of Local and Relational Open Data

Yusuke Kosaka
Keio University
Yokohama, Japan
kosaka@db.ics.keio.ac.jp

Shu Murakami
Keio University
Yokohama, Japan
shu@db.ics.keio.ac.jp

Thomas Laurent
Keio University
Yokohama, Japan
thomas.laurent@db.ics.keio.ac.jp

Kento Goto
Keio University
Yokohama, Japan
goto@db.ics.keio.ac.jp

Motomichi Toyama
Keio University
Yokohama, Japan
toyama@ics.keio.ac.jp

ABSTRACT

There are currently massive amounts of public data, also refereed to as open data, for example stock price data or weather data. However, such data is distributed in a variety of ways, such as downloadable files like CSV or XML files, or through API calls to web services. Each data source thus requires a specific workflow, making it a burden for the users to process and use this data. This barrier to use diminishes the openness of this data We thus propose the Remote Table Access (RTA) system, a simple and safe architecture for publishing, i.e. giving open read only access to relational data, and easily integrating it with the user's local data. RTA enables the user to query relational open data and their own local data seamlessly through a single SQL query. To allow this, we designed a three parties architecture featuring a client-side application, an optional server-side module and a "Public Table Library" (PTL). The client side application processes the RTA query and fetches the necessary data, the server side system acts as an agent between the remote database and the client, offering added security as well as scalability in terms of connections, and the PTL list all the published data and stores its access information. We implemented an early prototype of this architecture as a proof of concept. We validated it against two datasets, including data from the TPC-C benchmark and make it available[1]. Our results show the feasability of RTA and possible significant reduction of query processing time mainly because of the reduction on transmission volume by condition pushing and semijoin.

ACM Reference format:
Yusuke Kosaka, Shu Murakami, Thomas Laurent, Kento Goto, and Motomichi Toyama. 2017. RTA: A Framework for the Integration of Local and Relational Open Data. In *Proceedings of IDEAS '17, Bristol, United Kingdom, July 12-14, 2017,* 8 pages.
https://doi.org/10.1145/3105831.3105852

[1]http://db.ics.keio.ac.jp/RTA/demo.html

1 INTRODUCTION

Open data is published data that is freely accessible, usable, and republishable by any party without any restrictions, except possibly attributing the origin of the data. Examples of such data include information made available by government agencies, weather records or stock prices.

Open data is currently published in four main ways. The first two methods consist of making the data available as downloadable files; either human readable such as PDF or HTML or more structure and machine-friendly such as CSV or XML. The third method is to make the data available through API calls to web services. Finally, open data is also being made available through a new format called Linked Open Data, or LOD [1].

While the two first methods make it easy to retrieve the data, as one only needs to download a file, they introduce other difficulties. Indeed, once the file or files have been acquired, the user still needs to parse and process them to integrate them with his local data or to use them in any kind of application. This parsing process can be very heavy, if not impossible, for files such as PDF files. Furthermore the acquisition an parsing of the data has to be repeated each time the data is updated, which can happen very often. The third method allows for better integration of the data into an existing framework but requires the data user to write a data-specific program to access the data and the data owner to design, implement and maintain a web service and an API for this service. Furthermore, if the user needs to retrieve data in a way that the data owner has not foreseen they can be limited by the capabilities of the API. Finally, while LOD provides a framework for the publication and integration of multiple data sources, it is not easily integratable with the relation database paradigm.

Relational databases are currently used in a variety of situations and applications and we consider that there is a real need for easy interfacing of open data with local relational data. We consider that the current distribution methods of open data do not offer an efficient solution to this need. Indeed, data published as files can be more or less well structured and the parsing and formatting of this data into a relational form can require considerable effort. An effort that has to be renewed for every new data source as no general convention exists for the presentation of open data. Similarly, when using API calls to access the data, the end user has to write a new program for each new API, which can become overwhelming if numerous data sources have to be integrated together. Finally, LOD provides data that is not formatted in a relation fashion, requiring considerable knowledge about the data at hand and data management to integrate it into a relational paradigm.

In order to solve these problems, we propose a system we call Remote Table Access (RTA). In RTA, the data publisher can register open data as relational data tables through a Web application, opening read only access to this table to all RTA users. Users can query the open data along with their local data, as if it was stored in their own database, through a single SQL query. By adopting such a scheme, we solve the above problems and make it possible to use open data smoothly for both the user and the publisher.

RTA relies on three systems: a client application that processes the user's query and fetches the remote data, an optional remote side module that allows the safe publication of data and the Public Table Library, which registers all the published data sources.

The rest of this paper is structured as follows: In Section 2, we give an overview of current open data remote access technologies and the challenges in this area. We introduce the motivation for RTA and use cases in Section 3, and we describe concrete architecture and query process in Section 4. In Section 5, we evaluate the RTA using a prototype implementation, and we conclude and introduce future work in Section 6.

2 RELATED WORK

In this section we describe the current remote and open data access systems and technologies. First we review Linked Open Data techniques and then introduce different methods for relational database remote access.

2.1 Linked Open Data

Linked data is a method for storing data in a way that facilitates semantic queries. Linked data relies on the Resource Description Framework (RDF) [2] to format the data as subject-predicate-object triples. Several large scale projects, such as DBPedia [3] which publishes information extracted from Wikipedia as linked data, aim at sharing linked data, creating the world of Linked Open Data.

SPARQL [4], is a querying language for RDF, aiming at bringing the expressivity and search power offered by SQL over relational data to linked data. Still, most data is now stored as relational data, making the integration of linked data with the data at hand an uneasy task. This is why we will concentrate on relational open data in this project.

2.2 Remote access to relational databases

Some efforts have been made to allow querying of remote data in relational databases, we will now review the most notable examples.

Remote Database Access, or RDA, is a protocol standard for accessing information on remote databases. Despite early efforts at building proof of concept implementation, such as [5], the protocol has been largely ignored by the major DBMSes. Thus, despite it technically providing a basis for open relational data RDA can not be used with current real systems.

The ISO/IEC 9075- 9:2016 standard deals with the management of external data in SQL. This standard defines the notion of wrapper for external data in SQL, a notion that manifests itself as funciton such as postgres_fdw, Oracle DBLINK or mysql federated in current implementations. These wrappers, like RTA, provide a means to integrate remote and local data in SQL but do not provide a mechanism for data publication. Furthermore, they require a considerable configuration effort from the user, which RTA avoids.

Federated databases offer a way to manage treat a group of databases as a single entity. This permits easy querying of data from heterogeneous sources. An early proposal for a federated databases architecture is given in [6]. RTA also offers the possibility of seamlessly integrating several data sources, but in a read only fashion, for reasons we'll highlight later. Furthermore, RTA introduces the notion of publicizing the data, which is not part of the federated databases model.

Graywulf [7] is a platform that provides reusable components for efficient storage, conversion, statistical analysis, and display of scientific data stored in Microsoft SQL Server. It is similar to our concerns as it provides unified user access to data though a slightly modified SQL that provides a transparent representation of the distributed data. Still, the scope of this project is much more restricted than RTA as it only considers one DBMS and is not intended for data publication.

CloudMdsQl [8] is a polystore system that enables users to retrieve data from multiple heterogeneous cloud data stores in a single SQL-like query. The CloudMdsQl query is then split into queries for each store and translated into the native querying language for each system. Although this principle resembles the functioning of RTA, CloudMsQl does not offer data publishing mechanisms or easy, configuration-less querying for the end user as RTA proposes . We believe the work done on CloudMdsQl in areas such as query optimisation can be adapted to our system and we are currently exploring this possibility.

3 MOTIVATION AND USE CASES

3.1 Motivation

There are currently massive amounts of public data, also referred to as open data, such as stock price data or weather data for example. However, such data is distributed in a variety of ways, such as downloadable files like CSV or XML files, or through API calls to web services. A user wanting to use such data has to write data-specific programs to fetch, process and integrate such data in their relational paradigm, making it a burden to use this data. As such, we do not consider open data to be truly open, as there is a significant cost to using it.

We therefore strive to offer a system that eases the use of open data, for both the user and the data publisher. By allowing data-owners to directly publish their relational databases, which are one of the main means of storing data; and allowing the user to access the data directly in a relational form (through SQL), we hope to see the world of open data expand.

Such a framework for relational open data could pave the way for easy implementation for a multitude of applications revolving around open data, such as stocks value monitoring. Furthermore, lowering the burden of publishing data would incentive data-holders to give access to their data, expanding the possible horizons even more.

3.2 Use cases

We will now illustrate our vision of RTA by describing several possible use cases. We first describe the initial application of RTA, open relational data, then we explore the potential of RTA for mobile use and finally review a possible extension to a "private RTA" to publish and share relational data inside an organization.

3.2.1 Relational open data. Relational open data is the initial application of RTA. In this use case, a public instance of RTA is used by publishers to give unrestricted read access to their data and by end users to easily integrate this open data with their current relational data. In this setting the published data is completely open.

Then RTA could be used to give open access to weather data or stock prices for example. This data could then easily be integrated and displayed by a web service, or queried and analyzed by any interested party.

As RTA would then represent an open read access point to the publisher's database, security concerns are bound to arise, which we will take into consideration. As the pool of end users is virtually limitless, performance and concurrency issues are also to be considered, should a data source become popular or a "hot-spot".

3.2.2 Mobile RTA. RTA as an relational open data framework could also be adapted to mobile devices. Considering the nature of mobile networks the volume of data retrieved in this case would of course be smaller. In this setting RTA could for example make the design of applications querying and displaying data such as public transport schedules easier.

A challenge in this situation would be to implement and interface RTA for the mobile device environment (e.g SQLite).

3.2.3 Private RTA. Another possible use of RTA would be as a private instance to easily share data among an organization. As an organization's data is rarely publishable this use case would require a closed, private RTA. On the other hand, access mechanisms would be necessary to ensure that no entity outside the organization can access the internally published data. In this way, RTA could be seen as a read-only federated database.

A hybrid setting could also be envisioned, where a company opens a part of its data to the world while also using RTA as an internal sharing tool. In this setting, security would be primordial as RTA would have access to both open and sensitive data.

4 ARCHITECTURE

In this section we describe our proposed architecture. We first describe the general architecture and list the different modules of our proposed system. We then describe each module.

4.1 General architecture

RTA is based on a three-parties architecture: the "client" (or "local machine", L) who wants to query remote open data, the remote publisher (R) who makes some of their data public and the Public Table Library (PTL), that stores R's access information and makes it available to L.

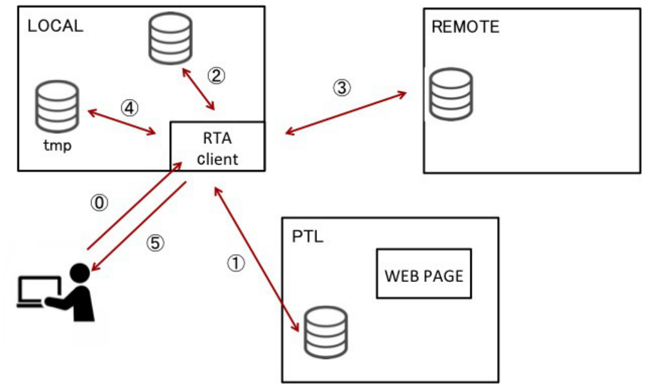


Figure 1: RTA Architecture (method A)

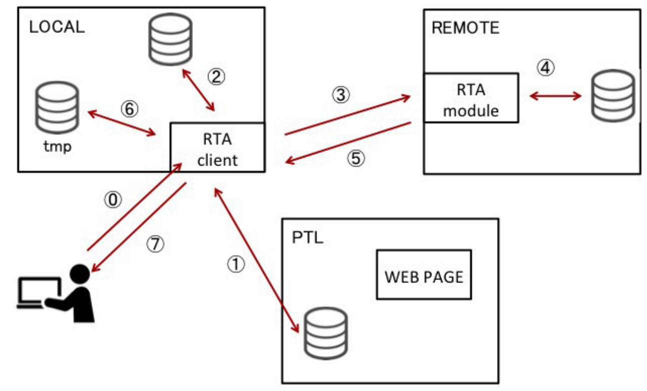


Figure 2: RTA Architecture (method B)

Figures 1 and 2 present this architecture. These two figures differ in the connection method used between L and R, see subsection 4.6 for further details. Let us describe the process of publishing and querying open data with RTA.

(0) (For method B) The administrator of R installs the RTA module on R.
(1) The administrator of R registers public tables to the PTL using a web interface.
(2) The user at L executes a query Q, the RTA Client scans for remote data calls (indicated by a # in the query)
(3) The RTA client fetches connection information for R from the PTL
(4) The RTA client fetches the needed data from R (through method A or B) and stores it in a local temporary database
(5) The remote and local data are joined and filtered
(6) The query results are returned to the user

We will now describe each module and step in more detail.

4.2 Public Table Library

The Public Table Library is a repository of all tables published through RTA. It offers a web interface that lets publishers register new tables and manage their existing publications, and enables end users to find published tables and their access names. The access name uniquely identifies a table in the RTA system. It is set by the publisher at registration and is used by end users in their queries.

Figure 3 shows the interface proposed to register a new public table in the system. The publisher enters the table access information, such as DBMS, hostname, and credentials, and finally a list of tables to be published. The publisher can then give a short description of the tables and their columns to facilitate readability for the end user. This information is then saved in the PTL and the table is added to the list of published tables. In its current version the PTL uses and stores database credentials, it is thus strongly advised to for the publishers to use a separated user specifically for RTA, whose permissions are limited.

Figure 4 shows the list of published tables as shown to the end user. When pressing the “details" button for a table, the user is shown the mdframed in figure 5. On this page, the user can see the name, type, and description of each column of the table.

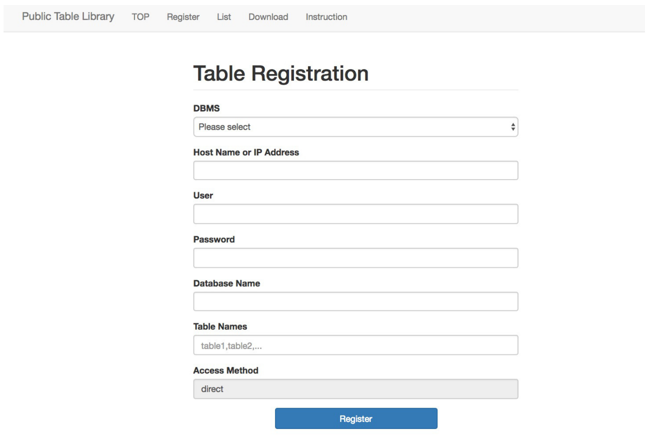


Table Registration

DBMS: Please select

Host Name or IP Address

User

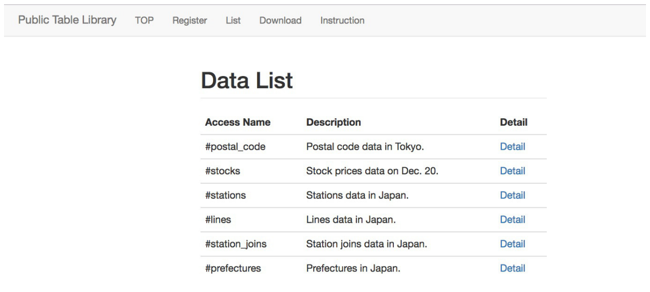
Password

Database Name

Table Names: table1,table2,...

Access Method: direct

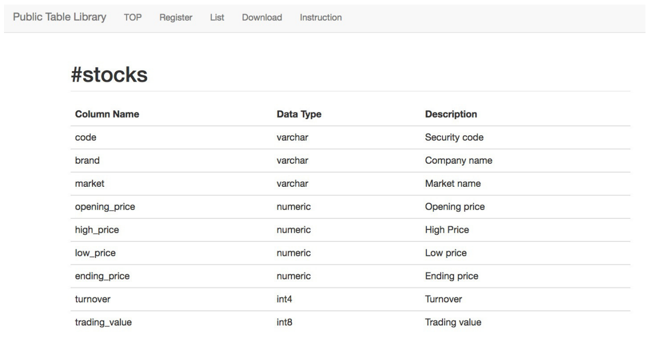
Figure 3: Public Table Registration



Data List

Access Name	Description	Detail
#postal_code	Postal code data in Tokyo.	Detail
#stocks	Stock prices data on Dec. 20.	Detail
#stations	Stations data in Japan.	Detail
#lines	Lines data in Japan.	Detail
#station_joins	Station joins data in Japan.	Detail
#prefectures	Prefectures in Japan.	Detail

Figure 4: Public Table List



#stocks

Column Name	Data Type	Description
code	varchar	Security code
brand	varchar	Company name
market	varchar	Market name
opening_price	numeric	Opening price
high_price	numeric	High Price
low_price	numeric	Low price
ending_price	numeric	Ending price
turnover	int4	Turnover
trading_value	int8	Trading value

Figure 5: Stocks Table Detail

4.3 RTA query parsing

RTA uses the syntax of SQL to express queries over the local and published data and enriches it with the character # to mark published data sources. In an RTA query, remote tables are marked by prepending the # character before the table name. This character is an invalid first character for a table name in the SQL syntax, avoiding any clash between a normal SQL table name and a remote marker. Parsing an RTA query is thus the same as parsing a regular SQL query.

In our implementation of we use a slightly modified version of JSQLParser [9] to parse the RTA queries on L.

4.4 RTA query processing

Once the query Q is parsed, the resulting representation is decomposed on L, and a local query $\mathrm{Q_l}$ and remote queries $\mathrm{Q_r}$ are created. Queries $\mathrm{Q_r}$ are then sent to the remote machines to be executed and the results stored in temporary local tables. The cached remote results and the local data are then joined to answer Q. We will now detail the process of decomposing Q, as described in algorithm 1 and illustrate it using the query in figure 6.

First, we examine the FROM clause of Q. For every remote table marked by a # we check that the table exists in the PTL and create a RemoteConnector, and object representing the table and containing its connection information. In our example we create one RemoteConnector for table stocks. For every local table, we add the table to the LocalConnector. We then process the SELECT clause. For each column in the select clause, if the column belongs to a remote table we add it to the corresponding RemoteConnector's selectItem, a list of the columns to fetch from this table, and if it belongs to the local data we add it to the LocalConnector's selectItem. Here we add s.ending_price to stocks' RemoteConnector and u.id, u.name and us.number to the LocalConnector. Finally we examine the WHERE clause for any additional columns to fetch from the remote or local tables. Here s.code is added to the RemoteConnector; us.user_id and us.code are added to the Local Connector.

Once the query has been processed like so, a remote query Q_r is produced for every RemoteConnector and a local Q_l is produced from the LocalConnector. Queries Q_r and Q_l for our example are shown in figures 7 and 8.

```
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
```

Figure 6: RTA query Q

```
SELECT s.ending price, s.code FROM stocks
```

Figure 7: RTA query Q_r

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

Figure 8: RTA query Q_l

Generation of the remote querie(s) Q_r is currently naive in RTA. The remote query currently fetches all the necessary tables to complete the query in their entirety, thus fetching much unnecessary data. This process can be vastly optimised, using well studied query optimisation techniques for distributed databases such as defined in [10] or [11]. A first, straightforward way to optimise it would be to integrate conditions in the RTA where clause that concern only the remote data into query Q_r. A second, more involved way would be to first run query Q_l locally and then use the results of this query in association with join conditions to fecth only remote data that can be joined with the local data. This can be done either though an IN clause in Q_r or by performing a semi-join on the remote server. This last possibility would require extra privileges to be given to RTA on the remote machine.

Algorithm 1 Query decomposition algorithm

```
1: Input: Parsed RTA Query (Q)
2: Output: Local and remote SQL queries($Q_l$, $[Q_{r,n}]_{n=1..}$)
3: for all tableName in Q's FROM clause do
4:     if tableName begins with # then
5:         if access name is registered with the PTL then
6:             add a new RemoteConnector to the list of RemoteConnectors with the
               connection information and access name
7:         else
8:             return 'This access name is invalid'
9:         end if
10:    else
11:        add the tableName to the LocalConnector
12:    end if
13: end for
14: for all column in Q's SELECT clause do
15:    local ← true
16:    for all remoteConnector[n] in remoteConnector do
17:        if the column belongs to table remoteConnector[n] then
18:            add column to remoteConnector[n]'s selectItem list
19:            local ← false
20:        end if
21:    end for
22:    if local then
23:        add column to localConnector's selectItem list
24:    end if
25: end for
26: for all column in Q's WHERE clause do
27:    local ← true
28:    for all remoteConnector[n] in remoteConnector do
29:        if column ∈ remoteConnector[n] AND column ∉ remoteConnec-
           tor[n].selectItem then
30:            add column to remoteConnector[n]'s selectItem list
31:            local ← false
32:        end if
33:    end for
34:    if local AND column ∉ localConnector.selectItem then
35:        add column to localConnector's selectItem list
36:    end if
37: end for
38: for all remoteConnector[n] in remoteConnector do
39:    create remote SQL query $Q_{r,n}$
40: end for
41: create local SQL query $Q_l$
```

4.5 Communication between the PTL and L

L queries the PTL for connection information to the remote machine(s) R using the access names used in the query. This is done through a GET request to an exposed endpoint on the PTL. The PTL then returns the connection method (see 4.6) and necessary connection information, for example the url or IP of R, and a username/password combination to connect to the database or an error if the table is not registered.

4.6 Data retrieval from L to R

After getting the access information from the PTL, L fetches the necessary data from R and stores it in a local temporary database. We propose two methods to achieve this. The publisher can choose either method to publish their data and the choice is irrelevant to to end user as everything is handled by our system.

The first method, that we call method A, is a traditional direct database connection. This requires to communicate the access credentials of the remote database to the client, which represents a security risk, as these credentials could be intercepted, by a third party, or by a malicious user. The burden is then on the publisher to manage the permissions associated with these credentials in order to protect themselves and to make sure their firewall allows outside connections to their database.

The second method, that we call method B, relies on a module installed on the remote machine. When the local machine queries the PTL it receives the hostname of this module and uses a REST request to fetch the data. Although this introduces an overhead at execution, it both improves the security of the system, as database credentials are not communicated and only read capabilities are exposed; and lessens the burden for the publisher, who only has to install the module, and does not need to worry about permissions or firewalls. While this method relies on REST requests, it hides the intricacies of these calls from both the end user, who only has to write an SQL query; and from the information holder, who does not have to design and implement an API for the data.

4.7 Query processing on R

When using connection method A, the query is simply handled by the RDMS on R. When using method B, the remote module receives the client's REST request, queries the database for the necessary data, transforms the query result into a JSON representation and sends the data as a response to the client.

```
SELECT T1.id, T2.name,
SUM (T2.number * T3.ending_price) as sum
FROM users_20170418172910 T1,
user_stocks_20170418172910 T2, stocks_20170418172910 T3
WHERE T1.id = T2.user id AND T2.code = T3.code
GROUP BY T1.id, T2.name
```

Figure 9: RTA final query Q_{fi}

4.8 Joining local and remote data

Once the remote data has been retrieved and stored in the local temporary database it is joined with the local data to produce the query results. This is done through a final query Q_{fi} as illustrated in figure 9. The temporary tables are named as the original tables and a timestamp is appended to distinguish different runs. These tables are currently re-created for each run future versions of RTA can leverage previously acquired data to process queries faster.

5 PERFORMANCE EVALUATION

In this section we describe the experiments we performed using our proof of concept prototype of RTA and review the obtained results. This current prototype is only compatible with mysql and postgresql.

5.1 Experimental Environment

We perform our experiments in the following environment. We use a Macbook pro 2015 under OS X 10.11, with a 2.7GHz Intel Core i5, 16GiB of RAM and running MySQL 14.14 as the client machine L. We use a virtual machine with 4 cores at 2.4Mhz, 3.7GiB RAM, CentOs 7.2.1511 and running postgreSQL 9.2.15 and mysql 14.14 as our single remote machine R. The PTL is hosted on the same machine as the remote database.

We perform our experiments in four different network environments: *LOCAL*, where the network is not used and all the data is stored on the local machine L, *WAN*, where the client L and the remote machine R are located on different networks, *MOBILE*, where L uses a (tethered) mobile network and *LAN*, where L,R and the PTL are all on the same network. LOCAL serves as a baseline, as the capabilities of RTA are not used in this case and the three other situations mirror the use cases described in 3.2.

5.2 Prototype and optimization mocking

As a proof of concept for RTA we implemented a naive prototype that fetches whole tables from the remote machine R before executing the query locally. This prototype is available at http://db.ics.keio.ac.jp/RTA/demo.html. This prototype validates our architecture but does not provide acceptable performance. We thus run our experiments mocking different possible optimizations that will be integrated to RTA in the future.

The optimized versions of RTA are: *WHERE*, in which where clauses that apply solely to the remote data are passed to R, thus fetching less data; *IN*, where all possible local data is first fetched and used to select the remote data using an IN clause; and *OPTIM* where both the other techniques are combined. We will refer to the real prototype as *NAIVE*.

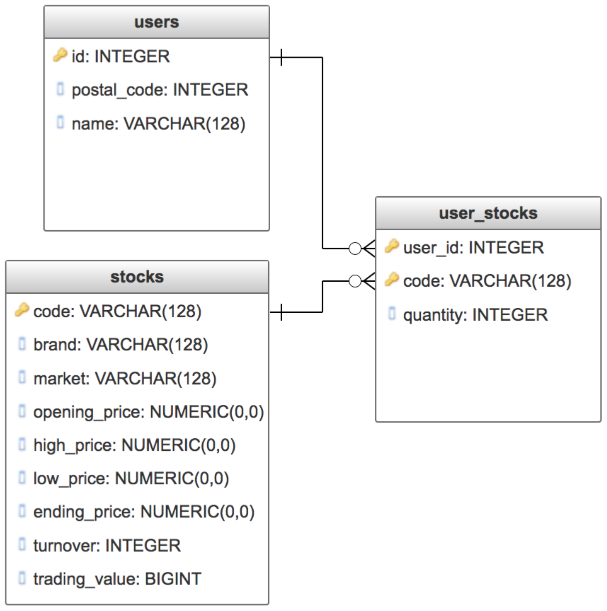


Figure 10: Schema for the stocks data

```
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 = 'TSE' GROUP BY
u.id, u.name
```

Figure 11: Query used in the validation experiment

Each version of RTA is compatible with both communication methods described in 4.6.

5.3 Experiments

5.3.1 Validation. In this first experiment we compare the *NAIVE*, *WHERE* and *IN* versions of RTA, using both communication methods and in every network environment (*LOCAL*, *LAN*, *WAN* and *MOBILE*) using a real world, limited dataset. The *OPTIM* version is not considered in this experiment as the *IN* clause generated by the query used subsumes the *WHERE* clause generated and is thus equivalent to the *OPTIM* version.

The dataset used consists of stocks' price data from the second section of Tokyo Stock Exchange. The data is from 2016/12/20, and was retrieved from http://k-db.com/. The data schema is as shown in figure 10, where tables user and user_stocks are located on L and the other table stocks is located on R.

We used the query shown in figure 11 to test the system and measured the execution time of the query for each situation.

5.3.2 Scalability in data volume. In this second experiment we aim to assess the scalability of RTA. Thus we compare all versions of RTA (*NAIVE*, *WHERE*, *IN*, and *OPTIM*), with both connection methods in all network environments (*LOCAL*, *LAN*, *WAN* and *MOBILE*), using different workloads.

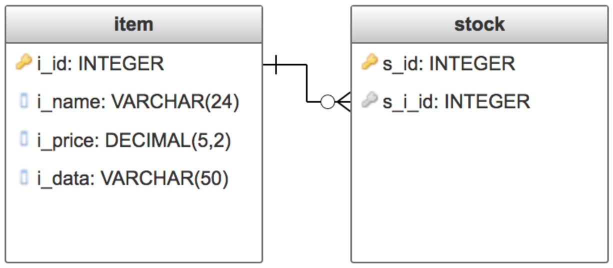


Figure 12: Schema for the TPC-C data

```
SELECT s.i_id, i.i_name, s.s_quantity*i.i_price FROM stock s,
#item i WHERE s.s_i_id = i.i_id AND MOD(i.i_id, 10) = 7
```

Figure 13: Query used in the scalability experiment

The data we used for this experiments is from the TPC-C benchmark. We used a Java implementation of the benchmark available at https://github.com/AgilData/tpcc. We only used the "stock" and "item" tables offered in this dataset. We show their schema in figure 12, the "stock" table being on the local machine and the "item" table on the remote one. The "item" table contains 10,000 tuples, which allows us to perform more at-scale experiments than our previous dataset.

In order to assess the scalability of RTA we pruned the dataset and measured the execution time of the query shown in figure 13 against different populations. This query also highlights the benefits of the *WHERE* and *IN* optimizations, as the *s.s_i_id = i.i_id* condition has a selectivity of 20% (we prune the "stock" table to ensure this), and the *MOD(i.i_id, 10) = 7* condition has a selectivity of 10%. Overall this query selects only 2% of the data.

5.4 Results

5.4.1 Validation. The average execution time of the query over five executions is reported in tables 1 and 2. As the *LOCAL* network environment does not use RTA we report these results only once, under the NAIVE label.

Let us first consider the effect of the network environment. The best performance is achieved by the *LOCAL* baseline, as RTA does not introduce any overhead, but of course this does not reflect an open relational data scenario. As expected in the other environments a significant communication overhead is introduced and the system performs better in higher throughput and lower latency environments. Still, we consider the introduced overhead reasonable, especially as such communication is inherent to open data.

We now compare the results based on the communication method between the local and remote machines. Method B introduces a significant overhead (18% overhead in average), which was to be expected as we introduce a middle man between the remote database and the local client and transform the database response into an intermediate representation that is then processed on the local machine. Still, we consider the overhead to be justified by the added security and scalability offered by such a system.

Table 1: method A, execution time in ms for the validation experiment

	NAIVE	*WHERE*	*IN*
LOCAL	482		
LAN	1731	889	665
WAN	1926	972	780
MOBILE	2685	1680	1190

Table 2: method B, execution time in ms for the validation experiment

	NAIVE	*WHERE*	*IN*
LAN	2240	1330	837
WAN	2608	1265	1108
MOBILE	2959	1807	1148

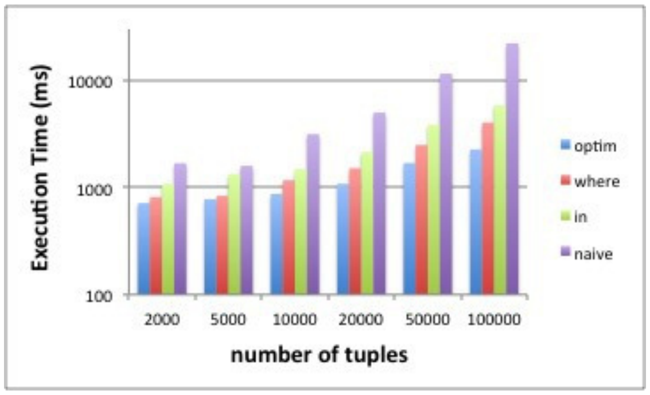


Figure 14: Execution time with method A, scalability experiment(DIRECT)

Finally, the results show that our proposed optimizations are efficient, as they reduce both the absolute execution time and the overhead of RTA compared to a local execution. These results show the importance of limiting the volume of communications for RTA's performance.

5.4.2 Scalability in data volume. The average execution time of the query over five executions is reported in figures **??** and **??**. We only report the results in the *LAN* network environment for space and readability, as the other results are similar. Full results are availbale at http://db.ics.keio.ac.jp/RTA/ideas2017.html.

The results show that the *NAIVE* implementation does not scale well. Indeed, the execution time of the query is nearly linear to the size of the remote table as the whole table is fetched every time. On the other hand, the mocked optimized versions perform much better, and significantly reduce the overhead of RTA. These results confirm the need for optimization in RTA and show its potential for working at large scales.

Furthermore, this experiment confirms that communication method B introduces a slight overhead. Nonetheless we consider that this overhead is acceptable, especially for the optimized versions of RTA.

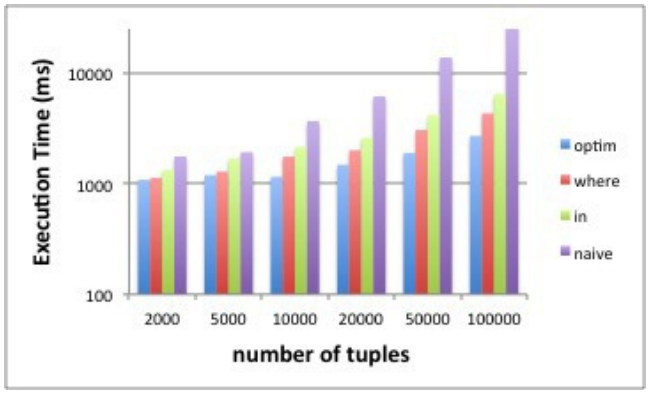


Figure 15: Execution time with method B, scalability experiment(WEB SERVICE)

6 CONCLUSION AND FUTURE WORK

In this paper we paved the way towards open relational data by proposing a framework for publishing relational data and integrating published data with local data. We implemented this framework as a prototype and validated it against two datasets. Our results show that through easy queries and with a reasonable overhead, remote published data can be integrated with local data.

Future work includes query optimization in order to limit communication between the local and remote machines and extension of our framework to cover different use cases and configurations. Finally, we will extend our tool to be compatible with more RDMS.

REFERENCES

[1] Tom Heath and Christian Bizer. Linked data: Evolving the web into a global data space. *Synthesis lectures on the semantic web: theory and technology*, 1(1):1–136, 2011.

[2] World Wide Web Consortium et al. Rdf 1.1 concepts and abstract syntax. 2014.

[3] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. Dbpedia: A nucleus for a web of open data. *The semantic web*, pages 722–735, 2007.

[4] Eric Prud, Andy Seaborne, et al. Sparql query language for rdf. 2006.

[5] Dean Arnold, Philip Cannata, Leigh Anne Glasson, Gary Hallmark, Bill McGuire, Scott Newman, Robert Odegard, and Harjit Sabharwal. Sql access: An implementation of the iso remote database access standard. *Computer*, 24(12):74–78, 1991.

[6] Amit P Sheth and James A Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys (CSUR)*, 22(3):183–236, 1990.

[7] Yogesh Simmhan, Roger Barga, Catharine van Ingen, Maria Nieto-Santisteban, Laszlo Dobos, Nolan Li, Michael Shipway, Alexander S Szalay, Sue Werner, and Jim Heasley. Graywulf: Scalable software architecture for data intensive computing. In *System Sciences, 2009. HICSS'09. 42nd Hawaii International Conference on*, pages 1–10. IEEE, 2009.

[8] Boyan Kolev, Carlyna Bondiombouy, Patrick Valduriez, Ricardo Jiménez-Peris, Raquel Pau, and José Pereira. The cloudmdsql multistore system. In *Proceedings of the 2016 International Conference on Management of Data*, pages 2113–2116. ACM, 2016.

[9] JSQLParser. https://github.com/JSQLParser/JSqlParser, 2011. Last accessed 28 March 2017.

[10] Philip A Bernstein, Nathan Goodman, Eugene Wong, Christopher L Reeve, and James B Rothnie Jr. Query processing in a system for distributed databases (sdd-1). *ACM Transactions on Database Systems (TODS)*, 6(4):602–625, 1981.

[11] M Tamer Özsu and Patrick Valduriez. *Principles of distributed database systems.* Springer Science & Business Media, 2011.