On The Performance Of Query Rewriting In Vertically Distributed Cloud Databases

Jens Kohler, Adrian Fiech, Kiril Simov, Thomas Specht

Abstract

Cloud Computing with its dynamic pay as you go and scalability characteristics promises computing on demand and cost savings compared to traditional computing architectures. This is a promising computing model especially in the context of *Big Data.* However, renting computing capabilities from a cloud provider means the integration of external resources into the own infrastructure and this requires a great amount of trust and raises new data security and privacy challenges. With respect to these still unsolved problems, this work presents an approach that uses traditional relational data models and distributes the corresponding data vertically across different cloud providers. So, every cloud provider only gets a small and logically independent chunk of the entire data, which is useless without the other parts. However, the distribution and the subsequent join of the data suffer actually from great performance losses, which are unbearable in practical usage scenarios. Therefore, we outline a query rewriting approach that parallelizes queries and joins in order to improve the performance again. We further implemented this approach based on the TPC-W benchmark and we finally present the performance results during in this work.

Keywords: Vertically Distributed Cloud Databases, Query Performance, Query Rewriting

I. Introduction

Cloud Computing with its dynamic pay as you go and scalability characteristics promises computing on demand and cost savings compared to traditional computing architectures. This is a promising computing model especially in the context of *Big Data.* Here, large computing capabilities can be rented on demand for analyzing, processing or storing *Big Data* volumes. Besides the increasing network bandwidth nowadays, new ways of storing such vast amounts of data (i.e. In-Memory) and new methods of representing information and knowledge (i.e. real-time analytics) have to be investigated. With respect to the latter topic, this work presents an approach that uses traditional relational data models and distributes the corresponding data vertically across different cloud providers. It has to be noted that renting computing or storage capabilities from external cloud providers requires the integration of a 3rd party into the entire architecture. It demands a great amount of trust when sensitive data should be stored at an external location somewhere in the cloud. This data security and protection challenges are still unsolved despite the fact that there are various approaches and research works that try to find adequate solutions. Basically, there are two directions that address such data security and privacy issues, firstly there is encryption of data and secondly, there is data distribution. Our work is based on security by data distribution, namely a vertical data distribution. In general, we vertically partition and distribute database data across various cloud providers in a way such that every provider only receives a small chunk of the data that is useless without the other chunks. This distribution suffers actually from a great performance loss, which is unbearable in practical usage scenarios. Thus, in this paper we present a distributed data access strategy that fetches data in parallel to increase the overall access performance. This query rewriting approach has the promising possibility to run the distributed queries in parallel in order to increase the overall query performance. We used a similar approach in one of our caching works [1] to build a client-based cache without adapting the queries but with transferring entire database tables into the cache.

We now start with a motivating example to illustrate our idea.

Motivating Example

To illustrate our approach in more detail we consider the following motivating example, based on the TPC-W CUSTOMER database table [2]:

CUSTOMER (C_ID, C_UNAME, C_PASSWD, C_FNAME C_LNAME, C_ADDR_ID, C_PHONE, C_EMAIL, C_SINCE, C_LAST_LOGIN, C_LOGIN, C_EXPIRATION, C_DISCOUNT, C_BALANCE, C_YTD_PMT, C_BIRTHDATE, C_DATA)

Basically, the table represents customer data that is necessary in a common web shop with a customer id (C_ID) as primary key. In our vertically distributed cloud setup, we store this table (containing i.e. sensitive data as year-to-day payment (C_YTD_PMT) or the current account balance (C_BALANCE)) logically distributed in two different public clouds. An exemplified distribution with 2 partitions could be stated as follows (whereas in a real world application scenario the distribution should consider that no sensitive columns are stored in the same partition):

CUSTOMER_p1 (C_ID, C_UNAME, C_PASSWD, C_FNAME C_LNAME, C_ADDR_ID, C_PHONE, C_EMAIL, C_SINCE)

CUSTOMER_p2 (C_ID, C_LAST_LOGIN, C_LOGIN, C_EXPIRATION, C_DISCOUNT, C_BALANCE, C_YTD_PMT, C_BIRTHDATE, C_DATA)

In this example the replicated primary key (C_ID) is used to *join* the customer data together. A user query is now formulated in *Hibernate Query Language* (HQL) to take advantage of the database abstraction of Hibernate. Thus, we are able to support different database systems with their different SQL implementations. It is even possible to use different database systems for the respective partitions simultaneously.

A sample query in Java could be formulated as follows:

```
List<Customer> customers = session.createCriteria(Customer.class).list();
```

This query clearly shows the main advantage of HQL, as it deals with objects rather than tables and relations. Above that, we use *Hibernate Interceptors* to adapt the HQL queries to our vertical distribution setup.

Our current implementation resulted in the SeDiCo framework (A *Se*cure and *Di*stributed *C*loud Data store). With this we demonstrated the technological feasibility of the presented vertical distribution. However, our approach is not feasible in practical real-world usage scenarios as it suffers from great performance losses [3]. At the moment we follow two different directions in order to accelerate the data query and manipulation. The first approach uses caching [4] and its variations, i.e. server-based, client-based, distributed caching etc. Our second approach follows the data distribution principle and distributes the queries against the databases as well [1]. This work follows the query distribution principle and aims at finding a generic way to rewrite queries such that they fit into our vertical distribution environment.

The main contributions of this paper can be summarized as follows:

- We analyze and measure the performance of distributed queries against vertically distributed cloud databases. Therefore, we implement the introductory example based on the TPC-W benchmark and its CUSTOMER table. We partition this table vertically and distribute the partitions across two different clouds.
- We regard this work as a foundation for our future work, in which we aim at finding a generic method that is able to rewrite arbitrary SQL queries and adapt them to our vertical distribution approach. Based on the performance results of this work, we are able to decide whether the query rewriting approach of this work is worth following any further.
- Finally, this work contains a detailed discussion and an analysis of how HQL and/or SQL can be transferred and rewritten such that queries can be adapted to vertically partitioned data models.

The remainder of this work is organized as follows: after this introduction we present other works that use different approaches in order to address similar challenges. We then define and formalize our problem in section III, before we present our approach and its implementation to tackle the problem of the slow data access in vertically distributed databases in section IV. In section V, we evaluate our Java-based implementation and present our achieved performance gain. We also contrast the results of this work to our previous works in section VI and we interpret the results and give an outlook to our future work plans.

II. Problem Formulation

The basic problem that we address in this work is the transformation of a database query into vertically distributed queries that run against vertically partitioned data sets in order to increase the overall query performance. We first have to mention that we concentrate our approach to conjunctive queries (i.e. queries that have their predicates connected with an AND) as they are the most common used form of queries in practical usage scenarios for both, OLTP and OLAP workloads [8].

The formalization of the problem according to [15] is as follows:

Our starting point is an HQL query:

```
List<Classname> objects = Hibernate.session.createQuery(
        "SELECT columnX, columnY FROM Classname
         WHERE columnX = 'x' AND columnY = 'y')                                    (1)
```

which can be formalized as:

$$\text{Objects (objectID, columnX, columnY) <-} \quad \pi_{\text{(objectID, columnX, columnY)}} (\sigma_{\text{(columnX=x AND columnY=y)}})^{\text{(TABLE\_NAME)}} \qquad (2)$$

The Hibernate framework then transfers this query into native standard SQL:

```
SELECT id, columnX, columnY
 FROM TABLE_NAME
 WHERE columnX = 'x' AND columnY = 'Y'.                                            (3)
```

In relational calculus this query is written as:

$$\pi_{\text{(columnX, columnY)}} (\sigma_{\text{(columnX=x AND columnY=y)}})^{\text{(TABLE\_NAME)}} \qquad (4)$$

Based on this query we aim at finding a generic rewriting rule, such that the query can be run against our vertically distributed database tables. Finally, the resulting query should have the following forms:

$$\text{Q1: ResultSet\_partition1 <-} \ \pi_{\text{(id, columnX)}} (\sigma_{\text{(columnX=x)}})^{\text{(TABLE\_NAME\_partition1)}} \qquad (5)$$

$$\text{Q2: ResultSet\_partition2 <-} \ \pi_{\text{(id, columnY)}} (\sigma_{\text{(columnY=y)}})^{\text{(TABLE\_NAME\_partition2)}} \qquad (6)$$

And the final result set (RS_{final}) of the query is the intersection of the distributed queries based on their unique identifies (id):

$$RS_{final} = \text{ResultSet\_partition1} \ \Pi \ \text{ResultSet\_partition2} \qquad (7)$$

The intersection described in (7) is a great advantage of conjunctive queries as non-conjunctive queries (e.g. query parameters that are connected with OR, or *average* or *sum* operations) use the union of the partitions. This shows that non-conjunctive queries are slower than conjunctive ones as optimization possibilities for the creation of a union are limited [8].

Another issue that we have to consider is the fact that Hibernate deals with objects instead of tables and relations. This is commonly referred to as *impedance mismatch* which is well-described and analyzed in [16]. Hence, it is not enough to just merge the result sets into an overall result set that meets all conjunctive criteria; we also have to create domain objects from the overall result set. Recall that query (1) uses the class name in the FROM clause instead of the table name and it also uses class properties instead of table columns as query parameters. Thus, we retrieve concrete domain objects instead of a generic result set. Nonetheless, the *Query-Parser* (Figure 1) analyzes and distributes the native SQL statement derived from the original HQL query (1) and here (in (3) and also in (5) and (6)) we receive generic result sets. So basically the last step is the transformation of the result set (RS_{final}) into a list of domain objects and this can be formalized as follows:

List<Classname> objects =

$$RS_{final} \leftarrow \pi_{(id,\ columnX,\ columnY)}\left(\sigma_{(columnX=x\ AND\ columnY=y)}\right)^{(TABLE\_NAME\_partition1,\ TABLE\_NAME\_partition2)} \quad (8)$$

Finally this step verifies that the merged result set RS_{final} is equal to the list of domain objects (List<Classname> objects), i.e.

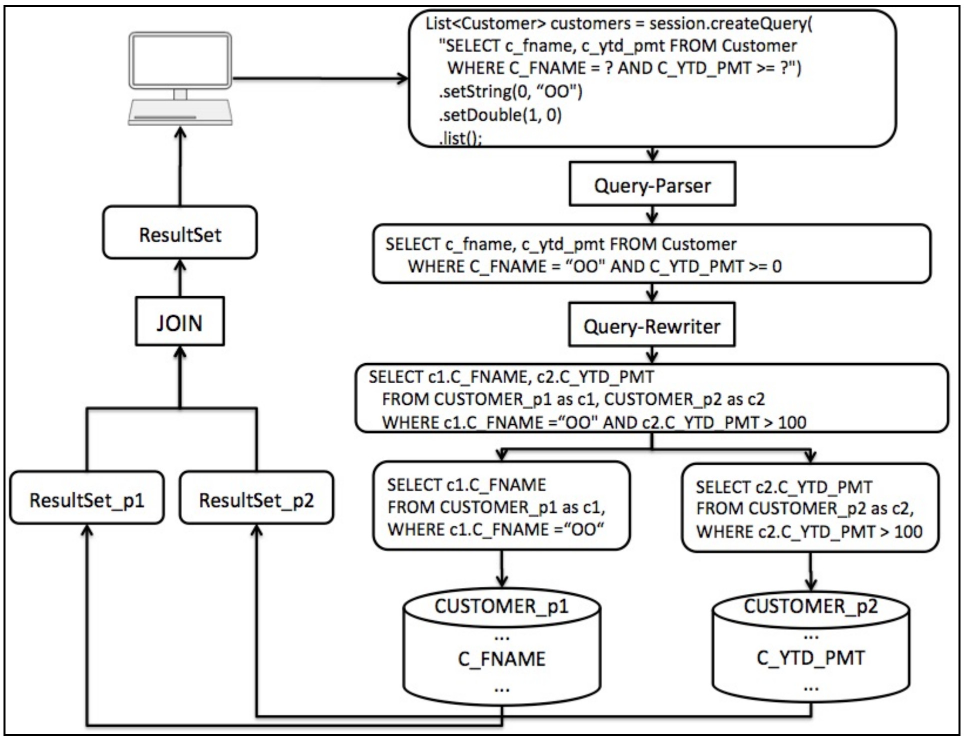
$$\text{List<Classname> objects} \equiv RS_{final} \quad (9)$$

which can be used to prove that the distributed queries return the same result as the non-distributed original query.

In the following section we now outline our approach to solve this formalized problem.

III. Approach

Figure 1 depicts our syntactic query rewriting approach with a concrete scenario derived from our motivating example above.



```
List<Customer> customers = session.createQuery(
  "SELECT c_fname, c_ytd_pmt FROM Customer
    WHERE C_FNAME = ? AND C_YTD_PMT >= ?")
  .setString(0, "OO")
  .setDouble(1, 0)
  .list();
```

```
SELECT c_fname, c_ytd_pmt FROM Customer
    WHERE C_FNAME = "OO" AND C_YTD_PMT >= 0
```

```
SELECT c1.C_FNAME, c2.C_YTD_PMT
  FROM CUSTOMER_p1 as c1, CUSTOMER_p2 as c2
  WHERE c1.C_FNAME ="OO" AND c2.C_YTD_PMT > 100
```

```
SELECT c1.C_FNAME
FROM CUSTOMER_p1 as c1,
WHERE c1.C_FNAME ="OO"
```

```
SELECT c2.C_YTD_PMT
FROM CUSTOMER_p2 as c2,
WHERE c2.C_YTD_PMT > 100
```

Figure 1: Query-Rewriting Approach

The starting point is a query formulated in either HQL or native SQL. In order to remain independent from a specific database vendor HQL queries should be preferred. The *Query-Parser* then analyzes the query syntactically and incorporates the actual query parameters. After that the *Query-Rewriter* parses the query and augments it with the partitioning information provided by the SeDiCo framework in form of an XML file. Based on this XML file the *Query-Rewriter* is able to decide which query predicate belongs to which partition and thus we are able to split the original query into sub-queries that then run against their corresponding partition. The entire approach is based on the assumption that this distribution has two substantial performance improvements. Firstly, the results are restricted faster and more easily as the potential overall result set is bounded to the number of tuples that both partitions return. As we deal with conjunctive queries, only the intersection (ResultSet_p1 Π ResultSet_p2, see Figure 1) of the two result sets have to be checked for the AND condition and not the entire data set. Secondly, the distributed queries and the join of the result sets are performed in parallel in different threads. Thus, the partitions are queried in two parallel threads and the result sets are merged (based on the AND condition) in parallel threads based on the number of available CPU cores of the querying client.

IV. References

[1] J. Kohler & T. Specht. “A Performance Comparison Between Parallel And Lazy Fetching in Vertically Distributed Cloud Databases.” In: *International Conference on Cloud Computing Technologies and Applications - CloudTech 2015*, Marrakesh, Morocco. 2015.

[2] TPC. “TPC Benchmark W (Web Commerce) Specification Version 2.0r.” 2003. http://www.tpc.org/tpcw/default.asp. [Accessed: 15-07-2015].

[3] J. Kohler & T. Specht. “Vertical Query-Join Benchmark in a Cloud Database Environment.” In: *Proceedings of 2nd World Conference on Complex Systems 2014*, Agadir, Marocco. 2014pp. 143–150.

[4] J. Kohler & T. Specht. “Performance Analysis of Vertically Partitioned Data in Clouds Through a Client-Based In-Memory Key-Value Store Cache.” In: *Proc. of The 8th International Conference on Computational Intelligence in Security for Information Systems*, Burgos, Spain. 2015.

[5] N. Hachani & H. Ounelli. “A Knowledge-Based Approach For Database Flexible Querying”, *17th International Conference on Database and Expert Systems Applications (DEXA’06)*. 2006.

[6] M. I. Hossain & M. M. Ali. “SQL query based data synchronization in heterogeneous database environment.” In: *2012 International Conference on Computer Communication and Informatics, ICCCI 2012*, Coimbatore, India. 2012pp. 1–5.

[7] C.-F. Liao, K. Chen, D. H. Tan, & J.-J. Chen. “Automatic query rewriting schemes for multitenant SaaS applications”, *Automated Software Engineering*. 2015. Vol. 1, No. 2015, pp. 1–34.

[8] A. Nash, L. Segoufin, & V. Vianu. “Views and queries”, *ACM Transactions on Database Systems*. Jul. 2010. Vol. 35, No. 3, pp. 1–41.

[9] G. Konstantinidis & J. L. Ambite. “Scalable Query Rewriting□: A Graph-Based Approach.” In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, New York, USA. 2011pp. 97–108.

[10] R. Chirkova, C. Li, & J. Li. “Answering queries using materialized views with minimum size”, *The VLDB Journal*. Oct. 2005. Vol. 15, No. 3, pp. 191–210.

[11] W. Z. Ming-Yee Lu. “Queryll: Java Database Queries Through Bytecode Rewriting.” In: *ACM/IFIP/USENIX 7th International Middleware Conference*, Melbourne, Australia. 2006pp. 201–218.

[12] MySQL. “MySQL 5.6 Reference Manual Including MySQL Cluster NDB 7.3 Reference Guide.” 2014. http://dev.mysql.com/doc/. [Accessed: 15-07-2015].

[13] Oracle. “Oracle® Database VLDB and Partitioning Guide 11g Release 1 (11.1).” 2007. http://docs.oracle.com/cd/B28359_01/server.111/b32024/partition.htm#i460833. [Accessed: 15-07-2015].

[14] B. Glavic & G. Alonso. “Perm: Processing provenance and data on the same data model through query rewriting.” In: *Proceedings - International Conference on Data Engineering*, Shanghai, China. 2009pp. 174–185.

[15] R. Elmasri & S. B. Navathe.*Fundamentals of Database Systems*, 7th editio. London, UK: Pearson Education, UK, . 2015.

[16] C. Ireland, D. Bowers, M. Newton, & K. Waugh. “A classification of object-relational impedance mismatch.” In: *Proceedings - 2009 1st International Conference on Advances in Databases, Knowledge and Data Applications, DBKDA 2009*, Gosier, Guadeloup. 2009pp. 36–43.

[17] T. Bezenek, T. Cain, R. Dickson, T. Heil, & M. Martin. “Characterizing a Java Implementation of TPC-W.” *Proc. of 3rd Workshop On Computer Architecture Evaluation Using Commercial Workloads (CAECW). HPCA Conference*. 2000. http://pharm.ece.wisc.edu/tpcw.shtml. [Accessed: 15-07-2015].