Faster Querying for Database Integration and Virtualization with Distributed Semi-Joins

Ramon Lawrence
University of British Columbia
Kelowna, BC, Canada
ramon.lawrence@ubc.ca

***Abstract*—Data integration and virtualization is commonly used to combine data for data analytics and reporting. A major challenge is handling large data sizes ("Big Data") as moving data across a network is extremely expensive and limits query processing. Business intelligence and data visualization software require rapid response times for users, and data virtualization is often limited for use cases involving joins across systems. The contribution of this work is a semi-join based approach to data virtualization joins that minimizes data movement and utilizes the extensive resources available in the database systems rather than performing query processing in the virtualization engine. The result is significantly less data movement which translates into faster query times and higher performance. Experimental results demonstrate that performance can be increased by an order of magnitude. A unique feature of the approach is that it does not require any special software installed above the database servers such as mediators and works directly using SQL queries.**

Keywords: semi-join, integration, distributed, virtualization, SQL query, analytics, Big Data

I. Introduction

Organizations have data distributed across many different data sources and frequently require data integration to combine data for reporting and planning. With the increasing popularity of do-it-yourself and just-in-time data analytics, it is often a challenge to get data in a suitable state for analysis. This is further compounded by large data volumes and diversity ("Big Data"). Data integration and virtualization allows several heterogeneous and diverse data sources to appear as one data source. Many data analysis tools, for example Tableau, support on-demand data virtualization by allowing users to write reports and queries that combine data from numerous sources yet display it in an integrated fashion. These tools are limited when the data volumes increase, especially when performing joins across sources, where the data must be extracted from the source and processed in the reporting engine.

This work adapts the semi-join operator for use in on-demand data virtualization scenarios. Semi-join [1] performs the join of two relations while preserving only the attributes of the first relation. It is useful when applied as a semi-join reducer to reduce the size of intermediate join results and data sent over the network. Semi-join was primarily applied in distributed databases [2] where the servers are coordinating together for query processing. For on-demand data virtualization, the servers are heterogeneous (e.g. MySQL and Microsoft SQL Server) with no collaboration. Access is via SQL commands that are coordinated by the virtualization server. The contribution of this work is an implementation and experimental evaluation of an SQL-based, semi-join for data virtualization. Experimental results demonstrate that utilizing the semi-join algorithm can improve query performance by an order of magnitude while at the same time reducing the load on the virtualization system by executing more of the query plan on the underlying data sources.

The organization of this paper is as follows. Section 2 contains background on semi-join and the requirements for on-demand data virtualization scenarios. Section 3 describes the semi-join algorithm and its variants. Experimental results are in Section 4, and the paper closes with future work and conclusions.

II. Background

Semi-join [1], [2] has been used for computing joins in distributed databases. A semi-join $R \ltimes S$ returns the tuples of R that match with S on the join condition. It is useful in distributed join processing as it may reduce the amount of data sent over the network. To compute $R \ltimes S$, the join keys of S are sent to the site with R for processing. This avoids communicating the entire relation R to perform the join. There has been research on query plans that can be improved with semi-joins and semi-join reductions [1], [2], and semi-joins were implemented in distributed database systems.

Semi-joins were expanded to 2-way semi-joins [3] to improve the reduction. Rather than sending the list of join keys, a compressed version can be used such as Bloom filters [4]. This reduces the amount of information transferred between the two sites with the addition of some local processing cost. Instead of a Bloom filter, PERF join [5] uses an ordered bitmap based on the order of the tuples in the relation to transmit join key information. This prevents collisions and is more efficient than using hashing but requires the tuples to be scanned in the same order.

Distributed querying consists of three phases [6]:

- local processing phase - performs selections, projections, and local joins on a source
- reduction phase - executes a series of semi-join reducers to reduce the size of the relations
- final processing phase - where all relations are sent to a processing site to combine to produce the query result

The use of semi-joins and semi-join reducers has reemerged with large-scaled distributed and cloud databases. There has been work on modifying the query optimizer to support semi-joins [7], and using semi-join reduction techniques for distributed and mediator systems [8] and large-scale parallel systems [9], [10]. In [10], track join is proposed that extends semi-join by generating an optimal transfer schedule for each join key. Further related work is the joining of data between a data warehouse and a distributed file system such as Hadoop [11]. In all cases, the goal of the optimization is to minimize data movement and process data where it resides.

This work applies semi-joins to on-demand, data virtualization scenarios required by ad hoc reporting and analysis tools. As described in [12], operational Business Intelligence (BI) allows the construction of dynamic reporting with low total cost of ownership by querying over a set of distributed and heterogeneous sources. Data transfer is the dominant cost in querying, and optimizers develop query plans to minimize data transfer. In many practical applications, the number of data sources can be quite large (greater than 5) for enterprise information integration queries. [12] provides details on modifying the query optimizer to use semi-join reduction using bind join operators [13]. A bind join performs a semi-join by executing as many parameterized queries on the source as there are distinct join key values. Bind joins do not scale if there are a large number of join key values as each value requires a separate query to be executed on the database.

A challenge with implementing a semi-join in a data virtualization architecture is that typically sources are heterogeneous and accessed via SQL using standardized interfaces such as ODBC/JDBC. Thus, any implementation of semi-join reduction must be executable without modifying the database system and achievable using SQL. This is a substantial difference from prior work on semi-joins implemented within a distributed database [2] or a large-scale parallel system using MapReduce [9]. There has been no prior work on implementing semi-joins by modifying SQL queries with the closest related work being bind joins.

III. SQL-based Semi-Join Algorithm

The SQL-based semi-join algorithm performs the typical steps of the semi-join algorithm. The description assumes two distinct, distributed SQL data sources, $db1$ and $db2$, capable of performing SQL queries. Assume $db1$ contains the smaller data set being joined. There is also a virtualization server responsible for coordinating query processing and final result production for the user. The semi-join steps are as follows:

- The virtualization server issues SQL query to $db1$ to extract data for the join.
- The virtualization server receives data extracted from $db1$ and sends data to $db2$.
- $db2$ processes the join and returns the result to the virtualization server.

Without performing a semi-join, the virtualization server would request data sets from both sources and perform the join itself. This is often not feasible if the data sets processed are large due to both cost of network transmission and local join processing cost. This variant of semi-join does not have $db1$ perform any query processing except for filters and local joins. The goal is to have the source with the larger data set perform all join processing and any further aggregation.

There are three variants of this SQL semi-join:

1) Result requires no attributes from $db1$ (small-side) - Semi-join is implemented by modifying the $db2$ SQL query to use an SQL IN clause which contains the join keys extracted from $db1$.
2) Result requires no attributes from $db1$ but has an outer join - Semi-join is implemented by modifying $db2$ SQL query to use a temporary table.
3) Result requires attributes from $db1$ - Semi-join is implemented by modifying $db2$ SQL query to use a temporary table or by performing final join processing on the virtualization server.

The following examples use the TPC-H schema. Consider this SQL query that illustrates case 1:

```
SELECT c_name,c_nationkey FROM db1.Nation
INNER JOIN db2.Customer
  ON c_nationkey=n_nationkey
WHERE n_name < 'GERMANY'
```

The source with the smaller relation, `Nation`, has no attributes in the final result. This query is executed by first extracting the data set from $db1$ using SQL:

```
SELECT n_nationkey FROM db1.Nation
WHERE n_name < 'GERMANY'
```

Then modifying the SQL query sent to $db2$ to be:

```
SELECT c_name, c_nationkey
FROM db2.Customer WHERE c_nationkey
  IN (0, 1, 2, 3, 4, 5, 6, 18)
```

The join is processed by $db2$. This approach is especially beneficial when the data set from $db1$ is small or when there is aggregation, in which case $db2$ can perform the join and aggregation, resulting in significantly less data transfer.

Since the join keys from $db1$ are sent to $db2$ via an SQL command, there are limits to the size of data in $db1$ that can be processed this way. These limits are bounded by the $db2$ command size. For example, the command size default for MySQL is 4 MB, but that can be increased to 1 GB.[1] Other databases have similar limits. PostgreSQL is 1 GB, and Microsoft SQL Server is 256 MB.[2] Depending on the join key size, this would allow in the order of tens of millions of rows from $db1$ to be sent to $db2$.

For cases involving outer joins, attributes required in the output from the small-side source, or larger data sets from the small-side source, other processing techniques can be used.

[1] https://dev.mysql.com/doc/refman/5.7/en/server-system-variables.html#sysvar_max_allowed_packet

[2] https://docs.microsoft.com/en-us/sql/sql-server/maximum-capacity-specifications-for-sql-server

If the query requires an attribute from the small-side source, the virtualization server can perform final join processing. Example query:

```
SELECT c_name, c_nationkey, n_name
FROM db1.Nation INNER JOIN db2.Customer
  ON c_nationkey=n_nationkey
WHERE n_name < 'GERMANY'
```

In this case, n_name would also be retrieved from $db1$ and stored in the virtualization server. The SQL sent to $db2$ is unchanged. To produce the final result set, the virtualization server joins each result row from $db2$ on n_nationkey to add n_name to each result row. This technique works okay for smaller $db1$ data sets but is not ideal if aggregation is performed after the join as then the aggregation may need to be applied at the virtualization server as well. Another approach is to use temporary tables. For data sets that are under the SQL command size limit, the temporary table can be introduced directly in the SQL statement. SQL query sent to $db2$:

```
SELECT c_name, c_nationkey, n_name
FROM customer, (SELECT
0 as n_nationkey, 'ALGERIA' as n_name
UNION ALL SELECT 1, 'ARGENTINA'
UNION ALL SELECT 2, 'BRAZIL'
UNION ALL SELECT 3, 'CANADA'
UNION ALL SELECT 4, 'EGYPT'
UNION ALL SELECT 5, 'ETHIOPIA'
UNION ALL SELECT 6, 'FRANCE'
UNION ALL SELECT 18, 'CHINA') N
```

Every database supports UNION ALL syntax, and many have support for multiple rows using the VALUES() clause, which allows for more space efficient encoding of the data in the SQL query. For larger data sets, a temporary table is created and insert statements are used to populate the data in $db2$ before the query is executed. There are limitations to this approach in many virtualization scenarios as the user may only have read access to the database, and creating temporary tables and inserting rows requires a fair amount of time. When possible, performance is improved by using a single SQL statement.

A. Analysis

The performance analysis of the SQL-based semi-join is similar to other semi-join algorithms. Let the relation produced by the SQL query on $db1$ be R and on $db2$ be S. Let the size of a relation R in bytes be denoted by $||R||$, and the number of tuples in R be denoted by $|R|$. The selectivity of the join is σ, which ranges from 0 (no tuples returned) to 1 (all tuples returned). The data transmission overhead of the semi-join is $2 * ||R||$ as R must be first transferred from $db1$ to the virtualization server then encoded in the SQL query sent to $db2$. The data transmitted is $2 * ||R|| + \sigma||S||$, which compares to $||R|| + ||S||$ for the regular join performed by the virtualization server. Even ignoring the significant local processing cost (CPU/memory) of executing at the virtualization server, the semi-join will be beneficial if the join is selective (i.e. σ is closer to 0).

For the second case, where attributes of R are required in the output, the data transmission is $2 * ||R|| + \sigma * (||R|| * |S| + ||S|| * |R|)$. There is still a potential benefit, but now the benefit may be reduced by transmitting attributes of R repetitively for each output row. Consider the previous example where n_name is a required field in the output. The overhead of transmitting n_nationkey, n_name to $db2$ for the join is small compared to the substantial cost of $db2$ returning n_name for each output row. In these cases, the optimizer must calculate the expected row sizes and may determine that it is better to perform the join on the virtualization server depending on the relative amount of semi-join reduction (σ) versus bytes added to each row transmitted.

The most significant benefit occurs when the semi-join is followed by other joins or aggregations in the query plan that can now be executed on $db2$. In this case, the output size is a function of the number of distinct grouping values and the size of each group row. As many analytical queries apply aggregation, using semi-joins may result in a data reduction of ten times or more.

B. Implementation

The semi-join algorithm is implemented in a Java-based virtualization system [14], [15]. The virtualization system contains a complete relational query engine and is capable of extracting data from any data source using JDBC. The virtualization system was modified to perform a post-optimization step on its query plan to detect potentially beneficial semi-join reductions and replaces joins planned for execution in the virtualization system with the semi-join. The system allows the user to explicitly request semi-join execution by extending the SQL outer join syntax with the key word SEMI JOIN:

```
FROM db1.Customer SEMI JOIN db2.Nation
  ON c_nationkey=n_nationkey
```

The optimizer uses the analytical cost functions and estimates selectivity and result sizes to determine when semi-join is beneficial. Multiple semi-joins may be used in a query plan.

IV. Experimental Results

Experiments were conducted with a TPC-H 10GB database[3] on two open source databases MySQL/MariaDB 5.5 and PostgreSQL 9.2. The database server machine was a dual-processor Intel Xeon E5620@2.4 GHz with 96 GB memory running RedHat Enterprise 7.4. The virtualization and reporting server was an Intel i7@2.8GHz with 16 GB RAM running Windows 7. The database server and virtualization server are connected via a high-speed local area network. All tests are the average of 5 runs.

The first experiment evaluates the most important use case that joins a smaller data set with a larger one. This happens frequently in reporting environments when a business has a large

[3]http://www.tpc.org/tpch/

data warehouse and BI users want to query it with additional information coming from smaller sources such as Microsoft Excel/Access and smaller databases in MySQL/PostgreSQL. Without semi-joins, the virtualization server may need to extract a significant amount of data from the warehouse and perform costly aggregation. The test query used is:

```
SELECT C.* FROM db1.Nation
INNER JOIN db2.Customer
  ON c_nationkey=n_nationkey
WHERE n_name < ?
```

The parameter value is varied such that the number of nations returned ranges from 0 to 25 representing a selectivity of 0 to 1 in steps of 0.04. The results are in Figure 1, and almost perfectly match the analytical formula as query cost is dominated by data transmission. There is a similar pattern for queries involving any number of attributes of `Customer`. Note that the data sets for `Postgres join` and `MySQL join` represent the query plan that has two queries that extract data from the underlying database and the join is performed on the virtualization server.

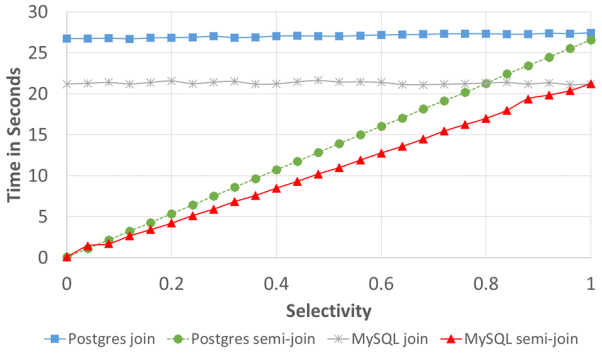

Fig. 1. Semi-join Performance for MySQL/PostgreSQL

The benefits are large when aggregation is required. Consider the query below and results in Figure 2. Computing the aggregation at the source *db2* saves most data transfer as well as being more efficient for the virtualization server.

```
SELECT c_mktsegment, SUM(c_acctbal),
       COUNT(*)
FROM db2.Customer INNER JOIN db1.Nation
  ON c_nationkey = N.n_nationkey
WHERE n_name < ?
GROUP by c_mktsegment
```

This query tested sending large data sets:

```
SELECT O.* FROM db1.Customer
INNER JOIN db2.Orders
 ON C.c_custkey=O.o_custkey
WHERE c_acctbal < ?
```

In this query, the number of join keys ranges between 0 and 1.5 million. The filter on *c_acctbal* was varied to allow for different selectivities. The results are in Figure 3. Both

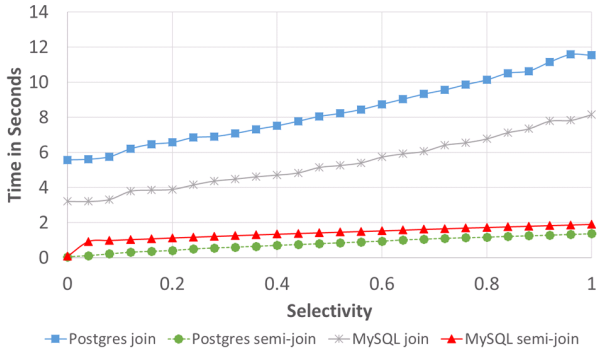

Fig. 2. Semi-join Performance with Aggregation Query

databases are able to handle a large SQL statement containing the smaller data set, and processing the join on *db2* is a benefit for almost all cases. This query also demonstrates the relative performance difference of MySQL and PostgreSQL. MySQL data transfer through its driver is more efficient than with PostgreSQL which has longer overall times. PostgreSQL consistently executes the SQL plan, and the semi-join performance is linear with the number of join keys. For MySQL, the optimizer initially selects a less efficient plan that explains the poor performance around selectivity 0.1 to 0.15.

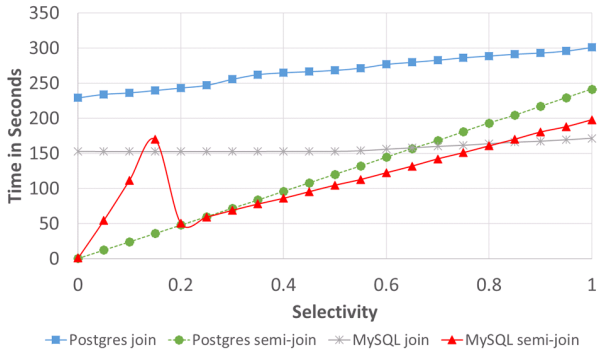

Fig. 3. Semi-join Performance with Large Data

The aggregation query below is for large data aggregation with results in Figure 4.

```
SELECT o_orderstatus, COUNT(*),
       SUM(o_totalprice)
FROM db1.Customer INNER JOIN db2.Orders
 ON C.c_custkey=O.o_custkey
WHERE c_acctbal < ?
GROUP BY o_orderstatus
```

When an aggregation is performed after the join, the virtualization server incurs both the cost in data transmission from the data sources as well as the cost of executing the join and aggregation. Aggregating at the data source reduces data transmission and is often more efficient as the database server typically has higher performance. In Figure 4, MySQL again has a poor optimization of the query for selectivities between

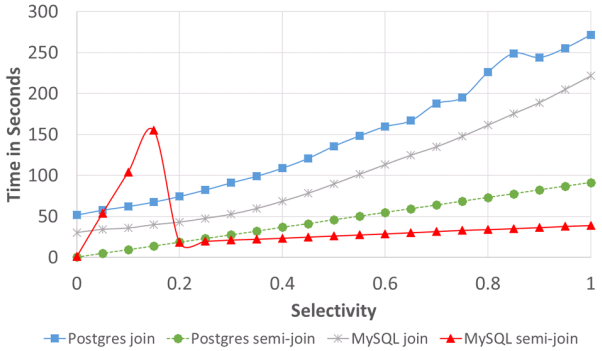

Fig. 4. Performance with Large Aggregated Data

0.1 and 0.15. The semi-join query performance improvement is due to sending less data to the virtualization server as aggregation is performed at the database.

Using temporary tables within the SQL query was evaluated using:

```
SELECT n_name, SUM(c_acctbal), COUNT(*)
FROM db2.Customer INNER JOIN db1.Nation
 ON C.c_nationkey = N.n_nationkey
WHERE n_name < ?
GROUP by n_name
```

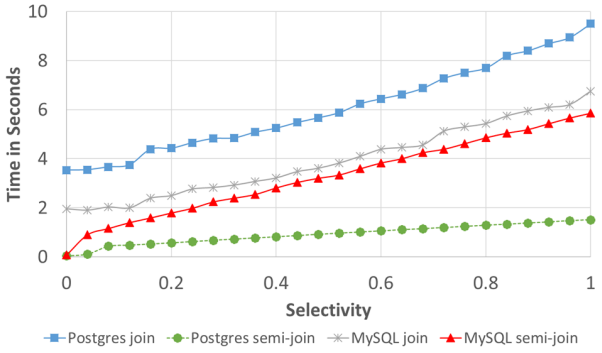

Fig. 5. Performance with Temporary Table

When using a temporary table, the results are more dependent on the local database query optimizer. For Postgres, there is always a significant benefit for performing the semi-join. For MySQL, it is less efficient performing GROUP BY so the time is not much faster than having the virtualization server perform the join and aggregation.

Overall, there is a benefit for performing semi-join for a wide-range of queries rather than doing the join at the virtualization server. Semi-join is more scaleable and reduces the query processing load on the virtualization server allowing for more queries and larger data queries to be processed.

V. Conclusions

This work introduced an SQL-based semi-join implementation for use in data virtualization and business intelligence reporting. The algorithm modifies SQL statements sent to the data sources to reduce the amount of data transferred without requiring modifications to the database system. This reduces the load on the data virtualization system and allows for handling much larger queries and data sizes as query processing occurs at the data sources rather than the virtualization engine. Experimental results demonstrate that the semi-join approach can reduce data transfer and improve query processing times by an order of magnitude. The approach is especially well-suited for on-demand business intelligence reporting applications processing Big Data, which are currently limited for large data sizes. Future work will improve the optimizer and expand the implementation and testing for more databases and queries.

References

[1] P. A. Bernstein and D. W. Chiu, "Using semi-joins to solve relational queries," *J. ACM*, vol. 28, no. 1, pp. 25–40, 1981. [Online]. Available: http://doi.acm.org/10.1145/322234.322238

[2] P. Valduriez and G. Gardarin, "Join and semijoin algorithms for a multiprocessor database machine," *ACM Trans. Database Syst.*, vol. 9, no. 1, pp. 133–161, Mar. 1984. [Online]. Available: http://doi.acm.org/10.1145/348.318590

[3] N. Roussopoulos and H. Kang, "A Pipeline N-way Join Algorithm Based on the 2-way Semijoin Program," *IEEE Trans. Knowl. Data Eng.*, vol. 3, no. 4, pp. 486–495, 1991. [Online]. Available: https://doi.org/10.1109/69.109109

[4] J. K. Mullin, "Optimal semijoins for distributed database systems," *IEEE Trans. Softw. Eng.*, vol. 16, no. 5, pp. 558–560, May 1990. [Online]. Available: http://dx.doi.org/10.1109/32.52778

[5] Z. Li and K. A. Ross, "PERF Join: An Alternative to Two-way Semijoin and Bloomjoin," in *CIKM '95*. ACM, 1995, pp. 137–144. [Online]. Available: http://doi.acm.org/10.1145/221270.221360

[6] M. Chen and P. S. Yu, "Combining join and semi-join operations for distributed query processing," *IEEE Trans. Knowl. Data Eng.*, vol. 5, no. 3, pp. 534–542, 1993. [Online]. Available: https://doi.org/10.1109/69.224205

[7] K. Stocker, D. Kossmann, R. Braumandl, and A. Kemper, "Integrating semi-join-reducers into state of the art query processors," in *ICDE 2001*, 2001, pp. 575–584. [Online]. Available: https://doi.org/10.1109/ICDE.2001.914872

[8] V. Josifovski, T. Katchaounov, and T. Risch, "Evaluation of join strategies for distributed mediation," in *ADBIS 2001*, 2001, pp. 308–322. [Online]. Available: https://doi.org/10.1007/3-540-44803-9_24

[9] J. Daenen, F. Neven, T. Tan, and S. Vansummeren, "Parallel evaluation of multi-semi-joins," *PVLDB*, vol. 9, no. 10, pp. 732–743, 2016. [Online]. Available: http://www.vldb.org/pvldb/vol9/p732-daenen.pdf

[10] O. Polychroniou, R. Sen, and K. A. Ross, "Track join: distributed joins with minimal network traffic," in *SIGMOD 2014*, 2014, pp. 1483–1494. [Online]. Available: http://doi.acm.org/10.1145/2588555.2610521

[11] Y. Tian, F. Özcan, T. Zou, R. Goncalves, and H. Pirahesh, "Building a hybrid warehouse: Efficient joins between data stored in HDFS and enterprise warehouse," *ACM Trans. Database Syst.*, vol. 41, no. 4, pp. 21:1–21:38, 2016. [Online]. Available: http://doi.acm.org/10.1145/2972950

[12] N. Dieu, A. Dragusanu, F. Fabret, F. Llirbat, and E. Simon, "1,000 tables under the form," *Proc. VLDB Endow.*, vol. 2, no. 2, pp. 1450–1461, Aug. 2009. [Online]. Available: https://doi.org/10.14778/1687553.1687572

[13] I. Manolescu, L. Bouganim, F. Fabret, and E. Simon, "Efficient querying of distributed resources in mediator systems," in *CoopIS 2002*, 2002, pp. 468–485. [Online]. Available: https://doi.org/10.1007/3-540-36124-3_27

[14] T. Mason and R. Lawrence, "Dynamic Database Integration in a JDBC Driver," in *ICEIS*, 2005, pp. 326–333.

[15] R. Lawrence, "Integration and Virtualization of Relational SQL and NoSQL Systems Including MySQL and MongoDB," in *Computational Science and Computational Intelligence*, 2014, pp. 285–290.