 CMES, vol.113, no.2, pp.171-181, 2017

Uniform Query Framework for Relational and NoSQL Databases

J.B. Karanjekar[1] and M.B. Chandak[2]

Abstract: As the data managed by applications has evolved over the years with the arrival of Web 2.0, a large number of new databases have been developed to manage various types of data. While the traditional relational databases continue to exist, NoSQL databases which are document oriented or key-value stores or columnar continue to evolve and are embraced very rapidly across the industry. It is not just the type of data handled by these databases that is different but also the query language they use is also different. This paper talks about a uniform query framework that can be used for traditional relational databases and NoSQL databases. This query framework can also perform joins, aggregates, filter on data from various data sources in a single query.

Keywords: NoSQL, RDBMS, data tetrieval, inter-database joins, SQL, Cassandra, MongoDB, Calcite, MySQL.

1 Introduction

The rise of cloud computing and Web 2.0 has opened an ongoing debate regarding the pros and cons of using traditional Relational Database Management Systems or document and key-value based NoSQL (Not-Only SQL) implementations. While the NoSQL databases are being embraced at rapid pace, no one is rejecting the relational databases just yet. What has helped NoSQL databases gain the popularity is its ability to scale out horizontally which would not have been possible without cloud computing. Amazon's Elastic Compute Cloud (EC2) on the AWS platform was released in 2006 and took advantage of low-cost servers and high-bandwidth networks first offered by Grid and Utility computing methods [Foote (2017)]. The merits of reliability, scalability and accessibility of data quickly led the public to see the values of using Infrastructure-as-a-Service over dedicated or shared hosting solutions, and several copycat companies have sprung up in recent years as a result.

For database administrators, the challenge has become choosing which among a wide variety of emerging database types to implement when hosting in the cloud. The relational model has its advantages in documentation, simplicity, familiarity, data integrity and reliability. SQL implementations in the cloud are capable of far more

[1] Shri Ramdeobaba College of Engineering and Management, Nagpur, India and Izel Technologies Pvt Ltd., Pune, India

[2] Shri Ramdeobaba College of Engineering and Management, Nagpur, India

complex queries and aggregates than other solutions [Cattell (2010)]. They support transactions to ensure only atomic changes are made to your data while keeping a single master copy by which all other copies replicate. The drawbacks come when a developer needs to scale outward across multiple servers. Standard RDBMS's have trouble with efficiently expanding a database due to their inherent complexity of organization [De Witt, Floratou, Patel, Teletia and Zhang (2012)].

On the other hand, SQL implementations are either key-value based, columnar or document-based and have grown in popularity because of their ease of scalability. NoSQL implementations are based on entities and support many of the functions of RDBMS such as sorting, indexing, projecting and querying but are not as reliable or effective when using complex database models. Transactions that guarantee atomic consistency are not supported and updating the database across multiple entities is an eventual process. Joins and ACID guarantees are traded off in favour of transaction speed. Scalability is where NoSQL earns its edge wherein quick ability to scale out or shrink the database allows administrators to have flexibility to meet their requirements [Chi, Pierre and Wei (2012)].

One of the challenges faced by NoSQL database family is lack of standards in NoSQL space. For so many years before NoSQL gained momentum, there was just one query language that was good enough for all the databases. Now there are many flavours trying to establish their own standards. Through this paper we are presenting a framework with which traditional SQL style of query mechanism can be implemented for RDBMS and NoSQL database systems. This framework also allows users to query different databases through a single query and perform operations such as joins, aggregation, nested queries and filters.

In this paper we are focusing on providing a uniform query framework for four data storage implementations which are MySQL, MongoDB, Cassandra and CSV files. MySQL has become one of the most popular open-source implementations of RDBMS for online applications. MySQL is highly efficient with complex structured data models and queries. MongoDB is currently the most popular NoSQL solution largely because of its simplicity of use and efficiency in clustering data. Unstructured data such as that gathered by social media websites is presumed to be best handled by NoSQL implementations. Cassandra is a massively scalable columnar NoSQL database which provides users the ability to store large amounts structured and semi-structured data. Cassandra's ability to scale out across multiple data centers is what make companies turn to Cassandra to store their data in the cloud.

2 Related work

The comparison between traditional RDBMS and NoSQL database has been widely studied in the distributed systems and database communities. While RDBMS has SQL, there are many different mechanisms developed to retrieve data from NoSQL databases. For instance, Cassandra provides a python based cqlsh utility [Cqlsh] as well as libraries and modules for different languages such as Python, JAVA, Go, C++, etc. [Cassandra Client Drivers]. Similarly, MongoDB has the mongo shell to query data and also drivers

and client libraries for almost all popular programming languages [MongoDB Drivers]. MongoDB also provides connectors for Hadoop.

However, the query language used by these databases is different. MySQL uses SQL, Cassandra uses Cassandra Query Language (CQL) and MongoDB uses its own JSON query syntax (representing queries as JSON). No doubt there are tremendous advantages and convincing reasons why these databases use these different query languages, but one question does come up again and again why there is no native support for a standard query language.

Couchbase community has developed a database query language called N1QL [N1QL] which extends the SQL for JSON objects. However, it does not support columnar database like Cassandra. As a result, there is a need for a uniform framework for a standard query mechanism for RDBMS as well as NoSQL databases.

3 Architecture

The main component of the architecture (Fig. 1) is the framework layer that provides the management framework for the entire setup. The various database systems should be connected to the framework layer. The nodes of any particular database systems can be added and removed from the setup without impacting other database systems.

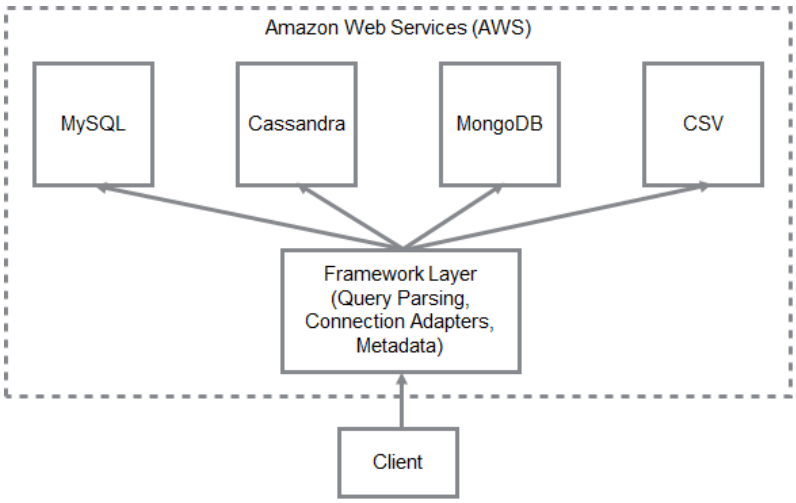

Figure 1: System architecture

The framework layer is a dynamic data management system. It contains many of the components that comprise a typical database management system but it deliberately omits components like storage of data and algorithms to process the data from storage. It is intentionally kept out of the business of storing data as it is only meant for mediating between different data storage and processing engines. It is responsible for maintaining metadata, accepting requests from clients, processing the queries, and sending the results back to the client.

To understand the architecture in detail we will discuss about connections adapters, schema discovery, query parsing and optimization in following sections.

3.1 Connection adapters

The framework layer maintains the connection configuration for all the databases in the setup as shown in Fig.2. It is stored in the form of JSON document which contains connection details for the database, relevant schemas/keyspaces and the schema factory for Apache Calcite [Apache Calcite]. A schema factory for a given database system initializes a schema for that system in the framework. At the initialization of the framework, these adapters are used for database connections and schema initializations.

```
{
  version: '1.0',
  defaultSchema: 'mysql',
  schemas: [
    {
      name: 'mysql',
      type: 'custom',
      factory: 'org.apache.calcite.adapter.jdbc.JdbcSchema$Factory',
      operand: {
        jdbcDriver: 'com.mysql.jdbc.Driver',
        jdbcUrl: 'jdbc:mysql://ec2-54-218-116-96.us-west-2.compute.amazonaws.com:3306/research_test',
        jdbcUser: 'user',
        jdbcPassword: 'password'
      }
    }
  ]
}
```

Figure 2: Sample configuration file for MySQL

3.2 Schema discovery

The framework uses Apache Calcite for schema discovery [Apache Calcite]. When we define the connection adapters, the schema factory and connection details are used to scan the tables available in that data source. When a query is parsed and it is planned to use certain tables, Calcite reads those tables and performs required operations.

We are using connection adapters and schema factories for Cassandra, CSV, MongoDB and MySQL. MySQL adapter uses JDBC schema factory which is powered by Apache Avatica [Apache Avatica].

To elaborate how schema discovery takes place let's take an example of CSV schema factory. When the schema factory is initialized with details of CSV model, it initializes a schema object using the path of the directory where CSV files are located. The jobs of this schema object are to produce a list of tables. The schema scans the directory and finds all files whose name ends with ".csv" and creates tables for them. The table objects in turn implement Calcite's "Table" interface. This interface reads the data from CSV files when a query is parsed and data is to be fetched from these files.

3.3 Query flow

A big advantage of the framework is its support for the industry-standard SQL query language. It does not matter what query retrieval mechanism the underlying data source supports. The SQL query is first received by the framework layer as shown in Fig. 3 which then parses it, optimizes it, and identifies the downstream databases that are required to process the query. Once the databases are identified, the "Table" interface in Calcite corresponding to those databases read the portion of the data that is required to fulfil the query. When the data is read from individual nodes, the framework layer performs final processing on the data and streams the result back to the client.

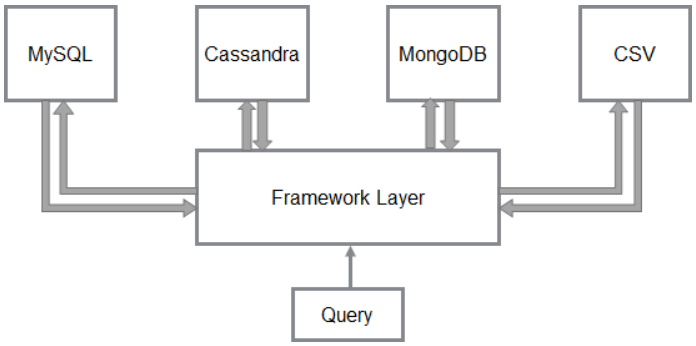

Figure 3: Query flow

3.4 Query parsing and optimization

The framework layer uses Calcite, an open source framework, to parse incoming queries [Apache Calcite]. When the query first comes to Calcite, it parses the query and converts into relational algebra. The output of the parser component is a language agnostic, computer-friendly logical plan that represents the query.

When it comes to reading a table, the "Table" interface we talked about in sessions 3.2 and 3.3 would work fine for small and moderate sized tables. However, if a table has a hundred columns and a million rows, we would not like to fetch all the data for every query. We would want to optimize the operation by finding an efficient way to access the required data. The optimization is achieved by applying planner rules that transform the query based on its relational algebra. Calcite applies the rules on the relational algebra of the query in order to transform it into a form that has low cost.

To elaborate query optimization with an example, consider the following query which exhibits federated join of a table named "weekly_sales" from MySQL and another named "stores" from Cassandra.

```
SELECT ca."store_type", SUM(my."weekly_sales")
FROM   "mysql"."weekly_sales" my, "cassandra"."stores" ca
WHERE  my."store_number" = ca."store_number"
    AND my."week_date" = '2011-11-25'
GROUP  BY ca."store_type";
```

After parsing, the query plan would look like as shown in Fig. 4 where the data from both the sources would be joined and Filter operation would be performed on the combined data.

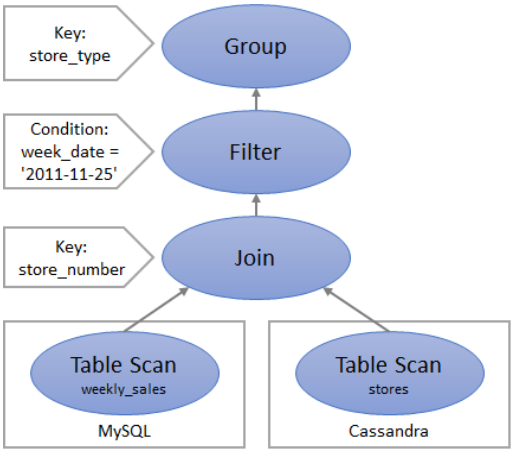

Figure 4: Execution plan before optimization

Above query plan produces the result as expected but the challenge that Calcite helps in overcoming when joining two different data sources is the optimization using transformation rules or the planner rules. We want to achieve optimization wherein the query produces the correct result and does it as efficiently as possible with lower cost.

When the query transformation rules are applied, the plan changes as shown in Fig 5. In this example the filter is pushed to MySQL database before the join. There are two transformation rules that come into play here. One is you can push a filter into an input of an inner join if the filter does not reference columns from the other input. And the second is you can push the filter down to the data source if the data source is independently able to execute the filters.

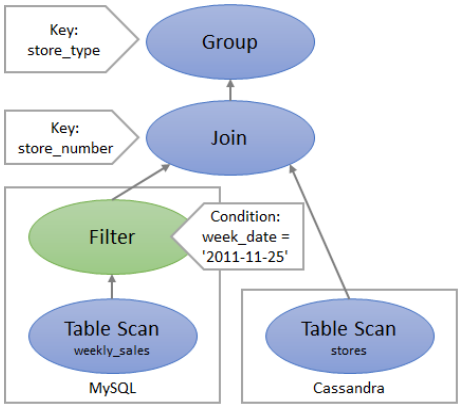

Figure 5: Execution plan after optimization

Calcite optimizes queries by repeatedly applying planner rules to a relational expression. A cost model guides the process, and the planner engine generates an alternative expression that has the same semantics as the original but a lower cost.

4 Data model

Keeping in view the strengths and weaknesses of relational and NoSQL databases, we are using both highly structured data model and also semi-unstructured data model. For structured data we are using historical sales data released by Walmart for 45 of their stores located in different regions [Walmart Store Sales Data] as shown in Fig. 6 and for unstructured data we are using a dataset containing user comments released by Reddit [Reddit Comments May 2015 Data]. We are using a separate AWS EC2 instance for each of the databases (MySQL, Cassandra and MongoDB) and identical sample data is uploaded to each one of them.

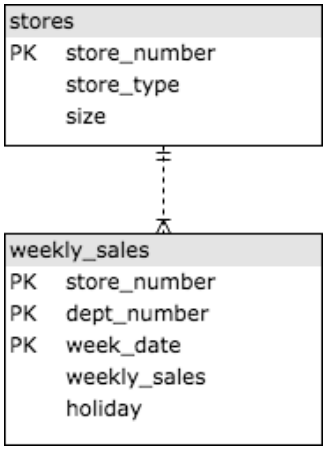

Figure 6: Entity relation model for structured data

5 Query execution

In this paper we discuss the core ideas in the context of read-only database operations using SELECT queries with joins, aggregates, filters and nested queries. These operations are passed through Calcite's algebra builder in which every query is translated to relational algebra and represented as a tree of relational operators [Apache Calcite]. The query planner rules transform the expression trees using mathematical identities that preserve semantics. For example, it is valid to push a filter into an input of an inner join if the filter does not reference columns from the other input.

Optimization of queries take place as mentioned in Section 3.4 by repeatedly applying planner rules to a relational expression. As mentioned in the Schema Discovery section, once the query is parsed and it decides which tables to use from each database, Calcite

invokes the table objects to read the data and performs required operations.

6 Database operations and results

In the following sections we will see the SQL queries that are used to operate upon the dataset we have uploaded to our databases. The SQL queries would fetch the data from all the databases mentioned earlier namely MySQL, Cassandra, MongoDB and CSV. These queries are similar to real-world queries used to solve practical problems. With this framework, they would perform inter database joins, filters, aggregates and nested queries to get required data.

6.1 Inter-database join

To demonstrate inter-database join, we use following query to find out store_number, store_type and the size of the store where sale has gone beyond USD 500,000 in any of the given weeks. In this query, an equijoin is performed between a collection named weekly_sales from MongoDB and a table named stores in MySQL database.

```
SELECT my."store_number", my."store_type", my."size"
FROM   "mongo"."weekly_sales" mo, "mysql"."stores" my
WHERE  mo."weekly_sales" > 500000
    AND mo."store_number" = my."store_number";
```

This query took 286 milliseconds. The schema names mongo and mysql mentioned in the query refer to the schema names with which MongoDB and MySQL databases were configured in the connection adapters respectively.

6.2 Aggregates

Aggregate functions such as SUM, AVG, COUNT, MIN, MAX are supported by the framework. In the following example we try to find out the SUM of weekly_sales grouped by store_type on the week ending on 2011-11-25 (a day after Thanksgiving). For this example, we use the combination of Cassandra and MySQL database.

```
SELECT ca."store_type", SUM(my."weekly_sales")
FROM   "mysql"."weekly_sales" my, "cassandra"."stores" ca
WHERE  my."store_number" = ca."store_number"
    AND my."week_date" = '2011-11-25'
GROUP  BY ca."store_type";
```

This query took 363 milliseconds. The schema names cassandra and mysql mentioned in the query refer to the schema names with which Cassandra and MySQL databases were configured in the connection adapters respectively.

6.3 Filters

Filter is a relatively simple operation however; we are documenting it in this paper to demonstrate how it can be applied even to CSV files. Almost all database systems support filters but with this framework, it can be applied to CSV files too with the WHERE clause.

```
SELECT *
FROM   "csv"."weekly_sales"
WHERE  "weekly_sales" > 500000.0;
```

In this example we are filtering weekly_sales greater than USD 500,000. This query took 433 milliseconds. The schema name csv refers to the schema name with which CSV file is configured in the connection adapter.

6.4 Nested queries

The nested queried are supported as shown below. In this example we try to find out the difference of weekly_sales between type A and type B stores on the special holiday weeks.

```
SELECT a - b
FROM   (SELECT SUM(CAST(csv."weekly_sales" AS FLOAT)) AS a
       FROM   "csv"."weekly_sales" csv,
              "mongo"."stores" mo
       WHERE  mo."store_number" = CAST(csv."store_number" AS INT)
              AND csv."holiday" = 'TRUE'
              AND mo."store_type" = 'A') AS x,
      (SELECT SUM(CAST(csv."weekly_sales" AS FLOAT)) AS b
       FROM   "csv"."weekly_sales" csv,
              "mongo"."stores" mo
       WHERE  mo."store_number" = CAST(csv."store_number" AS INT)
              AND csv."holiday" = 'TRUE'
              AND mo."store_type" = 'B') AS y;
```

This query took 2242 milliseconds. In this query, the sub-queries find out the sum of weekly_sales for a given store_type on special holiday weeks. The output of the sub-queries is used to find the difference in sales. The two tables used in this query come from two different databases. The holiday data comes from CSV whereas store_type comes from MongoDB.

You may also notice that we are using CAST function to convert store_number and weekly_sales into INT and FLOAT respectively. This is because these columns come from CSV where the data type is not explicitly defined and every column is considered TEXT by default. Thus, casting them to required data type is necessary for joins and aggregate functions.

6.5 Queries on semi-structured data

A mentioned earlier, we are using the dataset containing user comments released by Reddit as semi structured data. It is a single database table containing information about the user and the comment that he has logged. This data is loaded in all four database systems in question namely MongoDB, Cassandra, MySQL and CSV.

To demonstrate a query on this semi structured data, we will try to find out comments where the string 'happy people' is mentioned.

```
SELECT *
FROM   <schema>."reddit_comments"
WHERE  "body" LIKE '%happy people%';
```

This query works on all the database sources in question when appropriate schema is specified in the query.

7 Conclusions

Considering above data operations using SQL queries it has been demonstrated that a uniform query framework has been developed to fetch data from multiple heterogeneous database systems. These systems can be traditional relational databases, or NoSQL systems or even flat files such as CSV. With this framework, a layer of abstraction is provided and the user does not have to worry about the complexities of the underlying database system and the query mechanism used natively on that system. The user can directly use the SQL statements to fetch data from various sources.

Database operations such as filtering, aggregates, joins and nested queries are easily and seamlessly possible with this framework across database systems. It should also be noted in all the above examples that some of the operations which are not natively allowed in NoSQL databases can be carried out in this framework. For instance, Cassandra does not allow filtering by default. However, with this framework filtering can be performed without any issue. Similarly, MongoDB does not allow joins. However, when we query MongoDB collections using this framework, we can easily join them not only with other collections but also with objects from other databases.

There is a scope for analysis and research on the performance aspect of the framework and a comparative study against the individual database systems is needed.

References

Apache Avatica Documentation: https://calcite.apache.org/avatica/docs/.

Apache Calcite Documentation: https://calcite.apache.org/docs/.

Cassandra Client Drivers. *Cassandra Documentation:* http://cassandra.apache.org/doc/latest/getting_started/drivers.html.

Cattell, R. (2010): Scalable SQL and NoSQL Data Stores. *ACM SIGMOD Record*, vol.39, no. 4, pp.12-27, doi:10.1145/1978915.1978919.

Chi, C.; Pierre, G.; Wei, Z. (2012): Scalable Join Queries in Cloud Data Stores. *2012*

12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, pp. 547-555, doi:10.1109/CCGrid.2012.28.

Cqlsh: *CQL shell. Cassandra Documentation*: http:// cassandra. apache. org/doc/ latest/ tools/cqlsh.html.

DeWitt, D.; Floratou, A.; Patel, J.; Teletia, N.; Zhang, D. (2012): Can the Elephants *Handle the NoSQL Onslaught? Proceedings of the VLDB Endowment*, vol.5, pp.12, pp. 1712-1723.

Foote, K. D. (2017). *A Brief History of Cloud Computing*: http:// www. dataversity. net/ brief-history-cloud-computing/.

MongoDB Drivers and Client Libraries. *MongoDB Documentation*: https://docs. mongodb.com/manual/applications/drivers/.

N1QL: *What is N1QL? Couchbase*: https://www.couchbase.com/products/n1ql.

Reddit Comments May 2015 Data: https://www.kaggle.com/reddit/reddit-comments-may-2015.

Walmart Store Sales Data: https://www.kaggle.com/c/walmart-recruiting-store-sales-forecasting/data.