Middleware for Supporting “Big Data” Analytics across a Database Cluster

Daniel Salowe, Shayan Patel, Sahil Shah

Advisor: Zachary Ives, PhD
Abstract

The lack of affordable and open-source support for efficient, big data analytics across a relational database cluster is a problem in the technology industry. Specifically, there exists a need for an open-source distributed database middleware for off-the-shelf databases (MySQL). We approached the project by first deciding on the most important needs of a middleware addressing these issues. This project focuses on three main components, namely fault tolerance, distributed JOIN operations, and support for computationally intensive operations. By focusing on these three issues, we have developed an infrastructure that future developers can build upon, while also maintaining usefulness as a standalone middleware. The system detects node failures and automatically reroutes queries to replica nodes. Our distributed JOIN operation leverages nodes as computation resources to achieve efficient JOINs. The middleware works the best in situations where computation-heavy operations are to be applied on relevant data before or after the SQL operations. Our system distributes the computations as well as the query to work around issues such as memory constraints that are encountered on a single machine. Our work has led to a system that is accessible via a web user interface, allowing users to define arbitrary computations and run queries on large amounts of data.

1. Introduction

As the world becomes more digitized, more data is being collected through every application and commercial product. There have been countless efforts to reconcile this large collection of data with efficient ways to house and query the datasets. Recently, the advent of “big data” has sprung up all around the database community, impacting the emerging application of statistics, information technology, and computer science onto various fields that are starting to rely on data analytics to guide their business decisions. We believe that our middleware efficiently and reliably provides small-time developers a simple way of prototyping computations with user-defined set of distributed databases. Notably, by supporting off-the-shelf products like MySQL, Java, and IP Address configurability for nodes, we hope to keep our software more available to the public (i.e. open source).

The technology itself is not necessarily unique from infrastructures embedded in commercial-grade products, but our middleware is targeted at small-time developers interested in rapid prototyping who are already familiar with a standardized set of tools. Recently, there has been a lot of related work in this field as distributed systems have become crucial in supporting scalable products. Google’s MapReduce [1] run with Google File System [2] (GFS) and Apache’s open source versions called Hadoop and Hadoop File System (HDFS) [3], were the basis and inspiration of our work on a distributed MySQL platform. However, their file systems are proprietary and cannot be generalized to other kinds of hardware. Additionally, the MapReduce algorithm is fundamentally different from what our middleware accomplishes because we are supporting native MySQL (i.e. relational database) queries and do not rely on simple, two-step computations. The HashJoin algorithm we implement is a derivative of MapReduce. Other more modern products include Apache Spark [4] and Presto [5] have gained traction as industry-grade ways to support continuous, big data operations with commercial
products. These middlewares require connections to distributed database or file system architectures like HDFS and Cassandra to source their data; because our middleware is built on top of MySQL users can create, update, and query tables natively, without relying on a third party file system. This allows for minimal setup and requires basic familiarity with relational databases, ideal for small-scale developers.

2. Approach

In the course of work on this project, we used a wide variety of tools. The middleware was implemented in Java. The web user interface was tested and run on Jetty. The nodes in the distributed infrastructure were deployed on Amazon Web Services (AWS). ANTLR, a third party library for parsing MySQL commands, was also utilized in the project.

2.1 System Architecture

The traditional use of a MySQL database (visualized in Figure 5) comprises of a server or machine connecting to a single MySQL instance to either update or query data in a MySQL table. The common approach used to perform some form of computation across the data stored in a MySQL table requires writing a program that queries all relevant data from the table and running the computation on the result of the query. However, there are a number of disadvantages to this model of database usage. Since there only exists a single instance of the MySQL database, a permanent failure of the instance would lead to loss of data while a temporary failure would affect the availability of the data. Furthermore, there is a limit to the amount of data a single MySQL database instance can store and one can expect an increase in latency of query or update computations as the amount stored in the database increases.

Our middleware (visualized in Figure 6), designed to tackle the issues of using a single database instance discussed above, implements a distributed database system. The system contains an arbitrary number of nodes, each connected to an arbitrary number of database instances. Each node is responsible for maintaining a connection to each database instance assigned to it, and running any query or update to be performed on the instance. There also exists a master node, which maintains a list of nodes in the system and periodically pings each node to make sure it is alive, while also intelligently routing updates or queries to the appropriate nodes. Additionally, the master node also hosts a User Interface through which the user inputs a query or update to execute on our distributed database system.

The data stored across all database instances in the platform collectively makes up the full dataset in our system. The master node defines a sharding algorithm based on hashing which it uses to route an update or query to the appropriate node and thus the appropriate database instance. Each node is aware and can communicate with every node in the system. This is required for our implementation of MySQL operations, such as JOIN, which needs to shuffle data across nodes. The user of the middleware is responsible for maintaining a configuration file, which specifies each node and its MySQL instance, as well additional
metadata for the system such as the replication factor of the data.

Since there is no bound on the number of databases that can be added to our system, there is no bound on the amount of data that can be stored in the system and thus effectively supports scalability. Furthermore, since each database instance contains only a chunk of the full dataset, the efficiency with which native MySQL computes query results or performs updates is maintained. Our system supports replication based on a user defined replication factor, and thus the failure of a database instance neither leads to loss of data nor affects the availability since the query or update can be routed to the secondary copy instead.

### 2.2 Parser/User Interface

Our system features a web user interface. The user interface shown in Figure 3 allows a user to input in a query and a computation class. After the necessary operations are run, the results are displayed on the web page. There is also a live update of the number of active and inactive nodes. Our middleware required the use of an intelligent parser to run distributed MySQL queries. Since our system was distributed, even simple queries such as INSERT or SELECT needed to be intercepted to provide the correct results. Every query could not just be sent to every node without interception. In the case of an INSERT query, the parser needed to recognize the table and the certain values from the queries; these values are needed to hash to the correct node or replicas. For SELECT queries, the same command could be sent to all of the nodes; if a node were down, there would be a SELECT routed to a backup table. Other commands such a CREATE TABLE could also be sent to every node. This wide variety of usage scenarios requires our parser to know exactly what query was run each time.

Because we wanted to support native MySQL queries, we implemented a parser so that we can interpret the user commands and run the appropriate branch of code. We used a third party tool called Another Tool for Language Recognition (ANTLR) to break down the user-inputted SELECT statements containing JOINS. We modified a specific MySQL grammar and used it as an input into ANTLR, which generated a set of Java files; the ANTLR-generated Java files model the user-inputted query as a tree, as seen in Figure 4, and allow us to do a depth-first parse of the query. On top of the depth-first parser, we built a schema for organizing what columns, tables, and aliases mapped to what joins and maintaining an order of execution. The tool is especially critical in scenarios where the user has nested or multi-joins. For simpler queries like updates and table creations, we perform manual parsing and pattern matching using native Java libraries.

### 2.3 Fault Tolerance

As discussed earlier, fault tolerance is a crucial part of distributed systems. The practicality of distributed system relies heavily upon its data reliability. Our system creates replicas of data on other nodes. The replication factor is a user defined parameter; the default is set to two. Our replication model works in a circular fashion. For a system with $n$ nodes with $r$
replicas, any arbitrary node $i$ will have replicas stored from node $i+1 \% n$ to node $i+r \% n$. The modulo allows wrap around and guarantees an equal amount of replicas on each node. Our system currently can detect when a node crashes and can reroute messages such an INSERT and SELECT to the replicas. The user will see no discernible difference in operation. Currently, JOINs are not fault tolerant.

### 2.4 Distributed JOINs

Since performing a JOIN operation on each MySQL instance is not sufficient to obtain the full result of the JOIN, our middleware implements a Hash Join [6] to compute the full set of join collisions. The middleware leverages the numerous nodes in the system to parallelize the computation of join collisions. The shuffle of the data amongst the nodes and the join collision detection is based on a user configurable hashing algorithm (SHA-1 by default). Any join collision is backed up into an intermediate MySQL table on each database instance, and any subsequent queries or updates to the join result are routed to this intermediate table. Our middleware performs multi-joins incrementally by first performing a join on two tables, and any further joins on the intermediate table resulting from the previous join iteration and the next remaining table.

### 2.5 Computations

Another feature of our middleware is the support for arbitrary computations to be performed before or after queries. This allows users to implement memory and resource intensive functions that would execute too slowly on a single node system. By running these computations on numerous nodes, we leverage the distributed processing power of multiple machines. Potential uses of the computation infrastructure include clustering, genomics analysis, and image recognition. In order to execute arbitrary computations, the user is required to write a Java class that implements an interface we have defined. The interface contains functions that can be run either before the shuffle stage of the JOIN or after the execution of the user inputted query. The interface also allows custom shuffling of data amongst the nodes during the JOIN.

### 3. Results and Measurements

To measure our system’s performance, we focused on two main metrics: scalability and speed. To measure speed, a baseline measurement was needed. Our Baseline JOIN (1 Node) was run after adding data from another node. This was done to compare to our system that has shards data across multiple nodes. Thus in the baseline measurement, all the data was be copied to a single database and then the JOIN was performed natively on the MySQL instance.

The first experiment, to test the performance of our system, was run on setups that contained 2 nodes, 4 nodes and 8 nodes, and the resulting data was compared with the
baseline measurement. This test involved 6 million total rows of data in each setup. Figure 1 below shows the outcome of the first test. The values on the left side of the graph correspond to the first test. The baseline JOIN operation on 1 node running a native MySQL JOIN took 213.5 seconds. The 2 node system took 467.33 seconds to run to completion, almost double the time of the baseline. However, when the system contains 4 nodes, the JOIN execution time drops down to 249 seconds. Finally, the 8 node system performs better than the baseline with a time of 139 seconds. This test shows that our system performs better with increasing number of nodes, since the latency of data communication will be overcome by the parallelization of join computation.

The second experiment tests the scalability of our system. It ran on setups of 2 nodes, 4 nodes, and 8 nodes. In this experiment, each node contained the same amount of data rather than the same total data across the nodes. Each database instance stored 3 million rows of data. Therefore the 8 node system had 24 million total rows, the 4 node system had 12 million total rows and the 2 node system had 6 million total rows. The right hand side of Figure 1 shows the results of this test. Even though, the 8 node system contains 4x the amount of data as the 2 node system, there is no significant increase in latency to compute the JOIN. This shows that the system is scalable; one can add reliably add more database instances to the system without a significant increase in latency of query computations.

![Figure 1](image)

A third experiment was run to measure the effect of adding a computation after the JOIN result was obtained. In this case, we simply measured a single node setup with native MySQL JOINs and our middleware with 4 nodes in the system. The single node setup had to perform the JOIN and sequentially perform computations 4 times on a $\frac{1}{4}$ of the data each time to provide the same result. The results of the experiment are shown in Figure 2 below. The results show that the bottleneck for the 1 node system is the computation; the computation takes more than half of the time for 1 node. However for the 4 node system, the opposite is true. This is expected
since the resource intensive computations are performed in parallel on our system as opposed to the single node system.

![Time Trials with JOINS](image)

Figure 2

4. Ethical/Privacy Concerns

A major ethical concern with the usage of any database system is the security and privacy of the data stored in the system. This is especially relevant in a distributed database system that transfers data from node to node, as is the case with our middleware. For instance, while performing a JOIN, our middleware shuffles unencrypted data from the relevant tables over the Wide Area Network. Furthermore, our middlewares requires the user to input the database passwords in a configuration file, which is a security vulnerability.

If this project was to be released as an open source software, we would need to implement the proper security safeguards against these issues. We could use public and private keys to encrypt data exchanged amongst the nodes in our system. Furthermore, we could also set up a more secure way to gain access to the database instance most likely requiring the use of public and private keys instead of passwords.

5. Discussion/Future Work

Given more time, there are a few areas where we could make improvements to our middleware. Specifically, we can improve performance by implementing the middleware in a lower language like C, which allows us to more tightly couple our middleware with our hardware architecture. This gives us more control over data structures used by the middleware and avoids some of the memory overheads presented by Java. We can also improve performance by
utilizing more multithreading when detecting JOIN collisions. Additionally, we would like our algorithm to support more robust JOIN queries (e.g. nested, outer join, etc.) and accept more SQL commands in general, like aggregation, grouping, and ordering. We would also like to make our fault-tolerance mechanisms more effective. This would require further extensive testing and potential changes to our architecture and the way our middleware routes queries in the case of faulted machines.

6. Conclusion

Overall, we hope this middleware will prove useful to beginner and intermediate level developers. With the proposed enhancements in future, we expect the efficiency and reliability of the system to offer an alternative to single node systems. The potential to open-source this project opens a whole new world for enhancements as well. With more people working on a system like ours, more features can rapidly be added to the system and the better the system can be optimized.

7. References

[1] Google MapReduce paper


[3] IBM Information on Hadoop

[4] Apache Spark Website
http://spark.apache.org/

[5] Presto Website
https://prestodb.io/

Appendix

Distributed MySQL Database System
This is a distributed database system that is optimized for big data analytics. It focuses on the distribution of JOINS and computations on large amounts of data.

MySQL Control Panel

Enter a Query
Enter computation class

Submit

6 Nodes Active
2 Nodes Down

Figure 3
Figure 4

Figure 5
Figure 6